A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches

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

[1]KTH Royal Institute of Technology
[2]Roma Tre University
[3]UCLouvain

Abstract

High-speed ASIC switches hold great promise for offloading complex packet processing pipelines directly in the high-speed data-plane. Yet, a large variety of today's packet processing pipelines, including stateful network functions and packet schedulers, require *storing* some (or all the) packets for short amount of times in a programmatic manner. Such a programmable buffer feature is missing on today's high-speed ASIC switches.

In this work, we present RIBOSOME, a system that extends programmable switches with external memory (to store packets) and external general-purpose packet processing devices such as CPUs or FPGAs (to perform stateful operations). As today's packet processing devices are bottlenecked by their network interface speeds, RIBOSOME carefully transmits only the relevant bits to these devices. RIBOSOME leverages spare bandwidth from any directly connected servers to store the incoming payloads through RDMA. Our evaluation shows that RIBOSOME can process 300G of traffic through a stateful packet processing pipeline (*e.g.*, firewall, load balancer, packet scheduler) by running the pipeline logic on a *single* server equipped with one 100G interface.

1 Introduction

Network Function Virtualization is an essential architectural paradigm of today's networks [32]. Operators create and manage complex packet processing pipelines by combining together Network Functions (NFs), which are then deployed on the infrastructure. Network functions that require simple computations are generally deployed onto *cost-effective* ASIC-based switches, whereas more complex packet processing computations must be deployed on *expensive* general-purpose CPUs or FPGAs due to the inherent difficulty and cost of designing complex ASIC circuits [4]. Unfortunately, the networking stack of general-purpose servers and FPGAs is significantly slower in processing packets than dedicated ASIC hardware counterparts, ultimately increasing the energy footprint and cost of operating a large network.

Deploying network functions that have to manage large amounts of frequently changing *stateful per-flow* information in a cost-effective manner (*i.e.*, entirely on an ASIC switch) has been an elusive goal.

To understand the requirements posed by multi-terabit per second stateful packet processing, we analyze a set of real-world CAIDA traces in the 2013–2019 period [6]. Through a linear regression, we observe that *i)* the number of active flow connections traversing a switch is 120 K for every gigabit of forwarded traffic and *ii)* there are 4 K new flow connections for every gigabit of forwarded traffic. This translates to 385 M active flows and 12.8 M new flow-table insertions per second on a 3.2 Tbps forwarding pipe. With a 17 B flow-state entry (*i.e.*, a 5-tuple + action), as in a Layer-4 load balancer, the memory requirement becomes 6.5 GB, which go beyond the stateful memory that is available on today's ASIC chips.

In this work, we aim at designing a stateful per-flow packet processor system that satisfies the following requirements:

- **Expressiveness**, by supporting a variety of complex stateful logic (*e.g.*, load balancers, packet schedulers).
- **High Throughput**, by achieving superior performance compared to existing expressive designs.
- **Dynamicity**, by supporting very frequent modifications to its stateful data structures.
- **Cost Effectiveness**, by reducing the costs and power consumption for operating this system.

Building a system that supports the above requirements is highly challenging. The expressiveness requirement requires a system to rely (at least to some degree) on general-purpose CPUs or FPGAs. We distinguish between two types of systems that require external resources:

- Systems that use **dedicated** external devices to realize complex NFs. An example within the first category is Tiara [46], a clever load balancer system that reroutes packets from a switch to 16 ports that are connected to FPGAs performing per-packet load balancer calculations. While such solutions are expressive, they are not cost-effective. Half of the devices connected to a 32-port programmable switch are used exclusively to perform stateful packet processing operations.

This reasoning motivates the next type of approaches.

- Systems that rely on **shared** external devices whose resources are primarily used for running customers' applications. Such systems embrace emerging disaggregation paradigms in which applications runs on resources that are combined together on demand. As an example, TEA [21] is the first system to efficiently enlarge the memory of the switch by cleverly crafting RDMA messages to access remote memory on shared CPU-based servers. TEA exploits the well-known large amount of spare of bandwidth and memory resources in datacenters [23]. This design allows operators to make better utilization of the resources available on an external general-purpose server: customers' applications run on general-purpose servers and any spare bandwidth and memory resources are used by the switch to store all the per-flow connection states that are required to process terabits of traffic. Unfortunately, as we show in our motivation section, TEA cannot support expressive network functions, as only the state is stored on external servers while the forwarding rules are applied on the ASIC switch, which does not support advanced logic.

In this work, we present RIBOSOME, a system that is expressive, flexible, cost effective, and achieves high-throughput packets processing. RIBOSOME relies on two fundamental observations. First, in a large fraction of stateful per-flow network functions the bottleneck is the bandwidth between the switch and the packet processor. For example, a load balancer saturates a 100Gbps interface with just 3 CPU cores on a real-world trace [12]. Second, many network functions do not need to analyze the entire packet but only the relatively small portion of a packet containing its headers.[1] To put things into perspective, a load balancer that operates on a 5-tuple (13 bytes) field, would only require receiving 13 B per packet instead of potentially 1.5 KB full size packets.

RIBOSOME relies on dedicated external packet processors to process packet headers while storing packet payloads on shared general-purpose servers without any CPU intervention (*i.e.*, using RDMA technology). More specifically, we leverage the advanced capabilities of emerging high-speed programmable switches to receive packets, split them into headers and payloads, and reconstruct them after the NF processor has updated their headers or re-schedule their transmission. By only processing packet headers, we overcome the bandwidth bottleneck at the dedicated devices, which allows us to process significantly higher numbers of packets on the same dedicated machine. As all data structures are handled by CPUs, we support high numbers of modifications to these data structures.

We motivate this design approach with the following observation: storing & fetching payloads are two operations that only require simple support for writing & reading on a memory. These memory operations are general, making it attractive to offload the storage & fetching of payloads on shared memory resources (*e.g.*, RDMA). Using shared resources to store payloads allow operators to make more efficient utilization of the memory resources existing in a network (such as a datacenter). We then rely on dedicated resources for processing packet headers. In this case, the rationale is that the performance achievable by a stateful network functions highly depends on temporal and spatial factors (*e.g.*, high cache-locality), and is therefore less suitable to be executed on shared resources. Finally, RIBOSOME brings benefits when an NF only needs to inspect a small part of a packet, *e.g.*, a load balancer. RIBOSOME does not bring benefits when the NF requires access to the entire packet (*e.g.*, a deep packet inspection function).

We implement RIBOSOME on an ASIC programmable switch, with FastClick [1] as the NF packet processor, and RDMA to store payloads on other servers. We evaluate RIBOSOME using an empirically-derived multi-100G traffic trace. Our micro-benchmarks show that a general-purpose server processes 70 Mpps on a single server, which would correspond to 560 Gbps of traffic with 1KB average packet size. Based on this estimate, we observe that one would need only three 100G ports on the programmable switch to process 1.6 Tbps of traffic (whereas systems like Tiara would need 16 ports).

We also evaluate the entire system using 4 RDMA servers and a single 100-Gbps dedicated server to process 300 Gbps of traffic. Our results show that the bandwidth requirements at the dedicated server are merely 20 Gbps.

To summarize, our contributions are:

- We propose a new disaggregation-based architecture to circumvent the inherent constraints of high-speed ASIC switches both in terms of logic and memory. We design a system to perform stateful packet processing that carefully splits operations between dedicated and shared resources, where headers are processed by dedicated servers while payloads are stored on shared resources.
- We present the first programmable buffer abstraction that is suitable for Tbps NF packet processing.
- We make the observation that today's deployment of NFs onto general-purpose CPUs is severely bottlenecked by the server bandwidth, thus motivating the splitting of the packets into headers and payloads.
- We demonstrate a single server processes up to 70 M small-size packets per second and the bottleneck moves onto the PCIe. With 3 servers, one could process 210 M packets per second, which is equivalent to roughly 1.6 Tbps of 1KB-size packets. We discuss future optimizations to overcome this limit with future-available hardware.
- We demonstrate in a small testbed that RIBOSOME processes 300 Gbps using a single dedicated NF processor, whose bandwidth requirement is just 20 Gbps.
- We demonstrate a 2.2× speedup over the state of the art for

[1] We do not claim novelty for this observation but rather for the novel trade-off achieved by the design of our packet processing architecture and our fine-grained evaluation.

running complex packet schedulers [11], on similar server hardware. By doing so, we show that we have fundamentally pushed the performance barrier achievable with a combination of a programmable switch and commodity servers.
- We release all our P4 and FastClick code for running RIBOSOME including a high-speed implementation of RDMA on the Tofino programmable switch [40].

2 Motivation

We start this section by quantifying the memory and flow-table update requirements of general ASIC switches. Based on an analysis of real-world traffic traces, we posit that today's (but also next-generation) ASIC switches do not have enough memory to support stateful per-flow NFs. We then quantify and discuss the limitations of the state-of-the-art systems using dedicated resources (*i.e.*, PayloadPark [13] and Tiara) as well as shared resources (*i.e.*, TEA [21]).

ASIC switches have constrained memory. Several existing approaches, such as SilkRoad [31], Cheetah [3] and SwiSh [47], propose to store the entire state required to operate a specific NF entirely on the memory available on the chip of an ASIC switch. However, the amount of memory available to store per-flow state on existing high-speed ASIC chips is a renowned constrained resource that may not be sufficient for NF applications that handle very large amounts of flows. We run some back-of-the-envelope calculations to upper bound the amount of state that could potentially be stored on an ASIC using SRAM technology. Assuming one could use the entire chip area for SRAM memory (*i.e.*, no I/O, no buffers), an 826 mm^2 chip using 7nm technology would only store at most 5GB of flow state in SRAM [7]. We show in this section that this amount of memory would suffice to only store ~10% of the state required on a multi-terabit per second switch. In practice, the amount of memory is below this estimate as typically I/O and buffers occupy roughly 50% of the chip area and some memory is used to implement the packet processing logic. For example, a 16-nanometer high-speed ASIC switch contains 1.5-15.4MB per terabit of forwardable traffic (*i.e.*, 10-100MB of SRAM on a 6.5 Tbps ASIC chip) [14, 29, 31].

To understand the implications of potential future trends on the feasibility of storing per-flow state on ASIC chips, we analyze CAIDA traces in the 2013 – 2019 period for the NYC and CHI monitored links, for which there are publicly available statistics [6].[2] The throughput for these traces ranges from 2 Gbps to 6.5 Gbps. Each trace contains the number of IPv4 and IPv6 forwarded packets, their mean packet size, the trace duration and the flows per second. The flow per second field represents the number of distinct flows in the trace divided by the duration of the trace.[3] We define a flow to be *active* if that flow has sent a packet in the last 30 seconds.[4] If a flow is active, it means the stateful processor must keep state for that flow. In the following analysis, we assume that one needs to deploy a basic load balancer that stores 17 B for each single flow (*e.g.*, 13 B for the 5-tuple and 4 B to store the private IP destination address). We define the *mean number of active flows* in a trace as the number of active flows divided by the duration in seconds of the trace multiplied by 30 (*i.e.*, the threshold to determine if a flow is active). In Fig. 1, we report the mean number of active flows (y-axis in millions, blue crosses) with respect to the trace throughput (x-axis) for each single trace in the studied period. We first observe that 3 out of 53 traces already require more memory resources than those available on a real-world pipe of a high-speed ASIC, *i.e.*, the traces require above 20 MB of memory as they contains more than 1.1 M active flows.[5] This means that storing state for a real-world trace recorded at roughly 10 Gbps would require roughly 20% of the existing memory on a 16-nm ASIC switch chip (which is in the 10-100 MB range).

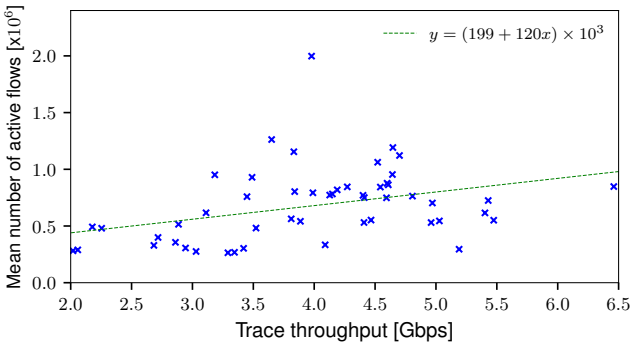

Figure 1: Number of active flows in historical CAIDA traces from 2013 – 2019 from Chicago and NYC.

We also plot a linear regression (green dashed line) that we use to estimate the memory requirements for a switch transporting terabits of traffic (such as recent 25.6 Tbps switches [5, 18]).[6] The steepness of the regression line is 120 K active flows per Gbps of traffic. We estimate the mean number of active flows on a 25.6 Tbps switch to be 3 072 M, which would require 52 GB of memory to store the corresponding state for a load balancer (*i.e.*, 17B per flow). This memory requirement is 300× higher than what is available on 7nm high-speed ASIC switches today [18] and roughly 10x the maximum amount of SRAM memory realizable on a

[2] We select this type of traces as we do not have access to datacenter traces at per-packet granularity (i.e., no sampling).

[3] We verified it by taking the Caida-nyc 2018-03-15 trace, counting the number of flows, and dividing by the trace duration, obtaining the same number.

[4] We use a 30-second threshold based on real-world timeouts used in the Facebook Katran load balancer [9].

[5] This is a lower bound since we take the mean of the active flows but peaks with higher number of active flows are likely to arise in the traces.

[6] The assumption may not be perfectly accurate but we do not have access to a datacenter trace at terabits per second speed and *per-packet* granularity (*i.e.*, no sampling).

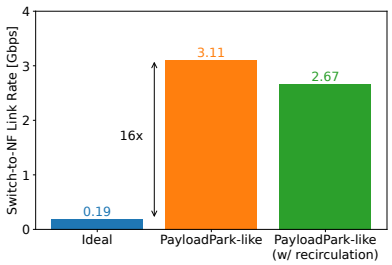

Figure 2: Ideal vs PayloadPark-like on CAIDA trace.

$825mm^2$ ASIC chip. Finally, we also note the memory on the pipe should be used for implementing other functionalities.

ASIC switches cannot support frequent per-flow state insertions from the control plane. Today's ASIC chips only support a limited amount of modifications to their flow-tables through the control plane [3, 31]. For example, a 16-nm 3.2 Tbps ASIC switch supports ~100 K flow-table entry modifications per second [46]. To the best of our knowledge, this is the highest (publicly available) table update frequency achieved by a 16-nm ASIC switch. Based on our analysis of the aforementioned CAIDA packet trace, we observe that the number of rule modifications is lower bounded by the amount of flows per second. Based on a similar regression, we obtain that the number of flow insertions per second grows by 4 K per Gbps of traffic. When we use this linear regression to estimate the amount of flow insertions on a 3.2Tbps ASIC switch, we obtain 12.8 M per-flow insertions per second, which is roughly 100× higher than the aforementioned 100 K flow-table modifications on an ASIC switch.

Now that we have quantified the amount of memory resources required to store the per-flow state of an NF, we discuss advantages and limitations of the three main state-of-the-art approaches to implement NF processors on top of programmable network hardware. We focus on the work that closely relates to our system and we defer the reader to Sect. 8 for a broad discussion of the existing related work.

NIC-based approaches that split packets are bottlenecked by the NIC speed. The *nicmem* system [35] is an NF accelerator that resides completely on a general-purpose server and does not involve any programmable switch. A packet arriving at the Network Interface Card (NIC) of a nicmem-equipped server is split into a header that is sent to the CPU cores and a payload that is stored on the small NIC memory. This approach comes with several benefits: higher hit cache ratio as payloads do not pollute CPU caches and 80% higher packet processing throughput. The inherent limitation of this work is that the throughput of the NF server is limited by the NIC speed (*e.g.*, a 100 G NIC can only process 10 M packets with size 1.25 KB). To process 800 Gbps of traffic, nicmem must connect to 8×100G ports on the switch. In this paper, we argue that an NF server should only receive the relevant bits (*e.g.*, the packet headers). This approach reduces bandwidth overheads toward external NF servers, reducing the number of ports on the switch that must be connected to dedicated resources. The remaining ports can be used for connecting the switch to other shared resources (where the payloads could be buffered) or devices. Moreover, traditional server- or NIC-based approaches (including nicmem) force the storage of the payload to be performed on the same machine that processes the header. Conversely, RIBOSOME decouples these two operations and it leverages spare memory resources in the network for performing the simpler payload storage.

Storing payloads at the switch does not mitigate bandwidth overheads. The *PayloadPark* [13] system also splits headers from payload but performs this operation directly on a programmable switch instead of the general-purpose server. The benefits of splitting a packet at the programmable switch instead of doing that at the server (as in nicmem) is that one can forward just the headers to the servers and store the payloads on the programmable switch. To implement a load balancer, a server could receive just 13 B from each packet (*i.e.*, the 5-tuple), thus potentially processing almost one billion packet headers per second through a 100 G interface. Unfortunately, PayloadPark suffers from an inherent constraint of high-speed ASIC devices. First, it is not possible to store the entire payload of a packet into the switch memory in a programmable manner.[7] Consequently, PayloadPark only stores the first 160 (352) bytes of payload for each packet without (with) recirculating it, thus letting most of the payloads still going to the external server.

To quantify the reduced performance gains, we have analyzed again the aforementioned CAIDA trace for three scenarios: *i)* when only a 54-byte header is sent to an external NF server, which we call *Ideal*, *ii)* when only 160 bytes of payloads are removed from the packet sent to the external NF server, which we call *PayloadPark-like*, and *iii)* a PayloadPark-like system that recirculates packets to store 352 bytes. Fig. 2 shows the link rate in Gbps between the switch and the external NF server (y-axis) for the three aforementioned systems. The ideal system requires 16× and 14× less bandwidth than the PayloadPark-like system with and without packet recirculation. We note that producing ASICs that would store larger parts of the packet in a programmatic manner would become significantly more complex and expensive [4] and are therefore neither available today nor in the near future.

Desiderata: large external memories shared with other applications. The goal of our work is to overcome the inherent limitations of ASIC switches and find an alternative design that keeps bandwidth requirements as close as possible to the ideal bar in Fig. 2. Since the memory on a switch cannot be used to store payloads in a programmatic manner, we focus our attention to leveraging external *unused* resources that are shared with customers' applications. As storing payloads only requires storing information into memory (without any

[7] The payload is temporarily stored by the switch while the headers are being processed.

complex logic), we focus our attention onto RDMA technology to store and retrieve payloads between a programmable switch and a set of external servers that are deployed to run customers' applications. Leveraging RDMA from a Tofino switch is not a new idea per-se as it has been already explored in TEA [21], a network function accelerator, and Dart [24], a monitoring system. We now discuss existing limitations of TEA, which will motivate our design. We note that some limitations of TEA are due to its design while some others are related to the functionalities that are today available on ASIC switches.

TEA cannot run complex NF logic such as packet schedulers. TEA [21] is the first framework to implement NFs using a programmable switch and leveraging additional RDMA-accessible memory to store per-flow state. In TEA, a packet is forwarded from the switch to the RDMA server that stores the rule used to process the packet. Both the packet and the rule are forwarded back to the switch, where the rule is applied to the packet. Unfortunately, this design does not support more complex per-flow network functions (*e.g.*, advanced load balancers, batch-based NF processing, etc.) since the only logic that can be performed in TEA is the one that is supported by the switch. For example, TEA cannot support advanced per-flow packet scheduler such as Reframer [11], where packets arriving at an NF are buffered for a few tens of microseconds and are then reordered to increase their per-flow spatial locality (*i.e.*, placing packets belonging to the same flow close to each other). The reason why TEA cannot support such NFs is that TEA can only "buffer" a single packet while reading its rule but it cannot buffer arbitrary sets of packets in a programmatic manner. We are not aware of any existing ASIC switch supporting such programmable buffers for packets.

TEA cannot handle per-flow rule insertions at high speed. When a new packet of a flow arrives at the switch, TEA states that "since it takes some time to complete an insertion operation, new entries are first inserted in to an SRAM stash". However, the insertions into the Stash are performed through the control-plane, which is renown to take up to 1ms to perform insertions [31].[8] In any case, the limit of flow-table insertions per second derived in Tiara [46] also applies to any table modification on TEA, which severely undermines the ability of TEA to perform a large number of flow-table insertions per second.

In the remaining sections, we address the following question:

> *"Can we design an NF packet processor that retains the high-throughput of an ASIC switch while supporting dynamic per-flow stateful network functions in a cost-effective manner?"*

[8] We do not have access to the original P4 code of TEA.

3 System Design

We now present an overview of RIBOSOME, a NF accelerator for stateful per-flow packet processing that relies on a novel design to overcome the limitations of existing architectures based on programmable switches and external devices.

Design space. We first divide the design space into *i)* systems built entirely *within a switch* and *ii)* systems using *external devices*. In the first category, realizing stateful packet processing entirely using ASIC-based switches is out of reach because of both memory limitations and limited modifications per second to the stateful data structures. In the second category (*i.e.*, systems with external devices), we further divide into two categories: *a)* systems that only use external *dedicated resources* and *b)* systems that also rely on external *shared resources*. In the following, we discuss these two types of systems and we refer the reader to Table 1 for a summary of the architectural and communication overhead differences

The table covers three types of operations (*i.e.*, the processing of the header, the storage of the packet, and the splitting and merging of the packet with the header (if any)) as well as the communication overheads in terms of bits and number of packets transmitted to the NF and the shared servers for each incoming packet at the switch.

Delegating all stateful packet processing functionalities to *dedicated* external FPGAs or CPUs (*e.g.*, Tiara [46], nicmem [35]) results in a high utilization of the switch ports to interconnect the external dedicated devices (*i.e.*, to process 800 Gbps of traffic, 8x100G ports on a switch must be connected to dedicated devices). PayloadPark [13] reduces bandwidth requirements toward externally dedicated devices. However, it only saves 1280 bits of bandwidth per transmitted packet, which only slightly reduces the number of ports on the switch that are connected to dedicated devices when the average packet size of a trace is in the 1 KB range.

Leveraging *shared* resources mitigates these overheads as ports on a switch can be connected to devices running other types of computations. Some recent work (*e.g.*, TEA [21]) delegates the storage of payloads on shared memory while relying on the switch to run the stateful packet processing logic. However, the logic implementable on an ASIC switch is limited (*e.g.*, no batch-based stateful processing as in packet schedulers or rate limiters). Moreover, it is difficult to use CPU-bypass technologies like RDMA to insert per-flow state inside the external server memory because RDMA only supports basic primitives (*e.g.*, Read, Write) and cannot be easily used to perform insertions at high-frequency [37]. Striking the correct balance in the usage of dedicated and shared resources and the architectural choices is the main goal of this section.

Our design principles. In this work, we explore a trade-off in the design space between the usage of dedicated and shared resources to accelerate stateful packet processing. Our observation from Sect. 2 is that any stateful packet processing should support *i)* high-speed insertions into per-flow state data

	Operations			Communication overhead (per packet)	
	Header processing	**Payload store**	**Split & merge**	**NF server [bits, # pkts]**	**Shared server [bits, # pkts]**
Traditional	NF server	NF server	-	pkt.size, 1	-
nicmem [35]	NF server	NIC	NIC	pkt.size, 1	-
Tiara [46]	NF server (fast path on FPGA on NIC)	FPGA and server CPU	-	pkt.size, 1	-
PayloadPark [13]	NF server	switch	switch	pkt.size - 1280 b, 1	-
TEA [21]	switch	RDMA server	-	-	pkt.size, 2
RIBOSOME	NF server	RDMA server	switch	pkt.header, 1	pkt.payload, 2

Table 1: Qualitative comparison among existing systems in the design space and RIBOSOME.

structures (in the order of tens of millions per second) and *ii)* more complex stateful logic (*e.g.*, batch-based processing) when deployed on a multi terabits per second switch. Our design is inspired by the following principles:

- **Offload complex logic to dedicated devices.** As ASIC switches support a limited number of flow-table updates per second and provide limited memory space, we argue that non-trivial network functions, whether for inserting high volumes of per-flow entries into the per-flow data structures or processing packets in a batch (*e.g.*, for scheduling), should be realized on dedicated general-purpose servers.
- **Process only relevant bits.** Our design targets network functions (*e.g.*, load balancers, NATs, rate limiters, packet schedulers) that do not require inspecting the entire packet, but rather just a few bytes such as a flow identifier. We therefore propose to only send the relevant bits to the dedicated general-purpose servers and store the payloads on shared servers while the headers are being processed. Splitting headers is not a new idea per-se (see [13, 35]), however we leverage it in such a way that the large gains materialize in practice, as shown in our evaluation section. Notice that our design also provides the possibility to disable the packet splitting for specific traffic classes. This allows the coexistence between RIBOSOME and NFs that require fully inspecting packets.
- **A programmable buffer on shared resources.** ASIC switches (including programmable ones) do not provide an interface for buffering packets in a programmatic manner. Packets are stored either while their headers are processed through the pipeline or in port queues. We argue that a network function system should be able to buffer packets in a programmatic manner, operate on batches of packets and schedule their transmission (to a certain degree of granularity, see Sect. 4). We rely on RDMA to bypass CPU and avoid wasting CPU cycles on shared machines. Note that our approach does not rule out the possibility of accessing other types of memory for storing payloads. We embrace disaggregation paradigms where the storage of payloads is performed on any shared memory resources in the network. As an example, switches could potentially support a programmable interface to store and fetch packets in an internal DRAM or HBM.

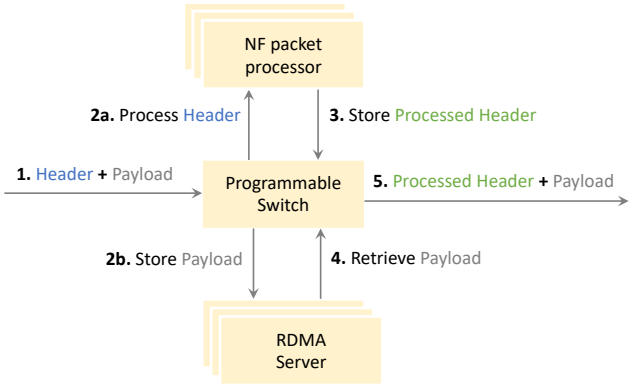

Figure 3: RIBOSOME overview.

To summarize, the main benefits of RIBOSOME are that it relies on dedicated devices only for realizing the NF processing logic and delegates the storage of the payload on external RDMA servers. RIBOSOME does not use any CPU cores on these RDMA servers. It only shares memory and NIC bandwidth with applications running on these servers. The benefits of RIBOSOME come with a cost: doubling the number of packets in a network since each packet will be split into a header and a payload packet.

System overview. RIBOSOME consists of a high-speed programmable switch, a set of *dedicated* external NF packet processors (*e.g.*, CPUs, FPGAs) and a set of *shared* servers. We leverage recent advancements in high-speed ASIC programmable switches [19], CPU-bypass memory storage (*i.e.*, RDMA [17]), and NF-specific CPU compiler optimiza-

tions [10] to design a system where dedicated packet processors only process the *relevant* portions of a packet while their payloads are stored on RDMA servers. We show a diagram of the high-level RIBOSOME architecture in Fig. 3. The programmable switch receives incoming packets (step 1) and splits each packet whose size is above a predefined threshold into a small header and a larger payload chunks. The programmable switch assigns an ID to both the header and the payload chunks. The switch assigns increasing IDs to each received packet within a predefined range (in a modulo manner). The switch forwards the header of the packet to one of the external NF packet processors (step 2a) and the payload to one of the shared servers (chosen hashing the flow 5-tuple) using RDMA (step 2b). The NF packet processors store the per-flow state needed to process any incoming packets. The NF uses this state to transform each incoming header into a new *processed header*, which is sent back to the programmable switch where it is stored on its small memory using the header ID as an index into an array in the switch SRAM memory (step 3). After storing a packet header, the programmable switch retrieves the corresponding payload from the RDMA servers (step 4). The programmable switch *combines* then the payload with the stored header using the ID and outputs the transformed packet (step 5).

We now discuss the different relevant operations and components of RIBOSOME, focusing on the main design challenges and our proposed solutions.

3.1 Splitting and Merging Packets

Deciding *i)* how many bits of a packet should be sent to the NF processor, *ii)* when a packets should be split into a header and payload, and *iii)* how to store the headers before the payloads are recollected are all questions that affect the overall performance of the system.

Challenges. Splitting a packet into a header and payload bring several benefits: it reduces both bandwidth overheads and cache pollution on the dedicated resources. However, splitting a packet also comes with some overheads: when we split a packet, *i)* we need to process a higher number of packets on the switch and *ii)* we need to use the switch memory to store the headers before recombining them with their corresponding payloads. More specifically, a single incoming packet arriving at a switch requires two packet processing if the packet is not split (*i.e.*, forwarding the packet to the NF and forwarding the modified packet from the NF to the output port) whereas a packet that is split results in 4 packet processing operations (*i.e.*, forwarding the header to the NF, forwarding the payload to the RDMA server, forwarding the NF response to the RDMA server to retrieve the payload, forwarding the recombined payload on the output port). Moreover, RDMA comes with limits on the number of operations per second that it can perform, which means transmitting small payloads may overload the server NICs without bringing any meaningful performance improvement.

Our approach. We devise a mechanism in RIBOSOME that splits a packet based on a threshold. There are two key thresholds in RIBOSOME: one threshold to specify when a packet should be split and one threshold to specify how many bits should be sent to the NF. We split packets at 72 bytes, considering the minimum Ethernet frame size of 64 bytes plus 8 bytes of additional custom RIBOSOME headers. Notice that the split threshold is configurable depending on the use case.[9]

RIBOSOME does not need to store any information on the switch when the packet has been split. Our system stores an header received from the NF on an array in the switch SRAM memory. Every time we split a packet, the switch increases the array index by one (modulo size of the array). This information is carried over in the header and the payload. When the header comes back to the NF, it is stored in the switch memory, and it also issues an RDMA Read Request to retrieve the corresponding payload.

The headers are stored until either *i)* the payload comes back to the RDMA server or *ii)* a new packet header is stored at the same array index, *i.e.*, the index pointer looped over the array size. We micro-benchmark the RDMA Read Request time (see Appendix F), finding that the maximum latency is at 4μs (with a 4096B payload). At 1.6 Tbps with an average packet size of 1 KB, this means we only need to store less than 800 headers in the array without risking to overwrite any header and combining it with the wrong payload. We configure the array with a size of 2K entries, which is large enough to guarantee that when the payloads come back, the header has not been replaced by a different header. By storing only 72 B of the packet header, RIBOSOME requires less than 60 KB of SRAM to store all the headers.

Our approach brings a significant advantage compared to alternative switch-based approaches like PayloadPark [13]. RIBOSOME only stores headers while retrieving payloads using RDMA, while PayloadPark must store payloads while waiting for the NF to process the headers. We observe that RDMA has more deterministic and lower response time than arbitrary NF processors. For instance, Batchy [25] reports processing times ranging from 100s of μs to few milliseconds to process a packet even for NFs that only look at packet headers (no heavy intrusion detection systems). At 1.6 Tbps with an average packet size of 1 KB and a response time of 1 ms, PayloadPark would need to store 185 MB of payloads, almost 10× the available memory on a 3.2 Tbps pipe.

Avoiding RDMA memory collisions. Packet payloads are stored contiguously in the external memory and are retrieved with the information contained in custom headers, hence there is no chance that two packets close in time read the same memory chunk. RIBOSOME conceptually uses the RDMA

[9] Ideally, the switch would only extract the bits relevant for a specific NF and batch the bits extracted from different packets into a single packet that is sent to the NF. Unfortunately, creating such batches is not supported by today's ASIC switches. We leave such optimizations as future work.

memory as a circular ring buffer, with a size provisioned large enough to prevent collisions.

Exploiting spare bandwidth on the RDMA servers. RIBOSOME leverages shared servers as remote buffers and it is therefore critical to use only the spare bandwidth of the server links without affecting the hosted services. Thus, RIBOSOME includes a control-plane mechanism that monitors the link bandwidth. When a link carries above a user-configured *back-off RDMA threshold*, the system stops sending payloads to the overloaded server.

Packets-per-second overhead. As RIBOSOME split packets, it also doubles the number of packets-per-second to be processed on both the switch (where we split the packets) and across the NF and RDMA-enabled devices when compared to a traditional approach. This is an inherent cost of splitting packets that is part of the RIBOSOME architecture. Table 2 compares per-packet processing overheads on each single component of three different NF approaches: *i)* Traditional NF processing where a switch sends the entire packet to an external NF processor, *ii)* a Payload-on-Switch (PoS) approach in which the payload is entirely stored on the switch (*e.g.*, PayloadPark), and *iii)* RIBOSOME. As it can be seen, RIBOSOME has the highest overhead. We first discuss the overhead on the switch. We note that several ASIC switches today support line-rate forwarding for small packets (*e.g.*, 300-Byte packets on general switches [18]). This means that RIBOSOME would support line-rate for packets with doubled size (*e.g.*, 600-Byte packets). This seems a reasonable trade-off in RIBOSOME as splitting a packet brings substantial benefits only when the packet is large-sized. As for the dedicated NF and shared RDMA-enabled servers, the number of PPS handled by the set of all these servers is twice as in a traditional approach. More specifically, the number of PPS handled by the NF servers is equal to that of the shared RDMA-enabled servers.

	Trad.	**PoS**	**RIBOSOME**
Switch	2/2	2/2	3/4
NF Server	1/1	1/1	1/1
RDMA Server	-	-	1/1

Table 2: Number of RX/TX packets for each approach (traditional, store payload on the switch, and RIBOSOME) in each component for each processed input packet.

3.2 High-Speed Reliable RDMA

To obtain a high-speed reliable RDMA implementation from the programmable switch, RIBOSOME has to overcome two main technical challenges. To support both RDMA Write and Read operations, Queue-Pairs (QPs) (virtual queues always composed of a *send* and a *receive* queue used to manage connections) must use the Reliable Connection transport mode. In this mode, the QP sends an acknowledgement for each packet received correctly, or a Nak in case of transmission problems. Detecting and recovering an RDMA Nak from a programmable switch is complex. For instance, receiving a *PSN Error Nak* would require transmitting the Nak-ed packet, and it is infeasible to store pending requests on the switch memory (especially RDMA Write operations that could contain a payload up to 4096 B). It is even more complex to recover from an *Invalid Request Nak*, triggered when the maximum number of outstanding RDMA Read Requests limit is reached. In fact, Infiniband specifications [42] limit the maximum number of outstanding Read Requests targeting a responder QP at any one time to a fixed amount (that is 16 with Nvidia Mellanox ConnectX-5 NICs [33]). To recover from this error, the QP state must be entirely reset.

We present a mechanism for recovering from the aforementioned failures with a minimal drop of packets in Appendix C. Moreover, we show in Sect. 5 how it is possible to handle multiple QPs, incrementing the maximum number of outstanding Read Requests, using a lower level API available within InfiniBand verbs [28].

4 Supporting Advanced Network Functions

Several common per-flow stateful operations require either batching a set of incoming packets (*e.g.*, [11, 25]), or keeping track of highly frequently arriving connections (*e.g.*, load balancers, NATs). In the following, we discuss three use cases for RIBOSOME that leverage advanced NFs whose logic would be difficult to realize on today's ASIC switches with large amounts of flows.

Stateful Load Balancers and NATs. Stateless load balancers suffer from high load imbalance [3] while software load balancers such as Maglev [8] must rely on many servers to remember which servers are taking care of every new selection. In RIBOSOME, we support stateful load balancers by storing the per-flow state on the NF processors and sending the headers of all packets to these processors. The main challenge is to support both high-throughput with millions of connections (which may not fit in the CPU caches) and forwarding rule insertions in the order of millions per second. In RIBOSOME, we use per-core Cuckoo++ [41] hash-tables, which have demonstrated superior performance [12]. The state management is using the recent FastClick's flow system, which finds a minimal per-flow state layout and handle state allocations and releases [2]. We support NATs similarly to load balancers.

Advanced per-packet software telemetry. Network telemetry is an indispensable component of today's networks, for both traffic optimization and security. Traditional monitoring tools such as NetFlow [16] rely on packet sampling and aggregation to perform off-path monitoring of the traffic, where "off-path" refers to the fact that network events are not detected on the data-path. Per-packet software monitoring has been proposed in *Flow [44], which however is also off-path,

thus cannot support advanced NFs. Emerging data-plane approaches such as ElasticSketch [45] detects network events directly in the ASIC data-plane using approximate data structures. However, ASIC switches have constrained memory for storing information about all flows. Through RIBOSOME, an operator leverages the large server memory to monitor in software and *on-path* all the packets processed by other advanced network functions, including load balancers and packets schedulers, which we discuss next.

Packet Schedulers. Packet schedulers are an example of network functions that determines the rate at which a flow or a traffic class should transmit traffic on a port or a destination server. Most importantly, they only need to inspect the packet headers and *buffer* packets for a limited amount of time (*e.g.*, on RDMA servers), which fits well within the RIBOSOME design. Realizing a per-flow packet scheduler on a hardware switch is hard for a number of reasons. Switches typically offer basic packet scheduler policies that scale to few traffic classes (up to 32 in general [43]). Realizing packet schedulers at the per-flow granularity or for more than 32 traffic classes is therefore hard to realize entirely in hardware.

RIBOSOME supports packet schedulers at the per-flow granularity, for instance, it support a per-flow leaky bucket rate limiter and the advanced Reframer [11] scheduler. RIBOSOME guarantees that packets belonging to the same traffic class leave the switch in the desired order. The main challenge in building a packet scheduler on RIBOSOME is that even if the NF processor reorders packets, there is no guarantee those packets will be output in the correct order from the RDMA servers. In fact, an RDMA Queue-Pair guarantees RDMA Read Request are served sequentially but Read Requests spread over different Queue-Pairs are not. Our key intuition is to guarantee that payloads of packets belonging to the same traffic class are read from the same RDMA Queue-Pair. The maximum throughput at which RIBOSOME guarantees an ordering of packets in a traffic class is limited by the hardware throughput of an RDMA Queue-Pair.

Example with a Reframer packet scheduler. We show an example of our approach implementing the Reframer packet scheduler [11]. Reframer is a NF that buffers packets for tens of microseconds and reorders them so that packets belonging to the same flow are transmitted back-to-back. We show that this functionality can be implemented on top of RIBOSOME.

Consider Fig. 4, where the RIBOSOME switch receives 8 packets from four ports belonging to four flows called *A* (black squares), *B* (yellow squares), *C* (blue squares), and *D* (green squares). Assume flows *A* and *B* are mapped to core 1 of the NF and they should be forwarded on port 1 of the switch while flows *C* and *D* are mapped to core 2 and they will end up on port 2. The switch is connected to one RDMA server which has 2 active Queue-Pairs; hence, all the payload data will be written on the same RDMA server. We use dashed orange (green solid) arrows to denote packets moving from the switch to external entities (from external entities to the

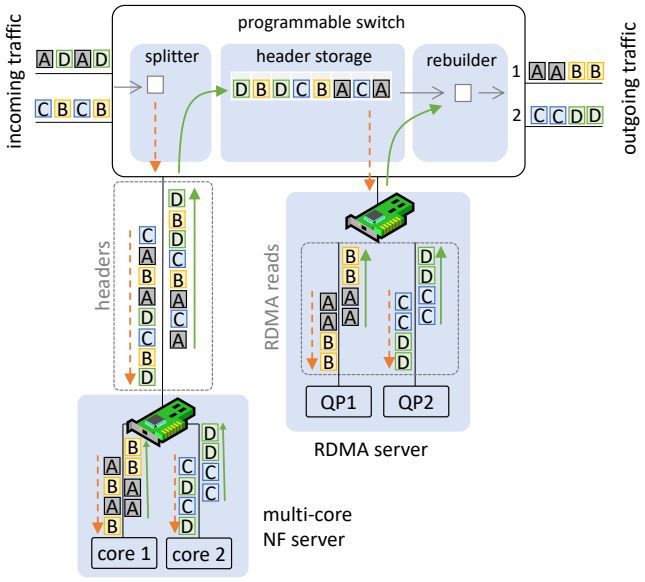

Figure 4: Supporting packet schedulers with RIBOSOME.

switch). The packets are initially split into a header that is forwarded to the NF server and a payload that is stored on the RDMA server (this RDMA Write operation is not shown in the figure). The headers will traverse the NF link in any extension of the partial order in which they arrived at the 4 ports. The switch tags headers with information about which RDMA server is used to store packets content (in this case, we have only one RDMA server and all packets have the same tag) [10]. On the NF side, core 1 (core 2) receives a sequence of packets $< B,A,B,A >$ ($< D,C,D,C >$) and reorders it into $< B,B,A,A >$ ($< D,D,C,C >$). To avoid Queue-Pairs overloading and to preserve packets ordering, the NF is enabled to add a new tag to packets determining the preferred Queue-Pair index for reading the content from the RDMA server. In this scenario, core 1 tags packets with QP1 and core 2 tags packets with QP2. Finally, the NIC forwards these headers back to the switch, possibly interleaving packets from the two cores. When packets arrive at the switch, the headers are stored in the order in which they are received and the corresponding RDMA Read Request are generated according to tags added by the NF cores. As a result, the RDMA server receives the Read Requests in two Queue-Pairs exactly in the order in which the corresponding NF cores have generated them. The payloads also return in the same order of the Read Requests.

5 Implementation

We implemented RIBOSOME's data plane in P4_16 language and compiled it to a Intel Tofino ASIC [19]. The server pro-

[10] We encode this tag in the MAC destination address which gives the flexibility to users to dispatch packets among cores based on corresponding RDMA servers if it is needed.

cess that manages RDMA connections and remote buffers is written in C++, using Infiniband Verbs [28].

Server Connection Establishment. For initializing a connection with the switch, the server agent takes as input the Infiniband interface name and the Server ID (an incremental index that starts from 0). It allocates a memory buffer enabled for remote write/read access. The process sends to the switch all the information to identify the connection using several different custom Ethernet frames (described in Appendix E).

The switch saves this information in different registers (`base_addr`, `rkey`, `mac`, and `ip`), using the Server ID as index.

After this initial setup, the process creates a user-defined number N_{qp} of Queue-Pairs. Each QP q_i ($i = 0, \ldots, N_{qp}$) is created using the Reliable Connection transport mode. A unique local Queue-Pair ID L_{q_i} is assigned by the NIC. Instead of pairing q_i with a remote QP, RIBOSOME fakes the connection to a remote endpoint, avoiding the need of a second Infiniband-capable device. A fake remote Queue-Pair ID R_{q_i} is computed using the formula $R_{q_i} = i + (Server\_ID * N_{qp})$. The remote and local initial PSNs are set to 0. At this point, the server sends the Queue-Pair information to the switch with a Server Queue-Pair Info frame, containing L_{q_i} and R_{q_i}. The switch stores L_{q_i} in the `qp` register at index R_{q_i}, it writes the `enabled_qp` register at index R_{q_i}, enabling the QP, and it also sets the register `psn` to 0 at the same index R_{q_i}.

RIBOSOME Headers. RIBOSOME uses two custom headers, that contain information needed to retrieve a payload and merge it with its correct headers. We provide a description of the headers and their fields in Appendix E.

Splitter. When a packet is received on a RIBOSOME enabled port, the switch checks that the length of the packet is higher than a definable threshold. If it is under the threshold, the packet is not split and sent to the NF. Otherwise, the switch applies a hash function (on a 4-tuple composed of *SrcIP, DstIP, SrcPort, DstPort*) to index a Match-Action Table to select the server and the index i of the QP q_i that will store the relative payload. This ensures that payloads of the same flow will be managed by the same server and QP, and avoids reordering packets belonging to the same flow. The switch reads the `enabled_qp` register: if the QP is not enabled (equal to 0), the packet is sent to the NF without splitting it. Else, the switch retrieves (using i) the server data from `mac`, `ip`, `base_addr` and `rkey` registers. The packet is then transformed into a RoCEv2 RDMA Write. The `PSN` field of the BTH header is set by reading and incrementing the register `psn` at index i. The switch also appends the Header Info, selecting an index h where the header will be stored after being processed by the NF. It also appends the exact padding bytes to align the payload to a 4-byte boundary. The packet is mirrored to the Egress pipeline, where it is truncated to the header size. The switch appends both Payload Info and Header Info, that will be used for reconstructing the packet after the NF processing.

We show a high level overview of the flow in Appendix D.

Rebuilder. When the switch receives a processed header from the NF, if it contains the Payload Split header, it means that the packet must be reconstructed, else the packet is normally routed. Before reconstructing the packet, the switch saves the processed header into several registers `hdrs` at the index h specified by the Header IDX field in Header Info. The packet is then transformed into a RoCEv2 RDMA Read Request. The switch reads the Payload Info header and retrieves the index i to read the information of the server that contains the payload (`mac`, `ip`, and `rkey` registers). The `PSN` field of the BTH header is set by reading and incrementing the register `psn` at index i. The switch fills RETH header with the Payload Address and Payload Length fields stored in Payload Info.

The RDMA Read Request is sent to the corresponding server, which answers with an RDMA Read Response containing Header Info and the payload. The switch parses the response, reads h from the Header IDX field of Header Info and uses it to load the right header from registers `hdrs`. It removes the additional padding and prepends the processed header, reconstructing the entire packet that is normally routed. The reconstructed packet is 4 bytes longer than the original one as the switch cannot remove the ICRC appended by RoCEv2.

We show a high level overview of the flow in Appendix D.

Spare Bandwidth Exploitation. To ensure that RIBOSOME uses only the spare bandwidth of the shared servers without affecting hosted services, we implement a control plane mechanism that monitors the actual usage of the links. If the per-port bandwidth usage is under a configured *back-off RDMA threshold*, RIBOSOME uses the link for storing payloads in the remote memory of the server. Instead, if the port usage is above the threshold, the switch stops using that link for payloads, preserving the bandwidth for services. In this case, RIBOSOME remaps the Match-Action Table that selects QPs and servers, equally redistributing the entries of the overloaded server among the others. When the port bandwidth usage goes below the threshold, RIBOSOME restores the original mapping of the table, re-enabling the server.

6 Evaluation

RIBOSOME is the first programmable buffer abstraction that is suitable for Tbps advanced NF packet processing. It performs stateful packet processing, carefully splitting operations between dedicated and shared resources, dedicated servers process headers and servers hosting customers' services store payloads without CPU interference. In this section, we demonstrate the performance gains achievable by RIBOSOME. All scripts, including documentation for full reproducibility, are available [39]. We aim to answer five main questions:

- *"How much* RIBOSOME *improves the per-packet throughput and latency gain on the NF server?"*
- *"How does the packet size impact the throughput gains?"*
- *"Can we build advanced NFs on top of* RIBOSOME*?"*

- *"What are the overheads on the RDMA servers?"*
- *"How many ASIC resources does* RIBOSOME *require?"*

RIBOSOME's data plane is deployed on a 64×100 Gbps Stordis BF6064X with Intel Tofino ASIC [19]. Four of its ports are connected to a 32×100 Gbps Edgecore Wedge100BF-32X with Intel Tofino ASIC. Four commodity servers run the server agent and are equipped with Intel®Xeon®Gold 6140 CPU @ 2.30GHz and Nvidia Mellanox ConnectX-5 NICs [33]. All CPUs are set at their nominal frequency. The testbed is wired with 100Gbps links. The experimental setup is depicted in Appendix A.

Workload generation. To generate different loads, we use an additional server equipped with Intel®Xeon®Gold 6140 CPU @ 2.30GHz, and Nvidia Mellanox ConnectX-5 NICs [33], connected to the 32-port switch. The switch multicasts the incoming traffic to three ports unless stated differently. We inject both synthetic and real-world traffic traces using FastClick [1]. Another server with the same hardware runs different NFs, also implemented in FastClick. All the experiments are repeated 3 times.

6.1 Throughput and Latency Gains

RIBOSOME enables multi-100G packet processing. Fig. 5 shows the throughput of the NF (in pps) and the output throughput of the system (in Gbps) for three systems: a baseline where the NF receives the entire packet (dashed-dotten orange line), a PayloadPark-like system that removes 160 bytes of a payload when transmitting a packet to the NF server (dashed light blue line), and RIBOSOME (solid dark blue line). The average packet size is 1KB. The x-axis is the packet rate injected by the traffic generator (in Mpps). The baseline rapidly saturates the available 100 Gbps link bandwidth, consequently capping the NF throughput to this rate. The PayloadPark-like performance shows that only a limited increase in throughput can be achieved as the switch can store just a small part of the payload. RIBOSOME achieves higher throughput by sending only the headers to the NF, showing the system can keep up the processing of the 300 Gbps input traffic. We note that only ~75 Gbps of payloads can be handled by the RDMA NICs in our testbed due to RDMA overheads. Therefore, the 300 Gbps of generated traffic is limited by the 4 RDMA servers. So in order to process 1.6 Tbps of traffic, RIBOSOME would need to be connected to 22 shared RDMA-enabled servers. The gains for this simple forwarding NF could potentially be even higher by deploying more RDMA servers, which we could not do in our testbed.

RIBOSOME improves latency. While one would expect delaying packets to recover the payload through RDMA takes time, the advantage of reducing the queue sizes and the transmission rate at the NF compensates. In Fig. 6, we show the median latency (y-axis) with respect to the input rate (x-axis). Even at a medium input rate, the latency of the baseline system (which does not split packets) increases, while the latency

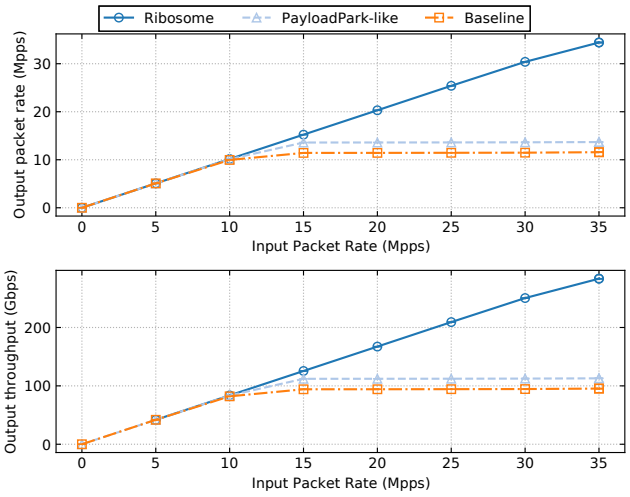

Figure 5: Bandwidth advantage of sending only headers to a forwarding NF, 1024 B packets.

of RIBOSOME is kept constant, achieving a 4× gain. We also verified that tail latency follows a similar trend: the baseline reaches up to ~500 µs tail latency because of the queuing happening on the NF server, while Ribosome maintains a constant ~60 µs latency.[11]

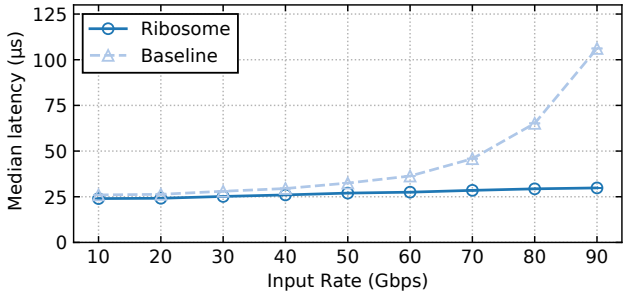

Figure 6: Median RTT latency for 1024 B packets sent by the generator to a forwarding NF under varying input rate.

RIBOSOME retains high performance regardless of the input packet size. Fig. 7 shows the output throughput (y-axis) of our baseline forwarding NF that does not split packets (dashed line) and RIBOSOME (blue line). The x-axis is the packet length in bytes. The split threshold is set to 64 bytes. Varying the packet size does not affect the overall throughput as the NF is still capable of processing 300 Gbps. Moreover, this also demonstrates that RIBOSOME is highly effective for the relevant real-world scenarios, where the average packet size ranges between 500 and 1K bytes [6]. We reach 300 Gbps of throughput already with 400 B packets (*i.e.*, 93Mpps). This graph demonstrates that the bottleneck with 1KB packets is

[11] Fig. 11 in Appendix B shows 99th percentile tail latency.

not the CPU but rather the limited number of RDMA servers. We hypothesize that RIBOSOME could potentially operate above 600Gbps (75Mpps) with a simple NF forwarder.

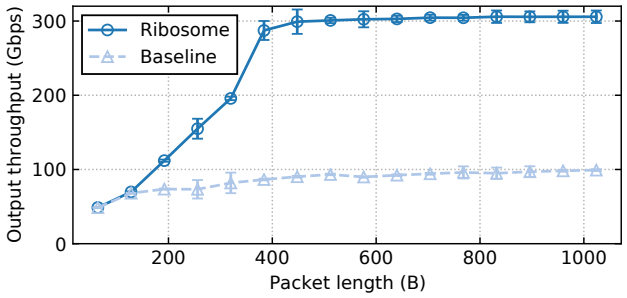

Figure 7: Bandwidth gain according to packet length.

6.2 Advanced Network Functions

We have successfully integrated and evaluated the three applications presented in Sect. 4 within RIBOSOME. Fig. 8 shows the RIBOSOME output throughput (in Gbps, y-axis) for the Reframer [11] advanced packet scheduler (blue solid line), a per-flow rate limiter (light blue dashed line), and a Layer-4 load balancer (orange dashed-dotted line). We vary the number of CPU cores used on the NF (x-axis). We replay a CAIDA trace that runs for 3.7s and contains 13.4 M flows.

RIBOSOME supports the Reframer packet scheduler at about 150 Gbps with 4 CPU cores and about 220 Gbps with 8 CPU cores (hyperthreading enabled). To put things into perspective, this level of throughput is almost 2.2× higher than the one achieved in the original paper on a single server (*i.e.*, 100 Gbps) [11]. We observe that a single Queue-Pair has a throughput limitations of 2.5 Gbps. This means that currently RIBOSOME, combined with Reframer, can only guarantee packet ordering for traffic classes whose throughput is at most 2.5 Gbps. We expect such limitations to ease in future generations of RDMA.

Similarly, Fig. 8 shows the performance of the per-flow rate limiter that independently tracks the rate of each individual micro-flow going through the switch and limits them using a per-flow token bucket. The rate-limiter NF achieves close to 300 Gbps similarly to the load balancer function. Both these NFs require keeping track of individual Layer-4 connections. In both cases, the NF server handles 3 M new flows per second, whereas an ASIC switch can only support ~100 K flow-table entry modifications [46].

RIBOSOME preserves the order of packets. As discussed in Sect. 4, there is a possibility that packets from different Queue-Pairs got interleaved during RDMA Reads. We investigate the amount of unordered packets by measuring the average Spatial Locality Factor (SLF) of the traffic on the RIBOSOME output and comparing it with the SLF value right after Reframer instances on the NF. It is the parameter used in [11] to measure the ordering level of the traffic. We observe that using our trace file, the SLF value on the NF server is 1.31 while it only drops slightly to 1.29 on the output of RIBOSOME, which demonstrates its ability to primarily preserve the order of packets according to the advanced scheduler.

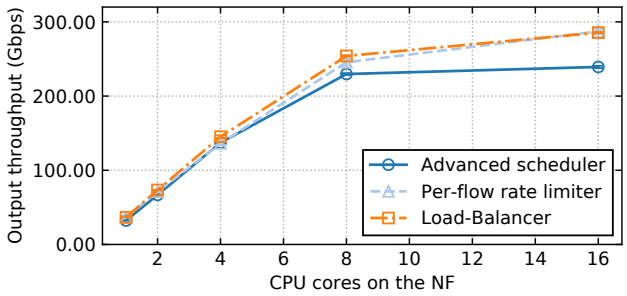

Figure 8: Throughput of three advanced Network Functions.

6.3 RDMA Interference Analysis

Storing payloads on RDMA servers may impact applications running on those servers, and specifically, their available network and memory bandwidth.

Little impact of RDMA on memory bandwidth and CPU. We generate RDMA operations that fill the NIC capacity (nearly 100 Gbps) and verify that the CPU load is 0% as RDMA traffic bypasses the CPU. We then use STREAM [30] to benchmark the CPU-to-memory bandwidth. We see that the available CPU-to-memory bandwidth decreases by ~25% when using 100% of the NIC for RDMA operations. Despite such bandwidth decrease, RIBOSOME leaves plenty of memory spare bandwidth on the RDMA servers.

RIBOSOME reactively releases network bandwidth resources from RDMA servers. We craft a synthetic trace where RIBOSOME does not split a specific traffic class and forwards it directly to the RDMA Server 1. This "unsplit" traffic gradually increases over time simulating an increased bandwidth demand on Server 1. We run this experiment without 3× multicasting enabled and we set the back-off RDMA threshold at 40 Gbps. Fig. 9 shows the input throughput of the "unsplit" traffic (violet line) and that of the four RDMA servers. We see that the RDMA throughput on each RDMA servers is around 7 Gbps at time 15s. When the "unsplit" traffic reaches roughly 33Gbps (at time 16s), this event triggers RIBOSOME's control mechanism, which stops sending payloads to Server 1 and redistributes the load on the other three servers. In future work, we will design an adaptive algorithm (instead of an on/off control mechanism) to share the NIC bandwidth in a fine-grained manner. This will also decrease the CPU memory controller utilization of RIBOSOME when the server is used for a network workload.

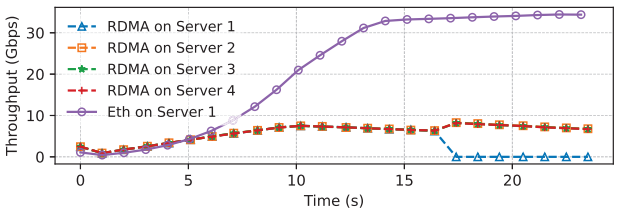

Figure 9: RIBOSOME stops sending payloads to Server 1 when it becomes overloaded.

6.4 ASIC Resource Usage

Table 3 shows the additional ASIC resources consumed by RIBOSOME based on the Tofino compiler's output. Overall, RIBOSOME consumes a similar amount of VLIW Instructions and Match Crossbar than TEA [21] and occupies a negligible amount of TCAM. For SRAM memory usage, we allocate $1.5K \times 15 \times 4B$ register entries to store headers while fetching the payloads with RDMA, which suffice to sustain the $4\mu s$ RDMA maximum response time. Moreover, RIBOSOME also relies on several registers to store RDMA connections data, which justifies the additional memory usage.

Resource	Additional Usage
SRAM	7.84%
TCAM	1.14%
VLIW Instruction	13.82%
Exact Match Crossbar	13.92%
Ternary Match Crossbar	0.69%

Table 3: Additional ASIC resources used by RIBOSOME.

7 Discussion

How many NF-dedicated ports to achieve full-throughput on a high-speed ASIC switch? Similarly to Tiara, RIBOSOME only achieves half of the switch throughput as half of the ports store payloads. One RIBOSOME server processes up to 80 Mpps for a forwarding NF. We need to use three ports of the switch exclusively for NF processing to process the equivalent of 1.6 Tbps of 1 KB packets. With advanced NFs, we must reserve 6-7 ports on the switch whereas Tiara requires 8 ports to connect its FPGAs. Replacing our NF servers with FPGAs may lower the number of dedicated ports to our lower bound of 3 ports, which is future work.

Can RIBOSOME offload heavy-hitter entries to the switch after an insertion? Yes, this is doable (whenever the NF function is realizable on a programmable switch) and similar to what TEA or CRAB [22] do. We believe this optimization is orthogonal to our approach and we leave it as future work.

8 Related Work

We discuss related work that we have not already mentioned.

Dedicated external devices. Several systems require sending the entire packet to the NF processor [8, 15, 20, 26, 34–36, 46, 49]. Gallium [48] enables offloading a part of the NF processing on the switch, but complex processing still needs to be executed on the NFs servers. In contrast, we minimize the amount of dedicated resources needed to run complex NFs by relying on shared resources to store and retrieve payloads, thus minimizing the number of ports on the switch connected to dedicated devices. Moreover, we present a novel programmable buffer that supports packet schedulers or batch-based NFs.

RDMA on a high-speed ASIC switch. TEA [21], SwitchML [38], and Dart [24] all propose to use RDMA directly on the Intel Tofino switch. TEA (SwitchML) implements reliable (unreliable) RDMA transport. TEA code is not publicly available. Unreliable RDMA does not support RDMA Reads. Dart sketches an implementation without providing code. We implement reliable RDMA, evaluate bottlenecks, and make all our code public.

9 Conclusion

We presented RIBOSOME, a high-speed stateful packet processor that reduces the amount of dedicated NF processors by carefully sending only headers to the external NFs. We showed in our testbed that RIBOSOME scales throughput by up to a factor of 3× with a single NF processor (potentially up to 10× with additional RDMA servers for storing payloads). We believe that RIBOSOME aligns with the current trends towards disaggregating architecture in which resources are shared for different purposes. We leave as future work the fundamental problem of further improving the performance of CPU-based NF processors given the observed high cost of retrieving the flow state from memory (especially when it resides outside the cache). Also, we will further investigate optimizations for reducing packet-per-second overheads introduced when splitting packets.

Acknowledgements

We would like to thank our shepherd Jeongkeun Lee and the anonymous reviewers for their insightful comments and suggestions on this paper. This work has been partially supported by the Swedish Research Council (agreement No. 2021-04212) and KTH Digital Futures. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889). Tom Barbette has been funded by an FSR Post-doc Fellowship from UCLouvain.

References

[1] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16, 2015.

[2] Tom Barbette, Cyril Soldani, and Laurent Mathy. Combined stateful classification and session splicing for high-speed NFV service chaining. *IEEE/ACM Transactions on Networking*, 29(6):2560–2573, 2021.

[3] Tom Barbette, Erfan Wu, Dejan Kostić, Gerald Q. Maguire, Panagiotis Papadimitratos, and Marco Chiesa. Cheetah: A High-Speed Programmable Load-Balancer Framework With Guaranteed Per-Connection-Consistency. *IEEE/ACM Transactions on Networking*, pages 1–14, 2021.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[5] Broadcom. Tomahawk4, 2022. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series.

[6] CAIDA . Trace Statistics for CAIDA Passive OC48 and OC192 Traces, 2019. https://www.caida.org/catalog/datasets/trace_stats/.

[7] Don Draper. TSMC's 5nm 0.021um2 SRAM Cell Using EUV and High Mobility Channel with Write Assist at ISSCC2020, 2020. https://semiwiki.com/semiconductor-manufacturers/tsmc/283487-tsmcs-5nm-0-021um2-sram-cell-using-euv-and-high-mobility-channel-with-write-assist-at-isscc2020/.

[8] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan 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.

[9] Facebook. Katran Load Balancer, 2021. https://github.com/facebookincubator/katran/blob/3fadb1eaaff719980a3cc9dc8870f88d442a40e1/katran/lib/bpf/balancer_consts.h#L100.

[10] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-Core 100-Gbps Networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Maguire, and Dejan Kostic. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2021.

[12] Massimo Girondi, Marco Chiesa, and Tom Barbette. High-speed Connection Tracking in Modern Servers. In *22nd IEEE International Conference on High Performance Switching and Routing, HPSR 2021, Paris, France, June 7-10, 2021*, pages 1–8. IEEE, 2021.

[13] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. *Parking Packet Payload with P4*, page 274–281. Association for Computing Machinery, New York, NY, USA, 2020.

[14] Vladimir Gurevich and Andy 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.

[15] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue 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.

[16] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys Tutorials*, 16(4):2037–2064, 2014.

[17] InfiniBand Trade Association. Supplement to InfiniBand Architecture Specification - Annex A17: RoCEv2, 2014. https://cw.infinibandta.org/document/dl/7781.

[18] Intel. Intel Tofino 3 Intelligent Fabric Processor Brief, 2022. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html.

[19] Intel. Intel Tofino Series, 2022. https://www.intel.com/content/www/us/en/products/

network-io/programmable-ethernet-switch/tofino-series.html.

[20] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald 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, April 2018. USENIX Association.

[21] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan 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.

[22] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the Load Balancer without Regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 193–207, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[24] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. *Zero-CPU Collection with Direct Telemetry Access*, page 108–115. Association for Computing Machinery, New York, NY, USA, 2021.

[25] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor 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, February 2020. USENIX Association.

[26] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong 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.

[27] Linux. perftest, 2022. https://github.com/linux-rdma/perftest.

[28] Linux. RDMA Core, 2022. https://github.com/linux-rdma/rdma-core.

[29] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native approach for detecting and mitigating volumetric DDoS attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3829–3846. USENIX Association, August 2021.

[30] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan 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.

[32] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, 2016.

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

[34] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. *SIGCOMM Comput. Commun. Rev.*, 43(4):207–218, aug 2013.

[35] Boris Pismenny, Liran Liss, Adam Morrison, and Dan 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, February 2022.

[36] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano.

FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.

[37] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is Turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association.

[38] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[39] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. Ribosome Experiments Github Repository, 2022. https://github.com/Ribosome-Packet-Processor/Ribosome-experiments.

[40] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. Ribosome Github Repository, 2022. https://github.com/Ribosome-Packet-Processor/Ribosome.

[41] Nicolas Le Scouarnec. Cuckoo++ hash tables: High-performance hash tables for networking applications. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 41–54, 2018.

[42] Tom Shanley. *Infiniband Network Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[43] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, April 2018. USENIX Association.

[44] John Sonchack, Oliver Michel, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 823–835, Boston, MA, July 2018. USENIX Association.

[45] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve 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.

[46] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong 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.

[47] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. 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, April 2022. USENIX Association.

[48] Kaiyuan Zhang, Danyang Zhuo, and Arvind 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.

[49] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine 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, November 2020.

A Experimental Setup

Fig. 10 depicts the experimental setup. The Traffic Generator is connected to the Multicast Tofino with a 100-Gbps link. The Multicast Tofino has four of its 100-Gbps ports connected to the RIBOSOME Tofino, and it multiplexes the incoming traffic from the Generator. It modifies each packet to have two additional copies with different 4-tuple values before sending them to the RIBOSOME Tofino. RIBOSOME Tofino is attached to four commodity servers running the RDMA server, and one server that implements the NF, all with 100 Gbps links. When a packet is reconstructed after the NF processing, the RIBOSOME Tofino sends it back to the Multicast Tofino. If

the packet is an original one (with no flow modifications), it is sent back to the Traffic Generator.

B RIBOSOME Tail Latency Impacts

Fig.11 demonstrates the impact of RIBOSOME on packets 99th percentile tail latency when the NF server is running a simple forwarder.

C Recovering RDMA Queue-Pairs

We illustrate the implementation of the QP Recover mechanism discussed in Sect. 3.

For supporting both RDMA Write and Read operations, Queue-Pairs are created using the Reliable Connection transport mode. This mode expects that packets are received in the correct order, by checking their PSN. If a packet is out of order, the QP sends back a *PSN Error Nak*, requesting the retransmission. RIBOSOME's implementation does not store RDMA Write or Read Request packets in the switch (it only keeps the current PSN). Hence, it is not able to retransmit a Nak-ed packet. Additionally, Infiniband specifications limit the maximum number of outstanding RDMA Read Requests on each QP. If there are more of such requests, the QP transits into an invalid state, sending an *Invalid Request Nak*.

To overcome these limitations, RIBOSOME exploits several QPs on each server, and it also implements a QP recovery mechanism, that allows to reset a QP in case of errors.

When the programmable switch receives a Nak, it puts the corresponding Queue-Pair q_i in a disabled state writing the `enabled_qp` register to 0. The index i to access the register is the `DestQP` field of the BTH header, that is the value $R_{q_i} = i + (Server\_ID * N_{qp})$. The `enabled_qp` register is periodically checked by a control plane script, that takes all the entries with a value of 0 and, for each index j, reads register `qp` at index j. If the Queue-Pair ID is set, the switch writes another register (called `qp_to_restore`) at index j with a value 1.

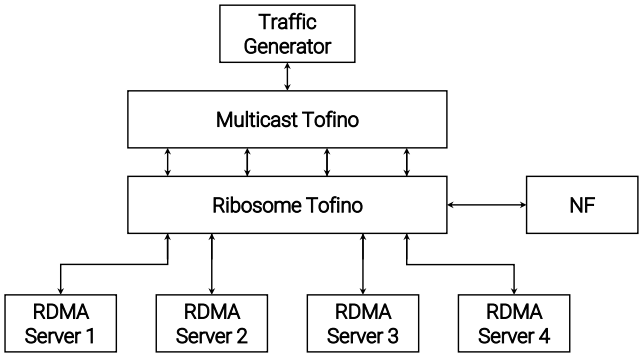

Figure 10: RIBOSOME testbed.

When receiving a packet in the Splitter component, if the Queue-Pair q_i is selected, and the value of `enabled_qp` register at index i is 0, and the value of `qp_to_restore` register at index i is 1, the packet is sent to the NF without splitting it. Also, the packet is mirrored to the Egress pipeline, and transformed into a simple Ethernet frame with `EtherType` = 0x4321, containing the index of the Queue-Pair to restore. The server agent has a raw L2 socket listening on the interface used to open the connection. When the aforementioned frame is received, the associated QP is reset and re-initialized. At this point, a Server Queue-Pair Info packet is sent to the switch, that re-enables the QP.

D Splitter and Rebuilder Components

We illustrate a high level overview of the two main RIBOSOME's components. Fig. 12 depicts the *Splitter* component, while Fig. 13 shows the *Rebuilder* component.

E Custom Ethernet Frames and Headers

RIBOSOME leverages on several custom Ethernet frames to identify a server connection, depicted in Fig. 14.

Also, the system uses two custom headers (showed in Fig. 15), that contain information needed to retrieve a payload and merge it with its correct headers:

a) **Payload Info**: appended to truncated headers when the packet is split. The Marker field is used by the switch to identify the header. The Payload Address indicates the starting address of the payload in the remote RDMA buffer. The Length field is the length of the payload in the buffer. The Index is used by the switch to request the payload on the same Queue-Pair used when sending the RDMA Write request.

b) **Header Info**: appended to the Payload Info when the packet is split. It is also prepended to the payload before sending it to the remote buffer. The Pad Count field stores the number of bytes of the Additional Padding field to align the payload to a 4-byte boundary as per Infiniband

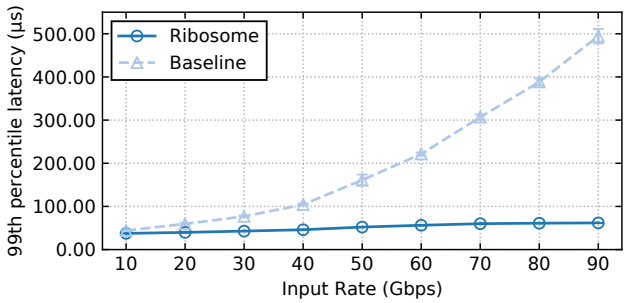

Figure 11: RIBOSOME 99th percentile latency of packets w/wo existence of Ribosome.

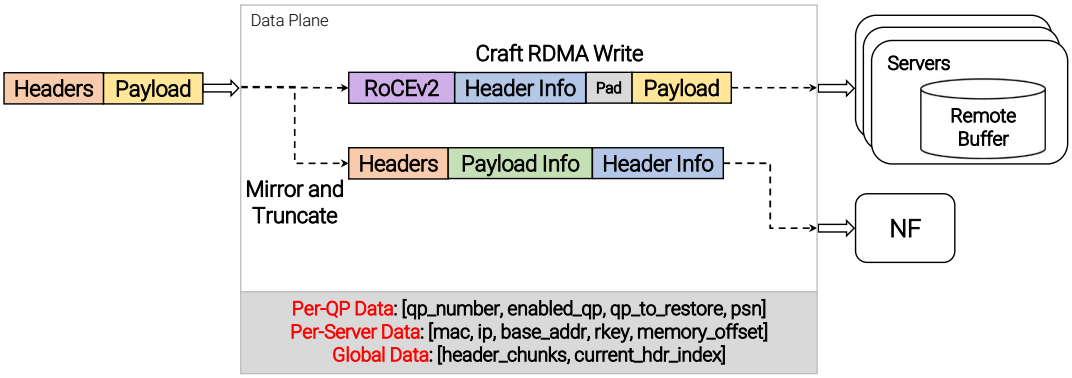

Figure 12: RIBOSOME's *Splitter* Component Overview.

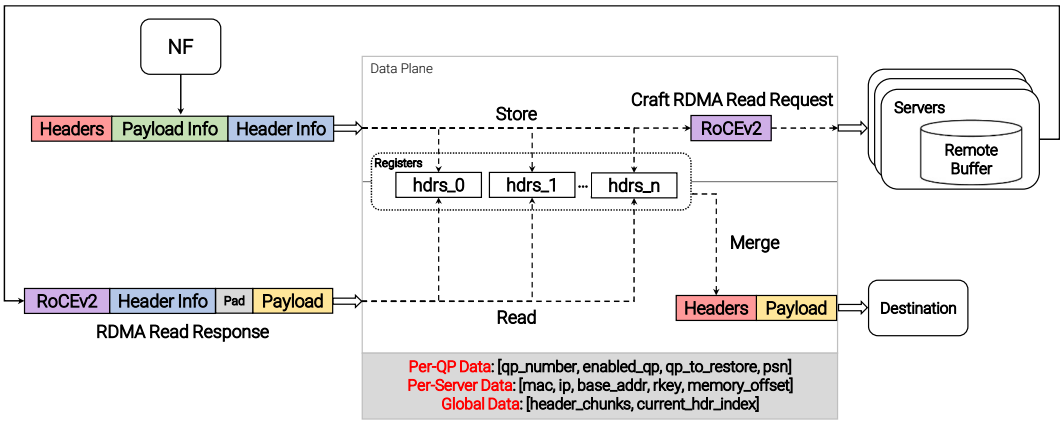

Figure 13: RIBOSOME's *Rebuilder* Component Overview.

specifications [42]. The Header IDX indicates the index of the register where the header processed by the NF is saved while the switch is fetching the payload from the remote buffer.

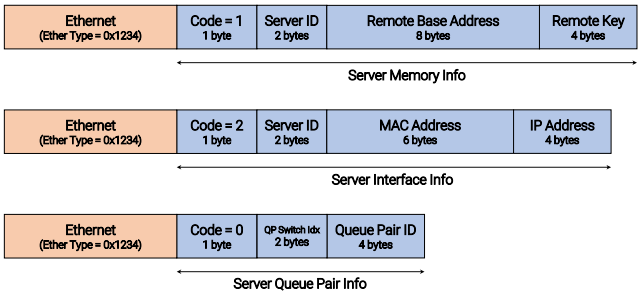

Figure 14: Server Info Ethernet Frames.

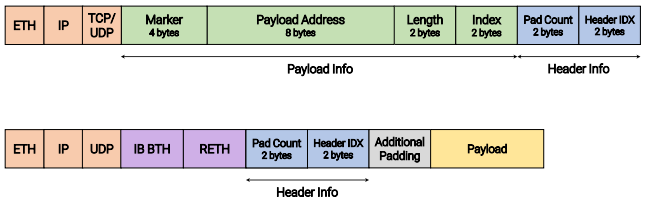

Figure 15: RIBOSOME Headers.

F RDMA Latency Microbenchmark

We performed a microbenchmark of RDMA operations using the Linux Infiniband `perftest` suite [27] on two servers, equipped with Intel®Xeon®Gold 6140 CPU @ 2.30GHz, and Nvidia Mellanox ConnectX-5 NICs. Fig. 16 shows the average latency of 1 K iterations (y-axis) of both RDMA Read and Write operations with different payload lengths (x-axis).

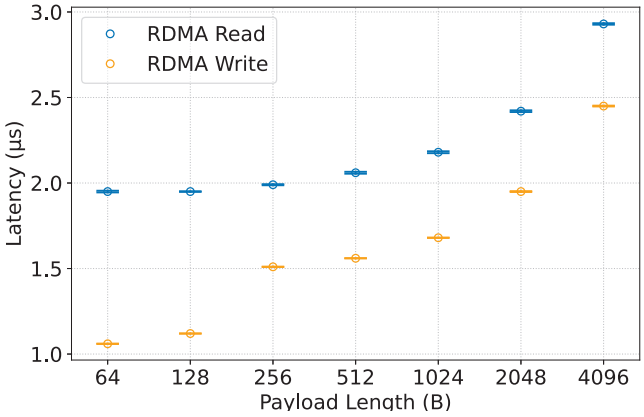

Figure 16: RDMA Operations Microbenchmark.