

Utilizing Hybrid P4 Solutions to Enhance 5G gNB with Data Plane Programmability

Mohsen Memarian*, Andreas Kessler*[†], Karl-Johan Grinnemo*, Sándor Laki[‡], Gergely Pongracz[§], Johan Forsman[¶]

*Computer Science Department, Karlstad University, Karlstad, Sweden

[†]Institute of Applied Computer Science, Deggendorf Institute of Technology, Deggendorf, Germany

[‡]Faculty of Informatics, ELTE Eötvös Loránd University, Budapest, Hungary

[§]Ericsson Research, Budapest, Hungary

[¶]TietoEvry, TBD, Sweden

Email: *mohsen.memarian@kau.se, andreas.kessler@kau.se, karlgrin@kau.se [†]andreas.kessler@th-deg.de,

[‡]lakis@inf.elte.hu, [§]gergely.pongracz@ericsson.com, [¶]johan.forsman@tietoevry.com

Abstract—The traditional method of data plane programming involves deploying a single P4 program to a single target. However, different targets have varying capabilities, functionalities, and support for various programming languages beyond P4. Therefore, disaggregating a single data plane program into multiple subprograms that run on different targets can allow us to leverage the strengths of each target, which becomes particularly important in the context of 5G, where some data plane processing functions, such as buffering and retransmission for RLC processing, cannot be effectively expressed in P4. This paper delves into the decomposition of a 5G gNB across a P4-programmable SmartNIC and an x86 server using DPDK-based processing, thus harnessing the strengths of each target. Our evaluation revealed that offloading certain processing to an x86 server can improve throughput by up to 50%, thanks to the scalability of DPDK applications' performance with the number of CPU cores. However, offloading does introduce a slight increase in latency, so the approach should be adjusted based on the specific needs and available resources.

Index Terms—5G gNB, P4, Data Plane, SmartNIC

I. INTRODUCTION

Data plane disaggregation leverages the specific strengths of various packet processing hardware to meet network requirements by distributing functionalities among them. By breaking down packet processing pipelines into separate components, some can be deployed on x86 servers, taking advantage of their abundant resources, while others can be accelerated using SmartNICs, FPGAs, or switch ASICs. Recent advancements in network programmability with the P4 language have enabled flexible, protocol-independent control over packet processing on reprogrammable hardware. P4 programs can be compiled for different hardware targets, including SmartNICs, switch ASICs, and even x86 DPDK code. SmartNICs are especially effective at offloading network functions to and from x86 servers, enhancing efficiency and reducing the processing load.

In 5G/6G networks, disaggregating data plane programs is essential for efficiently managing diverse traffic patterns and optimizing latency, throughput, and resource utilization [1].

In the context of a gNodeB (gNB), which is the 5G Next Generation base station supporting 5G New Radio and routing user data between the User Plane Functions (UPFs) and User Equipment (UEs), certain data plane functions can be executed on P4 programmable devices, such as encapsulating and decapsulating GTP-U packets and mapping QoS flows to Data Radio Bearers (DRBs) in the Service Data Adaptation Protocol (SDAP). However, the P4 language's expressiveness is significantly limited, making it challenging to describe complex packet processing operations such as loops, packet buffering, or cryptography on P4 programmable hardware targets. Consequently, the essential functions of gNB, like buffering and retransmission in the Radio Link Control (RLC) layer and header compression and encryption in the Packet Data Convergence Protocol (PDCP) layer, cannot be effectively implemented using P4. Therefore, the gNB's packet processing pipeline must be disaggregated, with some parts offloaded to P4 programmable devices while others require alternative solutions.

In this paper, we propose a scalable hybrid solution for gNB data plane disaggregation by distributing packet processing functions between a P4-programmable SmartNIC and an x86 server. Our approach leverages the strengths of each hardware target, utilizing the SmartNIC for low-latency packet processing, and the x86 server for handling more complex tasks that exceed the capabilities of P4, such as buffering. Our experimental evaluation revealed that while the SmartNIC is effective for certain tasks, it encounters a bottleneck when handling more complex gNB functions, particularly packet cloning, which led to a 40% reduction in throughput. By offloading some processing tasks to the x86 server, we were able to alleviate this bottleneck, increasing overall throughput by up to 50%, from 8.5 million packets per second (MPPS) on the SmartNIC alone to 12.75 MPPS with full offloading to the x86, due to the scalability of the DPDK performance as more CPU cores are allocated. However, this improvement comes with a trade-off, as offloading increased the average latency by over 135%, from 40 μ s with on-card processing to 94 μ s on the x86 server. These findings underscore that the decision to

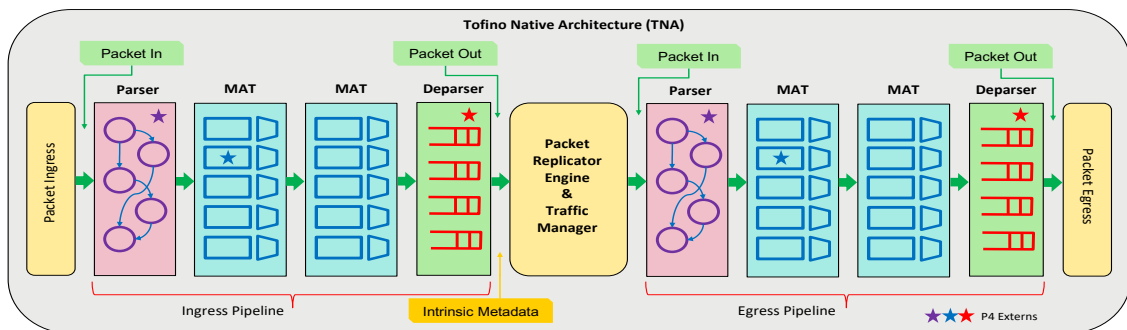


Fig. 1: Tofino P4 pipeline architecture.

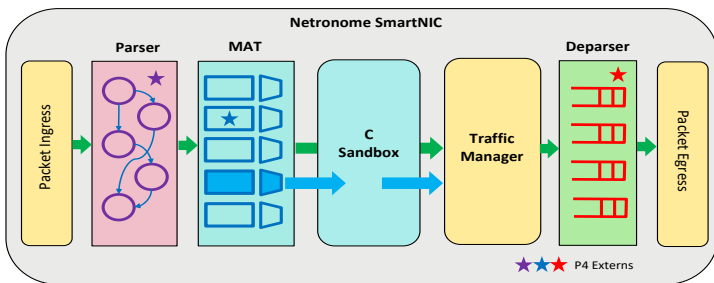


Fig. 2: P4 pipeline with C sandbox on Netronome.

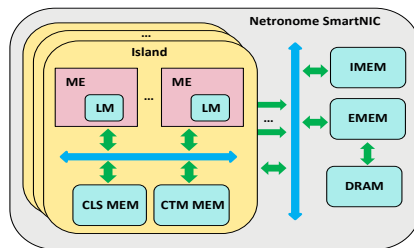


Fig. 3: Netronome memory architecture.

offload processing or keep it on the SmartNIC must be carefully balanced based on specific processing requirements and available hardware resources to achieve optimal performance in 5G gNB implementations.

The remainder of this paper is organized as follows: Section II provides background on P4 programmable data plane devices and 5G gNB air interface protocol stack. Section III reviews related work and outlines our contributions. Section IV details the design and implementation of the hybrid gNB architecture. Section V presents the experimental setup, methodology, and performance analysis. Finally, Section VI concludes the paper and suggests directions for future research.

II. BACKGROUND

The technologies relevant to this paper are discussed briefly in this section.

A. The Tofino Programmable Ethernet Switch

In the Tofino architecture, the switch’s operation combines fixed-function and programmable components, each playing a critical role in packet processing. As shown in Figure 1, fixed-function components such as Packet I/O (e.g., MAC), the Packet Replication Engine, and the Traffic Manager handle essential tasks like packet buffering, replication, queuing, and scheduling, which are beyond the scope of P4 programmability. However, there are specific programmable components within these fixed architectures that P4 programmers can control. These include the ingress and egress pipelines, consisting of the Parser, the Match-Action Table (MAT), and the Deparser. Tofino-specific externs are customizable features that allow users to extend switch’s capabilities beyond standard P4 constructs and can be utilized within the parser,

deparser, or MATs. They allow for advanced functions such as specialized hashing or complex stateful operations. Tofino-specific intrinsic metadata refers to predefined data fields automatically generated within the switch to convey critical status and control information between processing stages, ensuring seamless communication between programmable and fixed-function components. The combination of programmable and fixed-function processing, intrinsic metadata, and externs enables P4 programmers to design and implement efficient packet processing algorithms within the Tofino switch.

B. The Netronome SmartNIC

The P4 data path in the Netronome SmartNIC architecture, illustrated in Figure 2, consists of key components: the Parser, the MAT, and the Deparser. The P4 language enables hardware-agnostic programming of network devices, but specific device customizations may be needed for tasks such as stateful packet filtering, statistics collection, and QoS implementation. Netronome’s toolchain allows for customizations and integration of C-based custom applications, known as C Sandboxes, within the P4-defined data path. The toolchain maps the P4 reference architecture and custom C functions to the dedicated Micro Engines (MEs) in the SmartNIC, where each ME operates independently to avoid race conditions. As depicted in Figure 3, MEs are organized into islands, with each island sharing local memory resources. Due to differences in read and write latencies, developers must select the appropriate memory region for optimal performance. These memory regions include the ME’s Local MEMORY (LMEM) with the lowest latency, Cluster Local Scratch (CLS), Cluster Target Memory (CTM), Internal MEMORY (IMEM), and External MEMORY (EMEM) with progressively higher latencies. Each

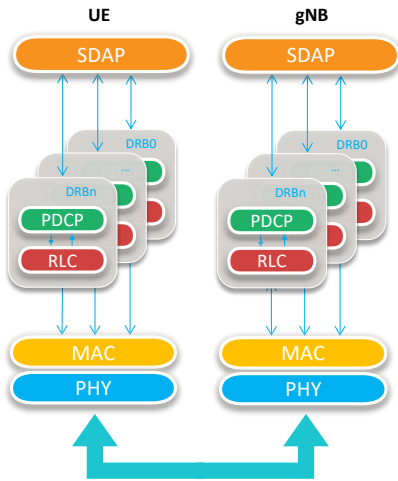


Fig. 4: The 5G NR user plane air interface protocol stack.

region is suitable for different tasks based on its capacity and access time, with EMEM offering the largest but slowest storage, often backed by external Dynamic Random Access Memory (DRAM).

C. The 5G gNB Air Interface Protocol Stack

The 5G User Plane Air Interface Protocol Stack comprises several layers, including SDAP, PDCP, RLC, MAC, and PHY. The SDAP layer receives user downlink data from the UPF and maps QoS flows from PDU sessions to DRBs. DRBs carry user data and ensure consistent packet treatment on the radio interface. The UE monitors the equivalent downlink mapping and applies it to the uplink. Each DRB involves PDCP and RLC entities at both the transmitter and receiver sides, handling uplink and downlink traffic. PDCP manages in-order packet delivery, header compression, and security functions such as ciphering and integrity protection. The RLC layer processes data through three modes: Transparent Mode (TM), Unacknowledged Mode (UM), and Acknowledged Mode (AM). TM passes data without added headers or segmentation/reassembly, which is suitable for broadcasting system information. UM handles segmentation and reassembly without error correction feedback, ideal for low-latency services. AM ensures reliable data transfer through error correction via ACK/NACK feedback and retransmission buffering. The MAC layer manages the scheduling and multiplexing of data streams, while the PHY layer is responsible for data's physical transmission and reception over the air interface.

III. RELATED WORK

Vörös *et al.* [2] suggested a hybrid gNB that combines Tofino with DPDK-based external services to manage complex functions such as retransmission. Singh *et al.* [3] introduced hybrid P4 pipelines for UPF and gNB, leveraging the capabilities of Tofino and x86 to improve throughput and latency, with Tofino handling most packet processing and DPDK managing unsupported functions. However, they do not leverage the capabilities of a SmartNIC, which we think will be more

common in modern platforms. A SmartNIC can be up to 15x cheaper than a Tofino [4], [5]. Although SmartNICs have lower individual performance, we can simultaneously scale the performance using multiple devices.

Several previous approaches have focused on the User Plane Function (UPF) and its acceleration with P4. MacDavid [6] developed a solution on Tofino, resulting in a high-performance UPF that optimizes throughput and latency. However, Tofino handles all processing, including GTP handling. To relieve the load on Tofino's limited resources, x86 can manage inactive users' traffic. Bose *et al.* [7] investigated programmable data planes for high-performance UPF, creating prototypes ranging from software-only DPDK-based versions to those offloading functions to programmable hardware. However, the UPF has a simple data plane that can be easily implemented in programmable hardware alone, and they do not consider this approach.

IV. DESIGN AND IMPLEMENTATION

Recent SmartNICs have programmable features that allow packet processing on the card. However, due to their limited resources and hardware processing capacities, resource-intensive functions may be better performed on the x86, which generally has more resources such as memory and CPU. The decision of whether packets should be processed on the SmartNIC or the x86 can be implemented by reading a packet marker on the SmartNIC. We can use flow specifications in a MAT to identify the corresponding DRB and determine which DRB entity should handle the processing. DRB is a logical channel that carries user plane data between the gNB and the User Equipment (UE).

In terms of modularity, each DRB may include entities on both the SmartNIC and the x86. The connection between DRB entities is facilitated through Virtual Interfaces (VFs) provided by the SmartNIC. The SmartNIC we use in this study can support up to 64 VFs created from Physical Interfaces (PFs) via Single Root I/O Virtualization (SR-IOV), which share the resources of the PF while remaining isolated. When these VFs are assigned to DPDK applications on the x86, the applications can directly access the PCI device, bypassing the host kernel and ensuring low-latency packet forwarding.

In terms of process management, SmartNIC supports parallel packet processing, while DPDK APIs facilitate managing multithreaded applications on x86. In a multithreaded DPDK environment, components like hash tables and ring buffers can be configured to ensure safe simultaneous access. For example, DPDK allows ring buffers to be configured as multi-producer/multi-consumer, single-producer/single-consumer, or a combination of both. Also, instead of a single large DPDK app, processing can be split into smaller applications, allowing more flexibility in distributing CPU cores between various processes. The sub-apps can communicate with each other through ring buffers.

Figure 5 depicts the packet processing workflow. Upon the arrival of a packet at the SmartNIC, the location for processing is determined by reading a packet marker. Both

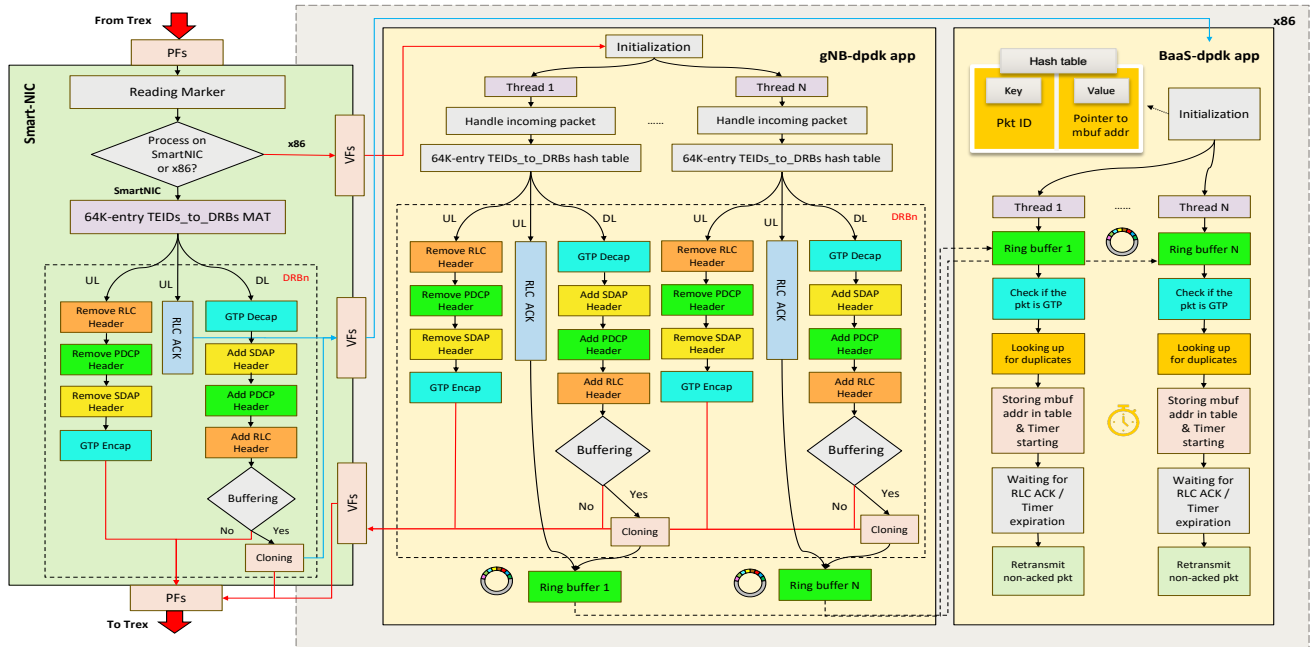


Fig. 5: Diagram of packet handling and offloading from SmartNIC to x86.

the SmartNIC and the host follow a specific packet processing path: A 64 K-entry MAT/hash table identifies the corresponding DRB entity based on flow attributes, such as the GTP Tunnel Endpoint Identifier (TEID). In the downlink path, GTP decapsulation occurs first, followed by adding SDAP, PDCP, and RLC headers. If buffering is enabled, the gNB app clones the packet and enqueues the cloned packet's memory buffer (mbuf) address to a ring buffer. On the receiving end, the Buffering as a Service (BaaS) app as a secondary DPDK app dequeues the packet's mbuf address from the same ring buffer. The BaaS app includes a hash table that keeps track of the memory addresses of packet copies until an acknowledgment is received or the timer expires, triggering retransmission to the UE. In the uplink path, packets are categorized as either RLC acknowledgment or RLC data. RLC acknowledgment packets are directed to the buffering service app, while RLC data packets undergo RLC, PDCP, and SDAP header removal before GTP encapsulation and forwarding to the UPF.

V. EXPERIMENTAL EVALUATION

Our evaluation seeks to address the following questions:

- 1) *Throughput performance*: What is the maximum throughput achievable using a SmartNIC with a simple L2 forwarding program? How much throughput can the SmartNIC inject into the x86 DPDK user space? How does throughput change as additional gNB operations are added on SmartNIC or the x86 DPDK app? What is the impact on throughput when gNB packet processing is split between the SmartNIC and the x86 host? How does performing cloning on a SmartNIC compare to doing it on an x86 host in terms of throughput and resource consumption? Lastly, how does storing packets on an x86 host affect both scenarios?

- 2) *Latency performance*: What is the latency impact of offloading packet processing to an x86 host compared to performing it on the SmartNIC itself? How does the latency change as each gNB operation is incrementally added to the L2 forwarding program?
- 3) *Scalability*: How does gNB processing scale with the number of CPU cores on an x86 host? What are the memory requirements for scaling the number of UEs on the SmartNIC?

A. Experiment Setup

To evaluate the performance of the gNB program, we set up a testbed with the Device Under Test (DUT) featuring an Intel Xeon Silver 4114 CPU running at 2.20 GHz with 20 cores, 32 Gbytes of DDR4 RAM, Ubuntu 20.04, and DPDK v.23.03. Figure 6 provides an overview of the experiment setup. The DUT is where the gNB program runs, and it's equipped with a Netronome Agilio CX 2x40G SmartNIC with 60 processing cores called micro-engines, each operating at 800 MHz, featuring eight hardware threads per core, and CRC acceleration optimized for packet parallel processing. The Netronome NFP Board Support Package (BSP) was released on June 29, 2018, and the Software Development Kit (SDK) version 6.1.0.0 is installed on the DUT. The DUT is directly connected via fiber to a traffic generator.

We are using an x86 server as a traffic generator running Trex v3.04 to produce downlink traffic at a line rate of 40 Gbps, with a packet size of 128 bytes, and varying destination IP addresses to mimic 64 k UEs. A script using the Trex and Scapy Python APIs manages the packet generation process. Trex measures throughput by sending these packets to the DUT, monitoring the rate at which they are processed and

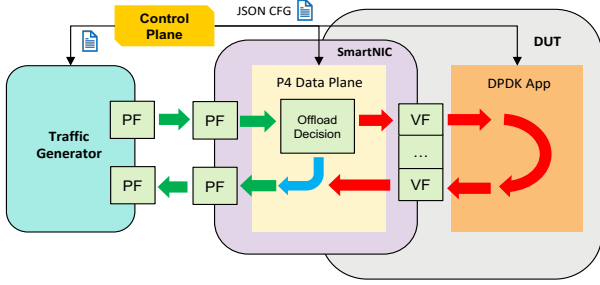


Fig. 6: Experiment setup for performance evaluation.

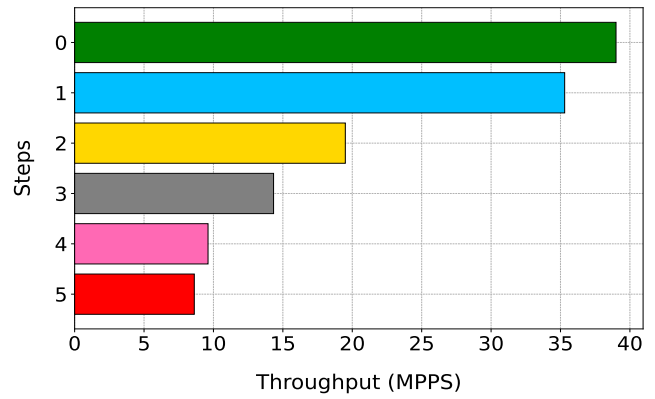


Fig. 7: Throughput comparison of the SmartNIC `gNB.p4` across various processing steps.

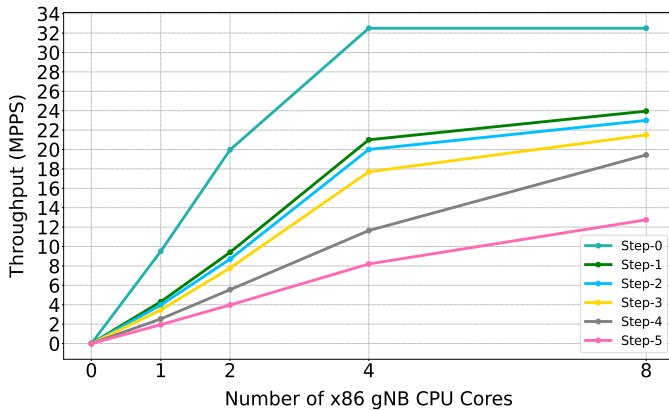


Fig. 8: Throughput comparison of the SmartNIC `p4wire.p4` + `x86 gNB.c` across various processing steps.

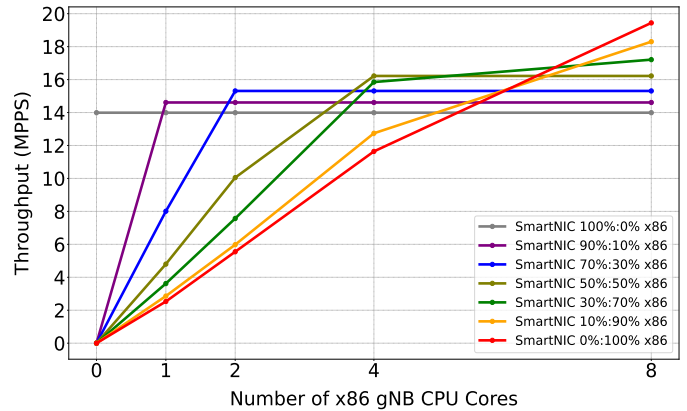


Fig. 9: Throughput vs. number of x86 CPU cores for different offloading ratios between SmartNIC `gNB.p4` and `x86 gNB.c`.

returned without any drops, with a confidence level of approximately $\pm 0.5\%$. Additionally, Trex measures latency by adding a 2-byte header to each packet, allowing precise tracking with microsecond accuracy. The packets that exit Trex, enter the SmartNIC P4 data plane, and, based on offloading decisions, either return to Trex after going through SmartNIC P4 data plane processing or go to the x86 user space DPDK application for further processing and then return to the SmartNIC and finally back to Trex. We use Trex’s High Dynamic Range Histogram (HDRH) method for latency measurement. While the per-packet latency timestamp provides microsecond resolution, similar to the ordinary method of measuring latency with Trex, the HDRH method offers greater accuracy by capturing detailed latency distribution data, with error lines shown on the latency chart to indicate the minimum and maximum observed latencies.

B. Baselines

In the evaluation section, we will compare our `gNB` implementation against two baseline setups:

1) *L2FWD.p4*: This is a simple P4 program developed for the SmartNIC. It receives packets from the PFs of the SmartNIC, swaps the source and destination MAC addresses, and then returns the packets to the PFs.

2) *P4Wire.p4* and *L2FWD.c*: This setup consists of a P4Wire program on the SmartNIC and an L2 forwarding DPDK application on the x86 host. The P4Wire program receives packets from the PFs, uses a MAT to decide the egress VF based on the ingress PF, and then forwards the packets to the designated VF. The DPDK application receives packets from the VFs, swaps the source and destination MAC addresses, and then sends the packets back to the VFs. Finally, the P4Wire program returns the packets to the PFs.

C. Throughput performance of `gNB` program

First, we address the question of what is the maximum throughput achievable using a SmartNIC with a basic L2 forwarding program and how throughput changes as additional `gNB` operations are added to the SmartNIC. In Figure 7, we can observe the impact of incorporating more complex operations into the `gNB` pipeline on the processing power of the SmartNIC. We divide the `gNB` processing pipeline into distinct steps, each adding incremental operations to the SmartNIC, to evaluate their impact on throughput. Starting at Step 0, basic L2 forwarding with MAC Swapping establishes a baseline throughput of approximately 39 MPPS. With the addition of GTP decapsulation in Step 1, the throughput slightly decreases to around 35 MPPS. Moving to Step 2, appending and popu-

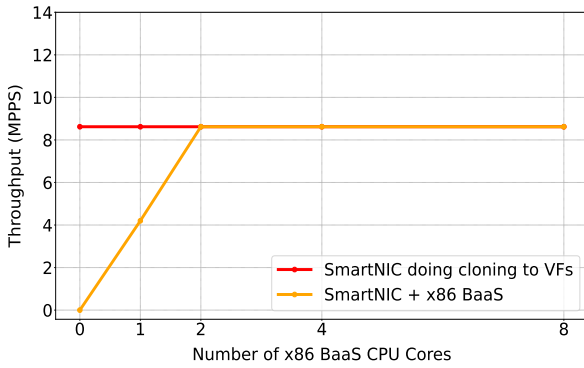


Fig. 10: Throughput of the x86 BaaS application when connected to SmartNIC `gNB.p4`.

lating the SDAP, PDCP, and RLC headers leads to a further reduction in throughput to about 20 MPPS. Step 3 introduces a 64 K-entry MAT, dropping the throughput to approximately 14.33 MPPS. In Step 4, cloning is performed on the SmartNIC, and the cloned packets are dropped immediately, the overall throughput decreases to roughly 9.60 MPPS. In Step 5, the cloned packets are sent to the x86 for buffering, as is necessary, the throughput reduces further to about 8.62 MPPS. Together, these two steps result in an overall throughput reduction of approximately 40%.

Second, we answer the question of how much throughput can the SmartNIC inject into the x86 DPDK user space, how throughput changes as additional gNB operations are added to the x86 DPDK application, and how performing cloning on a SmartNIC compared to doing it on an x86 host in terms of throughput and resource consumption. Figure 8 demonstrates how adding more complex operations to the gNB pipeline impacts the x86 processing power. We implemented a P4Wire program on the SmartNIC that forwards packets from PFs to VFs, reaching the x86 DPDK application directly without further on-card processing. In Step 0, we measure the maximum throughput the SmartNIC can inject into the x86, where the DPDK app receives and drops packets, achieving a rate of 32.5 MPPS. In Step 1, basic L2 forwarding with MAC swapping reduces the throughput to around 23.95 MPPS. Step 2 introduces GTP decapsulation, slightly lowering the rate to 23 MPPS. In Step 3, adding and filling SDAP, PDCP, and RLC headers decreases throughput further to 21.5 MPPS. Step 4, which involves a 64K-entry MAT, drops it to 19.44 MPPS. Finally, in Step 5, cloning packets on the x86 and enqueueing them into ring buffers lead to a 35% reduction, bringing the throughput down to approximately 12.75 MPPS. Up to four CPU cores, the throughput increases at a steeper slope, but beyond that, the slope begins to flatten. This likely happens because, initially, the SmartNIC isn't a bottleneck, so adding more CPU cores significantly boosts throughput. However, after a certain point, the SmartNIC can't inject packets fast enough, so adding more CPU cores gives smaller gains. It's also important to note that, before the cloning step, the x86 gNB program experiences a smaller percentage drop

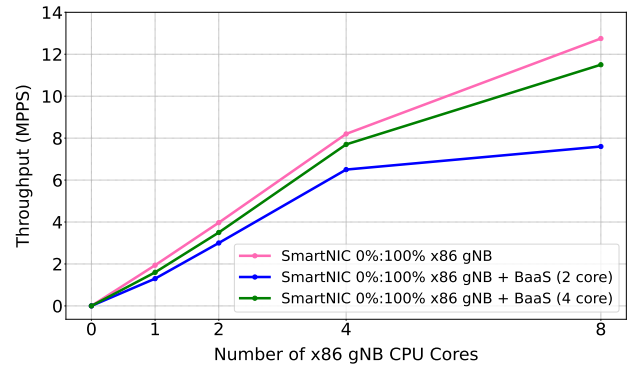


Fig. 11: Throughput comparison of the x86 BaaS application when connected to SmartNIC `p4wire.p4 + x86.gNB.c`.

in throughput compared to the SmartNIC gNB program, as illustrated in Figure 7. When analyzing the impact of packet cloning, Figure 8 (Step 5), shows that throughput on the x86 can decrease by up to 35% during cloning, while a similar operation on the SmartNIC results in a drop of 40%, as depicted in Figure 7 (Step 5). In the case of the SmartNIC, we consider the effect of directing cloned packets to VFs, while in the x86 case, we consider the effect of enqueueing cloned packets to ring buffers. In both scenarios, the goal is for the cloned packets to reach the BaaS application.

Third, we examine what is the impact on throughput when gNB packet processing is split between the SmartNIC and the x86. Figure 9 shows the overall throughput of the DUT based on the number of x86 CPU cores used. The graph indicates the effect of moving the gNB packet processing from the SmartNIC to the x86. We divide the packets into two groups based on a marker in the IPv4 Identification field, specifying their gNB processing target. The first group is processed on the SmartNIC, and the second on the x86, with both undergoing GTP, SDAP, PDCP, RLC, and MAT processing. Initially, 100% of the packets are processed on the SmartNIC. Gradually, we adjust the proportions, reducing the first group and increasing the second, until 100% of the packets are processed on the x86, with the SmartNIC solely forwarding packets to the x86 for gNB processing. The SmartNIC achieves a maximum throughput of 14 MPPS independently, as indicated in Figure 7 (Step 3). As the amount of offloaded traffic increases, more x86 CPU cores are required to handle the traffic. With 100% offloading and 8 CPU cores, the throughput reaches 19.5 MPPS, as shown in Figure 8 (Step 4). Offloading results in a gradual increase in overall throughput as x86 cores assist in processing a portion of the traffic. The overall throughput consistently falls between the extreme scenarios of all gNB processing on the SmartNIC and all gNB processing on the x86.

D. Throughput performance of BaaS program

In this sub-section, we investigate how storing packets on an x86 affects SmartNIC and x86 while they are doing the cloning. we evaluate the throughput performance of the BaaS application under two different scenarios. In the first

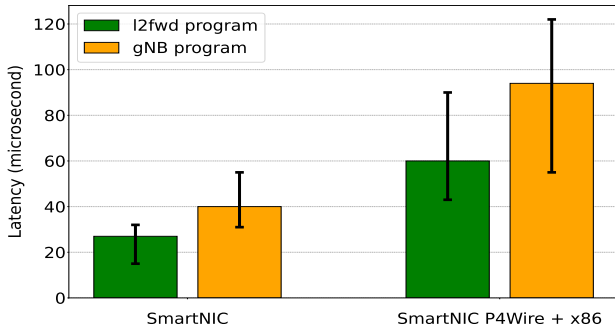


Fig. 12: Latency comparison at maximum non-drop rate: SmartNIC vs. SmartNIC P4Wire + x86.

scenario, the BaaS is directly connected to the SmartNIC via VFs. In this setup, the SmartNIC’s gNB program clones the packets and sends them directly to the BaaS for buffering. In the second scenario, the SmartNIC’s P4Wire program sends packets to the x86’s gNB DPDK application, which handles the packet cloning. The cloned packets are then enqueued into ring buffers, from which the BaaS application subsequently dequeues them for buffering.

Figure 10 illustrates the throughput of the x86 BaaS application in the first scenario, where it is directly connected to the SmartNIC. As previously discussed, the maximum throughput achievable from the SmartNIC, when it is engaged in cloning and forwarding packets to the VFs, is 8.62 MPPS. To handle this amount of packets, as indicated by the orange line, the BaaS application requires at least 2 CPU cores for buffering.

Figure 11 shows the second scenario, where the x86 BaaS operates as a secondary application connected to the primary x86 gNB application via ring buffers. As mentioned before, the maximum achievable throughput when the x86 gNB app clones and enqueues packets to the ring buffer is 12.75 MPPS. Achieving optimal performance requires balancing the CPU core allocation between the x86 gNB and BaaS applications. For example, assigning 2 CPU cores to BaaS while allocating 8 cores to the gNB results in a maximum BaaS throughput of only 7.6 MPPS, indicating a bottleneck where BaaS cannot efficiently process all the cloned packets from gNB. By increasing the assigned CPU cores for BaaS to four CPU cores, it can handle the majority of incoming packets, thereby improving overall performance.

E. Latency performance

In this section, we compare the latency of our gNB implementation with the baseline latency under two scenarios: on-card processing and offloading to the x86 host. Additionally, we analyze how each gNB operation contributes to the overall latency. We investigate the latency impact of offloading packet processing to an x86 host compared to performing it on the SmartNIC itself. By understanding the impact of different operations on latency, we can make informed decisions about whether to retain operations on the SmartNIC or offload them to the x86 host.

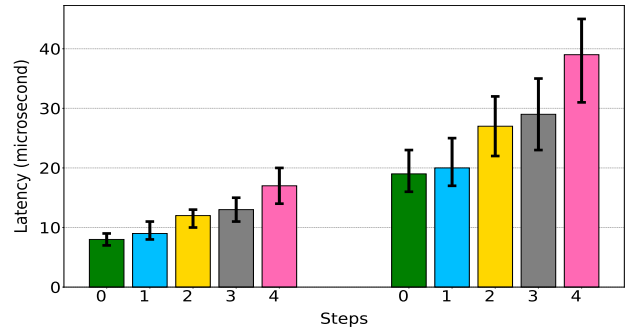


Fig. 13: Latency comparison at 10% of Line rate: SmartNIC vs. SmartNIC P4Wire + x86.

1) *On-Card Processing vs. Offload Alternatives:* We are assessing the latency impact of transferring processing from a SmartNIC to an x86 host. This involves measuring the effect on latency when offloading tasks, despite the x86 host having abundant resources. To begin, we create an L2 forwarding program (L2FWD) on both the SmartNIC and the x86 host to establish the baseline latency for these paths at the maximum non-drop rate:

- 1) $Trex \rightarrow PF \rightarrow SmartNIC\ L2FWD.p4 \rightarrow PF \rightarrow Trex$
- 2) $Trex \rightarrow PF \rightarrow SmartNIC\ P4Wire.p4 \rightarrow VF \rightarrow DPDK\ L2FWD.c \rightarrow VF \rightarrow SmartNIC\ P4Wire.p4 \rightarrow PF \rightarrow Trex$

To compare latency, we run the gNB program instead of the L2FWD program, once on the SmartNIC and once on the x86 host. This helps us see the additional latency introduced by our gNB implementation compared to the baseline latency.

The comparison in Figure 12 illustrates the latency between the baseline L2 forwarding and the gNB program, both implemented on the SmartNIC and as an x86 DPDK application connected to the SmartNIC P4Wire program. In the baseline scenario, packets experience an average latency of $27\ \mu s$ with on-card processing. However, this latency increases by 122% to $60\ \mu s$ when the same L2 forwarding is performed by the x86 DPDK application. When we replace the L2 forwarding with our gNB implementation, the latency increases by 135%, from $40\ \mu s$ for on-card processing to $94\ \mu s$ in the x86 offloaded case. Additionally, the error lines indicate that latency variability is higher in the offloading case compared to the on-card processing scenario.

2) *Impact of gNB Operation on Baseline Latency:* In the previous sub-section, we analyzed the overall latency of the gNB program. In this section, our goal is to show how each additional gNB operation impacts the baseline latency. To ensure a fair comparison, we carried out the measurements at 10% of the line rate. This approach is necessary because adding each gNB operation reduces the throughput, which could affect the overall latency observed by the TRex traffic generator.

Figure 13 demonstrates how adding more complex operations to the gNB pipeline affects baseline latency in two scenarios: on-card processing and processing packets on an

x86 DPDK application connected to the SmartNIC P4Wire. In the on-card processing scenario, the baseline L2 forwarding latency starts at $8\mu\text{s}$ in Step 0. Adding GTP decapsulation in Step 1 slightly increases latency to $9\mu\text{s}$. Step 2, which involves SDAP, PDCP, and RLC header handling, increases latency to $12\mu\text{s}$. Adding a MAT in Step 3 results in a minor increase to $13\mu\text{s}$, and in Step 4, cloning packets, with cloned packets sent to the BaaS via VFs while the original packets return to TRex, raises the latency to $17\mu\text{s}$. In the SmartNIC P4Wire and x86 DPDK application scenario, the same sequence of steps is applied. While the behavior across the steps mirrors the on-card processing scenario, with each step incrementally increasing latency, the average latency values are higher in comparison. Additionally, the variability in latency is more pronounced in x86 scenario.

F. Scalability

In this section, we investigate the challenges our implementation faces regarding performance scalability. We analyzed how increasing CPU cores affects DPDK applications performance and how memory constraints influence the ability to support a higher number of UEs.

1) *Scaling gNB/BaaS Apps Throughput with CPU Cores on x86*: We noticed in Section V-C (cf. Figures 8, 9, and 10) that the throughput increases as the number of CPU cores increases, as long as the SmartNIC or the gNB DPDK app does not become a bottleneck. This increase is due to the multi-threaded architecture of DPDK applications, where threads work independently, and components like hash tables and ring buffers (cf. Figure 5) are designed to allow safe simultaneous access. It's important to ensure that the number of CPU cores is always less than or equal to the number of VF interfaces. If we intend to increase the number of CPU cores, we must also increase the number of VFs to avoid thread contention during packet pulling from interfaces.

2) *Memory Requirements for Scaling UEs*: In our setup, each entry in the MAT corresponds to a UE, with the destination IP address as the key and the DRB number as the value. Therefore, the number of entries in the MAT equals the number of UEs. The MAT utilizes the EMEM cache memory within the Netronome SmartNIC, which is limited to 6 Mbytes. Table I outlines the behavior of MAT memory consumption in relation to the number of entries. As the MAT length increases, the number of supported UEs becomes limited. Increasing the MAT size from 1k to 10k entries raises EMEM cache consumption from 9.08% to 15.03%, a 1.6x increase. Expanding the MAT size further to 64k entries increases cache consumption to 40.97%, a 2.7x increase. EMEM cache variations are also reflected in the overall EMEM consumption. Netronome's P4 design restricts each table to a maximum of 64K entries but supports up to 256 tables. By using a series of tables, we can fully utilize the EMEM cache. To accommodate more UEs, part of the MAT could be offloaded to the x86 architecture, where more abundant memory resources are available.

TABLE I: Memory usage in SmartNIC

Memory Type	Size	Number of MAT entries		
		1K (%)	10K (%)	64K (%)
Local Memory (LM)	256K	43.86	43.86	43.86
Cluster Local Scratch (CLS)	64K	25.13	25.13	25.13
Cluster Target Memory (CTM)	256K	45.15	45.15	45.15
Internal Memory Unit (IMU)	4M	18.67	18.67	18.67
External Memory Unit (EMU)	2G	60.98	61	61.11
EMU Cache	6M	9.08	15.03	40.97

VI. CONCLUSION AND FUTURE WORK

We introduced a scalable hybrid approach to improve 5G gNB data plane programmability by splitting packet processing tasks between a P4-programmable SmartNIC and an x86 server, utilizing the SmartNIC for low-latency processing and the x86 server for complex tasks like buffering. Our experiments showed that while the SmartNIC handles simple tasks efficiently, it bottlenecks on complex functions such as packet cloning, and offloading to the x86 server increases throughput with a slight latency trade-off. The choice to offload depends on specific processing needs and hardware resources. For future work, we plan to explore the use of multiple SmartNICs to further scale the aggregate throughput, capitalizing on the modular design we have implemented.

ACKNOWLEDGEMENT

This work was carried out within the Data-driven Latency-sensitive Mobile Services for a Digitalized Society (DRIVE) project, partly funded by the Knowledge Foundation of Sweden, with additional funding from the European Commission through the HORIZON 6G SNS JU DESIRE6G (G.A. 101096466) and the Bavarian State Ministry of Education, Science, and Art through the High-Tech Agenda (HTA).

REFERENCES

- [1] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023.
- [2] P. Vörös, D. Kis, P. Hudoba, G. Pongrácz, and S. Laki, "Towards an in-network gpu-accelerated packet processing framework," in *2022 IEEE 2nd Conference on Information Technology and Data Science (CITDS)*, 2022, pp. 308–313.
- [3] 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*, vol. 22, no. 12, pp. 6921–6937, 2023.
- [4] Colfax International, "Netronome agilio cx dual-port 40 gigabit ethernet smartnic (product page)," <https://colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3018&idcategory=0>, accessed: 2024-07-25.
- [5] —, "Edgecore wedge 100bf-32x 32-port 100gbe bare metal switch (product page)," <https://colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3485&idcategory=0>, accessed: 2024-07-25.
- [6] 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)*, ser. SOSR '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 162–168.
- [7] 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*, ser. APNet '21. New York, NY, USA: Association for Computing Machinery, 2022, p. 57–64.