HiP4-UPF: Towards High-Performance Comprehensive 5G User Plane Function on P4 Programmable Switches

Zhixin Wen and Guanhua Yan, *Binghamton University*

https://www.usenix.org/conference/atc24/presentation/wen

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

HiP4-UPF: Towards High-Performance Comprehensive 5G User Plane Function on P4 Programmable Switches

Zhixin Wen
Department of Computer Science
Binghamton University
zwen7@binghamton.edu

Guanhua Yan
Department of Computer Science
Binghamton University
ghyan@binghamton.edu

Abstract

Due to better cost benefits, P4 programmable switches have been considered in a few recent works to implement 5G User Plane Function (UPF). To circumvent limited resources on P4 programmable switches, they either ignore some essential UPF features or resort to a hybrid deployment approach which requires extra resources. This work is aimed to improve the performance of UPFs with comprehensive features which, except packet buffering, are deployable entirely on commodity P4 programmable switches. We build a baseline UPF based on prior work and analyze its key performance bottlenecks. We propose a three-tiered approach to optimize rule storage on the switch ASICs. We also develop a novel scheme that combines pendulum table access and selective usage pulling to reduce the operational latency of the UPF. Using a commodity P4 programmable switch, the experimental results show that our UPF implementation can support twice as many mobile devices as the baseline UPF and 1.9 times more than SD-Fabric. Our work also improves the throughputs in three common types of 5G call flows by 9-619% over the UPF solutions in two open-source 5G network emulators.

1 Introduction

The emerging 5G technologies have the potential of revolutionizing various sectors in society such as manufacturing, healthcare, transportation, agriculture, national security, and entertainment. 5G's promised performance improvements over previous generations of mobile communication networks, such as up to 20 times faster than 4G LTE, significantly lower latency, and supporting communications for as many as a million devices per square kilometer [14], introduces significant technical challenges to its network architecture, particularly for its User Plane Functions (UPFs).

5G UPFs act as the data traffic gateway between the access networks (e.g., base stations) serving the User Equipment (UE) (e.g., mobile phones) and the Internet or other data networks. UPFs apply rules received from the 5G control plane to decide how packets should be classified, inspected, metered, accounted, marked, buffered, and forwarded. When deployed for core networks, 5G UPFs must be implemented to handle intensive traffic volume at high speed while meeting stringent QoS requirements. Various *software-based optimization* techniques have been proposed to improve 5G UPF performances on multi-core commodity servers, including direct packet delivery to userspace memory with Data Plane Development Kit (DPDK) [5,6,22], latency reduction based on Intel network adapters' Dynamic Device Personalization (DDP) features [6], parallel lookup over multiple fields in a packet header using a single VPP operation [5], and fast packet processing based on the eXpress Data Path (XDP) feature of new Linux kernels [16]. Software-based UPFs, however, do not present a cost-effective solution for the emerging 5G networks as they can achieve only 0.03 Mpps (Million Packets per Second) per US dollar (USD) or 0.14 Mpps per Watt [13].

There have been a few recent works aimed at implementing 5G UPFs on P4 programmable switches [24, 26–28, 32, 41], which can deliver impressively 0.4 Mpps per USD or 8.52 Mpps per Watt [13]. Unfortunately, the limited computational resources available on commodity P4 programmable switches pose a daunting challenge to implement the comprehensive 5G UPF features mandated by 3GPP specifications [10]. To circumvent this challenge, the state-of-art works have either ignored or simplified some essential features in the implementations or resorted to a hybrid approach of combining both P4 switches and extra computational resources. For example, MacDavid's UPF solution [27] includes rudimentary usage reporting, while SD-Fabric [28] allows usage reporting for only a small set of end devices by default. On the other hand, the 5G UPF solution developed by Singh *et al.* combines a fast 5G datapath using P4 programmable switches and a slow one implemented by DPDK-based software [32]. X-Plane [26] extends the capacity of the 5G UPF by leveraging external Remote Direct Memory Access (RDMA)-based DRAM in addition to the P4 programmable switch. The co-existence of separate data paths or rule storage, however, significantly increases the complexity of managing and synchronizing UE state information in the data planes of 5G networks.

In this work we aim to develop a new high-performance 5G UPF called HiP4-UPF, whose comprehensive features, except packet buffering, can be deployed entirely on a commodity P4 programmable switch. Towards this goal, we first implement a baseline UPF based on prior works to identify the key performance bottlenecks.Through performance measurements we discover the inefficiency of SRAM and TCAM usage due to both the dependency among the tables storing UPF rules and the redundancy of matching keys and data stored in these tables. We also observe that frequent usage reporting can cause high operational latency.

To overcome the rule storage challenge, we develop a tiered optimization framework, which consists of three complementary optimization techniques, including removing excessive dependency, splitting rule tables, and consolidating action data. To reduce the high operational latency incurred by frequent usage reporting, we propose a novel scheme that combines pendulum table access and selective usage pulling. Pendulum table access enables instantaneous rule updates while pulling usage data from the switch ASIC with two alternating table partitions. The selective usage pulling technique estimates the urgency scores of individual rules to prioritize the collection of their counter data from the switch ASIC.

We develop a prototype of HiP4-UPF on a Tofino-based P4 programmable switch and evaluate its performances. Our results show that HiP4-UPF increases the maximal number of UEs by an average of 101.95% over the baseline UPF. We also make modifications to HiP4-UPF to match the features implemented by SD-Fabric [28], an open-source data plane fabric based on P4 programmable switches. Our results show that HiP4-UPF can support 1.9 times more UEs on a P4 switch than the 5G UPF solution provided by SD-Fabric. We also compare the responsiveness of HiP4-UPF to the requests from the 5G control plane against alternative UPF implementations. Our experiments show that HiP4-UPF not only improves the throughputs in three common types of 5G call flows by 9-619% over the CPU-based UPFs in two open-source 5G network emulators, free5GC [2] and open5gs [7], but also significantly reduces the operational latency in comparison with two alternative P4 switch-based UPFs. Finally, the selective usage pulling scheme has been shown capable of reducing the mean usage reporting latency by 84.6%.

Our implementation of HiP4-UPF has been made publicly available at https://github.com/CyberSecurityScience/HiP4-UPF/.

The remainder of the paper is organized as follows. Section 2 provides the background of this work, including a primer on 5G UPF and a baseline UPF implementation. Section 3 presents the performance measurements of the baseline UPF as the motivation behind this work. Section 4 introduces three rule storage optimization techniques and Section 5 discusses how to reduce operational latency. Section 6 gives the performance evaluation results. Section 7 discusses related work. Section 8 makes concluding remarks about this work.

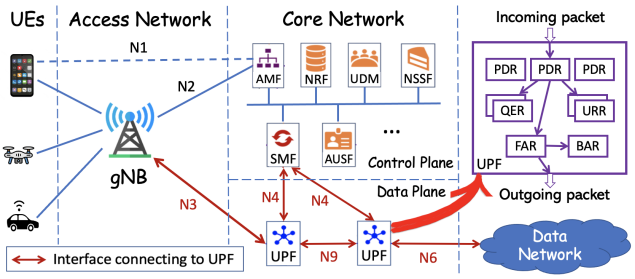

Figure 1: 5G network architecture. Key NFs are Access and Mobility Management Function (AMF), Session Management Function (SMF), Authentication Server Function (AUSF), Unified Data Management (UDM), NF Repository function (NRF), and Network Slice Selection Function (NSSF).

2 Background

This section first introduces the background about 5G UPF and then describes a baseline UPF based on prior work [27].

2.1 5G UPF Primer

Figure 1 illustrates the architecture of a typical 5G network. The UEs such as mobile phones connect with a 5G core network through an access network, which includes a collection of 5G base stations called gNBs. The control plane in the core network adopts a Service Based Architecture (SBA), whereby various network functions (NFs) communicate with each other through well-defined interfaces.

In the data plane, user data go through one or multiple UPFs between gNBs and external data networks such as the Internet and an IMS (IP Multimedia Subsystem). A UPF connects to a gNB via the N3 interface, where GRPS Tunnelling Protocol (GTP)-U tunnels are used to carry user data from/to the gNBs. These tunnels are terminated at the N6 interfaces, where the UPF forwards the user data to/from the external data networks. Optionally, multiple UPFs can be chained together in a data path for use cases such as home routed roaming. These UPFs are connected via the N9 interfaces, also based on the GTP-U protocol. Each GTP-U packet contains a Tunnel Endpoint Identifier (TEID) to identify its own tunnel.

Through the N4 interface, a UPF gets instructions from a Session Management Function (SMF) on how packets should be classified, inspected, metered, accounted, marked, buffered, and forwarded. The Packet Forwarding Control Protocol (PFCP), which runs on UDP, is used by the SMF to control the UPF. Each PDU session, which provides an end-to-end connectivity between a UE and a specific data network, is managed by a PFCP session between the SMF and the UPF.

2.2 Baseline UPF

To identify the performance bottlenecks in deploying UPFs on commodity P4 programmable switches, we implement a base-

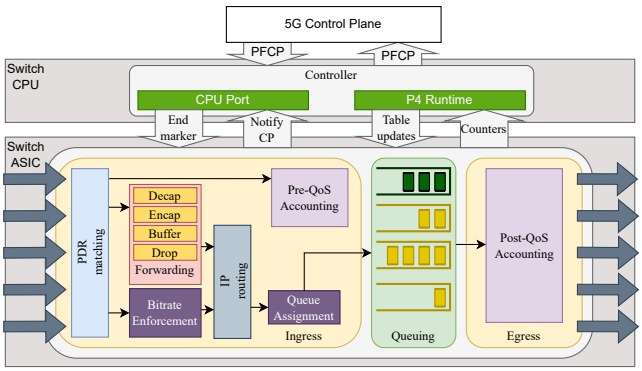

Figure 2: Architecture of the baseline UPF

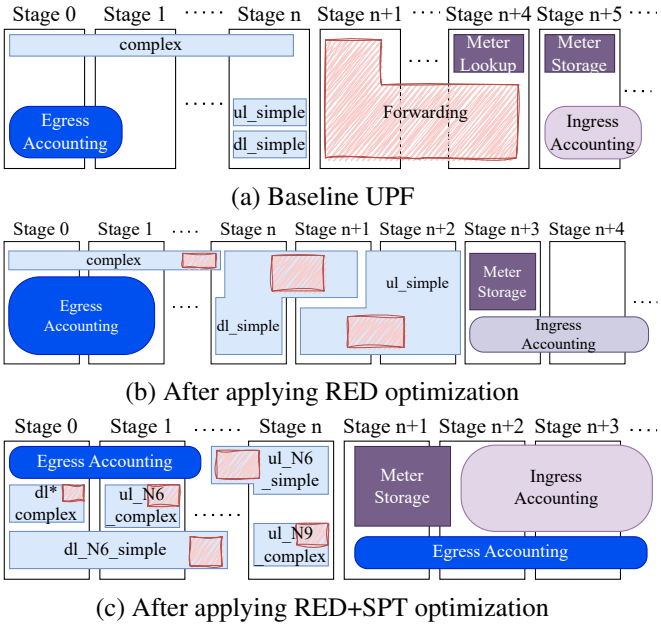

Figure 3: Illustration of table placement in the switch ASIC in different scenarios. Pink rectangles indicate forwarding data and purple rounded boxes indicate ingress accounting.

line UPF based on prior work [27] with its architecture shown in Figure 2. A primer on Intel Tofino-based P4 switches is given in Appendix A. The baseline UPF has two modules, *controller* and *packet processing pipeline* (PPP). The controller runs on the switch's CPU and is responsible for translating PFCP messages received from the control plane to rule table updates written in the P4 language. The PPP module, which runs on the switch ASIC, processes user data according to the rules provisioned by the control plane.

The PPP module consists of components that use these resources to process packets based upon the various rules provisioned by the control plane. These rules are maintained as follows in our implementation: ① **Packet Detection Rule (PDR)**: Each PDR is uniquely identified by a PDR_ID (24 bits). As done in prior work [27], we use three parallel PDR tables as follows. A *complex PDR table*, which is stored in TCAM, is used to map from a packet's packet detection information (PDI) (e.g., GTP-U TEID, QoS Flow Identifier (QFI), IP 5-tuple, packet direction (uplink or downlink), and Differentiated Services CodePoint (DSCP) markings) to its corresponding PDR_ID. We also use two *simple PDR tables* to keep exact match rules in the resourceful SRAM, one for uplink and the other for downlink. The uplink simple table matches packets based on their tunnel destinations and TEIDs, while the downlink one based on the UEs' IP addresses. There is a single entry per UE in either of these simple PDR tables. The structures of these tables are shown in Table 1. ② **Forwarding Action Rule (FAR)**: All FARs are stored in a *forwarding table*, which matches PDR_IDs to the corresponding forwarding actions. ③ **QoS Enforcement Rule (QER)**: QERs are used to enforce bitrates for selected PDU sessions based on TrTCM [21] meters, which are natively supported by Tofino-based switches. Our current implementation allows up to 1/8 of the rules in all PDR tables (simple uplink, simple downlink and complex) to use its own TrTCM meter. The meter objects allocated by the switch are stored in a *meter storage table*, indexed by their identifiers called METER_IDs. We also use a *meter lookup table* to find the METER_ID of the meter used for a given PDR_ID. ④ **Usage Reporting Rule (URR)**: A PDR can be mapped to an arbitrary number of URRs. It would be extremely wasteful of precious SRAM resources if we create a URR table that can accommodate the maximum number of URRs for every PDR. Hence, we use two tables to store counters for each PDR, one at the ingress port (*ingress accounting table*) and the other the egress port (*egress accounting table*), in the switch ASIC. The controller periodically pulls information from these counters and thereby generate usage reports based on URRs provisioned from the control plane. ⑤ **Buffering Action Rule (BAR)**: We do not implement buffering within the switch due to its limited resources. Instead, a buffering service is deployed on a different server. If the action given by a FAR indicates the need for buffering, the packet is forwarded to the buffering service.

The details of packet processing and usage reporting based on these rule tables are explained in Appendix B.

3 Motivation

Using the baseline UPF, we identify performance bottlenecks related to rule storage and operational latency.

3.1 Rule Storage

To understand how UPF rules are stored we consider a scenario where there are two exclusive sets of UEs. Each UE in the *simple UE set* has one rule in the ul_simple table and the other in the dl_simple table. Each UE in the *complex UE set* has two rules in the complex table. We use the complex/simple ratio to denote the ratio of the size of the complex

Table 1: PDR tables used by the baseline UPF. *set_ids* loads PDR_ID, QER_ID and FAR_ID from action data to packet metadata, *nop* does not do anything.

Table	Place	Match fields	Action	Action data
ul_simple	SRAM	TEID, QFI	set_ids | nop	PDR_ID
dl_simple	SRAM	Destination IP	set_ids | nop	PDR_ID
complex	TCAM	direction, TEID, QFI, IP 5-Tuple, DSCP	set_ids | nop	PDR_ID

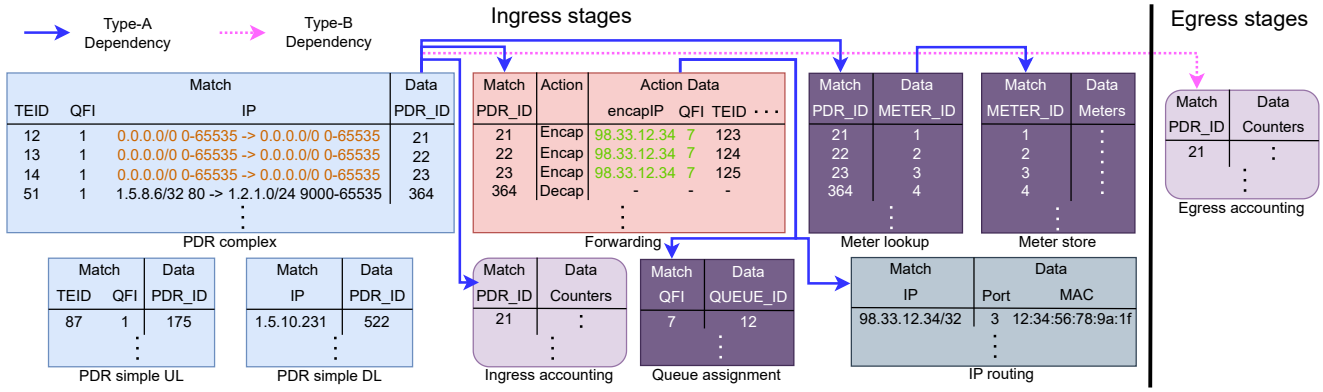

Figure 4: Illustration of rule storage inefficiencies by baseline UPF. Same matching keys and same action data are highlighted in brown and green, respectively. Best viewed in color.

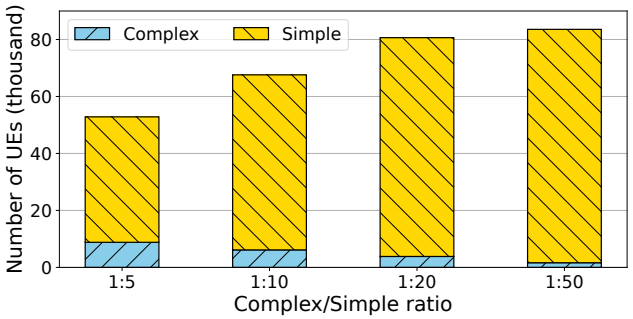

Figure 5: Maximal numbers of UEs supported by the baseline UPF under different complex/simple ratios

UE set to that of the simple UE set. To find the maximal number of UEs under a certain complex/simple ratio, we search the size of the simple UE set in an increment of 1024 while calculating the number of the complex UE set accordingly. The P4 compiler generates an error if the computational resources on the P4 switch cannot support the number of UEs requested. We vary the ratio among 1:5, 1:10, 1:20, and 1:50, and report the maximal number of UEs supported by the P4 switch in Figure 5. We observe that as the fraction of simple UEs becomes larger, more UEs can be supported. When the complex/simple ratio is 1:50, the P4 switch can accommodate at most 83.5 thousand UEs. By contrast when the ratio is 1:5, only 52.8 thousand UEs can be supported.

To gain deep insights into how the SRAM and TCAM resources are utilized, we use Intel's P4Insight tool [3] to identify the placements of the rule tables in the P4 switch's ASIC when the complex/simple ratio is 1:5. The table placements are shown in Figure 3(a). We notice that although the rule tables are distributed across all available stages on the Tofino ASIC and the utilizations of the SRAM and TCAM are 54.4% and 49.3%, respectively. Among them, only 2.3% of SRAM is used for each of the ul_simple and dl_simple tables, while 48.6% of the TCAM is used for the complex table. Also, 17.1% of the SRAM is used for usage accounting.

Inefficient rule storage can result from the dependency existing among different rule tables. We use Figure 4 to demonstrate how rules are stored by the baseline UPF. There are two types of table dependencies. *Type-A dependency* exists within ingress stages. For example, both packet forwarding based on the forwarding table and pre-QoS accounting based on ingress accounting table require the lookup results of the PDR tables, suggesting that the PDR tables must be placed at an earlier stage than the forwarding and ingress accounting tables as shown in Figure 3(1). Such dependency can lead to resource usage disparity among different stages, which affects the resource utilization on the P4 switch. *Type-B dependency* occurs between the ingress and egress phases. As shown in Figure 4, use of the egress accounting table also needs the PDR_IDs resulting from the lookup of the PDR tables. However, as the PDR_IDs are carried within the packets when they travel from the ingress to the egress phases, they effectively eliminate the necessity of placing the egress accounting table in earlier stages than the PDR tables.

There also exists ample redundancy among the different rule tables in the baseline UPF. Such redundancy can occur

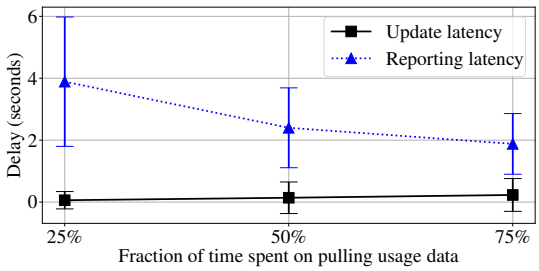

Figure 6: Operational latency observed for the baseline UPF

in both matching keys and action data. Figure 4 shows both redundant match fields stored in the PDR complex table and redundant action data stored in the forwarding table.

3.2 Operational Latency

The second performance bottleneck occurs when a 5G network requires both frequent usage reporting and short operational latency. Ideally, when 5G networks are used to support Ultra-Reliable Low-Latency Communication (URLLC) applications such as autonomous vehicles and augmented/virtual reality, the control plane latency should be less than 10 milliseconds [9]. The baseline UPF uses two accounting tables, one for pre-QoS accounting during the ingress phase and the other for post-QoS accounting during the egress phase. These tables need to be read from the switch ASIC to the CPU when performing usage reporting while they need to be written when there are rule updates (e.g., PDU session establishment). However, the P4 switch allows only one type of operation (read and write) at a time on each table stored in its ASIC. To circumvent this issue, we use two threads, one responsible for usage reporting and the other for rule updates. As usage reporting can occur frequently, we let the corresponding thread sleep for a certain period after it finishes pulling all counter data from the switch ASIC to allow rule updates by the other thread. We vary the sleep duration to control the fraction of time spent on pulling counter data from the switch ASIC by the usage reporting thread, which is denoted by β.

We conduct experiments to measure the delays of usage reporting and rule update by the baseline UPF. For reporting latency, we consider volume-based reporting: a report is triggered when a traffic quota is reached. Traffic is generated by Cisco's TRex tool [8], which runs on a server connected to the P4 switch through a 100G Ethernet cable. We record the time when traffic quota is reached at the traffic generator, denoted by t_g, and the time when the report arrives at the SMF, denoted by t_r. As it is difficult to measure the exact time when the traffic volume exceeds the quota at the P4 switch, we first measure the round trip time between the traffic generator and the P4 switch, denoted by t_p. After 20 runs, t_r is measured to be 0.2 ± 0.034 milliseconds. As t_r has negligible variation we estimate the report latency to be $t_r - (t_g - mean(t_p)/2)$.

For update latency we measure the delay between the times when a handover PFCP request message is sent by the SMF and when the response message from the UPF is received at the SMF. Figure 6 shows both reporting latency and update latency when we vary β among 25%, 50%, and 75%. The results illustrate the obvious contention between reporting latency and update latency: when we increase β from 25% to 75%, the mean reporting latency decreases from 3.9 to 1.9 seconds, while the mean update latency increases from 60 to 230 milliseconds with the 99-percentile update latency increasing from 1.5 to 2.7 seconds. Moreover, regardless of how β is chosen, both types of operational latency significantly exceed the desirable control plane latency of 10 milliseconds for URLLC applications in 5G networks. Therefore, there is a need for optimizing the baseline UPF to further reduce both rule update latency and usage reporting latency.

4 Rule Storage Optimization

Our performance measurement results shown in Section 3.1 reveal that the number of UEs supported on a P4 programmable switch can be affected by both rule dependencies and rule redundancy across different rule tables used by the data planes of 5G networks. This section describes a three-tiered optimization framework to overcome inefficient rule storage due to these challenges.

4.1 Remove Excessive Dependency (RED)

From Figure 3(a), we see that the two PDR simple tables are placed in the same last stage of the complex table (i.e., Stage n). This is because all the PDR tables must be placed before the forwarding, ingress accounting, and meter lookup tables, all of which process packets based on PDR_IDs obtained from the PDR tables. Moreover, these downstream tables have to store the mapping keys (i.e., PDR_IDs), which incur additional SRAM usage.

To overcome this issue, we eliminate the forwarding and meter lookup tables from the baseline UPF, and then for each PDR_ID, merge its associated action and action data in each of these tables into the corresponding entry in the PDR tables. However, we cannot remove PDR_IDs altogether from the PDR tables because they are needed to index the egress accounting tables, which cannot be merged into the PDR tables because they are used in the egress phase.

The table placement after applying RED is shown in Figure 3(b). We can see that the forwarding and meter lookup data have been merged into the three PDR tables (complex, dl_simple, and ul_simple). However, even with the RED optimization, the two simple PDR tables have to be placed no earlier than Stage n, where the complex PDR table ends, because the result of being a lookup hit or miss from the complex PDR table is known at this stage. This can lead to considerable resource under-utilization from Stages 0 to n-1, except

Table 2: List of split PDR tables. The *drop* flag decides whether to drop the packet. The *nocp* flag decides whether to notify the control plane. The *mark_dscp* flag decides whether to trigger DSCP marking; if set, the value from the *dscp_val* field is used for marking. The *buffer* flag decides whether buffering is needed.

Table name	Place	Match fields	Action	Action data
ul_to_n6_simple	SRAM	TEID, QFI	set_ids | nop	PDR_ID, drop, nocp, METER_ID, mark_dscp, dscp_val
ul_to_n6_complex	TCAM	TEID, QFI, IP 5-Tuple, DSCP	set_ids | nop	PDR_ID, drop, nocp, METER_ID, mark_dscp, dscp_val
ul_to_n9_n3_simple	SRAM	TEID, QFI	set_ids | nop	PDR_ID, drop, nocp, TUNNEL_IP, TEID, QFI, METER_ID
ul_to_n9_n3_complex	TCAM	TEID, QFI, IP 5-Tuple, DSCP	set_ids | nop	PDR_ID, drop, nocp, TUNNEL_IP, TEID, QFI, METER_ID
dl_from_n6_simple	SRAM	Destination IP	set_ids | nop	PDR_ID, drop, nocp, buffer, TUNNEL_IP, TEID, QFI, METER_ID
dl_from_n6_complex	TCAM	IP 5-Tuple, DSCP	set_ids | nop	PDR_ID, drop, nocp, buffer, TUNNEL_IP, TEID, QFI, METER_ID
dl_from_n9_simple	SRAM	TEID, QFI	set_ids | nop	PDR_ID, drop, nocp, buffer, TUNNEL_IP, TEID, QFI, METER_ID

that egress accounting can be performed in these stages due to its Type-B dependency on the PDR tables (see Section 3.1).

Following the same example shown in Figure 4, the RED technique merges the forwarding table and the meter lookup table into the PDR complex table. Although this operation expands the action data of the PDR complex table, it does not affect its total number of rule entries.

4.2 Split PDR Tables (SPT)

Our second optimization aims to reduce the action data for packet forwarding, which have been merged into the PDR tables by the RED scheme. The key observation here is that when packets are forwarded to different interfaces by UPF (i.e., N3, N6, and N9), their matching fields and action data needed are different. The SPT scheme splits the PDR tables into smaller ones based on forwarding interfaces.

Table 2 lists all the split PDR tables after applying this technique. Its ul_to_n6_simple and dl_from_n6_simple tables are the same as ul_simple and dl_simple, respectively, in Table 1, while its other tables are derived by splitting the complex table in Table 1. Based on these tables packets are processed as follows. ① Packets going to a DN through the N6 interface are processed based on the rules in *ul_to_n6_simple* or *ul_to_n6_complex*. As the UPF is a terminating one, it has the option of DSCP marking and the action data include both mark_dscp (a boolean indicating whether DSCP marking is necessary) and a specific DSCP value. ② For uplink packets going to another UPF via the N9 interface (i.e., this UPF serves as an I-UPF/UL-CL UPF) or those that go to another gNB via the N3 interface in case of N2 handover, they are processed based on the rules in *ul_to_n9_n3_simple* or *ul_to_n9_n3_complex*. As this UPF is not serving as a terminating one, DSCP marking is not needed. QFI is used to mark flows for QoS enforcement internally within the mobile network. As packet encapsulation is needed, a TEID and a TUNNEL_IP (i.e., the destination IP address used in the tunnel header) are included within the action data for the new GTP-U tunnel header. ③ Downlink packets from a DN via the N6 interface are processed by the rules in *dl_from_n6_simple* or *dl_from_n6_complex*. Before being forwarded to an I-UPF or a gNB, these packets are encapsulated with a tunnel header, which thus requires a TEID and a TUNNEL_IP in the action data. The action data also include the need for buffering or notifying the control plane. ④ Downlink packets from another UPF via the N9 interface need to be encapsulated within a new GTP-U tunnel header, thus requiring a TEID and a TUNNEL_IP. As these packets are forwarded among UPFs, they are processed only by the simple rules stored in *dl_from_n9_simple*.

As seen from Figure 3(c), the SPT optimization technique has the benefit of splitting the original PDR tables into smaller ones, which allow a mixture of simple and complex tables to fit into the same early stages. As each rule in the complex table in the baseline UPF is assigned to only one of the specialized tables after PDR table splitting, SPT does not affect the overall number of entries in all the PDR tables. However, by reducing the sizes of either match fields or action data for the complex rules, SPT decreases the storage space for the PDR tables.

4.3 Consolidate Action Data (CAD)

Our third optimization is motivated by the observation that multiple UEs can share some common action data. For example, different PDU sessions may have their downlink packets be forwarded to the same base station, classified by the same QFI, and processed with the same flags (i.e., *buffer*, *drop*, and *nocp*). Although these UEs cannot use exactly the same action

data due to different TEIDs, we can consolidate those shared fields into the same template, in hopes that the number of bits to represent the entire action data can be reduced.

It is noted that after our first two optimizations, the PDR_IDs in Table 2 are used only to look up the egress accounting table, as other action data have already been merged into the PDR tables. The key idea of the CAD optimization technique is to encode the templates as described above into these PDR_IDs. As an example, suppose that 200 UEs share the same downlink gNB IP 154.100.10.8 and QFI value 1. We can assign their downlink PDRs with PDR_ID values from 256 to 455, which allows a *single* ternary match 0b0001,xxxx,xxxx to match any PDR_IDs ranging from 256 to 511. Given any such matched PDR_ID, the UPF performs encapsulation with shared action data (Action=Encap, IP=154.100.10.8, QFI=1). For ease of presentation, we call each unique combination of action and action data an *action template*. Based on packet directions and action types (e.g., encapsulation or decapsulation), we define three types of action templates, each stored in a particular action template table. The action template tables, which are stored in TCAM, can be matched in parallel because the Tofino chip supports ternary match over multiple tables in each stage. With this design, all Boolean flags, QFIs and TUNNEL_IPs are removed from the PDR tables completely, because they can be derived from the much compressed action template tables instead.

The controller is responsible for assigning PDR_ID ranges to different action templates. As ternary match is based on a matching value along with masked bits, we denote a *PDR_ID range* as a tuple (r,m) such that it covers the PDR range $[r \times 2^m, (r+1) \times 2^m - 1]$. Following the previous example, a ternary match of 0b0001,xxxx,xxxx can be achieved by a PDR_ID range of $(1,8)$. For ease of presentation, we also use (R,M), where $|R| = |M|$, to denote a set of PDR_ID ranges with $\{(r_k,m_k)|r_k \in R, m_k \in M\}$.

The PDR_ID assignment problem is formulated as follows. Let $T = (t_1,t_2,...,t_n)$ be a collection of action templates which need to be assigned with actionable PDR_ID ranges. Also, let $L = (l_1,l_2,...,l_n)$ include the minimum number of PDU sessions that can be applied to each action template in T. Hence, for each action template $t_i \in T$, l_i provides a lower bound on the size of PDR_ID range assigned to t_i. Both T and L are providedby the Mobile Network Operator (MNO) during UPF startup. Our goal is to find a set of *disjoint* PDR_ID ranges (R_i,M_i) to each $t_i \in T$, such that the total number of entries in the action template tables (i.e., $\sum_i |M_i|$) is minimized. The optimization problem should be solved with the following constraints satisfied. *First*, for each t_i we have $\sum_{m \in M_i} 2^m \geq l_i$. The constraint is used to ensure that, for each action template, its allocated disjoint PDR_ID ranges should have the capacity to accommodate the minimum number of PDU sessions provisioned by the control plane. *Second*, there is an upper bound on the number of bits to represent each PDR_ID, which is denoted by Θ. *Third*, we consider assignments of only disjoint

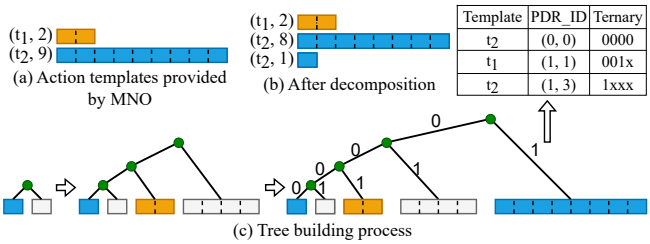

Figure 7: An example showing assignments of PDR_ID ranges to action templates by the bootstrapping algorithm

PDR_ID ranges to avoid non-deterministic behavior from the switch (if overlapping PDR_ID ranges are assigned to different action templates) or to prevent redundancy (if they are given to the same action templates).

We address this PDR_ID assignment problem with two algorithms. The *bootstrapping* algorithm finds an initial assignment scheme with both T and L provided by the MNO. As the PDRs provided by the control plane change over time with new PDU sessions established and old ones removed, such changes are addressed by the *runtime update* algorithm.

Bootstrapping. We use an example to explain the basic idea of the bootstrapping algorithm, whose details can be found in Appendix C. Let the provisioned number of bits for PDR_IDs be 4 (i.e., $\Theta = 4$). Figure 7(a) shows that initially two action templates, t_1 and t_2, are provided by the MNO, and they require at least 2 and 9 PDR_IDs to be assigned, respectively. Hence we have $T = \{t_1,t_2\}$ and $L = \{l_1 = 2, l_2 = 9\}$. The bootstrapping algorithm consists of two steps, *decomposition* and *tree-building*. To explain these two steps, we introduce a few notations here.

We use $log_2(x)$ to represent the binary logarithm of x. For each $z = (t,l)$, we define its *looseness* score to be $\gamma(z) = 2^{\lceil l \rceil} - l$. Intuitively speaking, $\gamma(x)$ indicates how many spare IDs there are if a smallest PDR_ID range is assigned to it. When the provisioned number of bits (i.e., Θ) is insufficient to encode all the action templates requested by the MNO, the decomposition step iteratively decomposes the one with the highest looseness score into two with a smaller looseness score combined.

In the same example, for (t_2,l_2), any PDR_ID range (r,m) assigned to it must have $m \geq 4$. However, as $\Theta = 4$, it means that the provisioned number of bits is insufficient to assign PDR_ID ranges to (t_1,l_1) any more. The decomposition step overcomes this issue by breaking (t_2,l_2) into two, $(t_2,8)$ and $(t_2,1)$, as seen from Figure 7(b). The reason why (t_2,l_2) is chosen because it has a larger looseness score (i.e., $\gamma((t_2,9)) = 7 > \gamma((t_1,2)) = 0$). After decomposition, there are three (t,l)-tuples, each having a looseness score of 0. Assuming that a smallest PDR_ID range is assigned to each (t,l)-tuple, the sum of the lengths of these ranges is only 11, which can be accommodated by the Θ bits provisioned by the MNO.

The tree-building step, which is illustrated in Figure 7(c), performs the task of assigning the PDR_ID ranges to the (t,l)-

tuples produced from the decomposition step. For each such tuple, it creates a leaf node whose capacity is defined to be the total number of IDs if a smallest PDR_ID range is assigned to it. The algorithm runs iteratively, and in each iteration the two nodes with the smallest capacities are merged. If the two nodes (u and v) have different capacities, the smaller one (i.e., u) is recursively paired with an empty leaf node of the same capacity (shown in white boxes in Figure 7(c)) to form a parent node with a doubled capacity (shown as circle nodes in Figure 7(c)) until the parent node has the same capacity as the larger node (i.e., v). After that u and v are paired to form a parent node with a doubled capacity. The process repeats until there is only a single root node. It is easy to see that the tree as constructed above must be a strict binary tree. With this tree, the depth-first traversal algorithm can be used to visit all nodes in the tree. Each node can be coded by the traversal path, which appends bit 0 if a left branch is taken or bit 1 if a right one is taken. Whenever a non-empty leaf node is visited, its associated template is assigned by a PDR_ID range (r, m), where r is the code of the leaf node and m is the binary logarithm of the node's capacity.

Runtime update. At runtime, old PDR rules can be removed or new ones can be added, suggesting that PDR_ID range assignments should be adjusted dynamically. Two respective procedures, *deallocate* and *allocate*, are thus developed. They both operate on the same strict binary tree with augmented fields and their details are given in Appendix D.

5 Operational Latency Optimization

Although the switch ASIC constantly updates the accounting data of each PDR according to its matched packets, they have to be collected from the switch ASIC to obtain the current usage data. As explained in Section 3.2, when there are a large number of PDU sessions, frequent usage reporting can cause excessively high operational latency. The root cause is that the P4 programmable switch does not allow simultaneous read and write operations on the same rule table stored in its switch ASIC. To overcome this challenge, we propose two techniques, *pendulum table access* and *selective usage pulling*, for HiP4-UPF. Pendulum table access uses two separate partitions for each accounting table in the switch ASIC to enable simultaneous read and write operations on the two partitions of the same accounting table, while the selective usage pulling scheme prioritizes the PDRs whose accounting data should be pulled with high urgency from the switch ASIC.

To accommodate these two methods, we restructure the original UPF Controller shown in Figure 2 into two components, *North Controller* and *South Controller*. The North Controller interacts with the 5G control plane (i.e., the SMFs) through PFCP protocols and performs PDR_ID assignments, while the South Controller interacts with the P4 switch's ASIC through its ASIC driver. The two controllers, both of which are executed by the switch CPU, communicate with each other through Unix raw sockets.

The split in functionalities between the two controllers strikes a balance between security and performance. The North Controller is implemented by the secure Rust programming language, whose strict enforcement of thread and memory safety can prevent code injection attacks posed by the insecure UDP-based PFCP protocol messages from the N4 interface. It uses the P4 runtime library to perform typically infrequent updates of all UPF tables but PDR tables. The South Controller is responsible for latency-sensitive operations including rule updates for the PDR tables and reading both the ingress and egress accounting tables. The South Controller, which is implemented in C++ on top of the native ASIC driver, enables fast custom data serialisation based on FlatBuffers [1]. The ASIC driver's APIs also allow multi-threaded table update and reading, so different tables can be read and updated at the same time, which cannot be achieved by the P4Runtime interface.

5.1 Pendulum Table Access

The South Controller uses two read threads and one write thread to parallelize the tasks of reading the usage data and updating the PDR tables. The two read threads are used for reading the ingress and egress counters, respectively. For both ingress and egress accounting, its accounting table has two non-overlapping partitions, which are referred as Accounting Table Partition (ATP) 1 and 2, respectively. Note that both ATPs, along with the PDR tables, are stored in the switch ASIC. Without loss of generality our following discussion considers only the read thread used for egress accounting. Each of the two ATPs can be in an either `READ` or `WRITE` state. When an ATP is in a READ state, its accounting data can be pulled from the switch ASIC. By contrast, when an ATP is a `WRITE` state, it means that its rules can be updated by the write thread but its accounting data should not be pulled by the read thread. The two partitions of the same accounting table operating in different states enable simultaneous read and write operations on the same accounting table stored in the switch ASIC.

As two separate partitions are used for the same accounting table, we need to ensure that the rules are coherently maintained in these two partitions under dynamic rule updates (i.e., rule insertion, rule modification, and rule deletion) while the accounting data read for each PDR should be accurate within a reasonable report delay. Towards this goal, the South Controller uses the following data structures. It keeps the index of the ATP used for reading and writing as I_r and I_w, respectively, during the current pulling period. Read/write accesses to both indices are protected by a mutex exclusion semaphore. The South Controller also stores two maps, M_{last} and M_{expire}, which maps PDR_IDs to timestamps. M_{last} records the last time when a rule entry is pulled from the switch ASIC, and M_{expire} denotes the time when a rule entry should expire. As

M_{expire} is used by both the read thread and the write thread, access to it is protected by a mutex exclusion semaphore. M_{last} is used only by the read thread so no protection is needed.

The read thread interleaving pulls the counter data from the two ATPs as follows. We use τ to denote the start time of the current pulling period. Once it finishes pulling the accounting data from ATP I_r, it performs the following: it notifies the North controller of the latest accounting data; for each entry with PDR_ID=k in ATP I_r, $M_{last}[k]$ is set to τ; for every PDR_ID $k \in M_{expire}.keys()$, if $M_{expire}[k] \leq M_{last}[k]$, it removes the corresponding entry from the ATP where it is stored and then notifies the North Controller of the deletion of this rule; it swaps I_r and I_w, which effectively changes the roles of both ATPs. Finally, τ is updated to be the current wallclock time, indicating the start of the next pulling period.

The write thread monitors new rule update requests from the North Controller. There can be three types of rule update operations, which are handled differently as follows.

(1) Rule insertion. For insertion of a rule that is newly allocated by the North Controller, the write thread simply adds an additional entry to the PDR table. A corresponding entry is also added to the ATP indexed by I_w. For a newly inserted rule k, its $M_{expire}[k]$ and $M_{last}[k]$ are initialized to be infinity and the current wallclock time, respectively. The write thread reports successful update to the North Controller, which further finishes the corresponding PFCP operation.

(2) Rule modification. Rule modification, which replaces an old rule with a new one, can occur frequently in a 5G network due to handover operations. Due to its complexity, we use an example illustrated in Figure 8 to explain how it is accomplished. We assume that at time t_0 the read thread starts reading from ATP 1. ⓐ The North Controller receives a PFCP request from the SMF, which requires an old rule with PDR_ID=10 to be replaced by a new one. The North Controller allocates a new PDR_ID=20 and then sends a rule modification request to the South Controller, which is received by its write thread at t_1. ⓑ The write thread updates the PDR table to produce an entry with PDR_ID=20 instead of 10. This update can be a single action data modification if the match keys are unchanged, or an entry deletion followed by an insertion in case of changed match keys. The write thread sets $M_{expire}[10]$ to be the current wallclock time (i.e., t_1). The write thread next inserts a new entry with PDR_ID=20 into ATP 2. As Intel's Tofino driver allows these table update operations to be performed within a single transaction, traffic accounting switches from PDR_ID=10 to PDR_ID=20 instantaneously. ⓒ The write thread reports successful update to the North Controller, which further finishes the corresponding PFCP operation. It is noted that this update procedure from ⓐ to ⓒ does not involve any time-consuming operation, thus allowing for low rule update latency.

ⓓ The read thread finishes pulling counter values from ATP 1 and sends them to the North Controller for usage reporting. For each rule with PDR_ID=k in ATP 1, it sets $M_{last}[k]$ to be t_0, the start time of the past pulling period. While doing so, it finds that $M_{expire}[k] > M_{last}[k]$, which means that there are some residual counter value left in ATP 1 for the old rule with PDR_ID=10. Finally, the read thread switches the roles of ATPs 1 and 2 by swapping I_r and I_w. This starts the next pulling period for which ATP 2 is used.

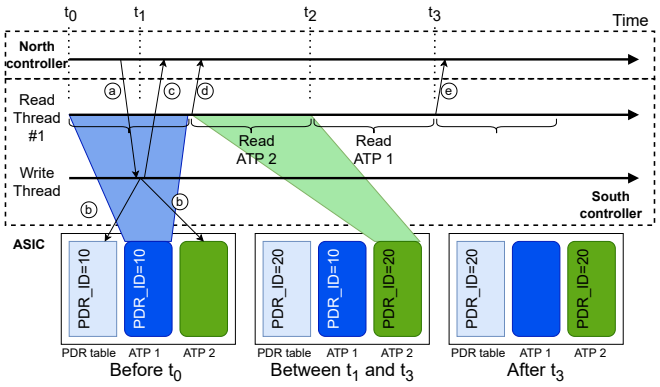

Figure 8: An example showing a rule modification operation

ⓔ At time t_3, the read thread finishes pulling the counter data from ATP 1, again. After updating $M_{last}[10]$ to be t_2, as it finds that $M_{expire}[10] \leq M_{last}[10]$, it deletes the entry from ATP 1 and notifies the North Controller of the completion of the rule with PDR_ID=10 in addition to the latest counter data. On the arrival of this notification, the North Controller frees PDR_ID 10 using its deallocation procedure. Later the North Controller will add counter values from both PDR_ID=10 and PDR_ID=20 to create usage reports. The read thread removes PDR_ID=10 from M_{last} and M_{expire}.

Note: Consider a different case where PDR_ID=10 is in ATP 2 when a rule modification request arrives at the write thread at time t_1. The final counter data for PDR_ID=10 is read during the second pulling period and, as done in Step ⓔ, the North Controller is notified of its completion at time t_2.

(3) Rule deletion. Suppose that the rule has PDR_ID=k. The write thread removes PDR_ID=k from the PDR table and sets $M_{expire}[k]$ to the current wallclock time. The write thread reports successful update to the North Controller, which further finishes the corresponding PFCP operation. Later when $M_{expire}[k] \leq M_{last}[k]$, the same procedure in Step ⓔ happens: the South Controller notifies the North Controller of the completion of rule PDR_ID=k, which allows the latter to deallocate PDR_ID=k.

Analysis: If a PDU session has multiple PDR_IDs assigned to it, it is possible that their final counter data are reported to the North Controller out of order with respect to the times at which they are assigned. Following the same example in Figure 8, if PDR_ID=20 is requested to be deleted within the first pulling period (which is ended by Step ⓔ), the final counter data for PDR_ID=10 is reported to the North Controller at time t_3, but the final counter data for PDR_ID=20 is reported to the North Controller at time t_2 even though PDR_ID=20 is

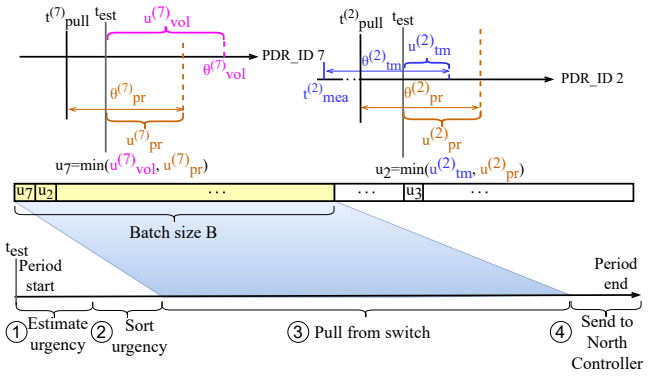

Figure 9: HiP4-UPF's usage reporting timelines

assigned to the PDU session later than PDR_ID=10. Hence the North Controller needs to wait for counter data for all the PDR_IDs assigned to the same PDU session before it reports its overall usage to the SMF.

5.2 Selective Usage Pulling

It is often the case that counter data are not immediately needed to generate usage reports. To further reduce operational latency, we propose a selective usage pulling scheme, which prioritizes those PDRs with high urgency of usage reporting. Let $S = (s_1, s_2, ..., s_n)$ denote the set of PDRs in the UPF. The North Controller sends URRs along with PDR updates to the South Controller. The South Controller sorts these PDRs based on their urgency scores.

To support selective usage pulling, the read thread performs additional steps as illustrated at the bottom of Figure 9. ① The thread loops through *all* PDRs among *all* PFCP sessions' URRs and update their urgency scores stored in a sorted map U. ② The PDRs in U are then sorted based on their updated urgency scores. ③ The read thread pulls the counter data of PDR_IDs belonging to the current ATP for up to B PDRs with the highest urgency scores from the sorted map U in a batch. By default we use $B = 10,000$. ④ The counter values are sent to the North Controller as discussed in Section 5.1. The time spent on Steps ① and ② is much shorter than the time spent on the pulling operation during Step ③.

We next explain how to estimate the urgency score u_i of each PDR s_i in Step ①. Suppose that the read thread estimates the urgency score at time t_{est} which is at the beginning of a pulling period. Let the last time at which the PDR's counter data were pulled be $t^{(i)}_{pull}$. For each PDR s_i we keep the number of bits that has already been counted as q_i.

We consider any URR provisioned by the control plane, R_j, with a set of associated PDRs. The same PDR can appear in two different URRs but the PDRs associated with the same URR must differ. As a URR can contain different types of triggers for usage reporting, we calculate urgency scores based on three factors, *volume*, *time*, and *periodic*.

Volume: Let $\theta^{(j)}_{vol}$ be the volume threshold for URR R_j which, if crossed, triggers usage reporting. We measure the observed bitrates of all PDRs, which are updated whenever their counter data are pulled. We use an Exponential Moving Average (EMA) method to estimate the bitrate with factor α: at every time step, the estimated bitrate b^i_{est} is updated as $\alpha \cdot b^i_{est} + (1-\alpha) \cdot b^i_{obs}$, where b^i_{obs} is the observed bitrate at the current step for PDR s_i. We set $\alpha = 0.95$.

As a volume-based reporting rule can pull usage data from multiple mutually exclusive PDRs, we can sum up the bitrate and counted volume for a URR R_j as $b^{(j)}_{est} = \sum_i b^i_{est}$ and $q^{(j)} = \sum_i q_i$, respectively, for all PDR s_i that appear in R_j.

The *volume urgency* of any PDR s_i, denoted by $u^{(i)}_{vol}$, can be estimated as $\min_j\{\max\{0, (\theta^{(j)}_{vol} - q^{(j)})/b^{(j)}_{est}\}\}$ with $s_i \in R_j$. That is to say, the volume urgency of PDR s_i is the minimum estimated time for any of the URRs containing s_i to reach its volume threshold. In Figure 9, PDR_ID 7 use volumes to trigger usage reporting.

Time: 5G networks allow time-based reporting to operate in two modes. The measurement can be immediately started when the URR is provisioned (mode A), or after the first matching packet has been detected by the UPF (mode B, which is the default option). In either mode, let $t^{(i)}_{mea}$ denote the time at which measurement is started for PDR s_i. Also let $\theta^{(i)}_{tm}$ be the time threshold which, if crossed, triggers usage reporting. Then the *time urgency*, denoted by $u^{(i)}_{tm}$, can be estimated as $\max\{0, \theta^{(i)}_{tm} - (t_{est} - t^{(i)}_{mea})\}$. In Figure 9, PDR_ID 2 requires time-based usage reporting.

Periodic: We define the *periodic urgency*, denoted by $u^{(i)}_{pr}$, as $\max\{0, \theta^{(i)}_{pr} - (t_{est} - t^{(i)}_{pull})\}$, where $\theta^{(i)}_{pr}$ indicates how frequently the PDR's counter data should be collected. We assume that there is an imaginary timer for two different use cases. First, if the PDR requires periodic reporting, the timer can be periodically fired to trigger pulling. In this case, we can simply set $\theta^{(i)}_{pr}$ to be the time interval of periodic usage reporting requested by the control plane. Second, if periodic usage reporting is not needed explicitly, the timer can still be used to prevent starvation caused by selective usage pulling and thus improve the accuracy of bitrate estimation in volume urgency calculation. When it fires, the urgency of the PDR is elevated to ensure that its counter data should be pulled immediately. We set $\theta^{(i)}_{pr}$ to be 5 seconds by default.

With these three types of urgency defined, the eventual urgency score for PDR s_i is defined as their minimum value: $u_i = \min\{u^{(i)}_{vol}, u^{(i)}_{tm}, u^{(i)}_{pr}\}$.

6 Performance Evaluation

For performance evaluation we have built a prototype of HiP4-UPF, which includes 31.1K lines of Rust code for the North Controller, 6.1K lines of C++ code for the South Controller,

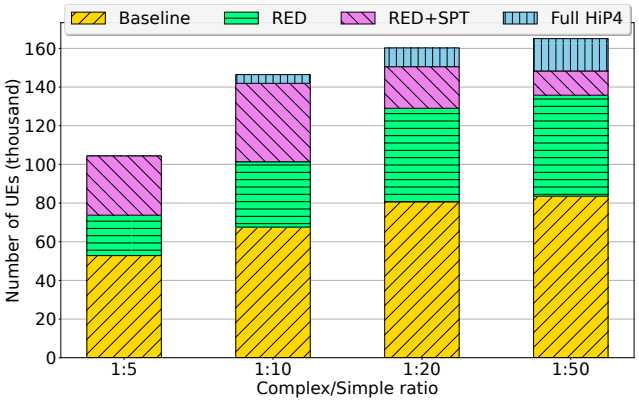

Figure 10: Maximal numbers of UEs supported under different optimization schemes

and 1.2K lines of P4 code for the data plane. Our implementation is based on a Netberg Aurora 710 P4 programmable switch with an Intel Tofino ASIC and an Intel Xeon D-1527 CPU, which runs Ubuntu 20.04 with kernel 5.15.83 on its CPU. Two x86 servers using Intel Xeon CPUs are connected to the P4 switch through 100Gb Ethernet cables. We use the P4 compiler and the switch ASIC driver from Intel's Bf SDE 9.11.2 software package. We use the P4Insight tool in the Bf SDE to obtain the resource usages of P4 programs.

For presentation clarity, we keep using the baseline UPF to refer to our UPF implementation based on prior work [27] (see Section 2.2). Our performance comparison experiments also consider other open source UPF implementations, including SD-Fabric [28], free5GC [2], and open5gs [7].

6.1 Number of UEs Supported

In this set of experiments we study how many UEs can be supported by HiP4-UPF. We model the same scenario in Section 3.1 where two exclusive UE sets are considered. As usually not many UEs need cross-UPF N9 interface, we keep a constant number of 2,048 UEs as exceptions. Hence, when SPT is applied, the ul_to_n3_n9_simple table is fixed to have 2,048 entries. If the size of the simple UE set is N_s, the sizes of both ul_to_n6_simple and ul_from_n6_simple are $N_s - 2048$. Moreover, we allocate half of the UEs in the complex UE set to the complex N9 table. Formally, if N_c denotes the size of the complex UE set, the sizes of the ul_to_n3_n9_complex, ul_to_n6_complex and ul_from_n6_complex tables are all $N_c/2$. We also use the same method in Section 3.1 to search for the maximal numbers of UEs supported under different optimization schemes.

Optimization effects. Figure 10 depicts the contribution of each optimization scheme to the improvement in the number of UEs supported under different complex/simple ratios. The results for the baseline case are the same as what are shown in Figure 5. Compared against the baseline UPF, the full optimization mode of HiP4-UPF (i.e., RED+SPT+CAD) increases the maximal number of UEs supported by 97.6%, 116.6%, 98.6%, and 95.0% at a complex/simple ratio of 1:5, 1:10, 1:20, and 1:50, respectively, which leads to an average improvement of 101.95% over all four cases. From our experiments we also observe that the TCAM usage decreases with a decreasing complex/simplex ratio. This agrees with our intuition as the PDR rules for the complex UE set are stored in the TCAM (see Table 2).

Within the three-tiered optimization framework, the average increase of the number of UEs supported due to RED, SPT, and CAD over the previous optimization level is 53%, 27%, and 5.2%, respectively. As we move up the optimization ladder, the SRAM usage becomes higher due to more UEs supported under the same complex/simple ratio. However, with the same complex/simple ratio, the TCAM usage under RED+SPT is lower than that under RED because the SPT optimization scheme makes more efficient use of the TCAM that stores the complex rules.

Comparison against SD-Fabric. SD-Fabric [28] is an open source programmable network fabric with various data plane features including 5G UPF. We compile SD-Fabric's data plane P4 program with its 5G UPF feature enabled while disabling its In-band Network Telemetry (INT) feature.

For fair comparison between SD-Fabric and HiP4-UPF, we make the following modifications. First, the vanilla SD-Fabric's UPF supports 100,000 UEs, among which only 4096 have usage counters associated. As SD-Fabric is designed to support private enterprise 5G deployment it does not require each UE to have a usage counter. We modify its code to have a counter assigned to each UE. Second, SD-Fabric and HiP4-UPF differ in their UPF rules. In the SD-Fabric's UPF module, packets match against UEs' IP addresses in the downlink direction and the UPF IP and TEID in the uplink direction. These two tables serve a similar purpose as HiP4-UPF's simple tables. In SD-Fabric, *all* UEs' packets go through application detection rules and are classified into one of at most five application types. SD-Fabric then use the session id and an APPLICATION_ID to decide the packet forwarding action and produce an usage counter identifier, which is further used to index a later usage accounting table. SD-Fabric does not support UPF-triggered usage reporting as HiP4-UPF can achieve. For fair comparison, we modify HiP4-UPF to apply the same application detection approach. We remove its complex and N9 tables and add an application detection table similar to the one used in SD-Fabric. A 4-bit `APPLICATION_ID` is produced by looking up this table and is later used as part of the matching data in the simple PDR tables. Due to these modifications on HiP4-UPF, we dub it as HiP4-UPF (AppDetect) for ease of presentation. Third, as 5G UPF is only part of SD-Fabric's comprehensive data plane solution, we remove some of its functionalities unnecessary for 5G UPF, such as VLAN, MPLS and ACL. We dub this simplified version as SD-Fabric (UPF only).

We use the same search method as described in Section 3.1

Table 3: Comparison between HiP4-UPF and SD-Fabric

Solution	#UEs	SRAM	TCAM
SD-Fabric	35.8k	34.2%	17.4%
SD-Fabric (UPF only)	55.3k	43.3%	4.2%
HiP4-UPF (AppDetect)	159.7k	85.4%	9.7%

to find the maximal number of supported UEs. The comparison results are shown in Table 3. By removing the unnecessary features, the simplified SD-Fabric has a lower TCAM usage than its vanilla version, but its SRAM usage has increased by 26.6% due to a 54.5% increase in the number of UEs supported. However, even with similar features implemented, HiP4-UPF (AppDetect) can support 1.9 times more UEs on the commodity P4 switch than the simplified version of SD-Fabric with about twice the SRAM and TCAM usages.

6.2 UPF Responsiveness

In this section we evaluate how quickly HiP4-UPF can respond to the PFCP requests from an SMF, which is adapted from its implementation in the VET5G testbed [38]. Each experiment considers 150,000 UEs and is repeated 200 times.

Comparison against CPU-based UPFs. For performance comparison we consider the CPU-based UPF implementations in two existing 5G network emulators, *free5GC* [2] and *open5gs* [7]. As the UPFs in both free5GC and open5gs run on commodity servers, their data plane performances are far worse than that of HiP4-UPF whose data plane functionalities are executed by the P4 switch's ASIC. We thus only compare different UPFs' responsiveness to the PFCP requests from the control plane. For fair comparison we let the SMF run on a separate server from the one that executes the free5GC or the open5gs UPF. The two servers are connected to the same regular switch through 10G Ethernet cables. The P4 switch is connected to the regular switch through a 10G Ethernet cable. Hence, the SMF has a similar route to the UPF in all cases.

We measure the throughput as follows. For each common type of operations, registration, deregistration, and handover, the SMF repeatedly sends 10 concurrent PFCP requests and waits for *all* of them to respond. After repeating this until one second has passed, we wait for the responses for the last batch of requests. The throughput is calculated as the total number of PFCP requests finished divided by the entire duration. Table 4 compares the throughput results for the three different UPF implementations. HiP4-UPF achieves the highest throughputs in all cases. More specifically, for the registration, deregistration and handover operations, HiP4-UPF improves the throughput by 197%, 619%, and 51% over open5gs and by 49%, 272%, and 9% over free5gc, respectively.

Comparison against P4 switch-based UPFs. We consider the baseline UPF described in Section 2.2, which uses two threads, and a single thread approach. The latter uses one thread to handle both read and write operations: by constantly monitoring the Unix socket for update requests, the thread performs rule updates if an update request is received and otherwise pulls counter data for only 100 PDRs.

Table 4: Throughputs of different UPF solutions measured in the number of operations per second. Both means and standard deviations are shown.

Operation	**open5gs**	**free5gc**	**HiP4-UPF**
Restration	362±19	723±40	1074±454
Deregistration	379±23	733±50	2725±1007
Handover	1796±195	2478±292	2713±455

We measure the handover response latency as follows. The SMF sends two concurrent handover requests every 10 milliseconds for 1 second (in total there are 200 requests) and then waits for them all to respond. The handover response latency is measured as the delay between the handover request and the corresponding response. The experimental results are depicted in Figure 12. Compared with the baseline UPF, the single thread approach improves the 99-percentile latency from 1.3 seconds to 361 millisecond but its medium latency increases from 4 to 9 milliseconds. By contrast, HiP4-UPF has a medium latency of 1 millisecond and a 99-percentile latency of 6.4 milliseconds. The much reduced latency results from its pendulum table access scheme which divides the original ingress or egress accounting table into two partitions to alternate read and write accesses, thus enabling the write thread to perform rule updates instantaneously.

6.3 Effects of Selective Usage Pulling

In this section we evaluate HiP4-UPF's performance improvement due to selective usage pulling. We use the same emulated SMF as discussed in Section 6.2. We use Cisco's TRex tool [8] to generate representative Internet traffic from UEs in the uplink direction. We choose the tool's SFR profile, which is created from captured real life traffic ranging from web browsing, emails, to voice and video calls. Each UE has a bitrate ranging from 100 Kbps to 600 Kbps.

Due to page limitation we only show the results about volume-based reporting latency, which is measured in the same way as in Section 3.2. Figure 11 shows the cumulative distribution function (CDF) of the reporting latency in one representative run with 150,000 active UEs. Due to selective usage pulling, the mean reporting latency is reduced by 84.6% from 3.12 to 0.48 seconds, more than 73.8% of all UEs have reporting latency less than 500 milliseconds , and 93.1% of them have reporting latency less than one second. Comparing these results with Figure 6 it is clear that selective usage pulling allows HiP4-UPF to significantly reduce the high reporting latency observed with the baseline UPF.

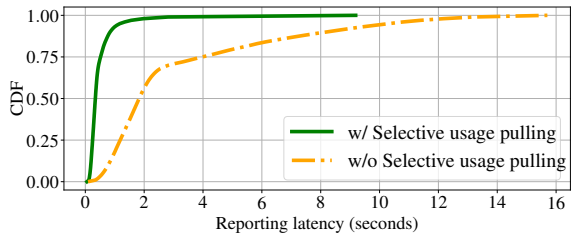

Figure 11: Volume-based reporting latency with 150,000 UEs

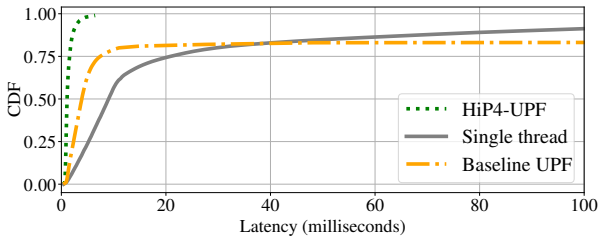

Figure 12: Handover response latency

7 Related Work

Optimization of packet classification. There have been a plethora of efforts on optimization of packet classification, which, according to work [20], can be classified into three groups based on the type of algorithms used: hash-based (e.g., Tuple Space Search [35]), decomposition-based (e.g, cross-producting [36], recursive flow classification [19], and Aggregated Bit Vector [11]), and decision tree-based (e.g., HiCuts [19], HyperCuts [31], HyperSplit [30], EffiCuts [37], Adaptive Binary Cutting [33], and ParaSplit [18]). There are also works aimed at balancing fast packet classification and fast rule updates, such as Tuple Space Search [35], PartitionSort [39] and TupleMerge [15]. These packet classification algorithms have been implemented in both hardware (e.g., TCAM-based network devices [17, 25, 34, 40]) and software (e.g., Open vSwitch [29]). While HiP4-UPF leverages the existing packet classification capabilities on a commodity P4 switch to implement 5G UPFs, its three-tiered rule storage optimization schemes leverage 5G-specific rule characteristics to improve efficiency of rule storage on the resource-constrained switch ASIC and thus increases its capacity.

Software-based optimization of 5G UPFs on commodity servers. UPF products by Ericsson [22], Metaswitch [5], and SK Telecom [6] all rely on DPDK for fast packet processing, as it allows packets received at the NICs to be directly delivered to the user-space memory. The SK Telecom's solution [6] further leverages Intel network adapters' DDP capabilities to reduce packet latency [6], while the Metaswitch's solution relies upon an optimized packet pipepine with match-action classifiers allowing parallel lookups over the packet header in a single operation [5]. Amaral *et al.* propose to leverage the XDP feature of new Linux kernels for fast packet processing in 5G UPFs [16]. Although software-based UPFs allow easy scaleout and inclusion of comprehensive features, they are not as cost effective as P4 switch-based UPFs [13].

5G UPFs based on programmable hardware. Bose *et al.* propose to offload packet processing to programmable dataplane hardware to improve UPF performance per unit cost or power [13]. In their work, PFCP messages from the control plane are processed by a thread running on the CPU, while packet forwarding rules are implemented on the smartNICs. However, their UPF solution can support the forwarding states of only 10K users. Their recent work [12] suggests that it is possible to further improve 5G UPF performance by offloading the majority of PFCP messages containing simple packet processing rules (e.g., those for IoT devices) onto programmable NICs or switches.

MacDavid *et al.* have provided the design of a 5G UPF, which can run on a real Intel Tofino-based programmable switch [27]. The development of HiP4-UPF has been heavily influenced by this work, which serves as its starting point for further performance optimization. Singh *et al.* propose a hybrid approach to combine a fast datapath based on the switch with a slow one, which is implemented on commodity services with plenty of resources for packet processing [32]. X-Plane [26] is a recent 5G UPF implementation, which extends TEA's original idea of storing lookup tables on inexpensive DRAMs accessible from P4 programmable switches [23]. X-Plane has proposed new methods to overcome the challenges of concurrent access to stateful data, slow table lookup, and out-of-order packets due to such a design. Our work is orthogonal to Singh's method and X-Plane as its goal is to optimize 5G UPF with comprehensive features, which, except packet buffering, can be deployed entirely on a commodity P4 programmable switch. Other UPFs based on P4 programmable switches have also been discussed in the literature [24, 28, 41]. Our experimental results in Section 6.1 have shown that HiP4-UPF can support 2.9 times as many UEs as the UPF implementation by SD-Fabric [28]. As few details are provided in [24, 41] and their implementation code is not publicly available, comparisons with these two UPF solutions are not conducted in this study.

8 Conclusions

Due to the increasing use cases of 5G networks, there is a urgent need to develop high-performance UPFs that can support a large number of UEs with low operational latency. In this work we present the design and implementation of HiP4-UPF, whose novel contributions include a three-tired optimization framework for rule storage and a hybrid approach combining pendulum table access and selective usage pulling to reduce operational latency. Our evaluation demonstrates that HiP4-UPF offers a high-performance solution to 5G UPF, whose comprehensive features, except packet buffering, can be deployed entirely on commodity P4 programmable switches.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback. This work is partially supported by the US National Science Foundation under award CNS-1943079 and Intel's Fast Forward Initiative 2022 Award.

References

[1] FlatBuffers. https://flatbuffers.dev/.

[2] free5GC. https://free5gc.org/.

[3] Intel® P4 Insight. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-insight.html.

[4] Intel® tofino™ native architecture - public version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

[5] Lighting Up the 5G Core with a High-Speed User Plane on Intel Architecture. https://builders.intel.com/docs/networkbuilders/lighting-up-the-5g-core-with-a-high-speed-user-plane-on-intel-architecture.pdf.

[6] Low Latency 5G UPF Using Priority Based 5G Packet Classification. https://builders.intel.com/docs/networkbuilders/low-latency-5g-upf-using-priority-based-5g-packet-classification.pdf.

[7] Open5GS. https://open5gs.org/.

[8] Trex low-cost, high-speed stateful traffic generator. https://github.com/cisco-system-traffic-generator/trex-core.

[9] URLLC Latency Requirements in User and Control Planes. https://resources.pcb.cadence.com/blog/2022-urllc-latency-requirements-in-user-and-control-planes.

[10] 3GPP. 5G; 5G System; User Plane Function Services; Stage 3 (3GPP TS 29.564 version 17.1.0 Release 17). Technical specification (ts), 3rd Generation Partnership Project (3GPP), July 2022.

[11] F. Baboescu and G. Varghese. Scalable packet classification. *ACM SIGCOMM Computer Communication Review*, 31(4):199–210, 2001.

[12] A. Bose, S. Kirtikar, S. Chirumamilla, R. Shah, and M. Vutukuru. AccelUPF: accelerating the 5G user plane using programmable hardware. In *Proceedings of the Symposium on SDN Research*, pages 1–15, 2022.

[13] A. Bose, D. Maji, P. Agarwal, N. Unhale, R. Shah, and M. Vutukuru. Leveraging programmable dataplanes for a high performance 5G user plane function. In *Proceedings of the 5th Asia-Pacific Workshop on Networking*, pages 57–64, 2021.

[14] Bytestart. How 5G will benefit businesses. https://www.bytestart.co.uk/how-5g-will-benefit-businesses.

[15] J. Daly, V. Bruschi, L. Linguaglossa, S. Pontarelli, D. Rossi, J. Tollet, E. Torng, and A. Yourtchenko. TupleMerge: Fast software packet processing for online packet classification. *IEEE/ACM transactions on networking*, 27(4):1417–1431, 2019.

[16] T. A. N. do Amaral, R. V. Rosa, D. F. C. Moura, and C. E. Rothenberg. An in-kernel solution based on XDP for 5G UPF: Design, prototype and performance evaluation. In *Proceedings of the 17th International Conference on Network and Service Management*, pages 146–152. IEEE, 2021.

[17] M. Faezipour and M. Nourani. Wire-speed tcam-based architectures for multimatch packet classification. *IEEE Transactions on Computers*, 58(1):5–17, 2008.

[18] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang. ParaSplit: A scalable architecture on FPGA for terabit packet classification. In *Proceedings of the 20th IEEE Annual Symposium on High-Performance Interconnects*, pages 1–8. IEEE, 2012.

[19] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 147–160, 1999.

[20] P. He, G. Xie, K. Salamatian, and L. Mathy. Meta-algorithms for software-based packet classification. In *Proceedings of the 22nd IEEE International Conference on Network Protocols*, pages 308–319. IEEE, 2014.

[21] J. Heinanen and R. Guerin. A two rate three color marker. RFC 2698, RFC Editor, September 1999.

[22] L. Johansson, P. Holmberg, and R. Skog. Energy-efficient packet processing in 5G mobile systems. *Ericsson Technology Review*, 2022(7):2–11, 2022.

[23] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *Proceedings of the ACM Conference on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 90–106, 2020.

[24] R. Kundel, T. Meuser, T. Koppe, R. Hark, and R. Steinmetz. User plane hardware acceleration in access networks: Experiences in offloading network functions in real 5G deployments. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 1–10, 2022.

[25] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary cams. *ACM SIGCOMM Computer Communication Review*, 35(4):193–204, 2005.

[26] Y. Liu, H. Nie, H. Cai, B. Jiang, P. Zhang, Y. Liu, Y. Yao, X. Wei, B. Lyu, C. Xu, et al. X-Plane: A high-throughput large-capacity 5G UPF. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2023.

[27] R. MacDavid, C. Cascone, P. Lin, B. Padmanabhan, A. Thakur, L. Peterson, J. Rexford, and O. Sunay. A P4-based 5G user plane function. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 162–168, 2021.

[28] ONF. SD-FABRIC. https://opennetworking.org/sd-fabric/.

[29] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vSwitch. In *Proceedings of the 12th USENIX symposium on Networked Systems Design and Implementation*, pages 117–130, 2015.

[30] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *Proceedings of the IEEE International Conference on Computer Communications*, pages 648–656. IEEE, 2009.

[31] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 213–224, 2003.

[32] S. K. Singh, C. E. Rothenberg, J. Langlet, A. Kassler, P. Vörös, S. Laki, and G. Pongrácz. Hybrid P4 programmable pipelines for 5G gNodeB and user plane functions. *IEEE Transactions on Mobile Computing*, 2022.

[33] H. Song and J. Turner. ABC: Adaptive binary cuttings for multidimensional packet classification. *IEEE/ACM Transactions On Networking*, 21(1):98–109, 2012.

[34] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended TCAMs. In *Proceedings of the 11th IEEE International Conference on Network Protocols*, pages 120–131. IEEE, 2003.

[35] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 135–146, 1999.

[36] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 191–202, 1998.

[37] B. Vamanan, G. Voskuilen, and T. Vijaykumar. Efficuts: Optimizing packet classification for memory and throughput. *ACM SIGCOMM Computer Communication Review*, 40(4):207–218, 2010.

[38] Z. Wen, H. S. Pacherkar, and G. Yan. Vet5g: A virtual end-to-end testbed for 5G network security experimentation. In *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, pages 19–29, 2022.

[39] S. Yingchareonthawornchai, J. Daly, A. X. Liu, and E. Torng. A sorted-partitioning approach to fast and scalable dynamic packet classification. *IEEE/ACM Transactions on Networking*, 26(4):1907–1920, 2018.

[40] F. Yu and R. H. Katz. Efficient multi-match packet classification with TCAM. In *Proceedings of the 12th Annual IEEE Symposium on High Performance Interconnects*, pages 28–34. IEEE, 2004.

[41] C. Zhou, B. Zhao, and B. Wang. A 100Gbps user plane function prototype based on programmable switch for 5G network. In *Proceedings of the 6th Asia-Pacific Workshop on Networking*, pages 83–84, 2022.

Appendix A: Primer on P4 Programmable Switches

In this section we briefly describe the working of Intel Tofino programmable switch. Figure 13 shows the overview of Intel Tofino architecture.

Packets from entering the switch are first parsed by a programmable parser to extract header fields and other metadata. The programmable parser is a state machine where in each state a certain amount of packet is extracted into Packet Header Vectors (PHVs). P4 allows for transitioning to different parser states based on extracted value to achieve handling of different types of packets. To ensure the state machine completes no loop is allowed and all paths from the start state must go to one of two end states: *accept* and *reject*.

Packets after being parsed goes into a set of match-action units (MAU) sequentially. With in each stage header fields or metadata extract during parsing are put into different tables in parallel and matched against (e.g., exact, range, and ternary

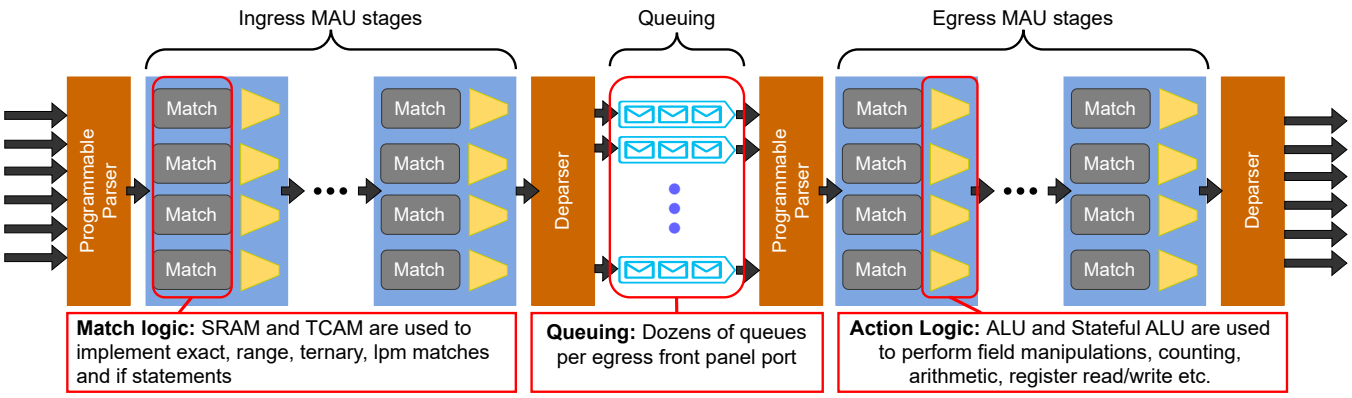

Figure 13: Overview of Intel Tofino

matching) value configured by the switch controller. Each match entry in a table has an action associated with it and will be executed if the match was a hit. Action can be a simple assignment to a header filed or metadata, or complex operations like CRC calculation, arithmetic operations, register read/write via stateful ALU, metering/counting etc.

The key limitation of this architecture is that in order to achieve high packet throughput, action within each stage is limited to one clock cycle operation (e.g., bitwise, add, subtract, logical shift, min/max etc.). More complex operations have to be split into multiple MAU stages which Tofino has a limited number of.

The P4 construct Registers is what make programmable switches so powerful as they can maintain states between packets. To achieve high throughput, registers also have their limitations: 1) they are MAU stage local, meaning they can only be read from and written to within that MAU stage. 2) actions based on registers are also limited to two clock cycles, meaning it can be just a bit more complex then regular actions, supporting additional operations like if statement and math operations.

For more details of Intel Tofino we refer the readers to [4].

Appendix B: Details of baseline UPF

In the baseline UPF, the controller and PPP modules communicate with each other via two interfaces. Through the *P4 Runtime interface*, rule table updates, which are translated from PFCP messages, are sent from the controller to the PPP module while the counters are read in the opposite direction. A *CPU port*, which is manifested as a Linux network interface, is used by the controller to inject end markers into the packet processing pipeline and by the PPP module to send Notify Control Plane (NoCP) packets to the controller[1].

[1]End marker in 5G is a special GTP packet sent toward the access network to facilitate packet reordering by the gNB during handover and Notify Control Plane (NoCP) packets are copies of the triggering packets sent to the controller in order to trigger certain events of interest, such as notification of downlink packet arrival.

(1) Packet processing. Using the rules defined in Section 2.2, the PPP module of the baseline UPF processes packets as illustrated in the upper right box of Figure 1.

The **PDR matching** component matches each incoming packet with the rules in the three PDR rule tables to obtain the PDR_ID, which will be later used to find the corresponding rules in downstream tasks. As the complex PDR table has higher priority over the simple ones, only a match miss from the complex PDR table enables the simple tables to be used for PDR matching. If there is no match from all three tables, the packet is simply discarded.

The PDR_ID given by a matching PDR is used by the **Forwarding** component to index the FAR table for the forwarding action, which can be *NoCP* (notify control plane), *Buffer*, *Drop*, *Decap* (decapsulation), or *Encap* (encapsulation). The *NoCP* action copies the triggering packet and sends it to the controller. The *Buffer* action forwards the packet to the buffering service via a dedicated front panel port. The *Drop* action simply drops the packet. The *Decap* action removes the tunnel header from an uplink packet and forwards it to the IP routing component. The *Encap* action adds a new GTP-U tunnel header to the packet, which requires the tunnel data. Hence its action data includes a 32-bit destination IP address, a 32-bit TEID and a 6-bit QFI. Given the IP address, TEID and QFI, *Encap* action is then used to create the new GTP-U tunnel header and sends the encapsulated packet to the IP routing component. For each packet received, the **IP routing** component is responsible for delivering it to the right egress port based on two *routing tables*, each with 2048 exact match entries and 512 Longest Prefix Match (LPM) entries. We use separate routing tables for routing encapsulated packets based on the tunnel headers and decapsulated packets using the original IP packet headers to achieve paralellization in packet routing to different interfaces.

Given the PDR_ID, the **Bitrate Enforcement** component

obtains from the QER table the identifier of the Two Rate Three Color Marker (TrTCM) meter used to mark matching packets. The meter identifier is zero if no meter is needed. Based on the packet colors marked by the TrTCM meters and traffic classes identified by QFIs, the **Queue Assignment** component assign packets to different queues identified by their 5-bit QueueIDs. The pre-QoS and post-QoS usage data are collected periodically by the **Pre-QoS Accounting** and **Post-QoS Account** components from the counters stored in the Ingress and Egress Accounting tables, respectively.

(2) Usage reporting. Usage reporting in a 5G network can be triggered periodically, or when a certain volume or time threshold has been reached. Given these different ways to trigger usage reporting, it is not a viable solution to push the measurement data collected by the P4 switch's ASIC to the control plane. Following the same approach in [27], we consider only *pulling* count statistics from the switch ASIC by the controller, from which usage reports are generated and then sent to the control plane using PFCP messages. Our implementation uses the P4 switch's built-in counters to track the number of bytes and the number of packets for each PDU session, both of which are *pulled* periodically from the switch.

Appendix C: Bootstrapping for initial PDR_ID range assignments

The decomposition step is performed by Algorithm 1. Given the list of (t_i, l_i)-tuples from the input, it first puts those with a looseness score of 0 to list B and others into a max-priority queue with their looseness scores as their keys (Lines 2-7). It also uses g to keep the total number of IDs that is needed if each (t_i, l_i)-tuple is assigned with a smallest PDR_ID range (Line 3). The loop (Lines 8-18) deals with the situation where there are not enough bits to assign g IDs. In each iteration, the (t, l)-tuple with the highest looseness score is extracted from the max-priority queue Q (Line 9) and then gets decomposed. The decomposition has three possible results: (I) the tuple can be split into two, one with a looseness score of 0 and the other with the same looseness score (Lines 10-12); (II) the tuple can be transformed into one with a smaller looseness score (Lines 13-15); and (III) the tuple can be transformed into one with a looseness score of 0. It is noted that Cases II and III are able to reduce g, the total number of IDs needed, while keeping the same number of (t, l)-tuples (in either B or Q). By contrast, Case I does not reduce g but increases the number of (t, l)-tuples by one after decomposition. Even so, the decomposition is instrumental in producing the latter two cases in future iterations of the loop. If decomposition can not overcome the second constraint, an error is raised (Line 20). Otherwise, all the (t, l)-tuples collected from both B and Q are returned (Lines 22-24).

The tree-building step, whose details are presented in Algorithm 2, is used to allocate a PDR_ID range for each (t, l)-tuple returned from the decomposition

Algorithm 1: Decomposition step in bootstrapping

Input: $T = (t_1, ..., t_n)$, $L = (l_1, ..., l_n)$, Θ
Output: List of (t, l)-tuples

1 $g \leftarrow 0$, $Q \leftarrow$ an empty max-priority queue, $B \leftarrow$ an empty list;
2 **for** $i \in 1..n$ **do**
3 $g \leftarrow g + 2^{\lceil log_2(l_i) \rceil}$;
4 **if** $2^{\lceil log_2(l_i) \rceil} = l_i$ **then**
5 Append (t_i, l_i) to B;
6 **else**
7 Insert (t_i, l_i) into Q with key $2^{\lceil log_2(l_i) \rceil} - l_i$;
8 **while** $(g > 2^{\Theta}) \wedge (Q$ is not empty$)$ **do**
9 Extract (t, l) from Q with the largest key;
10 **if** $l > 2^{\lceil log_2(l) \rceil - 1}$ **then**
11 Append $(t, 2^{\lceil log_2(l)) \rceil - 1})$ to B;
12 Insert $(t, l - 2^{\lceil log_2(l) \rceil - 1})$ into Q with key $2^{\lceil log_2(l)) \rceil} - l$;
13 **else if** $l < 2^{\lceil log_2(l) \rceil - 1}$ **then**
14 Insert (t, l) into Q with key $2^{\lceil log_2(l)) \rceil - 1} - l$;
15 $g \leftarrow g - 2^{\lceil log_2(l)) \rceil - 1}$
16 **else**
17 Append $(t, 2^{\lceil log_2(l)) \rceil - 1})$ to B;
18 $g \leftarrow g - l$;
19 **if** $g > 2^{\Theta}$ **then**
20 Raise an **error** of insufficient number of bits;
21 **else**
22 **while** Q is not empty **do**
23 Extract (t, l) from Q and append it to B;
24 **return** B

step. Towards this goal, it aims to build a strict binary tree in which each node is represented by a 5-tuple: $(template, capacity, allocated, left, right)$. For a leaf node v, $v.template$ stores the template for which a PDR_ID range should be assigned, while for an internal one its template field is set to be Null. For any node v, $v.capacity$ keeps the maximum number of IDs that can be possibly used by the nodes within the subtree rooted at v, while $v.allocated$ gives the total number of IDs that have already been allocated to the leaf nodes within this subtree. The algorithm uses a min-priority queue Q to maintain all the nodes created, where their capacities are used as the keys for insertions and extractions.

The tree-building step starts by creating leaf nodes for the (t, l)-tuples returned from the decomposition step (Line 3). For each leaf node, it is assigned with a smallest PDR_ID range to accommodate the l IDs requested. Hence, its *capacity* is set to the number of IDs allowed by this range, while its *allocated* field is set to be l. These leaf nodes are inserted into

Algorithm 2: Tree-building step in bootstrapping

Input: B: list of (t,l)-tuples, Θ
Output: A strict binary tree in which each node is represented by a 5-tuple: ($template$, $capacity$, $allocated$, $left$, $right$)

1 $Q \leftarrow$ an empty min-priority queue;
2 **for** $(t,l) \in B$ **do**
3 Create node v with 5-tuple $(t, 2^{\lceil log_2(l) \rceil}, l,$ Null, Null);
4 Insert v into Q with key $v.capacity$;
5 **while** Q has at least two items **do**
6 Extract u and v from Q with the smallest keys (i.e., $u.capacity \leq v.capacity$);
7 **while** $u.capacity < v.capacity$ **do**
8 Create node s with 5-tuple $(\text{Null}, u.capacity, 0,$ Null, Null);
9 Create node p with 5-tuple $(\text{Null}, 2 \cdot u.capacity, u.allocated, u, s)$;
10 $u \leftarrow p$;
11 Create node p with $(\text{Null}, 2 \cdot u.capacity, u.allocated + v.allocated, u, v)$;
12 Insert p into Q with key $p.capacity$;
13 Extract $root$ from Q with the smallest key;
14 **if** $root.capacity > 2^{\Theta}$ **then**
15 Raise an error of insufficient number of bits;
16 **while** $root.capacity < 2^{\Theta}$ **do**
17 Create node s with $(\text{Null}, s.capacity = root.capacity, 0,$ Null, Null));
18 Create node p with $(\text{Null}, 2 \cdot root.capacity, root.allocated, root, s)$;
19 $root \leftarrow p$;
20 DFS_assign($root$, 0);
21 **return** root;
22 **Function** `DFS_assign`(u, $code$)**:**
23 **if** $u =$ Null **then**
24 **return**;
25 **if** $u.template \neq$ Null **then**
26 Assign PDR_ID range $(code, log_2(u.capacity))$ to $u.template$;
27 `DFS_assign` ($u.left$, $2 \cdot code$);
28 `DFS_assign` ($u.right$, $2 \cdot code + 1$);

the min-priority queue Q based on their capacities (Line 4).

Next the algorithm keeps merging nodes u and v, which have the smallest capacities in Q (Lines 5-6). As these two nodes may have different capacities, we iteratively pair the smaller one (i.e., u) with a newly created sibling leaf node s, which has a Null template and the same capacity, attach them to a newly created parent node p, which also has a Null template but does the capacity, and then replace u with p (Lines 7-10). This process repeats until node u reaches the same capacity as node v. At that point, a new parent node p is created with u and v being its two children nodes (Line 11) and next inserted into the min-priority queue Q based on its capacity.

When Q contains only a single node, it is treated as a root node. If the root node's capacity has not reached the maximum value allowed by the number of bits provisioned (i.e., Θ), a new sibling leaf node is created to merge with the root node to obtain a new parent root node. The process repeats until the root node finally reaches the maximum capacity allowed (Lines 16-19).

It is easy to see that the tree as constructed is a strict binary tree. We can apply a depth-first traversal to visit all the nodes in the tree. Each node visited can be coded by the branches taken during the traversal: 0 for taking a left branch and 1 for taking a right branch. For each leaf node discovered whose template is not Null, a PDR_ID range (r,m) is assigned to the template where r is the node's code and m is the binary logarithm of the node's capacity (Lines 20, 22-28).

Appendix D: Allocate and deallocate procedures in runtime update

Each leaf node in the tree additionally keeps track of the list of used PDR_IDs, denoted by L_{used}.

Allocate: Given action template t, the following steps are performed. ① It first tries to find any existing node with the same template t. If the length of its L_{used} list is smaller than its capacity, one free ID from the node's PDR_ID range is assigned to it and then added to L_{used}. ② If none is found from the previous step, the procedure next tries to find an existing empty leaf node (i.e., its L_{used} list is empty) in the binary tree. If such an node exists, it is allocated to action template t and its first free ID is allocated and thus added to its L_{used} list. ③ Otherwise, the procedure finds an existing non-empty leaf node with spare IDs and then recursively splits it into two children nodes of equal capacity until one has an empty L_{used} list. After that, the empty child node is assigned to action template t, as done in Case ②.

Deallocate: When a PDR_ID is deallocated, it is removed from the L_{used} list of the leaf node whose PDR_ID range it falls into. If L_{used} becomes empty *and* the sibling node is also an empty leaf node, these two nodes are simply removed and their parent node is then transformed into an empty leaf node. This process repeats until the sibling node is not empty. The purpose of this step is to reduce fragmentation of PDR_ID ranges in the binary tree.