Ralf Kundel, Lisa Wernet, Fridolin Siegmund, Leonhard Nobach, Hans-Joerg Kolbe, Tobias Meuser. *Massive QoS-aware Packet Queueing and Traffic Shaping at the Access Edge using FPGAs.*
To appear in the Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS), 2025.

Massive QoS-aware Packet Queueing and Traffic Shaping at the Access Edge using FPGAs

Ralf Kundel*, Lisa Wernet*, Fridolin Siegmund*, Leonhard Nobach†, Hans-Joerg Kolbe‡, Tobias Meuser*
*Multimedia Communications Lab, Technische University of Darmstadt, Germany
{*ralf.kundel, lisa.wernet, fridolin.siegmund, tobias.meuser*}*@kom.tu-darmstadt.de*
† Deutsche Telekom Technik GmbH, Germany
leonhard.nobach@telekom.de
‡ Radisys corporation, USA
hans-joerg.kolbe@radisys.com

***Abstract*—Large-scale packet queueing and scheduling is the basis for today's Quality of Service (QoS) in computer access networks, especially to achieve guaranteed high throughput and low latency. The throughput of a single network function, implementing the QoS functionality for a large number of customers, is in the range of hundreds of gigabits or even more. Therefore, a good performance of the underlying hardware is mandatory. While highly-performant fixed-function ASICs offer sufficient functionality for most data center use cases, as of today, they cannot support all functionality required for access networks, including QoS-aware packet queuing.**

In this paper, we first analyze mobile and residential Internet access requirements from an Internet service provider perspective, focusing on the QoS-aware packet queueing needs. Considering this analysis, we present a universal and generic FPGA design for high-performance packet queueing and scheduling. Our evaluation results show that FPGAs can be used to implement a deterministic QoS packet queueing system with high performance. This concept can extend today's programmable networking ASICs with the desired functionality as an offloaded "sidecar".

***Index Terms*—FPGA, Network Function Offloading, QoS, Queueing, Traffic Shaping, AQM**

I. Introduction

Computer Networks are the basis for almost every digital application and have become integral to daily life. As a consequence, the underlying networking technology has been subject to many improvements over the last decades.

The most important Quality of Service (QoS) metrics in computer networks are throughput, latency and jtter. To fulfill them deterministically, network functions within the data plane, responsible for forwarding packets within the network, are typically built with Application Specific Integrated Circuits (ASICs). The control plane, a co-located software implementation, manages these ASICs.¡

With the introduction of Software Defined Networking (SDN) concepts, *e.g.*, the OpenFlow protocol [1], the Interface between ASIC and control plane was opened to program the ASIC's flow tables within the capabilities of the fixed chip pipeline.

However, bringing innovation into networking switches with fixed-function ASICs and SDN is still very expensive, and development cycle times are high due to the nature of chip manufacturing. Especially for niche applications requiring only a few thousand chips, it is almost impossible to facilitate this with the latest silicon manufacturing technologies fulfilling current and future networking standards, *i.e.*, up to 800 $Gbit/s$ Ethernet.

To enable such niche functionality and to increase the flexibility in data centers in general, programmable packet processing chips were introduced in the last decade [2]. These chips still have an ASIC-like internal architecture; however, they provide a pipeline with generic functional blocks tailored for packet processing. A compiler can connect these blocks with some limitations and these programmable network switches can achieve a similar performance as their non-programmable predecessors. With the domain-specific and open-source P4 programming language [3], programmable ASICs can fulfill most requirements of today's networks. Thus, the most common data center functionalities and many niche applications can be implemented with the same reconfigurable networking chip, reducing innovation costs and time.

However, the supported functionality of these programmable and reconfigurable ASICs, tailored for packet header processing and high throughput in data centers, is still limited, and not all networking applications can be implemented. One such application is massive packet queueing and scheduling. For Internet service providers, queueing is of tremendous importance at the termination node for mobile and residential customers, *e.g.*, in 5G deployments, where advanced QoS mechanisms must be applied in networks while maintaining a high end-to-end performance [4].

For example, each customer's traffic shall be separated and treated independently without being affected by the network usage of other customers. For that, a huge number of queues and advanced scheduling mechanisms must be implemented in hardware, which is not feasible with most existing chips. Existing solutions for massive queueing mainly utilize proprietary, special-purpose chips, and they come along with high prices, limited functionality, and slow innovation cycles because they are designed for niche markets only.

In this work, we address the challenge of replacing tailored chipsets with fully programmable off-the-shelf FPGAs, taking the viewpoint of a Tier 1 Internet access service provider, focusing on FPGAs for the QoS-functionality that available

fully programmable chipsets cannot provide as of today. First, we provide an overview of the essential requirements of QoS processing chips in Section II. Second, in Section III, we present our adaptable FPGA design addressing these requirements. Following this, we present evaluation results showing the viability of FPGAs for QoS applications in access networks in Section IV. Last, we discuss approaches in related work and conclude this work with a summary.

II. Problem Statement

In this section, we identify the requirements on packet queueing systems for Internet service creation based on our own experience as a service provider and from related work:

Number of Queues: Internet service providers typically use different QoS classes for different services, *e.g.*, Voice over IP (VoIP). Each class is mapped to one queue for each connected customer to separate customers and different services of one customer. The number of QoS classes per customer can vary between 1 and 8, depending on the service provider architecture [5]. Further, the number of customers terminated by a single termination node is typically in the range between $10,000$ and $35,000$ customers. Assuming four QoS classes per customer, up to $140,000$ queues are required.

Throughput: We expect up to $100\ Gbit/s$ Ethernet at next generation termination nodes. Thus, this should be supported by the FPGA design as well, ideally at multiple ports simultaneously. Assuming $1314\ bytes$ average packet size [6], this leads to an average packet rate of $\sim 10Mpps$.

Counting: For legal, accounting, and network monitoring reasons it is required to count the number of packets and bytes for each customer or even with a higher granularity, *e.g.*, for a specific service used by a customer. Depending on the concrete requirements, between 1 and 12 counters per customer are required before and after the queueing system. Note that these counters can be implemented either within the queueing system, *i.e.*, an FPGA, or in a preceding/following programmable network switch.

Rate-Limiting: The packet queueing system is responsible for limiting the dequeueing rate of each queue according to its configuration, *e.g.*, $100\ Mbit/s$ for a 5G mobile user. As oversubscription of available resources is quite common in Internet access networks, hierarchical schedulers might be needed in addition to the per-customer limiting. For example, the sending rate should not exceed the maximum throughput of a 5G base station. In case of reaching a hierarchical limitation, flows with higher priorities, *e.g.*, VoIP traffic, shall be prioritized over best-effort traffic of other customers. This scheduling requirement is often referred as *hierarchical QoS*.

Further, in order not to overload the access network components, the rate-limiting should not cause packet bursts, *i.e.*, scheduling a bunch of packets at once and then having an extended wait period than scheduling each packet individually.

Further, in mobile networks a handover can occur from one base station to another. In this scenario, the QoS system must buffer all packets during this handover until the customer is reconnected to the new radio base station. For that, queues must be able to be used as a gate, *i.e.*, the rate of the queue is set temporarily to 0.

Even though the previously mentioned examples focus on mobile access networks, these requirements can be applied similarly to residential access networks, except for handovers.

Active Queue Management (AQM): If the number of arriving bytes is constantly higher than the configured dequeueing rate, packet loss is unavoidable. However, the decision when and how the queueing system is dropping packets can strongly affect the end-to-end throughput and latency of network flows passing through. Active Queue Management (AQM) algorithms are dropping packets in an "intelligent" and network congestion control aware manner to ensure high throughput and comparably low latency at the same time [7], [8]. This is of tremendous importance for Internet service providers to offer high-quality products to their customers. As FPGAs provide the flexibility of implementing such algorithms in the data plane, an appropriate AQM should manage the queue level in the desired design.

Modularity and Reusability: Requirements on QoS systems vary from country to country due to regulatory laws, and between the mobile and residential access scenarios. Even though FPGA bitfiles can be easily exchanged, the implementation should be as reusable as possible. By that, *e.g.*, only the hierarchical scheduler behavior can be adjusted, or the packet counting functionality is parameterized.

III. Design and Implementation

In this section, the design of the FPGA-based QoS system is presented. First, we provide an overview of all components, including the data and control flow. Second, we present substantial parts of the design in detail. All components are created in a modular design, allowing the replacement of only a single component to fulfill the needs of another use case.

A. Overall Design

Figure 1 depicts the main components of the design and their interfaces. Different clock domains and peripheral functionality are not shown. Note that the design contains only vendor-specific IP Cores for Ethernet, PCIe, and DDR4 memory access and could, therefore, be migrated to a platform of another FPGA vendor.

First, the left-hand side $100Gbit/s$ IP Core receives the incoming packet and provides it as a 512-bit AXI4-stream [9].

Second, the `classifier` determines a queue id (`qID`) based on the packet. There are several ways to classify packets, *e.g.*, based on a lookup-table matching one or multiple packet header fields. In this work, we assume a preceding network switch classifying the packets and handing over the queue ID as the first 32 bits of the packet. The `classifier` removes this queue ID before handing over the actual packet to the `rx-handler`.

Packet Storing: As the packets might be queued for a longer time, *e.g.*, up to $100\ ms$, they are stored in external memory, while only the physical memory address and the packet length are stored inside the FPGA. Internal memory, *i.e.*, SRAM-based memory cells, do not provide sufficient capacity to store a large number of packets. For example,

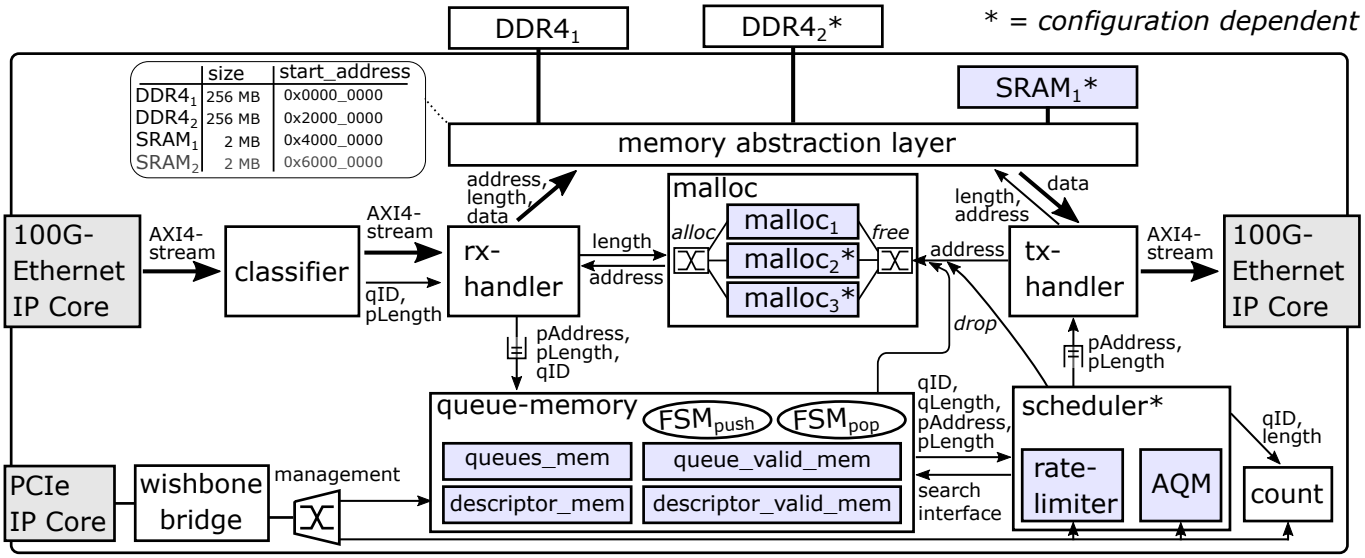

Fig. 1: Modular FPGA design for (H)QoS networking functionality. Packets are received and sent over 100G-Ethernet and stored in one or two DDR4 memory attached to the FPGA as well as in internal SRAM memory (configuration dependent).

an average queueing delay of $10\,ms$ at $100\,Gbit/s$ requires $125\,MB$ buffer memory.

For this, a central memory allocation unit (`malloc`) provides a physical memory address to the `rx-handler`. The incoming data stream is written on this memory address via a memory abstraction layer. This layer allows a simple replacement of memory technologies, *e.g.*, DDR4 with high-bandwidth memory (HBM). After storing the packet data, the `rx-handler` passes the packet metadata information, including the queue ID, to the `queue-memory`.

To avoid waiting situations or even deadlocks between `rx-handler` and `queue-memory`, a FIFO queue is used. This queue allows asynchronous processing in both modules by buffering up to four packet metadata to be enqueued.

The `queue-memory` stores the packet's address, length, and queue ID in internal data structures. Packets are separated in different FIFO queues, identified by the `rx-handler` as an ID. The internal logic of the queue-memory is described later in Section III-C.

Packet Transmission: For transmitting a packet, the `queue-memory` provides any non-empty queue ID and the queue length to the `scheduler`. Based on this information only, the scheduler determines if the top packet in this queue is allowed to be sent or not. While the queue ID is required for rate-limiting, the queue length is required by the Active Queue Management (AQM) module, which will be introduced later. Only if the `scheduler` allows the transmission of the packet, the packet's physical address and its length are deciphered and provided to the `scheduler`, which passes through this information to the `tx-handler`. Avoiding unnecessary early lookups of packet meta-information enhances fast scheduling decisions. For that, the rate limiter always assumes packets with the same size as the Maximum Transmission Unit (MTU) at decision time but the actual packet size when updating its state to prevent approximation errors.

While passing through the meta-information or declining the packet, the `scheduler` instructs the `queue-memory` to continue searching the next non-empty queue. Parallel to this, the `scheduler` triggers an attached counter module with the queue ID and packet length, holding each queue's packet and byte counters. Last, the `tx-handler` requests the original packet from the external memory and sends the packet out on an Ethernet Port via an AXI4-stream interface.

B. Memory

As mentioned before, we utilize a memory abstraction layer to abstract the read and write operations from and to the memory from the underlying hardware. One benefit of this memory abstraction layer is the possibility of simultaneously utilizing multiple external and internal memories in a single address range. This is highly beneficial, as our experiments have shown that a single DDR4 memory supports only up to $60\ Gbit/s$ to $80\ Gbit/s$ simultaneous read and write requests, due to the shared chanel for reading and writing, depending on the FPGA configuration.

Our proposed design utilizes two external DDR4 memories, each having $256\ MB$, and an FPGA-internal SRAM-based block memory of $2\ MB$ size. The DDR4 memories are aligned in $2048\ byte$ blocks and the internal SRAM memory in $512\ byte$ blocks. Each of these three memories has its own memory allocation unit, combined by a round-robin arbiter, providing free addresses to the `rx-handler`. However, physical addresses located in the SRAM memory address space are only used for small packets, *e.g.*, smaller than $512\ byte$. Note that if no memory slots in the internal SRAM memory are available, small packets can still be stored in the external DRAM, but the overall performance will be slightly worse. This has two advantages: First, smaller packets belong more often to a high-priority traffic class [6] and are typically scheduled faster, implying a shorter storage period in the FPGA. Second, short read and write bursts on the external DDR4 memory lower the total bandwidth of the memory due to the physical limitations of DRAM. The three memories are combined in one 32-bit address range, as shown

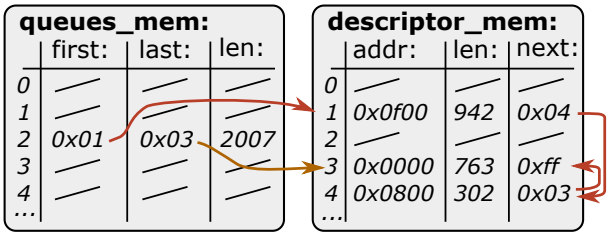

Fig. 2: FPGA-internal data structure for packet queueing.

in the top left of Figure 1. Further memories can be added similarly if needed. By that, they can be accessed in the same way; only the memory allocation must be aware of the heterogeneous memory. To increase the throughput by overcoming long waiting periods, we perform multiple AXI4 read and write requests with different IDs in an overlapping manner. For the `rx-handler`, we use up to 8 shared IDs in parallel; for the `tx-handler`, we use up to 12 IDs in parallel, 4 IDs assigned to each memory.

Through the usage of multiple memories, the order of packets may be changed, as memory B (e.g., SRAM) might have a shorter access time than memory A (e.g., DDR4). However, the order of packets in different customer queues does not pose a problem from a network perspective.

C. Data Structures for Packet Queueing

In this section, the internals of the `queue-memory` are discussed. Note that this module implements a large number of FIFO queues, *e.g.*, $140,000$ as mentioned before.

In general, the queues are implemented as a linked list, consisting of a pointer to the first and last descriptor entry, as shown in Figure 2. Further, each queue data structure holds the current length of the queue in *bytes*. The packet descriptor entries are stored in the `descriptor_mem`. For each new packet, a new entry in the `descriptor_mem` is created. This descriptor contains the packet information, *i.e.*, the address and length of the packet. Further, it contains a pointer to the next descriptor entry in the queue, which is *null* for the last packet in each queue. In the example of Figure 2, queue nr. 2 consists of three packets with a total length of 2007 *bytes*.

When a new packet push request enters the module, a free slot in the `descriptor_mem` is required. This slot is provided by an additional memory allocation unit, called `descriptor_valid_mem`. For each entry a one bit information is stored, indicating if the entry is used or free. Similarly, the memory allocation units for the external memories are implemented.

The last data structure for implementing the queueing behavior is the `queue_valid_mem`. Here, for each queue, a single bit indicates if at least one packet is enqueued in this queue or not.

The enqueueing process is controlled by a Finite State Machine (FSM), which pushes the packets into the corresponding queue and updates the aforementioned memories. If the maximum queue length is reached, the current packet gets dropped (called "taildrop"), and the `malloc` unit is notified to free the memory address of the packet.

Listing 1 depicts the pseudocode of inserting an incoming packet into the previously introduced data structures representing the queueing system. First, in Line 8 to 11, a new descriptor entry for the new packet is created. Following, if the new packet will be the first one in this queue (Line 13), the *first* pointer of this queue is set to this entry (Line 15) and the queue is set to be non-empty (Line 17). Otherwise, the *next* pointer of the old last entry in the linked list will be set to the newly added packet (Line 19). In Line 20, the last pointer of the queue is updated to the new tail entry, and in Line 21 the total queue length is increased by the size of the inserted packet.

This behavior is implemented as a four- or five-state FSM (depending on the clock frequency and the number of queues). In addition to this enqueueing logic, a second FSM is responsible for dequeueing packets in a similar way. To avoid inconsistencies, a lock mechanism between the push and pop FSM is required. This ensures that both FSMs never operate on the same memory addresses simultaneously.

The `queue_valid_mem` is used for scheduling packets, *i.e.*, identifying non-empty queues. While having a data width of 32 bits, in each clock cycle, 32 queues are checked to be non-empty by a single $\neq 0$ comparison. Accessing the *queues_mem* is only required if the `scheduler` decides to send a packet from the respective queue. From our past experience as a Internet service provider, most queues are empty most of the time. Therefore, this fast queue search mechanism improves the overall system performance significantly.

D. Hierarchical On-demand Token Bucket

One of the design goals was the capability to replace a single module for different scenarios, mainly the `scheduler`* module. As part of this work, we implemented three different scheduler designs:

The `no-scheduler` module accepts all packets proposed by the `queue-memory`. It is used for benchmarking the maximum throughput of the system.

```
1  func enqueue(packet p, int qID):
2    //check configured max. queue length
3    if (queue[qID].length > MAX_QUEUE_SIZE):
4      //drop the packet
5      malloc.free(p.address);
6      return;
7    //create descriptor_mem entry
8    descriptor = new descriptor();
9    descriptor.address = p.address;
10   descriptor.length = p.length;
11   descriptor.next = 0xffffffff; //null
12   //update queues_mem
13   if (valid_memory[qID] == 0):
14     //special case: push of first packet
15     queue[qID].first = descriptor;
16     //set queue to be non-empty
17     valid_memory[qID] = 1;
18   else:
19     queue[qID].last.next = descriptor;
20   queue[qID].last = descriptor;
21   queue[qID].length += p.size;
```

Listing 1: Simplified behavior of the enqueueing FSM.

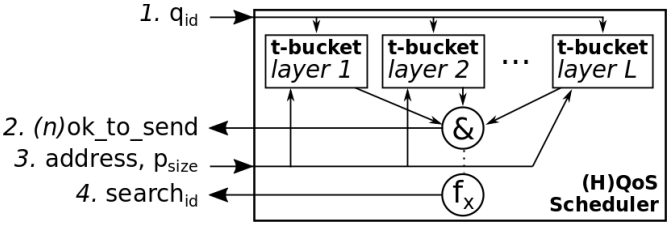

Fig. 3: Simplified hierarchical token bucket scheduler with parallel lookup. 1.-3.: dequeue interface, 4.: search interface.

The `simple-scheduler` enforces rate-limiting for each queue based on a token bucket mechanism. No dependencies between queues are considered. For example, similar schedulers are used in today's 4G/5G networks.

The `hierarchical-scheduler` provides L layers of token bucket rate-limiters to fulfill the needs of residential Internet access creation, as shown in Figure 3. Only if the scheduling of a packet does not pose a problem for all hierarchical layers, the packet will be sent.

The interface between `scheduler` and `queue-memory`, depicted in Figure 3, can be divided into four phases to ensure fast scheduling decisions: 1) The queue ID is provided to the `scheduler`, deciding if the packet should be sent. In addition, an Active Queue Management (AQM) algorithm can be consulted. 2) The `scheduler` notifies the `queue-memory` if the next packet of the indicated queue should be sent or not. 3) If the packet should be sent, the `queue-memory` dequeues the packet metadata, including its length and physical memory address. This information is then forwarded to the `tx-handler`. 4) Last, the `scheduler` informs the `queue-memory` to continue searching the next non-empty queue from a given starting queue ID. This is useful if a certain queue range, *e.g.*, $0-4095$, is not allowed to send a packet due to hierarchical restrictions, and the search should continue with queue ID 4096.

Frequently updating the state of many token buckets is impossible. Thus, a token bucket value $bucket_{i,old}$ and a timestamp $t_{i,last}$ of the update are stored for each queue. In addition, a rate value $rate_i$ stores the number of tokens provided per time for each queue. Only if a packet may dequeued, the token value is updated as shown in Equation (1):

$$bucket_{i,new} = bucket_{i,old} + (t_{now} - t_{i,last}) \cdot rate_i \quad (1)$$

E. Active Queue Management

One advantage of FPGAs is the customization of logic designs for specific purposes. As the primary use-case of this system is Internet service creation, we could integrate a well-suited Active Queue Management (AQM) algorithm into the design as part of the `simple-scheduler`. We chose one of the most prominent algorithms, CoDel [10], to be implemented.

This algorithm requires the queueing delay of the packet as an input. One approach would be to store a timestamp for each packet at enqueue time in the `queue-memory` requiring significantly more memory resources. Thus, we calculate the current queueing delay from the fill level of the queue and the rate-limiter configuration. By that, no additional per-packet state is required.

In addition, the CoDel algorithm requires a binary per-queue state, a 31 bit time register, and a 16 bit counter, leading to a memory width of 48 bit. The algorithm itself is implemented as a simple state machine, following its specification [10].

The output of the AQM is a two-bit information, either dropping or sending the packet. A central state machine in the `simple-scheduler` combines this output with the `rate-limiter` decision and notifies the `queue-memory` and `tx-handler` accordingly.

F. Control Plane Interface

The PCIe control plane interface allows for the writing of configuration memories and the reading out of counter values. As no high performance is needed, we build upon the lightweight wishbone bus protocol.

IV. Evaluation

In this section, we investigate the performance of the presented FPGA design. All results are measured on a Xilinx Alveo U200 FPGA running at 220 MHz clock frequency for the queueing logic and 300 MHz for the DDR4 memory controllers. All evaluated designs are routed without any timing violation.

The measurements are performed with the P4STA measurement framework [11], which measures latency and packet loss of each packet up to 100 $Gbit/s$ link speed. To achieve the highest possible time accuracy, i.e. below $5\,ns$, the input rate was limited to 99.9 $Gbit/s$ in all tests. All latency measurement results are corrected by the measurement overhead, *i.e.*, a fixed and constant latency caused by the measurement equipment and propagation delay of the used fiber optical cables. Therefore, the presented results describe only the time a network packet resides inside the evaluated FPGA design.

If not otherwise stated, each measurement run is 10 s long and consists of multiple millions of captured packets.

A. Throughput and Latency

To measure the theoretical maximum throughput and minimal latency, we instantiated the `no-scheduler` in the FPGA design with two DDR4 memory instances, as shown in Figure 1. This scheduler implementation accepts all packets immediately after being notified from the `queue_memory` and forwards them to the `tx-handler`. By that, no queue should be built up in the FPGA, and the base latency caused by receiving and sending packets through the DRAM can be measured. Further, the maximum achievable throughput can be determined.

The packet size of the following measurment results was 1474 $bytes$, a realistic packet size of best effort downstream flows in Internet access networks.

The plots in Figure 4 depict the measured latency time series for a system with 2048 and 262,144 queues, respectively. It can be observed that a higher number of queues causes a higher jitter in latency. This jitter is caused by the round-robin scheduler, which searches for non-empty queues. In one

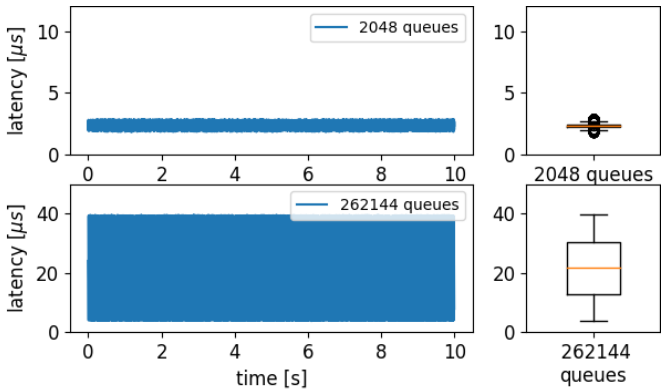

Fig. 4: Measured latency over time with no rate-limiting dependent on the number of queues. In both scenarios we use a 99.9 $Gbit/s$ input rate and a packet size of 1474 $byte$.

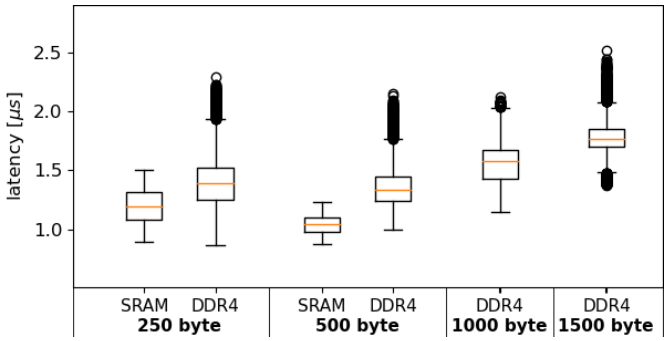

Fig. 5: Queueing latency distribution depending on the packet size and utilized memory technology at 100 $Mbit/s$.

clock cycle, concrete 4.5 ns at 220 MHz, 32 queues can be scanned. Assuming 262,144 queues, this would cause a jitter of 8192 clock cycles or 36.864 μs. In the case of 2048 queues, a theoretical jitter of 288 ns by the RR-scheduler is expected. However, the measured jitter is around 1.2 μs. We suspect that the remaining is caused by the non-constant behavior of DDR memory access and micro congestion in the FPGA design.

We experienced neither packet loss nor packet corruption for large packets with a size of 1474 $byte$. However, disabling one of the two DDR4 memories lowered the maximum throughput to around 70 $Gbit/s$ at the same packet size.

Additionally, smaller packets increase the packet per time rate but lead to packet loss at high loads. This fact is caused by the internal architecture and the performance characteristic of DRAM memory. For small packet sizes, the memory performance of DDR4 decreases due to the nature of DRAM [12]. To tackle this, we introduced in Section III-B the possibility to store small packets in internal SRAM. For 500 $byte$ sized packets and a single SRAM, we observed similar throughput performance characteristics compared to utilizing two external DDR4 memories. Also, we detected 0.6 μs less jitter than with packets of the same size stored in external DDR4 memory as a side effect. Figure 5 shows the measured latency distribution dependent on the packet size and used memory technology. In all cases, the input rate into the queueing system was limited to 100 $Mbit/s$.

First, it is noteworthy that the histogram of SRAM memory is tighter distributed than for DDR4 memory. Second, the

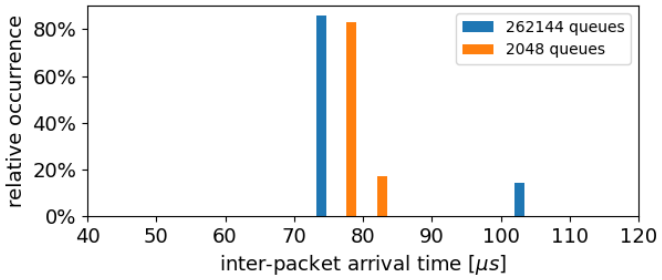

Fig. 6: Inter-packet arrival time for a configured rate of 100 $Mbit/s$, 1000 $byte$ packets and 262144/2048 queues with the `simple scheduler` and no AQM.

latency for 250 $byte$ packets is more widespread than for 500 $byte$ packets. This can be explained by the constant bytes per second input rate which doubles the rate of packets per second when the size is halved.

We conclude that for a mixture of big and small packets, *e.g.*, VoIP and normal Internet traffic, the total system performance can increase enormously by an additional internal SRAM memory. Furthermore, applications with strong jitter requirements can benefit from that.

B. Scheduler Verification

The *rate-limiter* should schedule packets of each queue with the assigned rate. In addition, it should cause as little packet bursts as possible. A perfect rate-limiter would always send only a single packet and the time between two packets is constant if the packet size is constant. A limiter tending to microbursts would send many subsequent packets in a row, and the waiting time until the next packet burst starts would be much longer in order to comply with the configured rate.

To verify the rate-limiter, we configured a rate of 100 $Mbit/s$ in the FPGA and sent 1000 $byte$ test packets with $\geq$ 200 $Mbit/s$ to ensure a permanently filled queue. Figure 6 shows the measured results for an FPGA design with 2048 and 262,144 queues. From this, we can deduce that always only a single packet was sent at one point in time and no micro-bursts occurred. In contrast to an ideal rate limiter, the time between two packets is not perfectly constant. This is caused by the time the scheduler requires to iterate over all queues, *i.e.*, 64 clock cycles for 2048 queues. By that, small microbursts are unavoidable if the configured rate is not integer divisible by the scheduler period.

C. Active Queue Management

Last, the functionality of the integrated Active Queue Management (AQM) algorithm will be investigated. Note that this implementation is only one example to show the integrability of AQMs in the proposed FPGA design. In Figure 7 the measured latency over time is shown for the FPGA implementation of the CoDel-AQM, which exhibits a similar behavior as the Linux kernel reference implementation. This behavior results from controlled packet dropping in the FPGA and the rate adaption of the TCP congestion control as a consequence of the detected packet loss. The actual latency falls below the desired value of 5 ms periodically, which is the expected behavior.

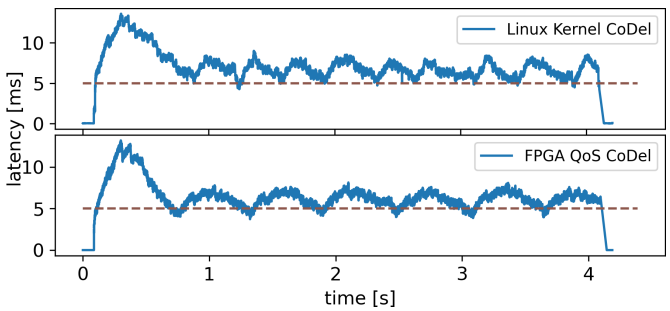

Fig. 7: CoDel Active Queue Management (AQM) comparison for three concurrent TCP flows. The FPGA implementation shows a similar behavior as the Linux Kernel reference implementation of CoDel.

TABLE I: Resource utilization for the Xilinx Alveo U200 FPGA. All configurations have a single 100 $Gbit/s$ Ethernet IP core for receiving and sending packets.

Configuration	LUT	FF	BRAM	URAM
2048 queues, 16384 buckets, 1x DDR4, no-scheduler	36,463 (3.1%)	51,104 (2.3%)	130 (6.0%)	6 (0.6%)
2048 queues, 16384 buckets, 1x DDR4, simple-scheduler	36,775 (3.1%)	54,445 (2.3%)	139 (6.0%)	8 (0.8%)
2048 queues, 16384 buckets, 1x DDR4, simple-scheduler, AQM	37,178 (3.1%)	54,596 (2.3%)	130 (6.0%)	9 (0.9%)
2048 queues, 16384 buckets, 2x DDR4, 1x SRAM, simple-scheduler, AQM	65,373 (5.5%)	88,797 (3.8%)	157 (7.3%)	73 (7.6%)
262144 queues, 131072 buckets, 2x DDR4, 1x SRAM, simple-scheduler, AQM	66,576 (5.6%)	89,108 (3.8%)	173 (8.0%)	416 (43.3%)

D. Resource Utilization

The resource utilization of the FPGA depends strongly on the configuration of the design. Table I shows the numbers for five exemplary configurations after synthesis for the Xilinx Alveo U200 FPGA. For all configurations, the number of counters is equal to the number of queues.

First, it is noteworthy that internal SRAM memory (*BRAM* and *URAM*) is the most utilized resource. This utilization is not surprising as the design consists of huge memories for queues, packet pointers, and counters. Second, we would like to point out the capability of scaling. For some use cases, having only a few thousand queues and less total buffer capacity is sufficient, *e.g.*, Internet service creation in sparsely populated areas. Such a configuration can also be built with low-budget FPGAs. The utilization of boolean Lookup Tables (LUT) and Flip-Flop memories (FF) is neglectable in all investigated configurations. We can summarize that the limiting resource is internal SRAM memory. The maximum utilization of this resource is limited by the signal routing constraints of the FPGA compiler. The last example represents the upper bound configuration, *i.e.*, which is feasible for the synthesis tool, at the FPGa clock frequency of 220 MHz.

E. Energy

To assess the energy consumption of the presented FPGA design, we measured the electrical interface input power of a server with and without an integrated FPGA. The results are as shown in Table II. The measurements were performed with a smart power outlet with an accuracy in the range of 1% as indicated by the vendor `NETIO`. In both scenarios, the server (Dell R740 with an Intel Xeon Silver 4110 CPU) does not execute any processes besides the underlying operating system. In the scenario with an FPGA, we 1) measured the energy power input while no packets are processed in the design (*idle*), and 2) while forwarding 99.9 $Gbit/s$ of packets through the FPGA.

TABLE II: Measured energy consumption of the FPGA design (maximum configuration: 262k queues, three memories, AQM) in idle mode and under 99.9 $Gbit/s$ load in a server.

	idle	99.9 Gbit/s	Δ
Server	$75W$		
Server + FPGA	$98W$	$108W$	$10W$
Δ	$23W$		

The results show that the additional power consumption of an FPGA with the proposed design in idle is around 23 W, including all overheads. The additional variable power consumption is 10 W at 99.9 $Gbit/s$.

Compared with the calculated energy consumption of the synthesis tool, namely $10.5W$, this is much higher due to the several energy consumers despite the FPGA chip, *i.e.*, FPGA board overheads as well as the server system itself.

It is noteworthy that most energy is used to operate the server holding the FPGA card and not the card itself. An integrated design, *i.e.*, into network switching hardware [13], could strongly lower the total energy consumption. Nevertheless, the energy consumption of processing the packets in software would be much higher. Depending on the concrete software and FPGA implementation, we assume a factor between 10 and 100 under load [14].

The results show that it is feasible from the energy perspective to integrate this or a similar FPGA design into network switches as an additional QoS processor. Assuming 1,000 $Gbit/s$ FPGA bandwidth distributed over two FPGAs, we expect an additional power consumption of less than 230 W in idle and additional 100 W when operating at maximum bandwidth. Note that this is an upper-bound estimation as we consider the idle FPGA power consumption to be linear to the maximum throughput.

F. Control Plane Performance

At startup and runtime, the configuration of queues and schedulers must be updated by the control plane to set each customer state according to its subscription. Further, counter values must be read by the control plane. For this, memory-mapped I/O operations are performed, typically over PCIe.

Table III shows the measured access times for 32-bit read and write requests on the FPGA internal registers for different scenarios. Each scenario was executed and measured 100 times to be statistically significant.

First, we evaluated a local controller accessing the FPGA directly through a simple I/O kernel module and via memory-mapped I/O. Second, we started a local gRPC server, allowing access to the FPGA via a simple API. This approach allows

TABLE III: Measured control plane speed for sequential 32 bit configure write and read requests on the PCIe interface.

	kernel module	memory mapped	gRPC local	gRPC remote
write	$5.37\mu s$	$1.25\mu s$	$270.4\mu s$	$14822\mu s$
std. dev. (w)	$0.46\mu s$	$0.22\mu s$	$49.84\mu s$	$5154\mu s$
read	$11.11\mu s$	$5.67\mu s$	$271.2\mu s$	$15010\mu s$
std. dev. (r)	$2.40\mu s$	$3.74\mu s$	$41.46\mu s$	$6311\mu s$

disaggregating data and control plane logic as suggested by Software Defined Networking (SDN) concepts.

For the local controller, it is noteworthy that the overhead of the context switches between controller and kernel lowers the speed approximately by a factor of 4 in the case of writes and by a factor of 2 for reads. Note that a kernel module is not needed at all, as it would only be required for interrupt handling and DMA transfers, which are typically not required.

Using gRPC significantly reduces read and write throughput, especially in the case of a remote controller running on a second server in a remote data center. This is caused by the implementation, which executes read and write requests sequentially and blocking. Therefore, remote controllers should implement an API allowing parallel calls, write request batching, and non-blocking operations.

Nevertheless, assuming a local gRPC control plane application with non of the above mentioned optimizations, it would be possible to set up $10,000$ customer sessions in the FPGA within $2.7\ s$. This exceeds the setup rates achievable by common control planes in service provider environments by more than an order of magnitude and thus will not act as a significant bottleneck.

G. Discussion and Summary

Our evaluation results show that FPGAs are a viable solution for massive packet queueing in computer networks if off-the-shelf switching ASICs do not provide the required functionality. The presented design, optimized to run at $100\ Gbit/s$ link speed, provides sufficient performance and could be instantiated multiple times on a single FPGA to serve several Ethernet ports in parallel and to achieve a total higher throughput.

While this work focused only on internal SRAM and external DDR4 memory, the availability of novel FPGA chips with integrated High Bandwidth Memory (HBM) opens up new possibilities. For example, state outsourcing of unused queues or counters might be possible.

Further, FPGAs allow the integration of advanced scheduling and active queue management algorithms in the network data path to ensure a high QoS level.

Last, we would like to mention the possibility of integrating FPGAs into (programmable) network switches. Similar to accelerator cards in data centers, this would provide a universal and flexible platform allowing to describe almost any network functionality while combining the benefits of FPGAs and conventional programmable network switching ASICs.

V. Related Work

The use of FPGAs for building network functionality has been discussed in related work before. Naous *et al.* presented the first generation of the NetFPGA hardware and project in 2008 as a universal platform for packet switching research [15]. In addition to FPGAs, Shrivastav proposed in 2019 the novel concept of programmable Push In Extract Out (PIEO) queues, which allows schedulers to be programmed in hardware [16]. The PIEO concept can be embedded in programmable switching ASICs without requiring reconfigurability on the bit level (as it does on FPGAs). Utilizing FPGAs for packet queueing and Active Queue Management (AQM) was discussed by Sivaraman *et al.* in 2013 [17]. The authors illustrate that FPGAs are well suited to implement many different AQMs in hardware. Additionally, they state that there is no single algorithm that fits the need of all scenarios, and therefore, the reconfigurability of FPGAs is highly beneficial. In accordance with this, Xu *et al.* proposed an AQM implementation on FPGAs, tailored for datacenter networks, allowing high throughput and no packet loss [18]. Furthermore, the feasibility of AQM algorithms on P4 programmable networking switches, providing best-in-class performance, was shown with some limitations [19].

Sanchez *et al.* have shown the viability of FPGAs for advanced flow-based and QoS-aware network functions in 5G networks [20]. In our previous work, we proposed the concept of offloading residential Internet access functionality, except for massive packet queueing, on P4 programmable switches [5]. Activities within the Telecom Infra Project (TIP) are targeting an open platform for residential Internet access creation, called "OpenBNG", upon programmable hardware [21], in which our work perfectly fits.

VI. Conclusion

Powerful systems are needed to fulfill the need for high performance and deterministic packet queueing in Internet service provider access networks and other scenarios. While most commercial ASICs for packet switching, aiming at a data center market, do not provide sufficient queueing capabilities, FPGAs offer a promising alternative coming at scale and reasonable power consumption.

In this work, we proposed an FPGA design running at 100 *Gbit/s* link speed with more than $200,000$ queues. This design provides sufficient memory capacity realized in external DDR4 memory and the capability to implement complex scheduling algorithms. Our evaluation results show the feasibility of this approach and the desired performance characteristics. In scenarios of up to $100 Gbit/s$, we could not observe any unexpected packet loss or packet corruption. The measured electrical power consumption clearly shows that this approach is also highly energy efficient. The presented design is available as an open-source project to enable other researchers to benefit from this project and its building blocks[1].

Acknowledgment

This work has been supported by Deutsche Telekom through the Dynamic Networks 8 project, by the LOEWE initiative (Hesse, Germany) within the emergenCITY center [LOEWE/1/12/519/03/05.001(0016)/72] and the German Federal Ministry of Education and Research (BMBF) within the project "Open6GHub" under grant number 16KISK014.

[1] https://github.com/ralfkundel/massive_fpga_queueing_system

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[4] R. Kundel, T. Meuser, T. Koppe, R. Hark, and R. Steinmetz, "User plane hardware acceleration in access networks: Experiences in offloading network functions in real 5g deployments," in *Proceedings of the 55th Hawaii International Conference on System Sciences*. Computer Society Press, 2022, p. 1–10.

[5] R. Kundel, L. Nobach, J. Blendin, W. Maas, A. Zimber, H.-J. Kolbe, G. Schyguda, V. Gurevich, R. Hark, B. Koldehofe, and R. Steinmetz, "OpenBNG: Central Office Network Functions on Programmable Data Plane Hardware," *International Journal of Network Management*, vol. 31, no. 1, p. e2134, 2021, 25 pages. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.2134

[6] R. Kundel, J. Wallerich, W. Maas, L. Nobach, B. Koldehofe, and R. Steinmetz, "Queueing at the telco service edge: Requirements, challenges and opportunities," in *Workshop on Buffer Sizing*. Stanford, US: Workshop on Buffer Sizing, 2019.

[7] R. Adams, "Active queue management: A survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 3, pp. 1425–1476, 2013.

[8] V. Bothra, A. Peer, V. K. Singh, M. Maity, and R. Shah, "Codel-act: Realizing codel aqm for programmable switch asic," in *2024 IFIP Networking Conference (IFIP Networking)*. IEEE, 2024, pp. 468–474.

[9] ARM, "AMBA 4 AXI4-Stream Protocol," https://developer.arm.com/documentation/ihi0051/, Tech. Rep., 2010, [Online].

[10] K. Nichols, V. Jacobson, A. McGregor, and A. Iyengar, "Controlled Delay Active Queue Management," Internet Engineering Task Force, Request for Comments 8289, 2018.

[11] R. Kundel, F. Siegmund, J. Blendin, A. Rizk, and B. Koldehofe, "P4STA: High performance packet timestamping with programmable packet processors," in *Network Operations and Management Symposium (NOMS)*. IEEE/IFIP, 2020, pp. 1–9.

[12] S. Li, D. Reddy, and B. Jacob, "A performance & power comparison of modern high-speed dram architectures," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 341–353.

[13] APS Networks, "APS2140D (Tofino+FPGA) - Advanced Programmable Switch," https://www.aps-networks.com/products/aps2140d/, 2021, [Online; accessed 15-Jan-2022].

[14] Z. Xu, F. Liu, T. Wang, and H. Xu, "Demystifying the energy efficiency of network function virtualization," in *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, 2016, pp. 1–10.

[15] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "Netfpga: Reusable router architecture for experimental research," ser. PRESTO '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1–7.

[16] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 367–379. [Online]. Available: https://doi.org/10.1145/3341302.3342090

[17] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan, "No silver bullet: extending sdn to the data plane," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in networks*. ACM, 2013, p. 19.

[18] L. Xu, K. Xu, T. Li, K. Zheng, M. Shen, X. Du, and X. Du, "Abq: Active buffer queueing in datacenters," *IEEE Network*, vol. 34, no. 2, pp. 232–237, 2020.

[19] R. Kundel, A. Rizk, J. Blendin, B. Koldehofe, R. Hark, and R. Steinmetz, "P4-codel: Experiences on programmable data plane hardware," in *ICC 2021 - 2021 IEEE International Conference on Communications (ICC)*, 06 2021, pp. 1–6.

[20] R. Ricart-Sanchez, P. Malagon, P. Salva-Garcia, E. C. Perez, Q. Wang, and J. M. Alcaraz Calero, "Towards an fpga-accelerated programmable data path for edge-to-core communications in 5g networks," *Journal of Network and Computer Applications*, vol. 124, pp. 80–93, 2018.

[21] TIP, "Open bng - technical requirements," https://cdn.brandfolder.io/D8DI15S7/as/jx5654t6f5bx94crvfxwm57w/TIP_OOPT_Open_BNG_Technical_Requirements_v10docx.pdf, Tech. Rep., 2020, [Online].