AccelUPF: Accelerating the 5G user plane using programmable hardware

Abhik Bose*, Shailendra Kirtikar*, Shivaji Chirumamilla*, Rinku Shah+, Mythili Vutukuru*
Indian Institute of Technology Bombay*, Indraprastha Institute of Information Technology Delhi+
India
{abhik,shailendra,shivaji}@cse.iitb.ac.in,rinku@iiitd.ac.in,mythili@cse.iitb.ac.in

ABSTRACT

The latest generation of 5G telecommunication networks are expected to provide high throughput and low latency while catering to diverse applications like mobile broadband, dense IoT, and self-driving cars. A high performance User Plane Function (UPF), the main element in the 5G user plane, is critical to achieving these performance goals. This paper presents AccelUPF, a 5G UPF that offloads functionality to programmable dataplane hardware for performance acceleration. While prior work has proposed accelerating the UPF by offloading its data forwarding functionality to programmable hardware, the Packet Forwarding Control Protocol (PFCP) messages from the control plane that configure the hardware data forwarding rules were still processed in software. We show that only offloading data forwarding and not PFCP message processing leads to suboptimal performance in the UPF for applications like IoT that have a much higher ratio of PFCP messages to data traffic, due to a bottleneck at the software control plane that configures the hardware packet forwarding rules. In contrast to prior work, AccelUPF offloads both PFCP message processing as well as data forwarding to programmable hardware. AccelUPF overcomes several technical challenges pertaining to the processing of the complex variable-sized PFCP messages within the memory and compute constraints of programmable hardware platforms. Our evaluation of AccelUPF implemented over a Netronome programmable NIC and an Intel Tofino programmable switch demonstrates performance gains over the state-of-the-art UPFs for real-world traffic scenarios.

SOSR '22, October 19–20, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9892-3/22/10…$15.00
https://doi.org/10.1145/3563647.3563651

CCS CONCEPTS

• **Networks** → **In-network processing**; **Programmable networks**; *Network performance analysis*; **Mobile networks**.

KEYWORDS

5G core, 5G user plane, programmable networks, in-network computation

ACM Reference Format:
Abhik Bose*, Shailendra Kirtikar*, Shivaji Chirumamilla*, Rinku Shah+, Mythili Vutukuru*. 2022. AccelUPF: Accelerating the 5G user plane using programmable hardware. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '22), October 19–20, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3563647.3563651

1 INTRODUCTION

The mobile packet core connects the wireless radio access network (with base stations and mobile users) to external networks. The packet core consists of several control plane components that process signaling messages from mobile users (e.g., for authentication, setting up sessions to transfer data, handling mobility-related events) and the User Plane Function (UPF) on the data plane that forwards user traffic to and from external networks. The two planes communicate using PFCP (Packet Forwarding Control Protocol) messages that are sent by the control plane to establish, modify, and delete packet forwarding rules in the user plane, as shown in Figure 1. The most recent fifth generation (5G) telecommunication networks aim to support use cases with high throughput (~1 Gbps/user), very low processing latencies (<1 ms), stringent quality of service (QoS), and diverse traffic characteristics, e.g., enhanced mobile broadband, dense deployments of IoT devices, self-driving cars, AR/VR, high-speed entertainment in a moving vehicle, and delay-sensitive video applications [27, 42]. A high performance and low cost UPF is necessary for meeting these requirements.

Most state-of-the-art UPFs today are built as multicore-scalable software packet processing appliances running over

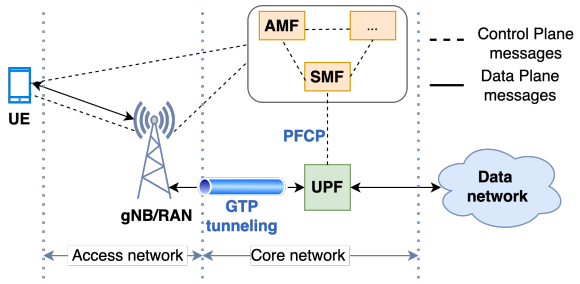

Figure 1: 5G Architecture.

commodity servers, and process traffic using a high performance packet I/O mechanism like the Data Plane Development Kit (DPDK) [11]. However, given the stringent performance requirements of 5G networks, and ever increasing network speeds running into a few hundreds of Gbps, prior work has proposed accelerating the UPF by offloading data forwarding functionality to programmable hardware [5–8, 18, 19, 22, 24, 31, 33], leveraging the availability of high-level programming languages like P4 [21] to program such hardware. Prior work has quantified the performance, cost and power savings of such offload [20]. Our evaluation comparing a production-grade DPDK-based software UPF with a state-of-the-art programmable hardware accelerated UPF (Table 1 in §5) also shows that hardware acceleration improves performance per unit cost by ~31%, and performance per unit power consumed by ~92%.

It is important to note that the hardware-accelerated UPFs in prior work offload only the user data forwarding to hardware. PFCP messages are still processed in software, and standard APIs exposed by hardware vendors are then used to configure packet forwarding rules in hardware. These APIs have a limited capacity of installing packet forwarding rules, which in turn limits the PFCP message processing capacity of today's offload-based UPFs that only offload data forwarding. Our measurements (Table 1 in §5) show that such an offload-based UPF can comfortably forward traffic at the 40Gbps linerate in our setup, but can process only a few hundred PFCP messages/sec.

But is the low rate of PFCP message processing in today's offload-based UPFs a bottleneck for real-life traffic? Some 5G use cases like IoT [34] or high mobility vehicular communication [32] are expected to frequently setup/reconfigure the data plane via signaling messages, roughly in the order of tens of seconds. For a UPF handling a few hundred thousand users [26, 39], this can translate to a few tens of thousands of PFCP messages to be processed every second, which is too high for today's offload-based UPFs to handle. Note that in our experiments, the DPDK-based software UPF was able to process a few thousand PFCP messages/sec on a single core, but we have already shown that software message processing is inefficient with respect to cost and power consumption. Therefore, for 5G use cases that generate significant amounts of signaling messages and require frequent reconfigurations of the user plane, neither the DPDK-based UPF nor today's offload-based UPF can deliver high performance, low power consumption, and low cost, all together.

So, why not simply offload the processing of PFCP messages to programmable hardware as well? PFCP message processing in programmable hardware is challenging for several reasons, as observed by prior work [20, 33]. PFCP message headers are variable in size, with multiple levels of nesting and several optional fields. Parsing such complex headers within programmable hardware is difficult because hardware is designed to run at linerate and is therefore constrained with respect to the instruction set and memory resources available. Further, the match-action tables of the programmable hardware that are used to store packet forwarding rules in today's hardware accelerated UPFs are configurable only from the software control plane via standard APIs, and cannot be directly modified from the switch data plane. While some switch memory (in the form of stateful register arrays) can be modified directly from within the data plane, such memory can only be accessed via a restricted interface of index-based access and is not as versatile as the match-action tables. Further, this limited switch memory may not accommodate the packet forwarding rules of all users, and is also not persistent across switch failures. Therefore, offloading the processing of PFCP messages to programmable hardware is non-trivial, and has not been attempted before in prior work to the best of our knowledge.

This paper proposes the design, implementation, and evaluation of AccelUPF, a programmable hardware accelerated 5G UPF that offloads most UPF functionality, including the processing of most types of PFCP messages and user data forwarding, to programmable hardware. Our key insight is to offload the processing of the more common and simpler patterns of PFCP messages to the fastpath on hardware, while handling the more complex and infrequent PFCP messages in software. Our design incorporates several novel ideas to handle common PFCP messages in the hardware fastpath. First, the hardware PFCP parser in AccelUPF identifies the mandatory and optional fields in the variable-sized, nested PFCP headers, and chooses the parser states dynamically based on the fields present in the received header (§3.3). Second, AccelUPF stores packet forwarding rules in stateful register arrays within the switch hardware. The register arrays allow only index-based access, so we use hash of the header fields of received packets as an index to access the array, handling hash collisions and switch memory overflows on the slow-path in software. Further, because PFCP messages and data traffic contain different header fields, we maintain packet forwarding rules across multiple register arrays indexed in different ways, in order to access them correctly across PFCP

and data traffic (§3.4). Third, we deploy a regular software UPF in the slowpath for traffic that cannot be handled in the fastpath within the programmable hardware, and we ensure that the UPF state is shared correctly across the software and hardware processing (§3.5). Finally, we use in-network replication of the register array state to protect against switch failures (§3.6).

We implemented AccelUPF on two different P4 programmable data plane hardware: an Agilio CX Netronome smart NIC [46] and an Intel Tofino programmable switch [25], using an existing production-grade standards-compliant software UPF [9] on the slowpath. Experiments with our AccelUPF prototypes show that AccelUPF achieves significantly higher PFCP message processing and data forwarding throughput, especially when normalized by cost or power consumed, as compared to both a pure software UPF as well as a hardware accelerated UPF where only user data forwarding has been offloaded. For use cases with significantly higher fraction of PFCP messages like IoT, AccelUPF provides up to 56% higher throughput than the best performing UPFs in prior work.

Our work makes the following contributions. (i) We show that prior work on programmable hardware accelerated 5G UPFs, where only data forwarding is offloaded to hardware, does not perform well for use cases which generate a high rate of signaling messages, because of the limited capacity of the software control plane APIs that install packet forwarding rules in the hardware. (ii) We design AccelUPF, a programmable hardware accelerated UPF that offloads not just the data forwarding but also most PFCP message processing to programmable hardware, and experiments with our implementation show significant performance gains over existing state-of-the-art UPFs for real-world traffic. (iii) Our design illustrates how one can offload complex control plane message processing to programmable hardware and our techniques are broadly applicable to other applications as well. (iv) By identifying the challenges in offloading PFCP processing to programmable hardware, our work can better inform future standardization efforts in 6G and beyond.

The rest of the paper is organized as follows. We begin with the background required to understand our work (§2), and then proceed to describe the design (§3), implementation (§4), and evaluation (§5) of AccelUPF. We then present related work (§6) and conclusions (§7).

2 BACKGROUND

This section provides the relevant background on 5G network architecture and programmable data plane hardware that is required to follow the rest of the paper.

5G architecture and procedures. Figure 1 shows the 5G architecture [4]. The 5G mobile packet core connects the

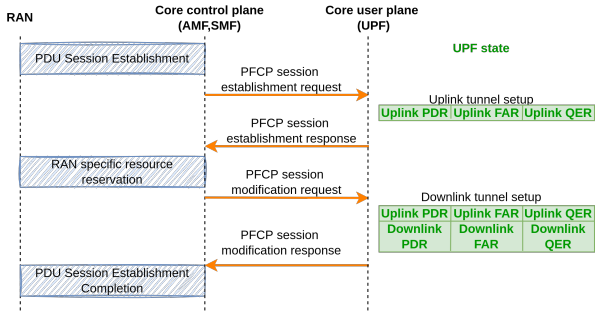

Figure 2: PDU session establishment callflow.

wireless radio access network (RAN)—which includes the User Equipment (UE) and the base station (gNB)—to other external networks. The packet core executes several *signaling procedures* on behalf of the UE. When a UE connects to a mobile network for the first time, it triggers an *initial registration procedure* via the base station at the Access and Mobility Function (AMF), which communicates with other components in the control plane of the packet core to authenticate and register the user. A registered UE that wishes to send data through the packet core must set up one or more *PDU sessions*, each with possibly different QoS requirements, using the *PDU session establishment procedure*. Such session-related procedures are coordinated by the Session Management Function (SMF) in the packet core. Once the sessions are setup, the actual traffic in the user plane is forwarded via the base station through one or more User Plane Functions (UPFs) through the packet core. The user plane traffic is encapsulated in GPRS Tunnelling Protocol (GTP) headers inside the packet core, and this tunnelling helps manage mobility of the UE in the network. A PDU session has two tunnels (identified by tunnel identifiers or TEID in the GTP header), one to carry uplink traffic from the base station and one to carry downlink traffic from external networks to the UE. On the uplink, the base station encapsulates the incoming UE IP traffic into GTP headers and the last UPF in the packet core performs the decapsulation. For downlink traffic, the UPF performs the encapsulation and the base station does the decapsulation. During its life time, a UE will trigger many other signaling procedures in the packet core, e.g., AN (access network) release procedure to move to an idle state after inactivity, service request procedure to reactivate itself when it wishes to communicate again, handover procedure to move to a different location, and so on.

PFCP messages. The SMF and the UPF communicate using Packet Forwarding Control Protocol (PFCP) messages that are exchanged over a UDP connection between both nodes [3]. A few important PFCP messages sent by the SMF to the UPF include the PFCP session establishment request, PFCP session modification request, and the PFCP session

deletion request, to establish, modify, and delete sessions at the UPF respectively. After processing these messages, the UPF sends back the corresponding response messages over PFCP as well, indicating the status (success or failure) of the request. A single signaling procedure of the UE such as a PDU session establishment can trigger multiple PFCP request/response exchanges between the SMF and the UPF. For example, we show a simplified UE initial PDU session establishment callflow in Figure 2. During this procedure, the SMF first sends a PFCP session establishment to the UPF to setup the uplink GTP tunnel, and later, after further communication with the base station, sends a PFCP session modification message to setup the downlink GTP tunnel. Several other signaling procedures like the AN release, service request, and handover will also involve one or more PFCP request/response messages exchanged between the SMF and UPF, e.g., to mark a session as idle/active or to switch the tunnel to another base station. The rate of PFCP messages received at a UPF will depend on the applications running on the UEs being served by the UPF. Several new use cases of 5G like dense IoT or high speed mobility are expected to generate high rate of PFCP messages. For example, a UE running an IoT application will frequently establish sessions, go idle and become active again, while transferring small amounts of data in between, leading to a relatively higher proportion of PFCP messages in its generated traffic.

UPF processing. The UPF primarily handles two types of incoming traffic: PFCP messages that setup, modify and delete various packet forwarding *rules* corresponding to UE data sessions at the UPF, and user plane (GTP) traffic that is then handled as per these established rules. There are several types of rules at the UPF, as shown in Figure 2. Packet Detection Rules (PDRs) help match the traffic of a session based on packet header fields, e.g., source/destination IP address/port number, or GTP TEIDs. With each PDR, we have other associated rules that specify the action to be taken on the traffic that matches the PDR: Forward Action Rules (FARs) specify the forwarding action to be applied on a packet (e.g., GTP TEIDs to use for encapsulation and decapsulation), QoS Enforcement Rules (QERs) specify the QoS that must be enforced (e.g., maximum bit rate allowed for the session), Buffering Action Rules (BARs) specify buffering requirements when the UE is idle, and Usage Reporting Rules (URRs) specify how usage reporting should be performed for billing and charging. These various PDRs and their associated FARs, QERs, BARs, and URRs are established, modified, and deleted via PFCP messages from the SMF to the UPF. Once these rules are in place at the UPF, user plane GTP traffic is handled by finding a PDR that matches the received packet, and executing the actions specified by the associated FARs, QERs, BARs, and URRs. Prior work that uses programmable data plane hardware to accelerate the UPF [5, 7, 8, 18, 22, 24, 33] proceses PFCP messages in the software control plane, installs the various rules in the hardware, and offloads only the GTP user plane traffic handling to the programmable hardware.

PFCP message structure. A PFCP message has a highly complex structure. A PFCP message has several Information Elements (IEs), which are used to create, modify, or delete the packet forwarding rules at the UPF. For example, a PFCP session establishment message contains the following IEs [3]: a node identifier of the SMF which sent this message, a unique session identifier (SEID) that identifies the session, followed by one or more IEs to create PDRs, FARs, BARs, QERs, and URRs. Now, while some IEs like the node ID, SEID, and the IEs to create PDR and FAR are mandatory, some other IEs are optional and need not always be specified. Furthermore, a PFCP session establishment message can have a variable number of IEs to create PDRs, and each of these PDRs can cross reference the same or different FARs, BARs, and so on. Each of these IEs to create PDRs and other rules have a nested structure with several smaller IEs contained within, which can further have mandatory and optional elements. To complicate things further, the 3GPP standards allow IEs to be present in any order inside a message. Therefore, parsing and processing a PFCP message is a highly complicated operation that is hard to fully implement within the restricted processing available in hardware. This is the reason why no prior work that uses programmable hardware to accelerate UPF proposes processing PFCP messages in hardware.

Programmable hardware. Before the introduction of programmable data plane hardware, a high performance packet processing network element like the UPF was either developed as a fixed function hardware appliance or as a software packet processing application running over commodity hardware. While a hardware implementation provided higher and more deterministic performance, a software implementation had the benefit of easy programmability to add new features. In contrast to fixed function hardware, programmable dataplane hardware can be easily programmed (and quickly reprogrammed) to perform complex packet processing functions, via code written in a high-level language like P4 [21]. Therefore, programmable data planes provide the best of both worlds, with the performance of a hardware implementation and the flexibility of a software implementation. Packet processing specifications written in a high-level language like P4 are compiled to a variety of targets, e.g., programmable hardware ASICs [1, 2, 36, 37], NPUs [13, 45], and FPGAs [10, 17]. Languages like P4 have several limitations put in place in order to ensure linerate processing of the software specification. They have limited expressiveness in terms of the supported instruction set and programming constructs. The packets cannot stall during the switch pipeline processing—they have to be either forwarded or dropped. The amount of on-board memory on such hardware is limited in capacity

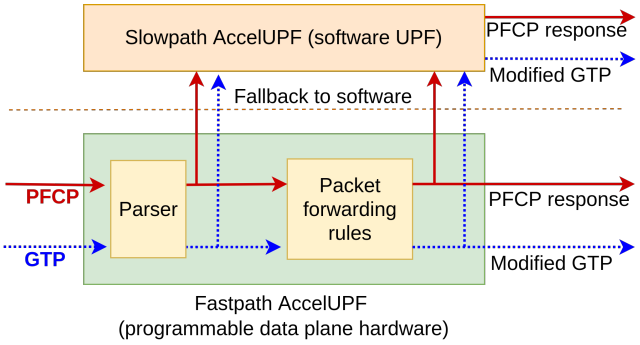

Figure 3: AccelUPF design.

(~few tens of MBs) [46], and provides a restricted storage model. Despite these limitations, researchers have observed substantial performance benefits by offloading applications to programmable hardware, though the functionality that is offloaded is often something that is simple enough to be executed within the limitations imposed by the hardware.

3 DESIGN

The goal of AccelUPF is to accelerate the performance of the 5G UPF using programmable data plane hardware. However, unlike prior work in this area, we aim to offload not just the GTP user plane forwarding but also the PFCP processing. We begin with describing the technical challenges that make this goal non-trivial to achieve.

3.1 Challenges

Complexity of PFCP processing. PFCP messages have a very complex structure, due to factors such as variable number of information elements (IEs), a large number of optional IEs in each message, nested structure of IEs, and the flexibility of ordering of the IEs within each message that is allowed by the 3GPP standards specification. Given this complexity, it is not easy to fully process all PFCP messages within the programmable hardware platforms available today.

Updating switch state from the data plane. Most applications that offload functionality to programmable hardware use the *match-action tables* available within the hardware platforms to store application state. Incoming packets are matched against these rules using the various fields in the packet header as keys, and the action corresponding to the matched rule is executed on the packet. While this key-based matching is often implemented efficiently using fast specialised hardware that can perform exact as well as ternary (wildcard) matches, today's programmable data planes only allow the match-action tables to be configured via from the software control plane via standard APIs, which can become a performance bottleneck under high PFCP traffic.

Hardware memory and compute limitations. The memory available to store application state in programmable hardware is limited, and may not be enough to accommodate the state of all users being served by a given UPF. The limited memory can also make some data traffic processing (e.g., buffering of data packets for idle users or for sessions that exceed their rate limit) hard to do within the hardware. Finally, the failure of the switch can result in the loss of application state stored in switch memory.

3.2 Design overview

We now provide an overview of AccelUPF's design (Figure 3), and the key ideas that help us address the challenges above.

Fastpath PFCP processing (§3.3). Given the complexity of processing PFCP messages in hardware, AccelUPF splits the PFCP processing into a fastpath in hardware and a slowpath in software. We identify the most frequent and simple patterns of PFCP messages that can be handled in hardware on the fastpath, but even these simple messages have a large number of optional IEs and several levels of nesting. To parse such messages correctly, AccelUPF identifies the finest granularity of IEs in the various PFCP headers, and chooses parser states dynamically based on the presence or absence of the various optional IEs.

Hardware data structures (§3.4). AccelUPF aims to avoid software control plane involvement in the fastpath of PFCP processing, and uses in-switch stateful memory called *register arrays* (that can be read and written from within the data plane itself) to store packet forwarding rules present in the PFCP messages. However, unlike match-action tables, register array access is only index-based and not key-based. So, AccelUPF uses the hash of header fields in received packets as an index to access packet forwarding rules within the register arrays. Computing this index is complicated by the fact that PFCP messages and GTP traffic have different sets of fields in the packet headers. Therefore, the data structures that store the packet forwarding rules in the register arrays are carefully designed so that the rules can be looked up via different indices for different types of traffic.

Software fallback (§3.5). AccelUPF uses the software slowpath as a fallback for sessions that cannot be handled in hardware. We ensure that the UPF state is shared correctly across the hardware and software components of AccelUPF, with clear state ownership to avoid race conditions.

Replication of switch state (§3.6). Unlike state stored in match-action tables that can be easily made fault tolerant using replication at the software layer, application state stored in register arrays in AccelUPF can be lost due to switch failures. To avoid losing forwarding rules state of users, AccelUPF replicates the switch state created due to hardware

processing of PFCP messages across other switches in the data plane using chain replication.

3.3 PFCP processing

AccelUPF handles the most common PFCP messages in the hardware fastpath, and redirects the remaining PFCP messages (and their associated data traffic) to the slowpath in software. PFCP messages that pertain to setting up and maintaining the PFCP association between SMFs and UPFs (also called PFCP node messages) are infrequent, happen outside the critical path of UE applications, and do not impact user-perceived performance in any way. Therefore, we handle all such messages only in the slowpath. Besides these messages, there are three main PFCP messages related to UE session management received at the UPF, which are the PFCP session establishment, modification, and deletion messages.

While a general PFCP session message can have a complex structure, with a variable number of packet forwarding rules per message, we argue that the message structure is simpler in the common case of a single application or service at a UE (e.g., mobile broadband or IoT) establishing a PDU session to transfer data. To see why, consider the initial PDU establishment callflow shown in Figure 2. Here, the SMF first sends a PFCP session establishment request to the UPF, which creates one uplink PDR (containing the UE's source IP address, GTP TEID, QoS flow identifier or QFI, and other packet header fields that identify a session's uplink traffic) and its actions. Later, the SMF sends a PFCP session modification request, which associates with the same session a downlink PDR (containing the UE's destination IP address and other packet header fields to identify downlink traffic) and its corresponding actions. Beyond these default rules, more PDRs for other applications can be added later to the same session via PFCP session modification messages, with each PDR created by a separate PFCP message. Therefore, PFCP messages in this common case do not necessarily need to contain more than one PDR in a single message.

Given this observation, the hardware fastpath of AccelUPF handles PFCP session establishment messages that create exactly one PDR, exactly one FAR, and at most one (optional) QER. Other PFCP session establishment messages, e.g., those creating more than one PDRs, are handled in the slowpath. We place similar restrictions on the PFCP session modification request also. Note that while we can extend our design to handle messages that create two (or any such small, fixed number of) PDRs in the hardware fastpath, handling an open-ended number of PDRs is infeasible in hardware.

Now, we note that designing a parser that can parse even this restricted set of PFCP messages is non-trivial for several reasons. First, the 3GPP specifications do not mandate a fixed order to the various IEs within a PFCP message. To parse

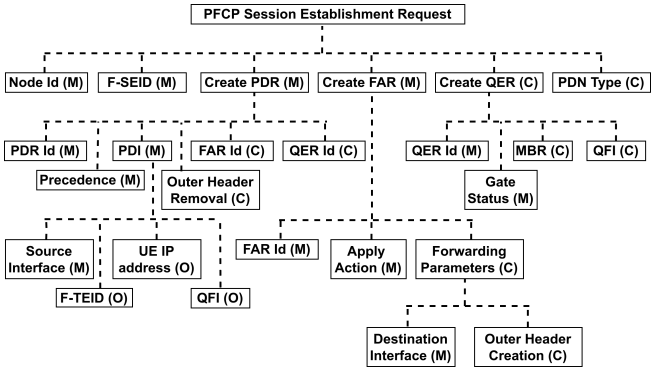

Figure 4: IEs implemented in the AccelUPF fastpath. M, C and O represent Mandatory, Conditional and Optional IEs respectively.

such messages correctly in hardware, we make the assumption that the various IEs within a PFCP message appear in the exact order in which they are specified in the standards documents. Without this assumption, the complexity of parsing a PFCP message in hardware becomes intractable as the various IEs cannot be represented as a directed acyclic graph (DAG) anymore. Even if this assumption does not hold in some existing 5G implementations, it is easy to enforce with minimal changes to control plane components like SMF.

Second, PFCP messages contain a large number IEs (321 in the latest specification version 17.5.0 [3]). Of these, some are classified as mandatory, some optional, and some conditional (i.e., mandatory or optional, depending on some condition being satisfied at the time of processing the message). Further, many IEs are nested IEs, with each nested level having its own set of mandatory, optional, and conditional IEs. For example, in a PFCP session establishment request handled by AccelUPF fastpath (Figure 4), the session identifier is a mandatory IE, while the UE IP address is an optional IE. The IE to create a QoS rule is a nested, conditional IE, which will be present for UEs specifying QoS rules, and absent otherwise. To parse such messages correctly, we identify the smallest units (simple non-recursive IEs or even part of IEs) that may be present or absent in a PFCP message. We classify conditional IEs as mandatory or optional suitably according to the PFCP message structure being handled in the fastpath. Now, between every pair of mandatory IEs, we generate multiple parser states and state transitions, guided by the absence/presence of the various optional IEs. At each stage of the parsing, if the next IE is optional, the parser looks at the first 16 bits of the next IE, and decides the next parser state based on which IE it finds next. For example, consider a sequence of 4 IEs, say, A, B, C, and D, in a message. Out of these IEs, suppose IEs A and D are mandatory, and B and C are optional. To correctly parse these optional IEs, we create the following parser states and state transitions: A $\rightarrow$ B $\rightarrow$

C → D, A → C → D, A → D, A → B → D. The actual parser state transitions will be guided by whether IEs B and C are present in the received message or not. We create multiple parser states in this manner for all optional IEs in the message. Across the entire PFCP session establishment message shown in Figure 4, this results in 28 parser states, 19 distinct paths in the directed acyclic parse graph (DAG), and 42 transitions between parse states.

Given the challenges in parsing PFCP messages in hardware identified above, we believe that future standardization efforts must incorporate simplifications to the PFCP message format, e.g., enforcement of a fixed IE order, a limit on the length and/or number of IEs supported in each message, in order to enable faster processing on hardware accelerators.

3.4 Hardware data structures

After parsing PFCP messages, AccelUPF stores the resulting packet forwarding rules inside stateful memory available within most modern programmable hardware platforms called register arrays. A register array is a storage abstraction for an array of P4 registers, where the width of the register and the length of the array are configurable. The register array entries can be read or written during the various stages of a packet processing pipeline directly from the switch data plane itself. While one can lookup entries in a match action table using a key (either exact or ternary), a register array entry can be accessed only using its index.

A strawman approach to store packet forwarding rules in register arrays would be to compute a hash over some packet header fields and use this hash as an index to access the register array entry required to process this packet. But, which packet header fields can we use? Every PFCP session message has a unique session identifier (SEID), which uniquely identifies a session. Therefore, one would think that all session-related states, including the PDRs, FARs, and other rules associated with a session, can be stored and retrieved in a register array using the hash of a SEID as an index. However, user plane traffic received from the UE (downlink IP traffic that must be encapsulated in GTP headers, or uplink GTP traffic that must be decapsulated) does not contain this session identifier in the GTP or IP layer headers. For example, given only the information within a GTP data packet (GTP TEIDs, source/destination IP addresses etc.), how does one identify the exact session that this packet belongs to, without knowing the index in the array at which this session is stored, and without having the ability to loop over all the entries in the array due to the constraints of hardware linerate processing?

To overcome this challenge, AccelUPF stores packet forwarding rules across multiple different register arrays, each accessed via a different index, as shown in Figure 5. The first

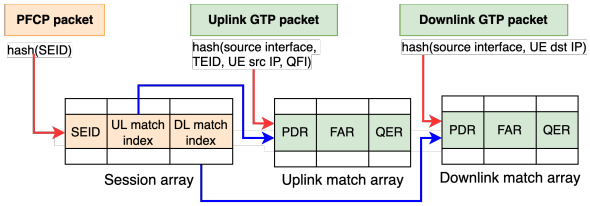

Figure 5: AccelUPF hardware data structure.

data structure is called a *session array*, which is accessed using the hash of the session identifier as an index. When a PFCP session message (establishment, modification, deletion) is received, we compute the hash of the unique SEID, and use this as an index to locate the register array entry at which this session's information is stored. This register array entry is then updated with information about the session as requested in the PFCP message. The second data structure is the *uplink match array*, which stores information about the PDR and the associated actions rules like FAR corresponding to uplink traffic. The index into this array is the hash of the various packet header fields that can be obtained from the GTP-encapsulated packets in uplink traffic (we use UE's source IP, TEID and QoS flow identifier, but other choices are possible too). An analogous data structure is the *downlink match array* that stores packet rules for downlink traffic of a session, and is indexed using the hash of the packet header fields present in downlink traffic (UE's destination IP in our case). Note that we will require different match arrays if we wish to use a different set of packet header fields for index computation. The entry of a session in the session array does not store the PDRs directly within the entry itself; instead it stores the indices of the uplink/downlink match arrays at which the uplink/downlink packet forwarding rules of a session are stored. This cross-linkage helps us avoid duplication of information across the various register arrays.

When AccelUPF receives a PFCP message, it computes the hash of the SEID, locates the entry corresponding to this session, and uses the stored indices to access all the packet forwarding rules (PDRs, FARs, etc.) of the session that are stored in the uplink and downlink match arrays. These various register array entries are then suitably updated to add/delete/modify rules based on the received PFCP request. A PFCP response indicating the status (success/failure) of the request is generated by modifying the contents of the request packet itself. When AccelUPF receives a data packet, it identifies whether it is an uplink or downlink packet, and uses the hash of the suitable packet header fields to index into the corresponding match array. It then verifies that the packet matches the PDR, and executes the actions specified within the FAR and other rules in case of a match.

AccelUPF currently supports only an exact matching of packet header fields with the information in the PDR. If

one has to perform other kinds of complex matching, e.g., PDR specifies a prefix and we must perform a longest prefix match, such matching cannot be performed using register arrays. AccelUPF handles sessions with such packet rules in the software slowpath. In addition to forwarding actions, AccelUPF must also enforce other rules corresponding to QoS enforcement, buffering data for idle users, and usage reporting. Of such rules, our implementation currently supports the enforcement of a session-wide aggregate maximum bit rate (AMBR) by computing per-flow rates using ingress timestamps and interpacket gaps. Support for more complex policies is deferred to future work. If a session exceeds its configured AMBR, its data packets have to be buffered, which is once again handled in the slowpath.

What if packets belonging to two different sessions hash to the same index within the match array? Our current implementation stores two entries in a hash bucket using dual-width registers and other such mechanisms available via P4 extern units on most programmable switches. We can also use techniques such as multiple hash functions to find alternate indices [38]. However, we will eventually face a hash collision, where two different sessions are contending for the same entry in the register array. Hash collisions are handled in AccelUPF by handling all traffic of the colliding session in the software slowpath.

3.5 Software fallback

Any PFCP message that cannot be handled within the hardware fastpath in AccelUPF is redirected to a software UPF running in host userspace, as shown in Figure 3. Examples of such PFCP messages include: (i) node-related PFCP messages that are not on the critical path of user-perceived performance; (ii) PFCP messages that contain a large number of PDRs and associated action rules, which cannot be easily parsed in hardware; (iii) sessions that require complex algorithms like longest prefix matching to match incoming data traffic to packet forwarding rules; (iv) sessions which hash to the same index in the register arrays. All such PFCP messages are redirected to the slowpath software UPF, which handles them normally, and creates suitable state in the form of packet forwarding rules in software. All subsequent PFCP session modification/deletion messages or GTP user plane packets of this session will also not find a matching rule in hardware and will thus be forwarded to, and correctly processed in, the software slowpath.

How are the session state and packet forwarding rules shared correctly across the hardware fastpath and software slowpath? Note that for most sessions, the state is created, modified, and deleted exclusively either within the hardware fastpath or in software. Therefore, the question of ownership of state is trivial to resolve in most cases. The only tricky scenario is when a session is initially created in hardware, but needs to fallback to the software slowpath midway due to reasons such as: (i) a session that was being handled by the hardware fastpath starts sending data at a rate beyond its configured maximum bit rate, and must fallback to the software for buffering, or (ii) we receive a PFCP session modification request for an already established session that was being handled in the hardware, and this PFCP message has a complex structure, e.g., a rule that requires longest prefix matching or one where multiple PDRs refer to the same FAR, and must therefore be processed in software. For such scenarios, all subsequent PFCP message processing and GTP forwarding of the session must *migrate* from hardware to software.

This state migration is accomplished as follows. When the hardware realizes that it can no longer process a certain session in the fastpath, it marks all the session states in the register arrays as invalid and under migration. All subsequent PFCP messages and user plane packets are forwarded to the software slowpath as a fallback. When a software slowpath receives a PFCP session modification or deletion message, or a GTP user plane packet, but does not find corresponding state in its data structures, it probes the hardware to find if this is a case of a session being migrated from hardware to software after initially being created in hardware. If it finds the corresponding state in the hardware register arrays as marked for migration, it copies this state to software, and deletes the corresponding invalid entry in the hardware register array data structures. All subsequent packets of this session will find the session and packet forwarding state in software and will be correctly handled in the slowpath. If the software UPF does not find the state to process a PFCP/GTP packet either in its own software data structures or after probing the hardware data structures, it drops the packet.

We note that an alternate design is possible where we handle complex PFCP messages in software and install the session rules directly to the hardware, allowing us to process future traffic of such sessions in the fastpath. However, this approach requires more frequent updates to the hardware rules from the software slowpath. Considering the hardware-software communication bottleneck and limited performance gains only for a small set of complex PFCP sessions, AccelUPF has not implemented this hybrid approach.

3.6 Fault tolerance of switch state

AccelUPF stores packet forwarding rules in register arrays, which are not persistent across switch failures. While a software UPF, or even a hardware accelerated UPF that only offloads GTP user plane forwarding, can use software-based mechanisms to replicate and persist session state across switch/host failures, AccelUPF cannot rely on software replication for hardware switch state. Therefore, AccelUPF relies

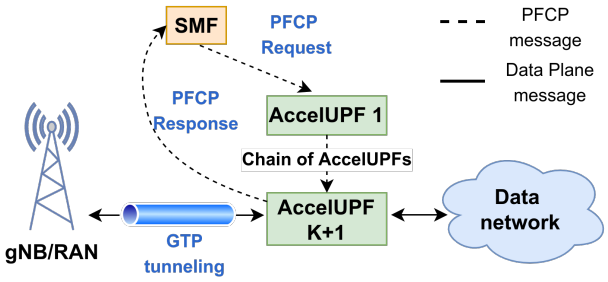

Figure 6: AccelUPF fastpath fault tolerance.

on techniques similar to those proposed in prior work for replication of switch state [23, 29, 30, 49]. To maintain packet forwarding rules in register arrays in a fault-tolerant manner, every update to the register arrays is replicated at $K + 1$ switches arranged in a linear fashion to achieve a K fault-tolerant system. The last switch in the chain replication acts as the primary UPF, and sends a response to the PFCP request. We use $K = 1$ in our current implementation. What if the primary switch fails before replication is completed? In such cases, we will not generate a response back to the SMF, and the PFCP reliability feature [3] will ensure that the SMF will detect the loss of the PFCP message via sequence numbers embedded on PFCP messages, and will retry the PFCP request once again at a new switch.

Note that our replication only increases the time required for PFCP message processing, and does not impact the performance of user traffic. This is because the GTP user plane traffic is only routed to the primary UPF from the RAN, and is re-routed to the backup switches in the replication chain only when the primary fails. Figure 6 shows the path taken by PFCP messages and GTP packets in our fault-tolerant AccelUPF design.

4 IMPLEMENTATION

This section describes the implementation of AccelUPF. The fastpath of AccelUPF is implemented in P4, and compiled to run on two targets: an Agilio CX Netronome 2x40GbE smart NIC [46] and an Intel Tofino based Edgecore Wedge 100BF-32X programmable switch [36]. For a smart NIC-based programmable data plane platform, the fastpath runs on the NIC and the software slowpath runs in the host userspace. For a switch-based platform, the fastpath runs inside the switch packet processing pipeline, and the slowpath can either run on the switch CPU or on a host connected to the switch.

We used two different hardware platforms to implement our hardware fastpath because these platforms have different internal implementations of abstractions like register arrays, on which our design depends heavily. The Netronome smart NIC platform uses multiple packet processing engines that process packets in a run-to-completion manner. Large register arrays are stored in memory that is shared across all engines, and one engine accessing a register may cause another engine to stall [47]. On the other hand, the Tofino programmable switch uses a pipelined design where the register array can be partitioned across multiple pipeline stages, resulting in better performance when accessing registers from the packet processing pipeline [28]. AccelUPF uses register arrays available at different part of the P4 pipeline (ingress, egress), and across different pipelines, provided they are independent from each other. AccelUPF distributes the forwarding sessions among the different sets of the register arrays by matching the most significant bits of the register array indices, which are generated by hashing the match fields in the incoming PFCP and GTP packets.

Our slowpath implementation builds upon a production-grade standards-compliant software UPF [9]. We worked with two different flavors of this UPF for our slowpath, one running on the kernel network stack, and another built on top of the high-speed DPDK packet I/O framework. Both the kernel-based and DPDK-based UPFs came with a multicore scalable design, with separate CPU cores processing PFCP messages and GTP user plane traffic, though the GTP forwarding throughput of the DPDK prototype was much higher. We made minor modifications to the software UPFs to work with our fastpath, e.g., to support migration of session state from the fastpath in case of fallback to software. We also use the unmodified pure software DPDK-based UPF to serve as a baseline in our evaluation.

As another baseline, we modified the software UPFs to offload the GTP user plane forwarding functionality to a programmable NIC/switch. In this GTP-offloaded UPF prototype, the UPF processes PFCP messages normally, and in addition to installing session state locally, also installs the packet forwarding rules within the NIC/switch hardware using the programmable data plane hardware APIs provided by the hardware platform vendors. We optimized this rule installation to work at the maximum rate supported by the hardware, by using multiple threads in the software control plane to install rules in parallel.

The software UPF also came with a load generator, a multi-threaded DPDK application that emulates the functionality of the UEs, RAN, and the packet core control plane. The load generator generates PFCP messages and GTP user plane traffic on behalf of emulated UEs, and provides knobs to vary the relative mix of PFCP messages and GTP traffic in the generated traffic.

5 EVALUATION

Setup. We use two different programmable hardware platforms to evaluate AccelUPF, shown in Figure 7. The setup

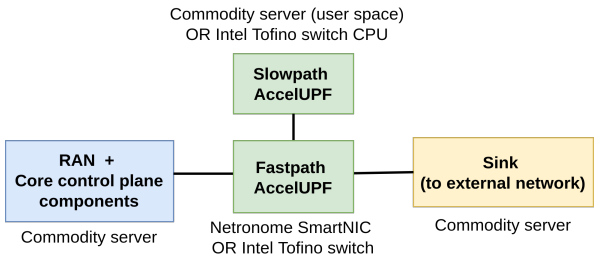

Figure 7: Experimental setup.

with the Netronome smart NIC contains three servers (AMD Ryzen 9 5950X processors@3.4GHz, 16 cores 32GB RAM), each connected to an Agilio CX 1/2x40GbE programmable smart NIC [46]. The first server hosts the load generator that generates PFCP and data traffic on behalf of emulated UEs, the second hosts the various UPFs we test, and the third hosts a sink that serves as the destination for the traffic generated from the load generator. The setup with the Tofino programmable switch consists of two servers (Intel Xeon Gold 6234 CPUs@3.30GHz processors, 24 cores, 128GB RAM) with 40Gbps NICs connected via an Intel Tofino Edgecore Wedge 100BF-32X programmable switch [25]. In this setup, the first server runs the load generator and the second server runs the sink. The slowpath software UPF runs on an internal Intel Pentium CPU D1517 @ 1.60GHz CPU connected to the data plane of the switch over the PCI bus.

Parameters and metrics. Across all experiments, we vary the mix of PFCP messages to GTP data packets in the UPF traffic by varying the knobs provided in the load generator. We measure the peak PFCP message processing throughput (messages/sec) and peak GTP data forwarding throughput from the load generator. We also measure the average PFCP processing latency and GTP forwarding latency at saturation. All experiments were run for a duration of 300 seconds, and we ensured that the load generator and the sink were not the performance bottleneck.

UPF variants. We compare the performance of AccelUPF (running on Netronome/Tofino) with the following baseline UPFs described in §4: a pure software DPDK-based UPF, a GTP offloaded UPF (with data forwarding offloaded to the Netronome/Tofino programmable hardware platforms).

5.1 Microbenchmarks

We first conduct several simple microbenchmarking experiments on all our UPF variants, and compare the PFCP message processing throughput/latency and GTP data forwarding throughput/latency. We also normalize the PFCP and GTP throughputs by the cost and power consumption of the corresponding commodity server or programmable hardware platforms [12, 14, 15], in order to measure performance per unit cost or power consumed. For the normalization, we scaled the cost of the server/switch/NIC according to the number of cores/ports actually used. All the results are shown in Table 1. We highlight some observations below.

PFCP throughput and latency. We first generate session establishment and deletion requests from emulated UEs in the load generator, and measure the maximum PFCP message processing throughput and average message processing latency or RTT for all UPFs, for a workload consisting only of PFCP messages. We find that AccelUPF has much higher PFCP performance (4.3M PFCP messages/sec on Tofino) as compared to the baseline UPFs, which could process only a few thousand PFCP messages/sec.

A few observations on the results shown in Table 1. (i) We note that the software UPF in our experiment was processing PFCP messages on a single core. One could argue that the performance of the software UPF can scale to a higher PFCP processing capacity by adding more CPUs, but this scaling would increase the overall cost and power consumption of the system. For example, one would need over 500 cores on our server hosting the software UPF for it to match the performance of AccelUPF. The PFCP performance normalized by cost or power consumed shows that AccelUPF is still more efficient than the software UPF, even if one were to scale the software UPF to a higher number of CPU cores. (ii) AccelUPF also performs much better than the GTP-offloaded UPF, both in terms of throughput and latency, because processing PFCP messages in hardware is much more efficient than processing them in software and installing the packet forwarding rules in hardware. To confirm that the hardware rule installation is indeed the bottleneck in the GTP-offloaded UPFs, we installed packet forwarding rules from a multi-threaded software controller program and measured the maximum rate at which the programmable hardware platform can install packet forwarding rule. This rule installation capacity turned out to be the equivalent of 4448 PFCP msg/s for Netronome and 11406 PFCP msg/s for Tofino, which provides an optimistic upper bound for the PFCP processing capacity on these platforms, even if one were to disregard all other processing overheads. (iii) AccelUPF performs much better on Tofino than on Netronome, because of the higher overhead of register access in Netronome (see §4). Reading and writing a single register takes around 150-590 clock cycles on the Netronome platform [47], and processing a PFCP message involves a few tens of register accesses. This explains the higher latency (and lower throughput) of AccelUPF on Netronome as compared to Tofino, which allows a much faster access to registers via its pipelined design [28].

GTP forwarding throughput and latency. Next, we establish sessions for 1K users ahead of time, and measure only the maximum GTP data forwarding throughput and average forwarding latency or RTT for all UPFs. We use IMIX packet size [16] and only uplink traffic (results for other packet

UPF design	PFCP				GTP			
	Tput (msg/s)	msg/s/USD	msg/s/Watt	RTT (us)	Tput (Mpps)	Kpps/USD	Kpps/Watt	RTT (us)
SoftwareUPF	8309	85.51	949.60	40	11.93	17.53	194.77	85
GTPOffload Netronome	1953	6.39	78.12	1470	10.51	17.20	210.20	71
GTPOffload Tofino	499	1.91	31.23	447	11.94	22.91	373.12	49
AccelUPF Netronome	794849	2601.80	31793.96	114	4.83	7.91	96.60	115
AccelUPF Tofino	4389254	16841.26	274328.37	35	11.94	22.91	373.12	49

Table 1: PFCP processing and GTP forwarding performance of UPFs.

sizes and downlink forwarding were similar). We find that all UPFs, with the exception of AccelUPF on Netronome, are able to process packets at the linerate of 40Gbps. Furthermore, the hardware-accelerated UPFs have a much better packet processing performance per unit cost or power consumed, as compared to the pure software UPF. The only outlier is the poor GTP forwarding throughput of AccelUPF on Netronome, which is once again due to the higher overhead of register access on the Netronome smart NIC platform. While the GTP offloaded UPF on Netronome stores its packet forwarding rules in match-action tables, AccelUPF uses in-switch register arrays which are shared across all packet forwarding engines in the smart NIC, leading to frequent stalls and poor packet processing performance. However, AccelUPF on Tofino has no such issues, and performs on par with the other state-of-the-art UPF variants. This experiment highlights the importance of choosing a good programmable data plane hardware platform to deploy AccelUPF on. If the underlying hardware platform cannot support efficient register access, AccelUPF may not be suitable for UPFs that see a large share of GTP traffic and relatively small PFCP traffic.

Overhead of chain replication. The above microbenchmarks of AccelUPF were obtained with the replication of switch state (along a chain of K+1 switches for a K-fault tolerance system) turned off. We measured performance with replication turned on, and we found no noticeable degradation in PFCP or GTP processing throughputs of AccelUPF. However, the PFCP latency of AccelUPF when replicating switch state at one other backup switch was 60% higher than that of AccelUPF with no fault tolerance. However, given the extremely low latencies of message processing in AccelUPF (few tens of microseconds in most cases), we do not expect this overhead to be a big concern.

Maximum number of user sessions. The maximum number of user sessions that can be supported by AccelUPF depends on the size of the register array memory available to store packet forwarding rules of a session, the number of separate pipelines available for the hardware platform with distinct register array, and the size of the hash computed by the hardware platform. Across both hardware platforms, we found that we could support 64K entries in each register array. This translates to a maximum of 64K user sessions for our Netronome platform, considering it has a single functional copy of each register array. Tofino has disjoint registers for its ingress and egress pipelines, so we can support 128K users in our current implementation. Note that for workloads generating frequent PFCP messages (say, every 10 seconds, which is the common value of the inactivity timer after which a session is marked as idle), the PFCP traffic generated by 128K users can only be comfortably and efficiently handled by AccelUPF, and by not any other UPF design we have evaluated. It is possible to increase capacity of AccelUPF further by using multiple pipelines present in switches. The Tofino switches support up to 4 pipelines (2 in our model), so it is possible to increase the capacity of AccelUPF to 512K users, which we plan to explore as part of future work.

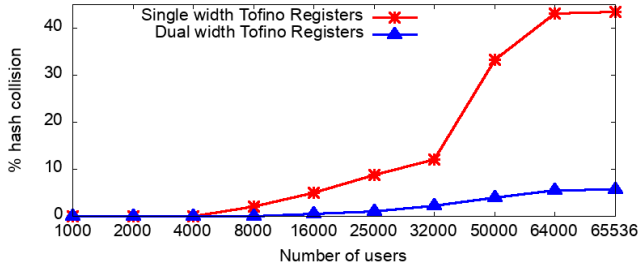

Figure 8: Rate of hash collisions.

However, as the number of users approaches capacity, hash collisions can become a problem. Figure 8 shows the percentage of sessions that see a hash collision as the number of active sessions at the UPF increases. These measurements were obtained from the AccelUPF prototype running on Tofino in two different cases. First, we store a single entry in a hash bucket using single width Tofino registers. In this case, we see that the number of hash collisions is relatively low until the system has about 32K users, and the hash collisions increase afterwards reaching to as high as 43% at the full capacity of 65k users. This would be a significant load for the slowpath software UPF. However, with using dual width registers, storing 2 entries in each hash bucket, we can bring the hash collisions to a low value (under 10%), and we can comfortably support 64K users in each register array.

Overhead of software fallback. AccelUPF needs to migrate an established session from hardware to software in certain scenarios, e.g., when a session exceeds its configured maximum bit rate. In such cases, when the processing needs to fallback to software, the first packet of the flow after migration will incur a high overhead, due to the software probing the hardware state and migrating it to the slowpath. We measured this overhead and found that the first packet processed in a session immediately after software fallback incurs a processing latency of around 2.4 milliseconds, as compared to the average case latency of around 100 microseconds. We argue that this migration overhead is not a major concern. The two main cases where a user session has to be migrated are when a user sends traffic beyond the configured rate limit, or when a session is modified using complex PFCP messages that cannot be parsed in the fastpath. In the former case, we argue that the user's traffic will suffer long delays due to buffering anyways, and the extra overhead of migration will not adversely impact performance. In the latter case, we expect future implementations of the SMF working with AccelUPF to evolve towards simpler PFCP messages, at least for UEs with stringent performance guarantees.

5.2 Real world traffic

We now present an evaluation of AccelUPF on real-world traces. We choose an IoT application that is likely to see a larger fraction of PFCP messages to GTP data, because an IoT device performs frequent signaling while sending small amounts of data intermittently. We obtained IoT packet traces from 5 different IoT applications [44]. These traces contain the packet sizes and timestamps of when packets were generated by various IoT applications. We then extrapolate the traces to add PFCP messages, to simulate the scenario where these IoT devices would be connected over a mobile telecom network as follows. We add PFCP messages corresponding to PDU initial session establishment for each IoT device at the start of the trace. Further, when an IoT device goes inactive after an idle period, we add PFCP messages corresponding to the AN release procedure that transitions a user from a connected state to the idle state. We also add PFCP messages corresponding to a service request when the user resumes activity once again. The inactivity timer is usually set to a few seconds [34] by network operators to reclaim radio resources of inactive users. We use a value of 10 seconds. After adding the emulated PFCP messages, the relative mix of PFCP and data traffic in the 5 traces (named trace A to trace E) is as shown in Table 2. We now generate PFCP and GTP traffic from our load generator in these ratios, obtaining other metrics like average packet sizes also from the trace. We measure the total packet processing throughput of the various UPF variants for these IoT workloads.

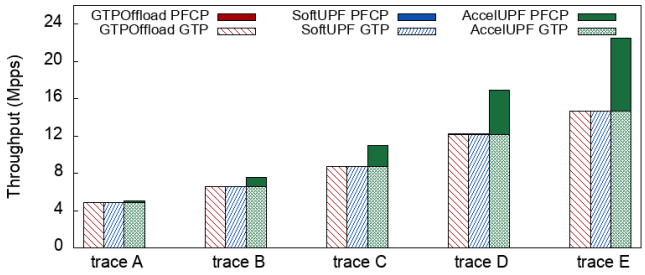

Figure 9: Comparison of UPFs for IoT traffic.

IoT Trace	A	B	C	D	E
PFCP %	3.65	12.53	19.63	28.86	35.79

Table 2: PFCP messages in IoT traces.

Figure 9 shows the PFCP and GTP processing throughputs (in pps) for the various UPFs (we omit GTPOffloadUPF Netronome and AccelUPF Netronome for clarity). We note that the average packet size in the IoT traces decreases from trace A to trace E, which resulted in an increased throughput (in Mpps) from left to right. The PFCP processing capacity of GTPOffload designs and software UPF was too low to be visible on top of the GTP throughput in our representation in Figure 9. We find that AccelUPF has around 57% higher throughput than SoftUPF or GTPOffloadUPF on Tofino for the trace E, which had 35.79% PFCP messages, and the increased throughput was primarily due to the higher PFCP message processing capacity of AccelUPF compared to the other two designs.

We note that under high PFCP traffic coming from a large number of UEs (beyond the capacity of the hardware fastpath), the AccelUPF slowpath will still have to support a high PFCP throughput. The software UPF on the slowpath (whose PFCP throughput per core is much lower than AccelUPF hardware, see Table 1) is scaled to run on multiple cores in such cases, to effectively handle the traffic coming to the slowpath. However, the number of cores required will be much lower than in the case of a pure software UPF, because the fastpath is expected to handle a bulk of the traffic.

6 RELATED WORK

State-of-the-art UPFs. Most production grade UPFs are built over kernel-bypass techniques like DPDK to achieve high user plane throughput in software. Neutrino [19] proposes a DPDK-based edge solution that replaces the 5G standards-based components by a 5G non-compliant control plane solution which is fast, reliable, and fault-tolerant. Metaswitch [6] uses a specialized processing engine (CNAP) in the software itself to achieve high throughput. Some UPFs

also use programmable hardware or specialized processing engines to offload some part of the UPF processing to hardware. Few proposals [7, 8, 18, 22, 24] offload the GTP encap/decap based forwarding to hardware, while some [31] offload packet steering to cores via deep packet inspection (DPI) of the inner IP header. Kaloom [5] offloads a subset of QoS processing (bit rate policing) along with GTP processing to the programmable hardware. TurboEPC [40] offloads the subset of 4G core signaling messages to the programmable hardware, but the proposed changes are not standards compliant. uP4 [33] offloads the 5G UPF user plane processing to programmable hardware and uses microservices that run on commodity hardware to process the corresponding PFCP signaling messages.

Our previous position paper [20] evaluated the costs and benefits of multiple 5G UPF designs (with and without hardware offload) and quantified the performance gains of user plane traffic offload. The work also identifies the PFCP processing bottleneck of the programmable data plane accelerated UPF when only data handling is offloaded, and proposes the offload of PFCP processing as well to programmable hardware.

Control plane offload. Much like AccelUPF, prior work has also proposed offloading the control plane logic of network functions (and not just the data plane) to programmable hardware, and highlighted the challenges with the same. Mantis [48] designs a control plane architecture over programmable switches that can react to data center network conditions within tens of μs to resolve congestion events that are microscopic in duration. Molero et al. [35] achieves line rate for internet routing by processing failure detection, distributed path-vector computations (shortest-path and BGP-like policies), and forwarding state updates, entirely within the data plane. D2R [43] implements fast reroute during network failure by performing route computation without control plane intervention. Lucid [41] presents a framework that simplifies the in-network implementation of control plane constructs such as stateful table data structures, periodic event triggers and event handler processing, packet buffering, traffic shaping, and synchronized state writes. Lucid also proposes a high-level language for writing control function code, and the compiler translates this code to the optimized target code for Intel Tofino switches. AccelUPF is complementary to, and strengthens the case for, frameworks like Lucid.

Fault-tolerance of switch state. With many stateful applications offloaded to the programmable data planes, protecting application state under switch failure conditions and concurrent state access is essential. Prior work has proposed state replication and fault tolerance solutions for such applications, some of which we leverage for fault tolerance of switch state in AccelUPF. Netchain [29] proposes protocols and algorithms that ensure strong consistency and fault-tolerance for an in-network key-value store. Choi et al. [23] and SwiSh [49] introduce new replication protocols for in-network state. Redplane [30] implements a fault-tolerant state store that ensures consistent application state access even if the switch fails or traffic is rerouted to another switch, while offering two consistency modes; strong consistency and bounded-inconsistency.

7 CONCLUSION

This paper presented the design, implementation, and evaluation of AccelUPF, a programmable data plane hardware accelerated 5G user plane function. Prior work on using programmable hardware to accelerate the mobile packet core user plane was restricted to offloading only the GTP user data forwarding functionality to hardware, while continuing to process PFCP messages that configure the packet forwarding rules in software. These designs perform badly when applications frequently reconfigure packet forwarding rules while sending little data in between (e.g., IoT applications), because the software control plane APIs that reconfigure hardware rules have a limited capacity. To overcome this bottleneck, AccelUPF offloads the processing of most PFCP messages to the programmable hardware as well, carefully working around the memory and compute constraints of the hardware platforms when processing the complex PFCP messages. Our experiments show that AccelUPF significantly improves UPF packet processing performance as compared to previous offload-based UPF designs, especially when the traffic has a high proportion of PFCP messages. Our work highlights the challenges in processing a complex protocol like PFCP in programmable hardware. Given the significant performance gains that accrue from processing PFCP messages in programmable hardware at the UPF, our work provides guidance on how the future versions of PFCP for 6G and beyond can evolve to make them amenable for acceleration using programmable dataplane platforms.

ACKNOWLEDGEMENTS

We thank our shepherd Hyojoon Kim, and the anonymous reviewers, for their insightful feedback. We thank the 5G testbed project, funded by the Department of Telecommunications, Govt. of India, for access to the various 5G core components. We thank Dr. Venkanna U. and his research team at IIIT Naya Raipur, especially Suvrima Datta, for providing access to their hardware setup during our initial work. We also thank the Fast Forward Initiative Hardware Grant Program by Intel® Connectivity Research Program (ICRP) for their grant of a programmable switch.

REFERENCES

[1] 2013. *Cisco highlights next big switch.* https://www.biztechafrica.com/article/cisco-announces-next-big-switch/5448

[2] 2015. *Cavium Xpliant ethernet switch product line.* https://people.ucsc.edu/~warner/Bufs/Xpliant-cavium.pdf

[3] 2017. *3GPP Ref #: 29.244. 2017. System architecture for the 5G System (5GS).* https://www.3gpp.org/ftp/Specs/archive/29_series/29.244

[4] 2017. *3GPP Ref #:23.501. 2017. System architecture for the 5G System (5GS).* https://www.3gpp.org/ftp/Specs/archive/23_series/23.501

[5] 2019. *The Kaloom 5G User Plane Function (UPF).* https://www.mbuzzeurope.com/wp-content/uploads/2020/02/Product-Brief-Kaloom-5G-UPF-v1.0.pdf

[6] 2019. *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

[7] 2020. *5G User Plane Function (UPF) - Performance with ASTRI.* https://networkbuilders.intel.com/solutionslibrary/5g-user-plane-function-upf-performance-with-astri-solution-brief

[8] 2020. *Optimizing UPF performance using SmartNIC offload.* https://www.mavenir.com/app/uploads/2020/11/Mavenir_UPF_Solution_Brief.pdf

[9] 2022. *5G testbed at IIT Bombay.* https://www.cse.iitb.ac.in/~5gtestbed

[10] 2022. *Altera.* https://www.mouser.in/manufacturer/altera

[11] 2022. *DPDK Overview.* https://doc.dpdk.org/guides/prog_guide/overview.html

[12] 2022. *Edgecore Wedge 100BF-32X 32-Port 100GbE Bare Metal Switch with ONIE - Part ID: Wedge100BF-32X-O-AC-F-US.* https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3485&idcategory=

[13] 2022. *EZchip.* https://www.radisys.com/partners/ez-chip

[14] 2022. *Intel XL710-BM2 Dual-Port 40G QSFP+ PCIe 3.0 x8, Ethernet Network Interface Card.* https://www.fs.com/products/75604.html

[15] 2022. *Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz.* https://www.intel.com/content/www/us/en/products/sku/91767/intel-xeon-processor-e52650-v4-30m-cache-2-20-ghz/specifications.html

[16] 2022. *Internet Mix (IMIX) Traffic.* https://en.wikipedia.org/wiki/Internet_Mix

[17] 2022. *Xilinx.* https://www.xilinx.com

[18] Ashkan Aghdai et al. 2018. Transparent Edge Gateway for Mobile Networks. In *IEEE 26th International Conference on Network Protocols (ICNP).*

[19] Mukhtiar Ahmad, Syed Usman Jafri, Azam Ikram, Wasiq Noor Ahmad Qasmi, Muhammad Ali Nawazish, Zartash Afzal Uzmi, and Zafar Ayyub Qazi. 2020. A Low Latency and Consistent Cellular Control Plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication.*

[20] Abhik Bose, Diptyaroop Maji, Prateek Agarwal, Nilesh Unhale, Rinku Shah, and Mythili Vutukuru. 2021. Leveraging Programmable Dataplanes for a High Performance 5G User Plane Function. In *5th Asia-Pacific Workshop on Networking (APNet).*

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review* 44 (2014).

[22] Carmelo Cascone and Uyen Chau. 2018. Offloading VNFs to programmable switches using P4. In *ONS North America.*

[23] Sean Choi, Seo Jin Park, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2019. Toward Scalable Replication Systems with Predictable Tails Using Programmable Data Planes. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking (APNet).*

[24] Zhou Cong, Zhao Baokang, Wang Baosheng, and Yuan Yulei. 2022. CeUPF: Offloading 5G User Plane Function to Programmable Hardware Base on Co-Existence Architecture. In *Proceedings of the ACM International Conference on Intelligent Computing and Its Emerging Applications.*

[25] Edge-core. 2022. *Quick Start Guide 32-Port 100G Ethernet Switch Wedge100BF-32X.* https://www.edge-core.com/_upload/images/Wedge100BF-32X_QSG-R01_EN-SC_0114.pdf

[26] Michaela Goss. 2022. *Macrocell vs. small cell vs. femtocell: A 5G introduction.* https://www.techtarget.com/searchnetworking/feature/Macrocell-vs-small-cell-vs-femtocell-A-5G-introduction

[27] R. E. Hattachi. 2015. *Next Generation Mobile Networks, NGMN.* https://www.ngmn.org/wp-content/uploads/NGMN_5G_White_Paper_V1_0.pdf

[28] Intel. 2021. *P416 Intel® Tofino™ Native Architecture – Public Version.* https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf

[29] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[30] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. 2021. RedPlane: Enabling Fault-Tolerant Stateful in-Switch Applications. In *Proceedings of the ACM SIGCOMM Conference.*

[31] DongJin Lee, JongHan Park, Chetan Hiremath, John Mangan, and Michael Lynch. 2018. *Towards achieving high performance in 5G mobile packet core's user plane function.* https://builders.intel.com/docs/networkbuilders/towards-achieving-high-performance-in-5g-mobile-packet-cores-user-plane-function.pdf

[32] Yuanjie Li, Qianru Li, Zhehui Zhang, Ghufran Baig, Lili Qiu, and Songwu Lu. 2020. Beyond 5G: Reliable Extreme Mobility Management. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM).*

[33] MacDavid, Robert and Cascone, Carmelo and Lin, Pingping and Padmanabhan, Badhrinath and ThakuR, Ajay and Peterson, Larry and Rexford, Jennifer and Sunay, Oguz. 2021. A P4-Based 5G User Plane Function. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR).*

[34] Foivos Michelinakis, Anas Saeed Al-Selwi, Martina Capuzzo, Andrea Zanella, Kashif Mahmood, and Ahmed Elmokashfi. 2021. Dissecting Energy Consumption of NB-IoT Devices Empirically. *IEEE Internet of Things Journal* 8, 2 (2021), 1224–1242.

[35] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. 2018. Hardware-Accelerated Network Control Planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets).*

[36] Barefoot networks. 2018. *NoviWare 400.5 for Barefoot Tofino chipset.* https://noviflow.com/wp-content/uploads/NoviWare-Tofino-Datasheet.pdf

[37] Recep Ozdag. 2012. *Intel Ethernet Switch FM6000 Series - Software Defined Networking.* https://people.ucsc.edu/~warner/Bufs/ethernet-switch-fm6000-sdn-paper.pdf

[38] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51 (2004).

[39] Javan Erfanian Rachid El Hattachi. 2015. *NGMM 5G white paper.* https://www.ngmn.org/wp-content/uploads/NGMN_5G_White_Paper_V1_0.pdf

[40] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. 2020. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *Proceedings of the Symposium on SDN*

Research (SOSR).

[41] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *Proceedings of the ACM SIGCOMM Conference.*

[42] Gábor Soós, Ferenc Nándor Janky, and Pál Varga. 2019. Distinguishing 5G IoT Use-Cases through Analyzing Signaling Traffic Characteristics. In *2019 42nd International Conference on Telecommunications and Signal Processing (TSP).*

[43] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. 2021. D2R: Policy-Compliant Fast Reroute. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR).*

[44] UNSW Sydney. 2021. *IOT TRAFFIC TRACES.* https://iotanalytics.unsw.edu.au/iottraces.html

[45] Netronome systems. 2020. *Agilio CX 2x10GbE SmartNIC.* https://www.netronome.com/media/documents/PB_Agilio_CX_2x10GbE-7-20.pdf

[46] Netronome systems. 2022. *Agilio CX 2x40GbE SmartNIC.* https://colfaxdirect.com/store/pc/viewPrd.asp?idproduct=2871

[47] Pablo B. Viegas, Ariel G. de Castro, Arthur F. Lorenzon, Fábio D. Rossi, and Marcelo C. Luizelli. 2021. The Actual Cost of Programmable SmartNICs: Diving into the Existing Limits. In *Advanced Information Networking and Applications.*

[48] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication.*

[49] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. 2022. SwiSh: Distributed Shared State Abstractions for Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22).*