QUEUE-MEM: Energy-Efficient Hardware Storage for Advanced Network Function Acceleration

Mariano Scazzariello[1], Tommaso Caiazzi[2], Hamid Ghasemirahni[3], Dejan Kostić[3], and Marco Chiesa[3]

[1]RISE Research Institutes of Sweden
[2]Roma Tre University
[3]KTH Royal Institute of Technology

Abstract

General-purpose CPU servers have been widely used to deploy Network Functions (NFs) thanks to their high flexibility and simplicity of deployment. Due to their high energy consumption, best practices advocate for only processing packet *headers* on CPU cores while temporary storing the corresponding packet *payloads* on either network interface cards or external RDMA-enabled memory.

We show that the seemingly minor decision of *where* to store a packet payload *greatly impacts* overall energy consumption in state-of-the-art NF systems operating at terabit-per-second speeds. In fact, we show that if one could *ideally* store packet payloads on today's *hardware switches*, while processing headers externally, one could reduce energy use by 1.8× to 10.9× compared to current practices.

In this paper, we introduce QUEUE-MEM, a general-purpose, energy-efficient storage solution to enhance NF deployment that is amenable for implementation with various existing hardware switches. Building QUEUE-MEM involves addressing significant challenges associated with payload storage, as hardware switches lack such functionality. By carefully exploiting the *buffer queues* of existing switches, we are the first ones to build and showcase a robust, energy-efficient packet processing pipeline capable of handling terabit-per-second speeds and supporting advanced per-flow network functions, all while using just a *single* commodity server connected to an ASIC switch.

1 Introduction

Network Functions (NFs) are a fundamental component of today's networks [38], supporting critical use cases ranging from enforcing security policies [45], to improving resource utilization [14], communication performance [28], and beyond. As NFs become increasingly essential, the choice of deployment platform — whether hardware-based (*e.g.,* ASIC) or software-based (*e.g.,* CPU) — becomes crucial to achieving low energy consumption, high flexibility, and advanced NF capabilities.

On the one hand, hardware ASIC switches offer the lowest power consumption per processed packets. Yet, realizing advanced network functions on ASIC silicon is difficult due to the constrained computational and memory resources as well as the lack of flexibility in logic modifications. Advanced NFs that operate on individual TCP/UDP flows are therefore difficult to support entirely in ASIC switches, *e.g.,* switches lack primitive data structures [4, 46] as well as memory to track all possible connections [7, 49, 61]. On the other hand, general-purpose CPUs have been widely adopted to support arbitrary advanced network functions in an easy-to-deploy and flexible manner. Their main drawback is however the higher energy consumption compared to ASIC switches, potentially up to a factor of 100× [26].

Recent approaches to minimize the involvement of power-hungry CPU-based devices advocate for *splitting* packet headers from packet payloads and to process *only* the packet headers on CPU or FPGA devices while temporarily storing the payloads somewhere in the network (*e.g.,* in the NIC of the CPU/FPGA device [46], opportunistically in the RAM of any server in a datacenter [49], on the switch itself [18]). Packet-splitting approaches promise dramatically increased throughput in an NF deployment.

In this work, we make the observation that even a simple operation, such as storing a payload, has profound implications on the energy consumption required for deploying NFs in a network. Our design analysis (at 1 Tbps) summarized in Fig. 1 indicates a 3.9× and 1.8× higher energy consumption when storing payloads through RDMA (as in Ribosome [49]) or on network interface cards (as in nicmem [46]), respectively,

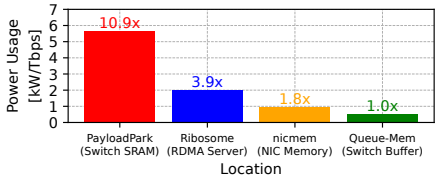


Figure 1: The location where a payload is stored impacts the estimated *total* power consumption of an NF deployment.

compared to storing payloads directly on the memory of the switch that splits headers from payloads. This is not surprising as transmitting payloads outside the switch results in undesirable overheads, including activating additional ports on the switch, processing on network interface cards, and external memory storage.

Temporarily storing the payloads of the packets on existing ASIC switches is, however, a complex task. ASIC switches do not offer explicit primitives to temporarily store and retrieve full-sized packets. *None* of today's existing approaches is able to store *entire* payloads on ASIC switches while processing their headers on external devices. As an example, PayloadPark [18] shows that existing switches allow programmers to only store and retrieve what the switch has parsed (*i.e.,* $\leq$ 160 Bytes). As such, PayloadPark must still transmit almost full-sized packets to external CPU processors, resulting in 10.9$\times$ higher power consumption compared to being able to store entire payloads on the switch.

We set four main requirements for an NF deployment:

- **power efficiency**: minimize *energy* to process traffic.
- **expressiveness**: support *advanced* network functions.
- **speed**: processing *terabits* per second of traffic.
- **compatibility**: runs on *existing* hardware.

In this work, we present QUEUE-MEM, a novel approach to temporarily store information (*i.e.,* packets) on existing hardware ASIC switches. Using QUEUE-MEM, we store payloads directly on switches and transmit headers to external CPU servers, ultimately achieving 1.8-10.9$\times$ lower estimated energy consumption compared to existing NF systems.

QUEUE-MEM explores a simple and unconventional idea that advocates for storing packets inside memory buffer queues that are *naturally* present in ASIC switches to absorb bursts of traffic towards the output ports (see Fig. 2). Such queues offer a FIFO-based abstraction (with ASIC-based priority packet scheduling), which is ideal to store payloads while headers are being processed on external NF processors. QUEUE-MEM makes the packet buffer memory of a switch *programmable*, *i.e.,* enables fetching data stored in the buffer based on external events (*e.g.,* reception of a header).

While being a simple idea, leveraging the packet buffer memory on existing ASIC switches is challenging. A switch forwards packets from the queue based on the speed of the outgoing port interface associated to a queue. This means that naively inserting a payload in a queue buffer may not result in storing that payload for the desired amount of time, *i.e.,* tens of microseconds for most network functions.

We tackle the above problem by exploiting existing functionalities for pausing and resuming the processing of packets in a queue in response to events that the switch handles at *data-plane* speed. These functionalities are commonly present to support the PFC protocol [21] and the DCQCN congestion control [65] on a majority of widely adopted switches [5, 24, 36]. At the high level, QUEUE-MEM *(i)* splits headers from payloads, *(ii)* sends the headers to an external NF, and *(iii)* stores the payloads in a queue by pausing that queue. QUEUE-MEM does not dequeue a payload from a queue (*i.e.,* resume processing packets on that queue) until the processed header returns from the external NF. While simple in theory, realizing this idea is non-trivial. First, existing operations to pause and resume a queue *do not operate at the per-packet level*, making it extremely difficult to pause and resume processing of packets in the queues at the correct time.[1] Second, one should not affect any background traffic that should not be sent to the external NF. Third, we need to deal with any potential packet drop (*e.g.,* a header), which may stall the processing of packets in a queue. Developing a system that effectively mitigates the "brittleness" presented by these challenges is a complex task.

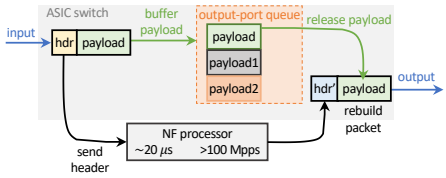


Figure 2: QUEUE-MEM stores payloads in the queue of a port. A high-performance external NF system today may process 100s million packets per second, each within $\sim 20\mu s$ of latency. The switch buffers payloads in the per-port queues until the processed headers are sent back from the NF.

Our key idea is to embrace the indirect and coarse-grained capabilities of the pause/resume functionalities of today's switches and guarantee that the system only dequeues *batches* of payloads when their corresponding headers have been processed, taking care of potentially dropped headers at the external NF. Guaranteeing this correct release of batches requires to carefully *time* with sub-microseconds levels of precision the pause/resume actions triggered by a header traversing the processing pipeline while recovering from potential packet losses.

We evaluate the robustness of our implementation of QUEUE-MEM in a realistic testbed under a different number and types of network functions and adversarial workloads. To the best of our knowledge, we are the first to run an NF system that processes *beyond* 1 Tbps of traffic running advanced NF chains (*e.g.,* rate limiters, Layer-4 load balancers, AES encryption of header fields) on a *single* CPU-based server, allowing us to consume 1.8-10.9$\times$ lower energy than alternative existing approaches. To summarize, our contributions are:

- We are the first to quantify the massive energy consumption overhead of different payload storage systems.
- We are the first to present an approach for storing *entire* payloads of data on a switch in a programmatic and energy-efficient manner without modifications to existing switches.

[1] Pausing/resuming queues has been proposed in Conweave [54] for a *different* problem with different requirements. Conweave does *not* require the same per-packet level of precision in pausing/resuming queues that we require in our system, making their approach unsuitable here.

HW	Base Power Usage	Additional Power per Gbps	Additional Power per Mpps
16-nm ASIC	108 W [26]	0.026 W [26]	0.312 W (1.5 KB pkts)
x86 Server	200 W	1.75 W (1.5 KB pkts) 2.36 W (64 B pkts)	21.07 W (1.5 KB pkts) 1.21 W (64 B pkts)
RDMA NIC	24.9 W [41]	-	-
FPGA	-	0.75 W [61]	9 W (1.5 KB pkts)

Table 1: Power consumption of different devices for deploying NFs.

- We present our key idea of exploiting the packet buffer memory that today's switches deploy for buffering packets during congestion events. We describe challenges in using existing operations for pausing and resuming queues directly from the data plane to control the storage of payloads of data in the queues. We overcome these challenges by devising a carefully timed batch-based mechanism to dequeue packets from their queues.
- We are the first to demonstrate through an implementation on an ASIC switch the ability of QUEUE-MEM to support the processing of *advanced* NF chains at over 1 Tbps with a single external CPU-based server and no additional resources assumptions on opportunistic resource availability (*e.g.,* Ribosome requires 14 additional RDMA-based shared servers for this level of performance).
- We will release all our P4 code for running QUEUE-MEM on programmable ASIC hardware.

2 Where to Store a Packet Payload?

We now explore the architectural design space for network functions with a particular focus on storing payloads, energy consumption, and performance. We start the discussion with a minimal background and then move to traditional designs and state-of-the-art mechanisms. We focus on *shallow stateful* NFs, which are widely deployed NFs that *(i)* process each packet only based on the header of the packet and *(ii)* must keep some state in the NF to process correctly the incoming packets. These network functions represent a large fraction of the existing NFs, including Layer-4 load balancers [12, 14], rate limiters [51], encryption/decryption of header fields [45], traffic optimizers [28, 55], and packet schedulers [16]. Network functions that need to process the entire packet (*e.g.,* deep packet inspection) go beyond the scope of this paper as the payload should be stored at the same location where the packet is processed.

A power model for estimating energy costs. We carry out our energy analysis based on *real-world testbed measurements* as well as information available on datasheets. We rely on the same methodology used in previous industrial work, *i.e.,* Tiara [61], with some differences (to make the comparison as fair and up-to-date as possible). In fact, we model the power consumption proportional to the number of processed Mpps rather than Gbps, as our testbed measurements show that the computational load is mainly influenced by the number of packets rather than the size of the packet. Even if we carry out our analysis on specific hardware configurations, we expect the overall trends to remain consistent across different hardware setups. Table 1 summarizes the results using a Layer-4 load balancer network function as a reference. We rely on the most recent real-world benchmark of programmable ASIC switches to derive the power consumption of a 32x100-Gbps programmable ASIC switch [26]. The base power consumption is 108 W and the cost increases by 26 mW per additional Gbps of forwarded traffic. Since the benchmark provides data only in terms of power per Gbps, we estimate the power consumption per additional Mpps by assuming a packet size of 1.5 KB, which results in ∼312 mW per Mpps. For CPU-based servers, we rely on our own benchmarks as the data from the Tiara paper is not up-to-date. In our testbed, we measure the power usage over 30 minutes (using a SmartMe power meter [53]) while running a load-balancer NF in FAJITA [17] on a server equipped with Intel®Xeon®Gold 6444Y @ 3.60 GHz, 128 GB of RAM and one Mellanox Connect-X 7 NIC. We carry out two separate measurements: (i) NF handling 100 Gbps of 1.5 KB packets (∼8 Mpps), and (ii) NF processing 100 Gbps of 64 B packets (∼142 Mpps), representing a workload focused on header processing. We verified that FAJITA can process >100 M headers per second per CPU socket, as stated in the paper [17]. Our measurements show that the idle server power consumption is ∼200 W with an added power draw of 21.05 W and 1.21 W per Mpps of processed traffic for 1.5 KB and 64 B packets, respectively. We do not consider the RNIC consumption in this value. To estimate the energy consumption of RNICs, we refer to the ConnectX-7 hardware datasheet, which specifies a power consumption of 24.9 W [41]. We do not adjust this value based on the Mpps, as RNICs lack load-dependent power-saving mechanisms, and therefore we assume constant consumption. For FPGAs, we again rely on hardware datasheets where an AMD Xilinx Alveo U50 equipped with 2x100-Gbps port interfaces, which has similar characteristics to the FPGA used in Tiara, consumes 75 W [57]. As we do not know the idle power consumption, we assume that the power consumption grows linearly from zero to 75 W based on the processed traffic, with 9 W per Mpps (with 1.5 KB packets). We verify that the power consumption using this approximation is in line with the estimate used in Tiara. We are now ready to analyze the cost of different NF deployments with a focus on the location where headers and payloads are processed and stored, respectively (refer to App. A for comprehensive details on the methodology and formulas).

Hardware ASIC switches are energy-efficient but poorly suited to support stateful NFs. Ideally, an NF processor would run entirely on the data-plane of an ASIC switch, striking the best Watt-per-packet performance in the NF design space. We show the energy consumption of this ASIC-only approach in Fig. 3 using the light green line (with right-facing triangle marks). In the figure, the x-axis denotes the number of packets per second and the y-axis shows the power

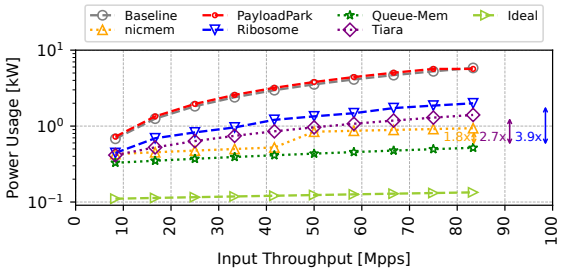


Figure 3: Power usage of different packet processing pipelines (considering an average packet size of 1.5 KB).

consumption of the NF deployment. Our analysis shows a cost of 134 W to process 1 Tbps of traffic on our 16-nm programmable ASIC switch.[2] Unfortunately, as evidenced in recent work advocating for CPU-based NF deployment, today's existing ASIC switches do not support advanced stateful NFs as these either require excessive memory to store the state or their logic is too complex for running on ASIC [49] as it is the case, for instance, of TEA [30]. For example, Tiara [61] has shown that an ASIC switch can only store state for 200 K connections, which is insufficient for existing workloads. Ribosome [49] has further shown that even a 10 Gbps real-world trace today would use 20% of the existing memory of an ASIC switch, and a 25.6-Tbps switch could only store 0.3% of the necessary flow state to process packets, even with just a single NF. Similar problems have been presented for inserting new state into key-value data structures, where ASIC switches could only insert 1 out of 100 flow states because of the limited insertion speed [7].[3]

Traditional NF processors are power-hungry. To implement complex NFs, today's state-of-the-art systems offload simple parts of NFs to ASIC switches (*e.g.,* a small forwarding/routing/ACL table) [63] and run more complex parts on external NF processors (*e.g.,* NFs requiring to keep per-connection state at high frequency and low latency). For instance, consider a traditional NF deployment where CPU-based NF processors are connected to a switch, from which they receive incoming packets and retransmit back the processed packets. Using our energy-consumption data from Table 1, we observe in Fig. 3 that this "baseline" approach (gray line with circle marks) requires 10 CPU sockets to process 1 Tbps of 1.5 KB packets (*i.e.,* 83 Mpps), amounting to 5.83 kW. This results in a significantly higher power consumption compared to the 134 W needed by a single 16-nm ASIC switch. Further, this approach wastes half of the bandwidth on the ASIC switch to move packets back and forth from external CPU servers.

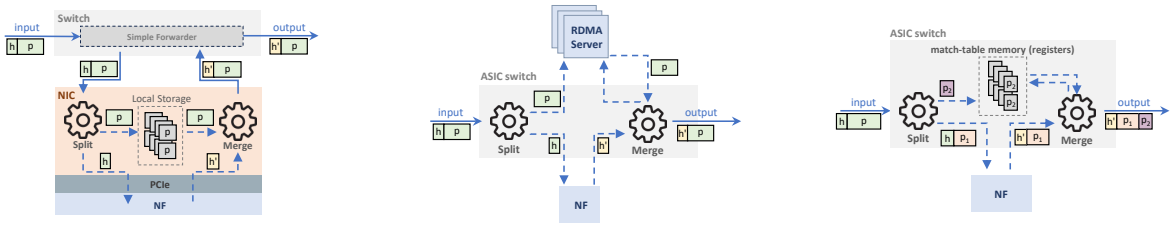
[2]The power consumption is even lower on new generation switches, *e.g.,* less than 1 W per 100 Gbps on Tomahawk 5 [6].

[3]Approaches like Switcharoo [7] to install flow states at data-plane speed suffer from multiple issues: (i) cannot implement complex NFs, and (ii) can only store states for sub-ms durations, whereas L4 load balancers require second- or minute-level storage.

FPGA-based packet processors consumes less yet significant amount of energy. Tiara [61] is a powerful load balancer system that reroutes packets from a switch to FPGAs (for the fast path) and x86 servers (for the slow path) performing per-packet load balancer calculations. Fig. 3 shows the power cost of a Tiara-like system (purple line with diamond marks). Tiara requires an ASIC switch, 10 FPGAs and one CPU socket. We assume that only one tenth of the total load goes through the slow path (*i.e.,* the CPU). Considering this, the total power consumption is 1.40 kW at 1 Tbps (*i.e.,* 83 Mpps), roughly 10× higher than an ASIC switch. Similar to CPU-based deployments, Tiara must reserve half of the bandwidth of the ASIC switch. Moreover, Tiara is tailored for load balancing and supporting other advanced network functions requires significant costs in developing code to synthesize on FPGAs.

Shallow network functions do not process payloads. Shallow NFs [18, 46] are widely deployed network functions that only process fields contained in the packet header, which have a negligible fixed size. One such example is the load balancer function used in Tiara, which only needs to process the 5-tuple of a connection. Other examples are access control lists, rate limiters, NATs, packet schedulers [16], transport optimizers [28], forwarding path validators [45], and beyond. Yet, the aforementioned packet processors transmit & receive full-sized packets from the switch to external packet processors. Receiving payloads has a significant impact on the performance of the NF processor (*e.g.,* by cache pollution [15, 16]). For these reasons, three recent systems have advocated for only processing packet headers on external NF processors (for better CPU cache hit-ratio and utilization) and storing payloads in various locations: on the NIC of the NF processor server with nicmem [46], on RDMA servers with Ribosome [49], or on the switch with PayloadPark [18]. These approaches promise to bring unparalleled NF performance by highly optimizing packet processing on power-hungry CPU devices. We analyze the actual energy consumption benefits of the above three approaches (using the high level architecture overviews from Fig. 4) in the following paragraphs.

nicmem: Splitting packets on the NIC of the server brings energy savings but wastes half of the switch ports. In nicmem (Fig. 4a), a traditional switch transmits the entire packet to an external CPU. The network interface card receives the packet, splits the header from the payload, stores the payload on a cache memory on the NIC, and processes only the header on the CPU, thus minimizing cache pollution. State-of-the-art systems have shown the ability of commodity servers to process packets at $>$ 100 Mpps [17]. Consequently, a single NF server can be equipped with multiple NICs to handle such a volume of packet headers on the CPU. The CPU model of our analysis supports up to 80 PCIe 5.0 lanes, enabling to have up to five NICs within a single physical machine. We show in Fig. 3 the



(a) nicmem (b) Ribosome (c) PayloadPark

Figure 4: A high-level comparison of three systems that split packet payloads and store them while the header is being processed on an external NF. *None* of these systems can store the *entire* payload on the switch.

power consumption of nicmem (orange line with top-facing triangle marks), which requires two servers equipped with one CPU socket and five NICs at 1 Tbps, resulting in a power consumption of around 0.93 kW per Tbps of processed traffic, still roughly 9× higher than an ASIC-only approach. As a drawback, nicmem consumes 2× more ports than an ASIC-only approach, as fully-sized packets must be transmitted back and forth between the switch and external packet processors.

Ribosome: Storing payloads on external servers is energy inefficient. In Ribosome [49] (Fig. 4b), a programmable switch directly splits the header from the payload, only transmits the header to an external processor, and opportunistically stores the payload on the memory of any server in a datacenter using CPU-bypass technology (*i.e.*, RDMA). We show in Fig. 3 the power consumption of Ribosome (blue line with bottom-facing triangle marks). Ribosome requires a single NF server (as it receives only packet headers), one ASIC switch, and at least 14 RDMA-enabled shared servers to store payloads (as stated in the paper, each RDMA server is able to process up to 75 Gbps of payloads). The power consumption of Ribosome at 1 Tbps reaches a total of 1.99 kW, roughly 20× the power consumption of a single ASIC switch.

PayloadPark: Storing only a few bytes from the payload does not bring energy savings. In PayloadPark (Fig. 4c), the authors aim to store payloads inside a switch using the local SRAM memory and stateful ALUs available on the switch. Unfortunately, existing programmable switches only allow storing bits that have been parsed. This limit is 160 Bytes (p_2 in Fig. 4c), leaving PayloadPark unable to store entire payloads on the switch and transmitting large fractions of a packet to external NF processors (p_1 in Fig. 4c). Fig. 3 shows the power consumption of PayloadPark (red line with dot marks), which requires 10 servers to process 1 Tbps of traffic, resulting in one of the highest power consumption around 5.66 kW. Even assuming an *ideal* switch that parses the entire packet (which would be extremely expensive in hardware), PayloadPark would require dedicated SRAM memory to store packets. However, SRAM memory is a scarce resource on a switch. In contrast, we rely on a small fraction of the memory for buffering packets that already exist on today's switches without affecting the SRAM memory available to store forwarding ACL, or firewall rules.

QUEUE-MEM: storing full payloads on ASIC switches would bring enormous energy savings. We analyze the potential benefits of storing the packet payloads entirely on the switch, which is the approach we take in this work, and we refer to as QUEUE-MEM. In this approach, an NF deployment only requires one CPU socket and one ASIC programmable switch to process more than 1 Tbps of traffic. Fig. 3 shows that the power consumption of such an approach is around 517 W when processing 1 Tbps of traffic. To put this number into perspective, storing payloads on the switch results in 1.8× and 3.9× energy savings compared to equivalent CPU-based NF deployments. Surprisingly, by storing the payload on the switch, a CPU-based approach becomes 2.7× more energy efficient than an FPGA-based deployment like Tiara.

Summary: the payload-storage location matters. While storing payloads is a relatively simple operation entailing a write and read operation, we demonstrated that the location where the payload is stored greatly impacts the total power consumption of the NF deployment, between a factor of 1.8-10.9× in CPU-based deployments. To the best of our knowledge, we are the first ones to analyze this phenomenon. Table 2 summarizes the comparison among existing systems.

In the following, we tackle the following question: *"Can we design an NF system that is power-efficient, easy-to-deploy, and supports advanced network functions?"*

3 A Queue-based Packet Storage

We envision a new approach for increasing the throughput of widely-deployed *shallow* NF packet processors. In our design, called QUEUE-MEM, we process headers on an external NF processor and we store packet payloads on the per-port queues of an ASIC switch. Switch manufactures design these queues to buffer packets whenever the rate of incoming packets directed towards an outgoing port exceeds the rate at which packets could be forwarded on that port, *e.g.*, during a traffic incast, congestion events, or differences in the input/output link speeds. The memory allocated to such queues is large, *i.e.,* 20 MB on 16-nm 3.2-Tbps switches and 64 MB on 7-nm 12.8-Tbps switches [19]. To put things into perspective, 5 MB of memory can be used to store payloads for up to 40 μs when

Processing	Network Throughput	Power Efficiency (W/Tbps)	Advanced NFs?
PayloadPark [18]	half ✗	5662	yes ✓
Ribosome [49]	half ✗	1996	yes ✓
Tiara [61]	half ✗	1402	yes ✓
nicmem [46]	half ✗	937	yes ✓
QUEUE-MEM	**full** ✓	**517**	**yes** ✓
ASIC-only	full ✓	134	no ✗

Table 2: Comparison among existing systems. "Half" throughput means that only half of the hardware switch throughput can be theoretically achieved.

receiving 1 Tbps of traffic.

We first explore the design space to examine where the payload can be stored within the switch without relying on external resources. We then present our approach for storing payloads in a queue by leveraging the Advanced Flow Control capability of new-generation ASIC switches, which enables per-queue pausing/resuming directly from the data plane [32].

3.1 Storing Payloads within the Switch

The ideal place to store the payloads is in the switch memory, as it prevents wasting bandwidth on the switch to transmit and retrieve payloads. Within current ASIC switches, there are two main memory areas suitable for storing data, both implemented using the SRAM technology: (i) ingress/egress packet buffers and (ii) match-table resources. Packet buffers are large memory areas utilized by the switch to temporarily store packets when processing them or during congestion events. To the best of our knowledge, packet buffers are not accessible with an API in a programmatic manner in any of the existing switches. Conversely, the match-table memory can be fully controlled by data-plane programmers for allocating match-action tables (*e.g.,* IP routing) or various stateful objects (*e.g.,* registers for network monitoring). One may decide to use the match-table memory for storing and retrieving payloads in the data plane, leveraging existing APIs, as already implemented in PayloadPark [18]. However, approaches based on match-table memory ultimately create *copies* of packets that anyway exist in the packet buffer memory of the switch, thus wasting those scarce memory resources needed to support state for certain packet processing functions. Increasing the SRAM memory on a switch to support the storage of 40μs of packets at 12.6 Tbps would cost $\sim$ 5k$, which is cost ineffective.[4]

An alternative: On-switch FPGA-accessible memory. Some market-available switches integrate both an ASIC and an FPGA within the same hardware box, with certain ports hardwired internally between the two chips [2]. With this setup, one can store payloads in the additional memory provided by the FPGA (*e.g.,* BRAM, HBM). However, this solution comes with two main drawbacks: (i) the effective throughput of the switch is *halved* due to hardwiring *half of the ports* between the ASIC and the FPGA chips and (ii) the increased buying and energy costs introduced by the FPGA, which are significantly higher than ASIC ones.

The approach taken in this paper carefully leverages the already-available packet buffer memory of ASICs to store and retrieve the payloads without a programmable support from the switch (*i.e.,* an API). We leverage the Advanced Flow Control (AFC) capability of new-generation programmable ASICs to meticulously control the egress queues [32]. AFC allows us to pause and resume processing of packets from any queue on the switch with data-plane events.

3.2 Creating a FIFO Buffer using AFC

We now describe the design of QUEUE-MEM, a queue-based packet storage based on the Advanced Flow Control feature of ASICs. This capability is commonly used to implement advanced queuing mechanisms, including PFC [21], with user-controlled parameters. We unconventionally take advantage of the possibility to programmatically pause/resume queues to implement a FIFO buffer inside the switch, holding payloads while their headers are processed on the external NF.

Key-idea: pause/resume queues to buffer payloads. AFC allows programmers to directly control a *single* queue state from the data plane based on packet processing events, *i.e.*, each packet can pause/resume a single queue transmission by setting a specific metadata in the ingress and egress processing pipelines. When the switch receives a packet, we leverage AFC to pause one of the available port queues in order to temporarily buffer the payload. At the same time, the header is forwarded to the external NF. Upon receiving the processed header back on the switch, it resumes the queue where the corresponding payload is buffered. Then, the header is temporarily stored in the egress. Subsequently, when the payload exits the queue, the switch recombines it with its corresponding processed header. The resultant packet is then forwarded to the output port.

Building a short-term FIFO storage. This simple idea allows exploiting the "hidden" buffer memory of the switch to store data in a programmatic manner. However, as the switch relies upon this buffer for forwarding, we can only store data for a short amount of time. Luckily, modern packet processors are capable of handling data in tens of microseconds, making it feasible to store payloads in the switch's packet buffer, without affecting its functioning, during NF processing. As mentioned at the beginning of § 3, 5 MB of memory can be used to store payloads for up to 40 μs when receiving 1 Tbps of traffic. With an external FPGA-based processor of packet headers, the processing latency may reduce by up to 10$\times$ (*i.e.,* 4 μs [61]), thus reducing even further the buffer memory requirements (*i.e.,* 5 MB for 10 Tbps).

Challenge: AFC cannot dequeue a single packet. Ideally, the data-plane would allow specifying the number of

[4] We assume a cost per-MB of a suitable SRAM chip to be roughly 80 $ [10], resulting in $\sim 5k$ $ for 64MB of storage.

dequeued packets upon resuming a queue. In this way, each NF-processed header would be able to instruct the switch to dequeue *one* packet from the front of the queue, corresponding to its previously stored payload. In practice, current implementations of AFC in P4-based switches provide a coarse-grained control of queues' transmission, with no guarantees on the number of dequeued packets. Indeed, if all packets are stored in the same queue, activating it would transmit all the enqueued packets before the next "pause queue" command is issued. The number of dequeued packets therefore depends on the time that it takes to issue a "pause" queue operation after a "resume" one.

Challenge: Delays in activating/deactivating AFC. Even if the metadata to control queue transmission can be set at any point in the programmable pipeline, current implementations of AFC only work at ingress/egress deparser level. This means that an ASIC may act upon the metadata value only when a packet reaches *the end of the pipeline* plus eventual non-deterministic queueing. Consequently, controlling queues from the data plane introduces a delay equal to the time needed to traverse the pipeline. For instance, suppose a processed header enters the switch and needs to resume and then pause again the queue containing its payload. Since each packet can interact with AFC only in deparsers, the header can control the corresponding queue twice: once in the ingress deparser and once in the egress deparser. Hence, after the queue is resumed in the ingress, the header must reach the egress deparser to issue the pause command. This implies that during queueing and egress pipeline traversal, other payloads may also be dequeued. This side effect must be carefully handled by the system to prevent packet drops.

Our idea: Batch-based queue control. Modern ASIC architectures commonly allocate multiple queues per port for implementing QoS traffic policies or advanced scheduling mechanisms [50,52]. In programmable switches, it is possible to specify the destination queue of each packet directly from the data-plane logic. We leverage this possibility in our design to distribute payloads among multiple queues, building batches of packets that are only released when all their corresponding processed headers return from the NF. Unfortunately, existing ASICs do not expose APIs for accessing queues' state (*i.e.,* paused or resumed) and the number of enqueued packets. Therefore, QUEUE-MEM implements a custom data-plane mechanism to track both the number of enqueued payloads and the forwarding state of each queue, and it *carefully* pauses/resumes queues in a way that guarantees batches are released without overlapping with each other. This challenging per-batch design eliminates potential packet drops caused by undesired payloads leakages.

3.3 System Design

We now present details about the QUEUE-MEM design. A high-level overview is shown in Fig. 5.

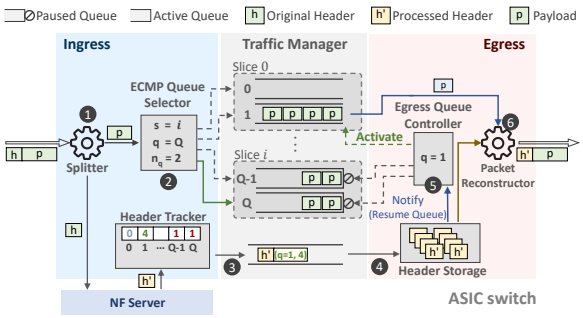


Figure 5: Design overview of QUEUE-MEM.

Splitting incoming packets and enqueueing payloads. At the high level, a packet *pkt* entering the switch ❶ is split into header h and payload p. As already mentioned, the main idea is to create fixed-size batches within the queues. We partition the available per-port queues into distinct slices, each of which is selected by packets based on a hash computed over their 5-tuple. This method reduces the chance that packets belonging to the same flow will be placed into different queues. To maximize the time a queue holding a batch remains unselected ❷ we exploit a round-robin selection method within each slice, which fills each queue one after the other. This mechanism maximizes the time a queue holding a batch remains unselected, ensuring the batch integrity during this period. To correctly recombine payloads with their post-NF-processed headers, we assign to both the header and its payload: (i) an incremental index *idx* referencing the memory array location where the header will be temporarily stored after NF processing, and (ii) an *id* that *uniquely* identifies a packet and ensures that the payload is *not* recombined with any other stored header, which could lead to vulnerabilities or data integrity issues. Moreover, the switch keeps track of the selected queue q for a packet and the number of packets in the queue, namely n_q, for each port. In case $n_q = 1$ (*i.e.,* the first packet of the batch), the switch instructs the ingress AFC to pause the queue q when processing the incoming packet. At this point, the switch forwards the header to the external NF, while it buffers its payload in the selected queue. For any other value of n_q, the switch simply forwards the header to the NF, and it buffers the payload in the queue without activating the AFC.

Storing processed headers and releasing buffered batches. Consider a processed header h' returning from the NF and entering the switch. The ingress pipeline ❸ keeps track of the number of processed headers for each queue, *i.e.,* $n_q^{h'}$. We assign this value to a metadata, that is forwarded to the egress pipeline along with h'. We dedicate a separate port queue (at highest priority) to prioritize headers forwarding, thus minimizing non-deterministic processing times. In the egress, the switch ❹ temporarily stores h' in several register arrays at the location specified by *idx*. Next, the switch checks if $n_q^{h'}$ is equal to the batch size: if so, h' is the last header of the batch, meaning that ❺ the queue q can be safely resumed to release

the payloads. When each payload p_i is dequeued, it reads the corresponding header register at location *idx*, it checks if its *id* is equal to the one stored, and, if so, ❻ it reconstructs the entire packet and forwards it. Note that QUEUE-MEM releases batches once all processed headers are received from the NF, without relying on time-based mechanisms.

Handling header drops. QUEUE-MEM handles possible header drops by employing an additional register s_q that tracks the status of each queue q, *i.e., PAUSED* or *RESUMED*. By combining s_q and the number of processed headers for a queue $n_q^{h'}$, the system is able to detect headers drops and to react upon them to restore a consistent state. So, when the switch receives an input packet *pkt* with $n_q = 1$ (*i.e.,* the first packet of the batch) that should be enqueued in the same queue q, if the previous batch has been correctly released, the value of s_q should be equal to *RESUMED*. Instead, if s_q is equal to *PAUSED*, it means that at least one header of the previous batch has been dropped. In this case, $n_q^{h'}$ is set to 0, s_q is set to *RESUMED*, and the ingress AFC is instructed to release the stalling batch. We cannot immediately re-pause queue q with the same input packet, as it is possible to modify queues' transmission once per ingress/egress pipeline. For this reason, *pkt* cannot be split, as its payload would be released along with the resumed batch. So, after resuming the queue, QUEUE-MEM resubmits the packet to re-process it as a new one. Further optimizations may be possible, *e.g.,* resuming a queue with packets enqueued in a different queue. Since we do not observe any significant negative impact on the effective throughput in our evaluation even under a heavy header loss rate, we leave these optimizations as future work.

3.4 Discussion

Setting the batch size. It is important to select a proper batch size, as this parameter impacts both buffer utilization and the delay experienced by payloads. Opting for a larger batch size allows exploiting a bigger portion of the buffer, but it also results in increased delays for payloads (especially for the first packet in the batch), which must wait for more headers before they can be released. Conversely, selecting a smaller batch size leads to reduced delays for payloads but shortens the time it can be buffered, as the algorithm cycles through the queues more quickly and may interpret delays as dropped headers. We suggest setting the batch size in a conservative manner, assuming a higher expected throughput, thus preventing re-using the same queues before the headers have been processed. By doing so, QUEUE-MEM guarantees that payloads are released *only* when all the corresponding headers have been processed by the NF and returned to the switch. To put things into perspective, consider a switch receiving 100 Gbps of packets with an average size of 1 KB and an external CPU-based NF with a processing time of $40\mu s$. According to the BDP, the switch has to buffer ∼500 payloads in its queues. Suppose that the switch has no processing latency (*i.e.,* all operations are instantaneous) and it can use 30 queues as buffers, the resulting batch size is 16 KB (*i.e.,* 16×1-KB or 32×500-B packets). When looking at the latency overhead induced by batching, consider the aforementioned calculations and an inter-packet gap $ipg = 80ns$. The additional latency introduced on the first packet of the batch is $b_s \cdot ipg = \sim 1.2ns$. At 10 Gbps, the delay is $\sim 8\mu s$, which is acceptable for inter-datacenter or user-facing packet processing. We leave as future work the development of a mechanism to dynamically adjust the batch size by monitoring the switch load.

Supported NFs. The design of QUEUE-MEM imposes a limit on the class of NFs it can support, which is determined by their processing time. Since the switch provides only limited buffer capacity, long NF processing times can exceed what the switch can support. For example, if an NF takes a second to process a header, then at line rate the switch would need to buffer an entire second of payloads. Based on BDP at 100 Gbps, this corresponds to roughly 12 M payloads, which is about 11 GB if each payload is 1 KB. Such a requirement is far beyond the buffer capacity of current-generation switches [27, 36, 58]. Nevertheless, QUEUE-MEM targets datacenter environments, where NFs typically have processing latencies on the order of 100 μs [33, 61].

Deployment model. Two key features enable a simple deployment of QUEUE-MEM: (i) it does not require any external (shared or dedicated) resources to store payloads, and (ii) it operates without modifications on the switch hardware or the NF side. This allows QUEUE-MEM to be deployed in a *plug-and-play* manner, overcoming all the operational complexities suffered by Ribosome [49]. In fact, using shared RDMA servers to store payloads implies that the switch must be a Top-of-Rack node in a datacenter, while QUEUE-MEM is a self-contained box that can be deployed (together with a directly-connected NF server) in any location of a fabric (see App. B) or a wide area network (with no large pools of shared servers). Recall that QUEUE-MEM adds a minimal amount of latency in the order of tens of microseconds due to batching payloads, which is still reasonable even for NFs processing intra-datacenter traffic, and negligible for other types of traffic (*e.g.,* user facing).

Implementing QUEUE-MEM on different platforms. We present our system design based on the Tofino 2 ASIC architecture. However, QUEUE-MEM can be also implemented on different programmable hardware architectures that support (i) packet trimming and (ii) per-queue control (such as PFC). Notable alternatives include state-of-the-art switches from Juniper [58], Broadcom [1], and Nvidia Mellanox [43], which offer the necessary features and have packet buffer pools comparable in size to that of the Tofino 2 ASIC [27, 36, 58].

4 Implementation

We implement QUEUE-MEM in P4_16 language (≈ 1420 lines of code), and compile it for the Intel Tofino 2 ASIC [23]

Resource	Usage
Stages	20
SRAM	24.30%
TCAM	0.60%
VLIW Instruction	8.00%
Exact Match Crossbar	8.70%
Ternary Match Crossbar	1.10%

Table 3: ASIC resources used by QUEUE-MEM.

using P4 Studio 9.11.1. We will publicly release all the code.

ASIC resources usage. Table 3 shows the additional ASIC resources consumed by QUEUE-MEM based on the Tofino 2 compiler's output. The implementation occupies a single pipe, leaving space for including additional user-defined logic. Overall, QUEUE-MEM consumes a negligible amount of VLIW Instructions, Match Crossbar, and TCAM. For SRAM usage, we allocate $16K \times 15 \times 4B$ register entries to store headers, which suffice to sustain 1.4 Tbps of input traffic used throughout the evaluation. A portion of the SRAM is also allocated for tables and registers that manage queues' states.

Data plane. QUEUE-MEM's data-plane implementation follows the design of § 3. Packet splitting is achieved by leveraging the packet mirroring and truncation capabilities of the switch ASIC [22]. We provision compiler's flags to set the split threshold (*i.e.,* size at which packets remain "unsplit") and the size of registers used for storing headers.

Control plane. Our QUEUE-MEM implementation introduces a suite of P4-Runtime APIs designed to facilitate runtime system configuration. Such APIs enable users to selectively activate QUEUE-MEM on specific switch's ports, giving also the possibility to dynamically manage payload buffering across specific queues, allowing the remaining ones to implement different traffic policies. In this way, network architects can configure the number of QUEUE-MEM's queues, the number of queues' slices and the size of payloads' batches, according to the workload, leaving enough resources for other switch operations. Operators are also allowed to filter a subset of NF traffic as "non-splittable".

5 Evaluation

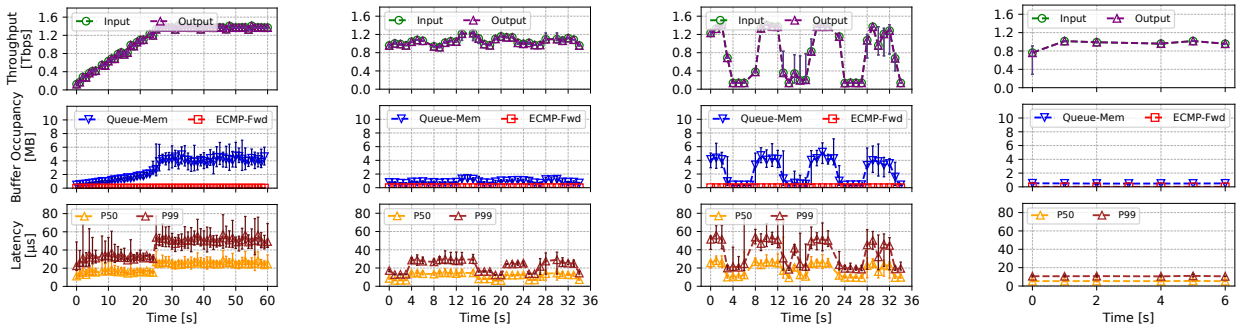
In this section, we assess the performance achievable by QUEUE-MEM. We will release all the code, including scripts for reproducibility. We aim to answer seven main questions:
Q1 *Does* QUEUE-MEM *handle different traffic patterns?*
Q2 *Does* QUEUE-MEM *recover from header drops?*
Q3 *Does* QUEUE-MEM *support advanced NFs?*
Q4 *Does* QUEUE-MEM *interfere with background traffic?*
Q5 *Does* QUEUE-MEM *affect the end-to-end latency?*
Q6 *Does* QUEUE-MEM *affect per-flow packet ordering?*
Q7 *Does* QUEUE-MEM *support real workloads?*
App. D provides additional evaluation results.

Testbed setup. QUEUE-MEM's data plane is deployed on a 32×400 Gbps Edgecore AS9516-32D with Intel Tofino 2 ASIC [24]. The switch forwards traffic based on ECMP. Fourteen of its ports are connected to a 32×100 Gbps Edgecore Wedge100BF-32X with Intel Tofino ASIC. Since each pipe of our Tofino 2 switch consists of eight ports, we allocate 7 of these 14 ports on one pipe, and the remaining 7 ports on another pipe. We then connect the external NF packet processor on the eighth port of each pipe (*i.e.*, two ports in total). The testbed is wired with 100 Gbps links. As our experiments utilize a quarter of the port capacity, we only exploit a quarter of the available port queues (including the priority queue dedicated to headers) to ensure fairness. Note that our testbed represents a prototype implementation based on our resources, yet one could expand the testbed by (i) using more pipes and (ii) moving to 400 Gbps links. App. C depicts a schema of the testbed.

External packet processor. We use two different types of external NF packet processors devices. Unless specified, the QUEUE-MEM switch sends the header to an *external switch* (*i.e.,* the 32×100G Tofino switch), which simply returns the header back on the same incoming port. The external switch introduces $\sim 3\mu$s of NF latency, similar to FPGA-based NF processors [61]. We use this mechanism to emulate a deployment scenario with an external FPGA-based NF (*e.g.,* Tiara [61]). For evaluating the ability of QUEUE-MEM in handling advanced NFs, we connect QUEUE-MEM to a real-world external CPU-based NF processor (more technical details in Q3), which incurs $\sim 15\mu$s of average latency.

Workload generation. To inject different loads, we use two generators: (i) the built-in Packet Generator on the 32-port Tofino and (ii) a server equipped with Intel®Xeon®Gold 6140 CPU @ 2.30 GHz, and Nvidia Mellanox ConnectX-5 NICs [42], also connected to the 32-ports switch. The switch multicasts the incoming traffic to all the 14 ports. We generate synthetic traffic using the 32-ports switch and we inject real-world CAIDA traces [8] from the server using FastClick [3]. To evaluate QUEUE-MEM's ability to handle real workloads, we generate traffic using `iperf` [13] and TRex [9]. All the experiments are repeated 10 times. We report mean, minimum, and maximum values for each data point.

Q1 Variable traffic and packet sizes. One key challenge addressed by QUEUE-MEM is the ability to sustain correct forwarding with variable traffic patterns (both in terms of throughput and packet size distribution). We tested the system using four different traces, three synthetic and two real-world traces, CAIDA [8] and MAWI [35] (results for MAWI are similar to CAIDA and shown in App. D). We consider different traffic patterns to generate varying levels of congestion within the switch's buffer. Fig. 6 shows the performance of QUEUE-MEM in such scenarios using a forwarder NF. For each scenario, the upper figures depict the input and output throughput (in Tbps) over the elapsed time (in seconds). The middle figures show how the buffer occupancy in bytes (y-axes) varies over the time (x-axis). To measure the impact of QUEUE-MEM on the buffer occupancy, we also report the buffer occupancy when QUEUE-MEM is



(a) Incremental. (b) Random. (c) Peaks. (d) CAIDA.

Figure 6: QUEUE-MEM throughput (upper), buffer occupancy (middle) and latency (lower) with different input traffic patterns.

disabled and traffic is simply traversing the switch according to ECMP (red line). Finally, the lower figures show how the average and 99th-percentile latencies (y-axes) change over time (x-axis). In Fig. 6a, we constantly increase the input throughput from 200 Gbps to 1.4 Tbps over 27 seconds (using 1.5 KB packets) and then maintain a steady rate of 1.4 Tbps for other 30 seconds. In Fig. 6b, every 3 seconds, we randomly change the input throughput on each port differently, ranging from 10 Gbps to 100 Gbps over 35 seconds. Rates are extracted from a distribution which privileges higher values. Fig. 6c shows how QUEUE-MEM reacts to sudden changes in the rate, going from 100 Gbps to 10 Gbps simultaneously on each port, and vice versa. Fig. 6d investigates the performance of QUEUE-MEM using a real-world trace with packets of different sizes and realistic inter-packet gaps. We replay on each port of the switch a CAIDA trace containing $\sim$16 M flows and an average packet size of 961 B that runs for 6s at 70 Gbps [17]. Overall, QUEUE-MEM efficiently handles input throughput in every scenario, ensuring almost zero packet drops ($< 0.0001\%$), demonstrating that the system is able to support realistic and variable traffic patterns. As for the buffer occupancy, we first observe that the baseline ECMP forwarder does not incur a high buffer utilization, since Internet traffic consists of many small flows and few bursty events [49] that are sustained well by the buffer memory of the switch not consumed to store payloads. QUEUE-MEM introduces an overhead that never exceeds 9 MB (out of the 64 MB available [25]), potentially leaving enough room to absorb sudden traffic bursts or incasts without resources contention even if it would be traversed by more datacenter-like traffic patterns, which may create incasts or large congestion. We observed a sudden increase in buffer occupancy around 1.3 Tbps of processed traffic, which reaches a very *high average per-port utilization* (*i.e.*, >90%) with some inevitable additional overheads in the internal packet scheduler, which does not result in any packet drops. As for latency, we observe that across all scenarios, the average remains below 30 μs, which is within the range of typical NF processing times in datacenters. The latency trend follows the buffer occupancy trend, suggesting that increases in buffer usage lead to latency overheads due to the switch's traffic manager operations. Similarly, the 99th-percentile latency mirrors the average, with values consistently staying under 60 μs, which is in line with previous findings [49].

Q2 QUEUE-MEM is resilient to header drops. To ensure the resiliency of QUEUE-MEM to header drops, we measured the throughput of the system while inducing controlled header drops on the NF. Fig. 7 shows the throughput in Tbps (y-axis) while varying the percentage of dropped headers on the NF (x-axis). We inject the same synthetic trace composed of 1.5 KB packets.The output throughput (purple line) is exactly reduced by the drop percentage set on the NF, demonstrating that the recovery mechanism outlined in § 3.3 works as expected. For instance, with an input throughput of 1.4 Tbps and a 50% header drop rate, the output throughput is 700 Gbps. For further evaluation that validates the robustness of the mechanism under adversarial workloads, refer to App. D.

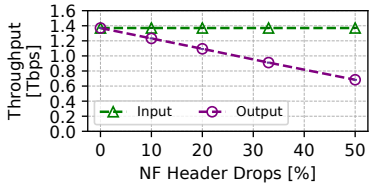


Figure 7: Throughput with deliberate header drops.

Q3 >1 Tbps on a single server. We now evaluate the QUEUE-MEM performance with a software-based packet processor, implemented using FAJITA [17]. We deploy FAJITA on a server equipped with 2× Intel®Xeon®Gold 6346 CPU @ 3.10 GHz, 256 GB of RAM and 2× Nvidia Mellanox Connect-X 6 Dx. NICs are connected to separate CPUs, with a FAJITA instance running on each CPU. Each instance handles traffic from one of the two pipes of Tofino 2. Fig. 8 depicts the output throughput (in Mpps and Tbps) over the input throughput (in Mpps) of two different stateful NF chains: (i) a chain composed of flow counter, stateful load balancer and per-flow rate limiter (blue line) and (ii) a chain composed of load balancer and a header encryptor that computes a 6 B fingerprint using AES (green line). We inject a synthetic trace

with 1.5 KB packets and a multicast factor of 14 on the 32-ports switch. The results show that the entire pipeline can process more than 1 Tbps of input traffic on a single server, which translates to about $\sim$ 100 Mpps of processed headers.

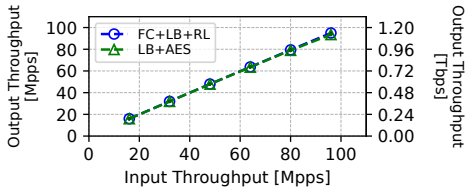


Figure 8: Throughput of advanced NF chains.

Q4 QUEUE-MEM avoids impacting the throughput non-NF-directed traffic. In this experiment, we show that QUEUE-MEM can be configured to avoid interfering with non-NF-directed traffic (*i.e.,* traffic that does not require NF processing and that should be solely routed by the switch). To assess the impact on non-NF-directed traffic, we connect two additional machines with 100 Gbps NICs to the testbed, serving as `iperf` client/server, generating 14 TCP flows at 7.14 Gbps (*i.e.,* a total of 100 Gbps from the client). Each TCP flow is routed through a different input port and forwarded to the server. For the NF-directed traffic, we replay the same CAIDA trace (at 70 Gbps) used in Q1 on each port. To route TCP traffic, we allocate a dedicated queue with a slightly lower priority than the queue used to receive the processed headers in QUEUE-MEM, on the same ports used by QUEUE-MEM to buffer the payloads. Fig. 9 shows how the TCP traffic reacts when NF-directed traffic is introduced (at the 8-second mark), by monitoring both the output throughput (in Tbps) and the TCP congestion window (in MB). The TCP traffic (blue line) remains constant for all the duration of the experiment. The TCP congestion window initially decreases when the 1-Tbps NF traffic suddenly appears at the switch, but quickly recovers to its pre-congestion level. The sum of the NF (red line) and TCP traffic is equal to the input (green line), demonstrating that QUEUE-MEM can handle additional traffic without throughput degradation.

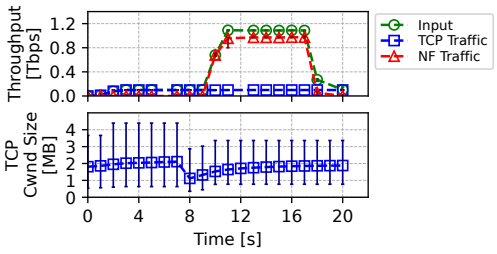


Figure 9: QUEUE-MEM throughput with NF and TCP traffic.

Q5 QUEUE-MEM does not affect end-to-end latency of "unsplit" traffic. In this experiment, we wonder whether buffering payloads on port queues impacts the latency of traffic that should not be split or is not destined to the NF server, even if relying on dedicated queues. To address this question, we measure the latency of traffic returned to the generator after being processed by QUEUE-MEM through the external FAJITA packet processor (described in Q3), which runs a stateful load balancer. Fig. 10 illustrates the end-to-end traffic latency in microseconds (y-axis) over time in seconds (x-axis). The injected traffic (green line) is another CAIDA trace lasting 7 seconds with a throughput of 95 Gbps. We modify the CAIDA trace so that the 66% of traffic is destined to the NF, while the remaining 33% is assigned to a specific IP subnet that serves as "background" traffic that only needs to be routed. As in Q4, we allocate a dedicated queue to route such traffic. We observe that the latency of the non-NF traffic, which only needs to be routed (blue line), remains unaffected by QUEUE-MEM. Similarly, packets not split by QUEUE-MEM (*i.e.,* packets with a payload smaller than 64 B) exhibit a stable latency of approximately 10 μs (orange line), which correspond to the processing latency introduced by the NF. The split traffic (red line) shows a median latency that ranges from 12 to 15μs, implying that QUEUE-MEM's processing adds a latency overhead of about 2-5 μs.

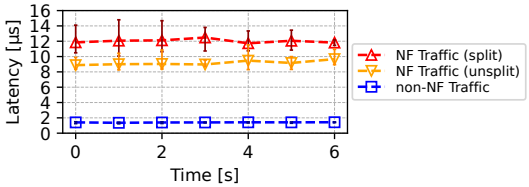
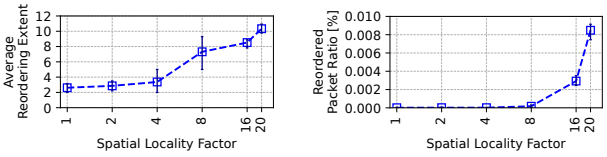


Figure 10: Median RTT latency for different types of traffic.

Q6 QUEUE-MEM introduces a negligible amount of packet reordering. To assess the level of packet reordering introduced by QUEUE-MEM, we measure two key metrics specified in RFC 4737 [39]: (i) reordering extent (*i.e.,* the maximum distance between two packets of the same flow after being processed by QUEUE-MEM) and (ii) percentage of out-of-order packets. These measurements are conducted while injecting 1 Tbps of traffic into the system. To evaluate QUEUE-MEM under varying traffic patterns, we repeat the experiments across different values of Spatial Locality Factor (SLF) metric, *i.e.,* the number of back-to-back packets per flow [16], ranging from 1 to 20. For context, typical datacenter traffic exhibits an average SLF of around 8 [16]. For our analysis, we insert a unique identifier when generating each packet and capture the traffic after being processed by QUEUE-MEM. We then analyze 15 million packets from the pcap file to compute the metrics. The results are presented in Fig. 11. Specifically, Fig. 11a shows how the reordering extent (y-axis) varies with the SLF (x-axis). As expected, a larger SLF, meaning more back-to-back packets from the same flow, leads to greater reordering extent. Nonetheless, even at SLF = 20, QUEUE-MEM maintains a reordering extent of around 10, which is comparable to other state-of-the-art NF processors [60]. Fig. 11b presents the percentage of out-of-order packets (y-axis) as a function of SLF (x-axis). The trend is consistent with the reordering extent: a higher

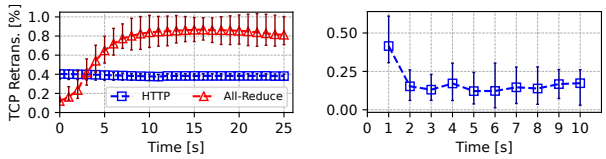
SLF results in a slightly higher percentage of reordering. However, across all scenarios, packet reordering remains minimal, with the worst case still under 0.01%.



(a) Reordering Extent. (b) Reordering Percentage.

Figure 11: QUEUE-MEM packet reordering analysis.

Q7 QUEUE-MEM sustains stable TCP performance under real workloads. To demonstrate the capability of QUEUE-MEM in handling real-world workloads, we conduct two additional experiments using actual TCP traffic. In both experiments, traffic is processed by QUEUE-MEM's switch, which sends the header to an external FAJITA NF running a flow counter, before recombining it with the payload and forwarding it to the server. In the first experiment, we use TRex [9] to inject two distinct workloads: (i) user traffic composed of HTTP POST requests, with 5 M connections at 80 Gbps and (ii) TCP-based All-Reduce collective job at 75 Gbps, representative of ML workloads (the trace is derived from the MVAPICH benchmark [56]). Since we do *not* observe any throughput degradation compared to a baseline forwarder that does not split packets, we focus on TCP retransmissions. Fig. 12a shows how the percentage of TCP retransmissions (y-axis) varies over time (x-axis). The results indicate that QUEUE-MEM manages the TCP connections with an acceptable level of packet reordering, maintaining retransmission rates around 0.4% for HTTP traffic (blue line) and approximately 0.8% for the All-Reduce workload (red line). In the second experiment, we evaluate QUEUE-MEM under standard Linux TCP stack. We use `iperf` [13] to inject 100 Gbps of traffic into QUEUE-MEM for 10 s, using 4 KB packets to emulate a typical datacenter MTU. Fig. 12b shows the percentage of TCP retransmissions (y-axis) over time (x-axis). The results demonstrate that QUEUE-MEM sustains stable TCP connections while maintaining retransmissions consistently around 0.2% throughout the experiment.



(a) TRex. (b) `iperf`.

Figure 12: QUEUE-MEM handling real workloads.

6 Related Work

We now discuss any related work not mentioned in § 2.

Dedicated external devices. Most systems send the entire packet to the NF processor [12,20,29,34,44,47,64]. Contrary, QUEUE-MEM only transmits headers, minimizing the amount of resources needed to run complex functions at Tbps speed.

Network functions directly on ASIC. Supporting network functions entirely in the data plane of an ASIC switch would result in high throughput, low-latency, and low-energy consumption. P4QRS [60] introduces a technique that utilizes switch queues to optimize packet processing tasks that require multiple computation rounds through recirculation, allowing stateless and complex network functions to be implemented directly on the ASIC. Several existing approaches, such as SilkRoad [37], Cheetah [4] SwiSh [62], Elastic-Sketch [59] and Sketchovsky [40] propose to store the entire state required to operate a specific NF entirely on the memory available on an ASIC switch. However, the amount of memory available to store per-flow state on existing high-speed ASIC chips is constrained and may not be sufficient for NF applications that handle large amounts of flows. Moreover, realizing complex logic directly on ASIC is a challenging task. Hence, we believe that for complex tasks the best choice is to exploit the high-throughput of ASIC switches, while delegating complex tasks to a dedicated external processors (*e.g.*, CPU, FPGA).

Disaggregated processing pipelines. Other works propose to disaggregate the processing pipeline to overcome the constraints imposed by ASIC switches or to obtain better performance by combining specialized hardware accelerators. TurboSwitch [48] buffers payloads via recirculation or multicast, but both cause unnecessary congestion. TEA [30] is the first framework to implement NFs using a programmable switch and leveraging additional RDMA-accessible memory to store per-flow state. Gallium [63] enables offloading a part of the NF processing on the switch, but complex processing still needs to be executed on the NF servers. ExoPlane [31] and Flightplan [55] integrate programmable switches with other dedicated programmable network hardware to deploy stateful NFs that require an amount of memory resources that is not available on ASIC chips. These approaches are orthogonal to QUEUE-MEM. If some network functions can be offloaded to ASIC switches, *e.g.,* to process packets belonging to heavy-hitter flows, the same approach could be deployed alongside QUEUE-MEM.

7 Conclusion

In this work, we observed that the payload-storage location severely impacts the energy consumption of an NF deployment. We introduced QUEUE-MEM, an energy-efficient system that leverages the existing packet buffer memory of ASIC switches to store packet payloads in a programmatic manner. QUEUE-MEM is the first ever advanced NF system that processes >1 Tbps of traffic, relying solely on a switch and a single CPU-based server. Our work spurs a question on *how complex* would it be for today's switch manufacturers to expose an API for controlling the forwarding of packets in a highly programmable manner, *e.g.,* dequeueing single packets using an identifier. Our work demonstrates the massive energy and performance gains achievable by such approaches.

Acknowledgments

We would like to thank our shepherd Boris Pismenny and the anonymous reviewers for their insightful comments and suggestions on this paper. This work has been partially supported by Vinnova (the Sweden's Innovation Agency), the Swedish Research Council (agreement No. 2021-04212), and KTH Digital Futures. This work has been partially supported by Knut and Alice Wallenberg Foundation (Wallenberg Scholar Grant for Prof. Dejan Kostić). This work has been partially supported by PRIN Project no. 2022TS4Y3N – EXPAND.

References

[1] P. Adrian, D. Dragos, H. Mark, N. Georgios, L. Jeongkeun, and R. Costin. Implementing packet trimming support in hardware, 2022.

[2] APS Networks. APS2140D Datasheet, 2022. https://www.aps-networks.com/wp-content/uploads/2022/06/APS2140D-Datasheet.pdf.

[3] T. Barbette, C. Soldani, and L. Mathy. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[4] T. Barbette, E. Wu, D. Kostić, G. Q. Maguire, P. Papadimitratos, and M. Chiesa. Cheetah: A High-Speed Programmable Load-Balancer Framework With Guaranteed Per-Connection-Consistency. *IEEE/ACM Transactions on Networking*, pages 1–14, 2021.

[5] Broadcom. Broadcom Trident 3 Platform Performance Analysis, 2019. https://docs.broadcom.com/doc/12395356.

[6] Broadcom. Broadcom Tomahawk 5: Powering the World's Highest Performance AI/ML Clusters, 2022. https://www.youtube.com/watch?v=A30nk8_e1WA&t=64s.

[7] T. Caiazzi, M. Scazzariello, and M. Chiesa. Millions of low-latency state insertions on asic switches. *Proc. ACM Netw.*, 1(CoNEXT3), nov 2023.

[8] CAIDA. The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019) - CAIDA, 2022. https://www.caida.org/catalog/datasets/passive_dataset/.

[9] Cisco. TRex - Realistic Traffic Generator, 2025. https://trex-tgn.cisco.com.

[10] DigiKey. IBM 04368CBLBC-28 SRAM Chip, 2024. https://www.digikey.com/en/products/detail/rochester-electronics-llc/04368CBLBC-28/12593844.

[11] D. Dutt. *BGP in the Data Center*. O'Reilly Media, 2017. page 11, paragraph "Connectivity to the External World".

[12] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 523–535, USA, 2016. USENIX Association.

[13] ESnet: Energy Sciences Network. iperf3, 2024. https://github.com/esnet/iperf.

[14] Facebook. Katran Load Balancer, 2021. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[15] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić. Make the most out of last level cache in intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[16] H. Ghasemirahni, T. Barbette, G. P. Katsikas, A. Farshin, A. Roozbeh, M. Girondi, M. Chiesa, G. Q. Maguire Jr., and D. Kostić. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 807–827, Renton, WA, Apr. 2022. USENIX Association.

[17] H. Ghasemirahni, A. Farshin, M. Scazzariello, G. Q. Maguire, D. Kostić, and M. Chiesa. Fajita: Stateful packet processing at 100 million pps. *Proc. ACM Netw.*, 2(CoNEXT3), aug 2024.

[18] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. Seltzer. *Parking Packet Payload with P4*, page 274–281. Association for Computing Machinery, New York, NY, USA, 2020.

[19] V. Gurevich and A. Fingerhut. P4_16 Programming for Intel Tofino using Intel P4 Studio, 2021. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf.

[20] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-Accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 195–206, New York, NY, USA, 2010. Association for Computing Machinery.

[21] IEEE. IEEE 802.1Qbb or PFC, "Priority-based Flow Control", 2011. http://www.ieee802.org/1/pages/802.1bb.html.

[22] Intel. $P4_{16}$ Intel Tofino Native Architecture – Public Version, 2021. https://raw.githubusercontent.com/barefootnetworks/Open-Tofino/master/PUBLIC_Tofino-Native-Arch.pdf.

[23] Intel. Tofino®Series, 2022. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[24] Intel. Tofino®2, 2023. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[25] Intel. Tofino®2 12.8 Tbps, 20 stage, 4 pipelines, 2023. https://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html.

[26] R. Jacob, J. Lim, and L. Vanbever. Does rate adaptation at daily timescales make sense? In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Jim Warner. Tomahawk-3, 2019. https://people.ucsc.edu/~warner/Bufs/tomahawk-3.html.

[28] L.-E. Jonsson, K. Sandlund, and G. Pelletier. The RObust Header Compression (ROHC) Framework. RFC 5795, Mar. 2010.

[29] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, Renton, WA, 2018. USENIX Association.

[30] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 90–106, New York, NY, USA, 2020. Association for Computing Machinery.

[31] D. Kim, V. Sekar, and S. Seshan. ExoPlane: An operating system for On-Rack switch resource augmentation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1257–1272, Boston, MA, Apr. 2023. USENIX Association.

[32] J. Lee. Advanced Congestion & Flow Control with Programmable Switches, 2020. https://opennetworking.org/wp-content/uploads/2020/04/JK-Lee-Slide-Deck.pdf.

[33] T. Lévai, F. Németh, B. Raghavan, and G. Retvari. Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 633–649, Santa Clara, CA, Feb. 2020. USENIX Association.

[34] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[35] MAWI Working Group. MAWI Working Group Trace Info (Apr 14th, 2025), 2025. https://mawi.wide.ad.jp/mawi/samplepoint-F/2025/202504141400.html.

[36] Mellanox. SN3000 Series, 2020. https://www.exclusive-networks.com/nl/wp-content/uploads/sites/21/2021/02/BR_SN3000_Series.pdf.

[37] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 15–28, New York, NY, USA, 2017. Association for Computing Machinery.

[38] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, 2016.

[39] A. Morton, G. Ramachandran, S. Shalunov, L. Ciavattone, and J. Perser. Packet Reordering Metrics. RFC 4737, Nov. 2006.

[40] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste. Sketchovsky: Enabling ensembles of sketches on programmable switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1273–1292, Boston, MA, Apr. 2023. USENIX Association.

[41] Nvidia. ConnectX-7 Specifications, 2023. https://docs.nvidia.com/networking/display/connectx7vpi/specifications#src-2572915597_Specifications-MCX715105AS-WEATSpecifications.

[42] NVIDIA Networking. NVIDIA Mellanox ConnectX-5 adapters, 2021. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/.

[43] V. Olteanu, H. Eran, D. Dumitrescu, A. Popa, C. Baciu, M. Silberstein, G. Nikolaidis, M. Handley, and C. Raiciu. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 761–777, Renton, WA, Apr. 2022. USENIX Association.

[44] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud Scale Load Balancing. *SIGCOMM Comput. Commun. Rev.*, 43(4):207–218, aug 2013.

[45] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat. *SCION: A Secure Internet Architecture*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[46] B. Pismenny, L. Liss, A. Morrison, and D. Tsafrir. The Benefits of General-Purpose On-NIC Memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, pages 1–18. Association for Computing Machinery, New York, NY, USA, Feb. 2022.

[47] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, Feb. 2019. USENIX Association.

[48] M. Scazzariello, T. Caiazzi, and M. Chiesa. Deliberately congesting a switch for better network functions performance. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pages 1–6, 2024.

[49] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa. A High-Speed stateful packet processing approach for tbps programmable switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1237–1255, Boston, MA, Apr. 2023. USENIX Association.

[50] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, Apr. 2018. USENIX Association.

[51] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, Feb. 2020. USENIX Association.

[52] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, Feb. 2020. USENIX Association.

[53] Smart-me. Intelligent Plug Datasheet, 2024. https://static.digitecgalaxus.ch/Files/2/3/6/8/5/2/5/smart-me_Datenblatt_ch.pdf.

[54] C. H. Song, X. Z. Khooi, R. Joshi, I. Choi, J. Li, and M. C. Chan. Network load balancing with in-network reordering support for rdma. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 816–831, New York, NY, USA, 2023. Association for Computing Machinery.

[55] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 571–592. USENIX Association, Apr. 2021.

[56] The Ohio State University. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot, 2025. https://mvapich.cse.ohio-state.edu/benchmarks/.

[57] Xilinx. Alveo U50 Data Center Accelerator Card Data Sheet, 2020. https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds965-u50.pdf.

[58] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi. Using trio: juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 633–648, New York, NY, USA, 2022. Association for Computing Machinery.

[59] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 561–575, New York, NY, USA, 2018. Association for Computing Machinery.

[60] Y. Yoshinaka, Y. Koizumi, J. Takemasa, and T. Hasegawa. High-throughput stateless-but-complex packet processing within a tbps programmable switch. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pages 1–12, 2024.

[61] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, apr 2022. USENIX Association.

[62] L. Zeno, D. R. K. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, Renton, WA, Apr. 2022. USENIX Association.

[63] K. Zhang, D. Zhuo, and A. Krishnamurthy. Gallium: Automated software middlebox offloading to 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*, SIGCOMM '20, page 283–295, New York, NY, USA, 2020. Association for Computing Machinery.

[64] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, Nov. 2020.

[65] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

A Methodology for the Power Usage Analysis

We now outline the approach we used for calculating the power usage analysis presented in § 2. We derive our calculations from the data provided in Table 1, describing the formulas we use to compute the power usage of each analyzed architecture (see Fig. 3).

We refer to the ConnectX-7 hardware datasheet for RNIC energy consumption [41], in the following *rnic*. Since RNICs do not implement load-dependent power-saving mechanisms, we assume a constant power consumption and do not scale it with the Mpps. For servers, instead, we consider the optimal case where the CPU power consumption is proportional to the throughput in Mpps. We conducted three distinct measurements, each lasting 30 minutes: (i) idle power usage (in the next, $server_{idle}$); (ii) power usage while processing 100 Gbps of traffic with 1.5 KB packets (corresponding to roughly 8 Mpps) using a load-balancer NF implemented with FastClick [3] (in the next, $server_{NF}^{1.5KB}$); (iii) power usage while processing 100 Gbps of traffic with 64 B packets (corresponding to roughly 142 Mpps) using the same load-balancer NF (in the next, $server_{NF}^{64B}$). Finally, we subtract *rnic* from each of these measurements to isolate the server contribution excluding RNIC consumption.

We can now compute the packet processor's additional consumption when processing full-sized 1.5 KB packets:

$$NF_{extra}^{1.5KB} = \frac{server_{NF}^{1.5KB} - server_{idle}}{Mpps_{1.5KB}^{100Gbps}} \quad (1)$$

We also account for the additional NF power consumption when processing 64 B packets, which represents the case of handling only packet headers:

$$NF_{extra}^{64B} = \frac{server_{NF}^{64B} - server_{idle}}{Mpps_{64B}^{100Gbps}} \quad (2)$$

In the following, we describe how we obtained the formula for each system depicted in Fig. 3. As discussed in § 2, we assume that all link capacities are set at 100 Gbps with input packets' size of 1.5 KB.

Baseline. To process traffic, a traditional packet processing deployment would require a dedicated NF server for each input port of the switch, which translates into the following:

$$Baseline_{power} = ASIC_{power} + (NF_{power} \cdot n_{NF}) \quad (3)$$

The first term represents the programmable ASIC consumption varying the number of input ports (in_{ports}) and their input throughput in Mpps (in_{tput}), assuming uniform traffic distribution across all ports:

$$ASIC_{power} = ASIC_{base} + ((ASIC_{extra} \cdot in_{tput}) \cdot ASIC_{ports}) \quad (4)$$

The values of $ASIC_{base}$ and $ASIC_{extra}$ are taken from Table 1. This value accounts both input and NF ports:

$$ASIC_{ports} = in_{ports} + n_{NF} \quad (5)$$

The second term considers the additional consumption of the NF servers, with one of them allocated for each input port:

$$NF_{power} = server_{idle} + (NF^{1.5KB}_{extra} \cdot in_{tput}) + rnic \quad (6)$$

$$n_{NF} = in_{ports} \quad (7)$$

PayloadPark. As for the baseline, we have that the PayloadPark power consumption is composed of two terms:

$$PayloadPark_{power} = ASIC_{power} + (NF_{power} \cdot n_{NF}) \quad (8)$$

Where $ASIC_{power}$ is computed as in (4). PayloadPark, by storing 160 B of payload on the switch, reduces the value of n_{NF} since it increases the maximum throughput achievable by each NF server. We compute the number of NF servers as:

$$n_{NF} = \left\lceil \frac{in_{tput} \cdot in_{ports}}{Mpps^{100Gbps}_{1.34KB}} \right\rceil \quad (9)$$

Where $Mpps^{100Gbps}_{1.34KB}$ is the number of millions of packets per second in 100 Gbps, considering 1.34 KB packets.

We calculate the NF power consumption assuming that each NF server processes $Mpps^{100Gbps}_{1.34KB}$:

$$NF_{power} = server_{idle} + (NF^{1.5KB}_{extra} \cdot Mpps^{100Gbps}_{1.34KB}) + rnic \quad (10)$$

nicmem. nicmem does not perform packet splitting on the switch, and full-size packets are sent to each NF server. But, the payloads are stored in NIC memory while only the headers are forwarded to the packet processor, thereby reducing CPU load. Prior research has demonstrated that a single CPU socket can handle up to ∼178 Mpps of 64 B packets [17]. Hence, increasing the number of NICs on the same server could enhance overall power efficiency. Based on this insight, we consider utilizing additional NIC slots within a single server. The power usage is computed as:

$$nicmem_{power} = ASIC_{power} + (NF_{power} \cdot n_{NF}) \quad (11)$$

Where $ASIC_{power}$ is computed as in (4) and $ASIC_{ports}$ is the same as (5). Since we consider that each NF server can utilize n_{nic} NICs, the number of NF servers is computed as:

$$n_{NF} = \left\lceil \frac{in_{ports}}{n_{nic}} \right\rceil \quad (12)$$

We can now compute NF_{power} as:

$$NF_{power} = server_{idle} + ((NF^{64B}_{extra} \cdot in_{tput}) \cdot n_{nic}) + (rnic \cdot n_{nic}) \quad (13)$$

We use NF^{64B}_{extra} to represent the CPU overhead, as nicmem processes only packet headers on the CPU.

Tiara. Tiara is a powerful load balancer system that reroutes packets from a switch to FPGAs (for the fast path) and x86 servers (for the slow path) performing per-packet load balancer calculations. In their evaluation, authors deploy Tiara using an ASIC switch, 10 FPGAs and one CPU. So, the formula for the power consumption is the following:

$$Tiara_{power} = ASIC_{power} + (server_{power} \cdot n_{server}) + (FPGA_{power} \cdot n_{FPGA}) \quad (14)$$

The total switch consumption is:

$$ASIC_{power} = ASIC_{base} + (ASIC_{extra} \cdot ((in_{ports} \cdot in_{tput}) + (n_{FPGA} \cdot in^{90\%}_{tput}) + (n_{server} \cdot in^{10\%}_{tput}))) \quad (15)$$

Since the majority of input traffic is processed by the fast path, we consider that FPGAs process 90% of the input traffic ($in^{90\%}_{tput}$), while CPUs process 10% of the input traffic ($in^{10\%}_{tput}$).

For the same reason, the $server_{power}$ is computed by considering that the CPU processes $in^{10\%}_{tput}$:

$$server_{power} = server_{idle} + (NF^{1.5KB}_{extra} \cdot in^{10\%}_{tput}) + rnic \quad (16)$$

The value of $FPGA_{power}$ is taken from Table 1.

Ribosome. Ribosome splits packets on the switch, but it needs several RDMA servers to store payloads. Therefore, the total power usage is computed as:

$$Ribosome_{power} = ASIC_{power} + (NF_{power} \cdot n_{NF}) + (RDMA_{power} \cdot n_{RDMA}) \quad (17)$$

Since only one NF server is needed to process 1 Tbps of input traffic, we consider $n_{NF} = 1$ in our analysis. Moreover, as stated in the paper [49], each RDMA server can process up to 75 Gbps of payloads. Therefore, considering that the total throughput to be managed by the RDMA servers is:

$$tput_{RDMA} = \frac{(in_{tput} \cdot in_{ports}) \cdot (1500 \cdot 8)}{1024} \quad (18)$$

The number of RDMA servers is computed as:

$$n_{RDMA} = \left\lceil \frac{tput_{RDMA}}{75} \right\rceil \quad (19)$$

Aside the input ports and the one directed to the NF, Ribosome wastes an additional port for each RDMA server. The final switch consumption is therefore:

$$ASIC_{power} = ASIC_{base} + (ASIC_{extra} \cdot ((in_{ports} + n_{RDMA} + n_{NF}) \cdot in_{tput})) \quad (20)$$

Note that, for the ASIC power consumption of the NF server port on the switch, we chose to consider it as $n_{NF} \cdot in_{tput}$ rather than $n_{NF} \cdot in_{tput} \cdot in_{ports}$, since the power consumption of a switch, being designed to operate at line rate, should not be heavily influenced by the number of processed pps.

The power usage of Ribosome on the NF server is:

$$NF_{power} = server_{idle} + NF^{64B}_{extra} \cdot (in_{tput} \cdot in_{ports}) + rnic \quad (21)$$

For RDMA servers contribution, we debated whether to include the idle CPU power of RDMA servers in Ribosome's power calculations, considering Ribosome assumes these servers are always available to run customer applications. On the other hand, any workload consolidation scheme would have to keep them running, so we assume there is a 50% chance that each Ribosome server is kept running specifically to support Ribosome. Additionally, we assume that Ribosome operates at maximum speed on NICs, optimizing resource utilization but limiting the available bandwidth for customers' applications. Thus, we take 1/2 of idle server power into account for power calculations even though Ribosome consumes the entire bandwidth on the NIC. Therefore, we only consider half of the idle CPU power plus the additional consumption induced by the RNIC:

$$RDMA_{power} = \frac{server_{idle}}{2} + rnic \quad (22)$$

QUEUE-MEM. QUEUE-MEM buffers payloads in the switch queues, without requiring any additional dedicated device aside from the NF processor:

$$QueueMem_{power} = ASIC_{power} + (NF_{power} \cdot n_{NF}) \quad (23)$$

As in Ribosome, one NF server is enough to process 1 Tbps of input traffic, so in the analysis we consider $n_{NF} = 1$ and NF_{power} is computed as in (21). By not utilizing external dedicated devices, QUEUE-MEM saves switch's ports, thus reducing the total power consumption:

$$ASIC_{power} = ASIC_{base} + (ASIC_{extra} \cdot ((in_{ports} + n_{NF}) \cdot in_{tput})) \quad (24)$$

B Deployment as an Exit Point-of-Delivery

Fig. 13 shows how QUEUE-MEM can be deployed inside the Internet-facing Point-of-Delivery (PoD) [11] of a datacenter to enhance the performance of a variety of NFs including security firewalls, proxies, NATs, and more. This approach significantly reduces the number of dedicated NF servers required to manage inbound/outbound fabric traffic.

C Experimental Setup

Figure 14 depicts our testbed described in Sec. 5. We utilize a multicast switch right before the QUEUE-MEM to increase the offered load by 14× as we need to examine the performance of QUEUE-MEM at high rates (*i.e.,* 1.4 Tbps in our experiments). Also, note that there are two 100 Gbps links connected to the NF server with two different NUMA nodes to achieve the desired rate in the testbed.

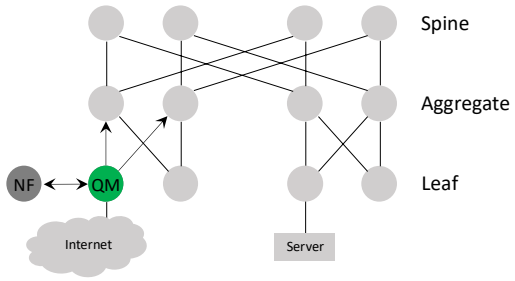


Figure 13: Example of QUEUE-MEM as an exit PoD.

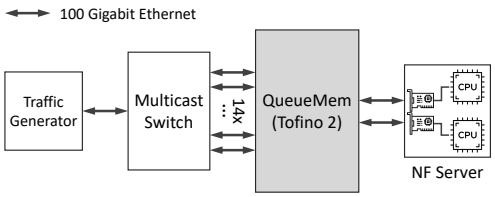


Figure 14: QUEUE-MEM testbed.

D Additional Evaluation

QUEUE-MEM scales linearly w.r.t. the input throughput. Fig. 15 shows the output throughput (in Mpps and Tbps) and the input throughput (in Mpps) of QUEUE-MEM while sending 1.5 KB packets to a forwarder NF. We compare QUEUE-MEM with two systems, a baseline that sends the full-size packet (without splitting) to the external NF and a PayloadPark-like system that can only store 160 B of each payload in the switch memory. As expected, the baseline is capped by the NIC bandwidth and cannot go beyond 100 Gbps. The PayloadPark-like system produces only a small increment in the output throughput, reaching about 115 Gbps. Instead, QUEUE-MEM is able to sustain the 1.4 Tbps of input traffic, while sending about 76 Gbps of headers to the NF, and achieving zero packet loss. This experiment demonstrates that QUEUE-MEM can successfully process >1 Tbps of input traffic without using any external resource to store payloads.

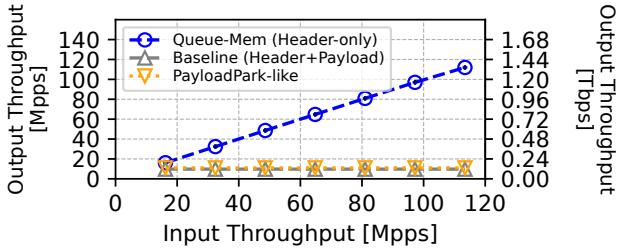
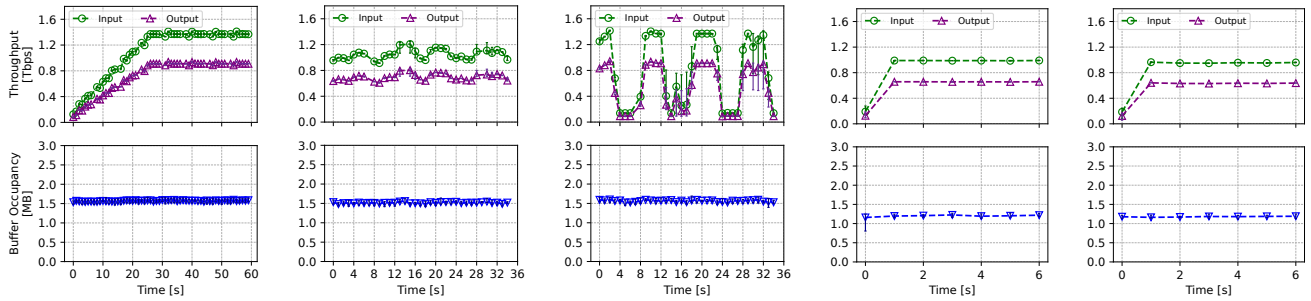


Figure 15: Forwarding NF throughput with 1.5 KB packets.



(a) Incremental. (b) Random. (c) Peaks. (d) CAIDA trace. (e) MAWI trace.

Figure 16: Throughput (upper) and buffer occupancy (lower) with different input traffic patterns and 30% header drops.

One can argue that Ribosome could achieve the same level of performance. But, to sustain 1.4 Tbps of traffic, Ribosome would need at least 19 RDMA servers (each RDMA server can process up to 75 Gbps of payloads [49]), requiring a more complex deployment, end-host modifications on the network stack, strong assumption on the availability of resources, and a higher power usage compared to QUEUE-MEM.

QUEUE-MEM is able to handle drops with different traffic patterns. To ensure that QUEUE-MEM's recovery mechanism correctly works with different traffic patterns, we performed an experiment running the same workloads of Q1 and inducing a controlled 30% NF header drop in each scenario. Fig. 16 depicts the results. The upper figures show the input and output throughput (in Tbps) over the elapsed time (in seconds), highlighting that the recovery mechanism is effective in all the scenarios. Indeed, the output throughput of QUEUE-MEM is inline with the induced drop of 30%. The overall buffer occupancy (lower figures) is lower w.r.t. Q1, since the switch receives back less packets from the NF.

MAWI trace. To further evaluate QUEUE-MEM with real-world traffic, we repeat the experiment from Q1 using a MAWI trace. This trace contains approximately 14 million packets with an average size of 1.4 KB and spans 6 seconds at 70 Gbps. As shown in Fig. 17, the results are consistent with those obtained using the CAIDA trace, demonstrating similar trends in throughput (top figure), buffer occupancy (middle figure), and latency (bottom figure).

Even dropping a packet for each batch, QUEUE-MEM does not suffer any additional throughput degradation. As further proof of the robustness of the QUEUE-MEM algorithm, we measured the throughput in the same scenarios of Q1 and the MAWI trace, while deliberately dropping the last packet of batches on the NF. In this scenario, a batch is released only when a new batch begins to fill the same queue. The batch size for QUEUE-MEM is set to 5. Fig. 18 shows how throughput (y-axis) varies as the percentage of batches experiencing a dropped packet increases (x-axis). The input throughput (green line) remains constant at 1.4 Tbps for all the experiments. The output throughput (purple line) is

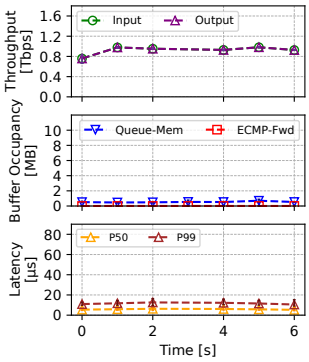


Figure 17: Throughput (upper), buffer occupancy (middle) and latency (lower) while injecting the MAWI trace.

inline with the percentage of batches with drops. Indeed, by dropping the last packet of all the batches, *i.e.,* the 20% of total packets, decreases the output throughput to 1.12 Tbps.

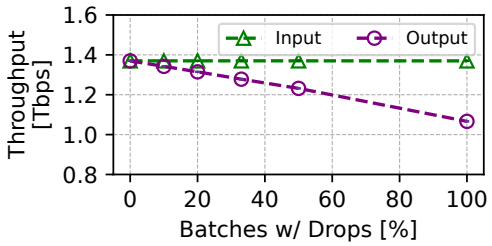


Figure 18: Throughput while deliberately dropping the last header of the batches.