Disaggregating Stateful Network Functions

Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, and James Grantham, *Microsoft;* Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, and Balakrishnan Raman, *AMD Pensando;* Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula, *Microsoft*

https://www.usenix.org/conference/nsdi23/presentation/bansal

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Disaggregating Stateful Network Functions

Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai‡, Mario Baldi‡, Krishna Doddapaneni‡, Arun Selvarajan‡, Arunkumar Arumugam‡, Balakrishnan Raman‡, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, Srikanth Kandula
Microsoft and AMD Pensando‡

Abstract– For security, isolation, metering and other purposes, public clouds today implement complex network functions at every server. Today's implementations, in software or on FPGAs and ASICs that are attached to each host, are becoming increasingly complex, costly and bottlenecks to scalability. We present a different design that disaggregates network function processing off the host and into shared resource pools by making novel use of appliances which tightly integrate general-purpose ARM cores with high-speed stateful match processing ASICs. When work is skewed across VMs, such disaggregation can offer better reliability and performance over the state-of-art at a lower per-server cost. We describe our solutions to the consequent challenges and present results from a production deployment at a large public cloud.

1 Introduction

All major cloud providers implement stateful network functions at their servers. These network functions are essential for network virtualization (e.g., private address spaces [75, 88]), enhanced security (e.g., stateful firewalls [14, 15]), load balancing [56, 87], QoS [62, 68, 92] and cost metering [5, 9, 20].

The key challenges in implementing stateful network functions in a virtualized context are three-fold:

- First, the state that must be maintained and accessed at line-rate can be per flow (for stateful firewalls) or per endpoint (to virtualize IP addresses) and can exceed 100MB for many virtual machines. Programmable switches [24, 25] have small SRAMs and are hence appropriate only in niche cases such as to only support a small subset of all flows [83, 85] or in bare-metal settings where the cloud provider has no access to the servers [41]. The most widely-used NF implementations today combine software in host virtual switches [52, 57, 88] with FPGAs or smart NICs that are directly attached to the servers [6, 58].
- Next, attaching FPGAs and smart NICs to each server is wasteful because these cards must be provisioned to meet the peak anticipated usage at each server but the actual usage is far below the peak most of the time, and on most servers. Moreover, VMs that use networking features which are only supported by the latest FPGA or smartNIC cannot be deployed on older hardware; in turn, this can lead to a sizable waste of non-networking resources.
- Finally, the tail performance is limited today. For example, in the three largest public clouds, we will show that the number of new connections per second a VM can support is well below the NIC capacity and is likely bottlenecked on stateful NFs that are applied on each new connection. Customers who deploy middleboxes in VMs (such as the Palo Alto VM-series firewall [33]) to secure access to their other VMs are forced to deploy many more middleboxes to offset the performance limitations in the provider's NF processing [35].

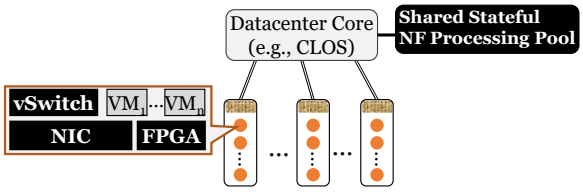


Figure 1: Today, stateful network functions are implemented on-host in virtual switches, NICs and FPGAs shown with a dark background. We propose to disaggregate stateful NFs, i.e., also process them in shared resource pools located elsewhere in the datacenter.

We propose to disaggregate the processing of stateful network functions into shared off-host resources pools as shown in Figure 1. Similar to how a customer can pick the CPU or memory for a VM, customers can also now pick a floating network interface (fNIC) which explicitly specifies NF requirements (e.g., # new flows per second, # concurrent flows) as well as the network capacity in Gbps. When deploying a VM, we allocate resources for the fNIC either on-host (that is, on the vswitch and the FPGA), or at an off-host shared resource pool or some combination at both locations.

We call out a few advantages from such a disaggregation

Stateful NF	State at each VM	Computation
Private address spaces [2, 10, 22]	A dictionary that maps customer's private addresses to the provider's physical addresses; one entry per remote endpoint that the VM speaks with.	Lookups, adds and deletes into the mapping dictionary
Stateful ACLs [32, 36]	Per ongoing flow that has passed the ACLs, a hashmap containing the flow's five tuple and the reverse five tuple	Lookups, adds and deletes into the per-flow hash table
Billing [5, 9, 20]	Total bytes, sliced by windows and per billable communicating entity such as a datacenter or a cloud service; also, bursts and peak rates	Multiple counters and sketches
Stateful NATs, load balancers	Per ongoing flow, the new *flow* to masquerade as.	Lookups, adds and deletes into the rewrite dictionary

Table 1: Some example stateful network functions that are implemented at cloud VMs and the associated state and computation. For more details, see [52, 57, 88].

of stateful NFs. First, we will show that for most of the time most of the VMs require much fewer NF processing capacity than the peak. We can thus reduce cost and power usage by equipping servers with less capable FPGAs or smart NICs and handling all of the spillover load at the shared resource pools. Next, we can deploy VMs which require novel NF processing on any server in the datacenter, including on older hardware, as opposed to restricting to just servers that have the latest FPGAs or smart NICs. As noted above, doing so reduces the fragmentation of non-networking resources. Third, we show how to increase the tail performance for VMs well beyond what is achievable from using the single FPGA or smartNIC that is attached to the host; for example, the number of new connections-per-second a VM can accept may only be limited by its NIC capacity. Doing so reduces cost and eases the deployment of middlebox VMs.

There has been much work on resource disaggregation. Disaggregating stateful NFs is similar in some ways to prior works that disaggregate resources such as memory, storage or GPU [45, 63, 74, 91] but there are a few key differences. One challenge is with regards to implementing a high-performance shared NF resource pool. The pool must simultaneously support large state and high-speed packet processing (e.g., 100s of GBs of states at multi Tbps packet processing rates). Doing so requires coherent access over a large memory at a high-speed. Programmable switches [24, 25] can process at multi Tbps but only have about 1GB of SRAM per switch. We have implemented an appliance which can be thought of as a bag-of-NICs wherein each NIC contains match-processing-unit ASICs that are programmable in P4 as well as a large coherently-accessible memory. Each appliance has 12 NIC cards, each card has a power draw of 75W, 16GBs usable for NF state and can process duplex packets at 100Gbps.

Another novel challenge from disaggregating stateful network functions is that fault tolerance shifts from a fate-sharing mode to a single point of failure. That is, when an FPGA or a smart NIC fails, only the VMs on the corresponding host fail but when an appliance (in the shared NF resource pool) fails the impact is felt by any VM whose fNIC happens to be allocated on that appliance. Naïvely replicating the state of network functions is hard because both primary-backup style replication [44, 51] and Paxos-like protocols [46, 48, 50, 81] queue requests while the state is being replicated. In the case of stateful NFs, requests can be any packet that changes state and so holding requests at speeds of hundreds of Gbps will require a very large packet buffer. We show how to replicate state *in-line* by ping-ponging packets between the replicas (pairs of programmable NICs) effectively buffering the state-changing requests on the network wires.

To the best of our knowledge, we are not aware of any prior work that disaggregates stateful NFs such as connection tracking firewalls or uses programmable bag-of-NICs appliances or supports in-line state replication. Some works offload specific stateful NFs to top-of-the-rack switches [41, 83, 85, 95] but do not support a rich class of NFs and the memory limit on Tofinos restricts them to only speedup a small subset of flows. Andromeda [52] deploys dedicated software middleboxes to process NFs but does not support stateful NFs citing concerns such as "state loss during upgrade or failure" and "transferring state when offloading". We discuss other related work in §7. To sum up, our key contributions are:

- We build a case to disaggregate stateful NFs by studying the functions and telemetry at a large cloud provider (§2).
- We present Sirius which disaggregates a rich class of stateful network functions onto pools of P4 programmable NIC cards. We show how to replicate state inline between pairs of nearby cards such that individual card failure does not adversely impact ongoing connections (§3.2). We discuss multiple disaggregation design-points including those that split or migrate the load of a VM across different NF processors (§3.3) and show a programmable NIC implementation that achieves better performance-over-cost than state-of-the-art (§4).
- We report results from a production deployment which, in part, show that when VMs offload onto Sirius, their stateful NF processing capacity improves by about 10×.

2 Background and Motivation

2.1 Stateful Network Functions

Table 1 lists some stateful network functions that are supported by public cloud providers. As the table notes, some NFs must maintain per-flow state whereas others keep state at coarser granularity. Counters and sketches are used to measure network usage for billing and diagnostics [73]. Customers

can also configure (add to) the stateful NFs on their VMs. As exemplars, we discuss two kinds of stateful NFs that are widely used in production but less well known academically.

First, virtual network peering [8, 12, 23] allows VMs in different virtual networks to communicate. Doing so requires all VMs in participating vnets to know how to map a virtual address belonging to any vnet into the corresponding physical address. This mapping must be kept up-to-date whenever VMs are deployed or migrated. Today's vnet peering implementations cache the map in a stateful layer at VMs [8, 12, 23].

Next, private links [7, 11, 29] let VMs communicate with PaaS services that have public IPs on a more direct path. For example, when a VM in AWS reads from an EBS volume which has a public IP, naïvely, such traffic must go via the cloud egress gateway similar to traffic to any public IP and then turn back towards the cloud store. Private links are more efficient and secure by having such traffic go directly to the cloud storage; a stateful layer at each VM encapsulates the outgoing traffic based on the VMs vnet id and the private IP address of the PaaS service and, in the reverse direction, a stateful layer at the PaaS service remembers which virtual and physical addresses a flow comes from when decapsulating (so-called *stateful decap*) and uses that state to encapsulate packets so that they go back to the appropriate VM.

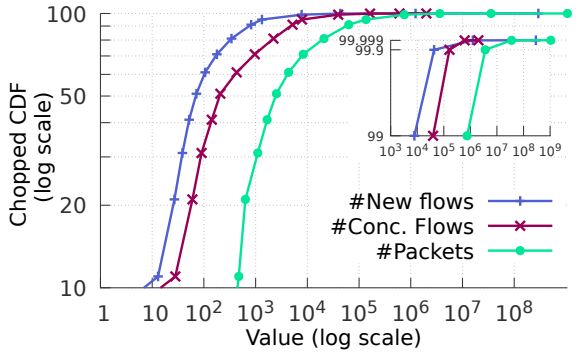
To sum, associated with each VM in the public cloud, providers implement numerous stateful network functions. Among the NFs considered in this paper, the connection-tracking firewalls, NATs and load balancers are the most intensive – they all require per-connection state. The total state to maintain per VM is often large since there can be hundreds of distinct *rules* to apply: one per private link, stateful ACL or vnet peer. NF actions on new connections are often implemented in software due to complexity and ease-of-programmability [52, 57, 88] whereas the per-packet actions are implemented in FPGAs or ASICs [6, 58].

2.2 NF workload at a public cloud

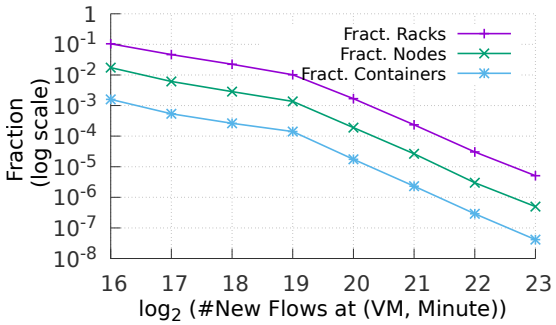
We characterize the usage of network functions at Azure. For each VM and each minute, we obtain the number of newly established flows, the number of active flows and the byte and packet counts. Our results here summarize metrics from a three month period. A typical minute has reports from $O(10^8)$ VMs and $O(10^7)$ nodes. Our key findings are as follows.

Skew in load for NFs: Figure 2a shows that the load for network functions is skewed; we measure load in terms of the number of newly arriving flows which must be verified to be policy compliant, the number of concurrently active flows for which state has to be maintained and the number of packets being exchanged. We see that the median load is multiple orders of magnitude smaller than the peak load. The inset zooms in on values further on the tail.

When the load is skewed, provisioning every host for the peak (e.g., by adding FPGAs or smart NICs), can be costly and most of the NF processing capability remains unused. We aim to provision hosts for the average load and handle the excess load using a disaggregated, logically shared, pool.



(a) Cumulative distribution function (CDF) of the NF Load at VMs; axes are in log scale and points on the CDF are unique (VM, minute) tuples.



(b) VMs that have high NF load are spread among many nodes and racks.

Figure 2: Characterizing the workload for stateful NFs at a large public cloud; data collected across over 10^8 VMs in a three month period.

Metric	Containers	Nodes	Racks
σ / μ	14.23	5.00	0.67
99th / μ	13.54	10.49	2.52

Table 2: Coefficient-of-variation (=stdev σ/ avg. μ) of the number of newly arriving flows per minute at each VM compared to the same metric when rolled up into the nodes or racks that contain the VMs.

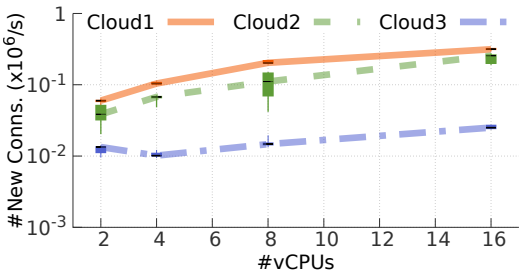
Containers with high NF load are spread throughout the network: Figure 2b zooms in on VMs and timewindows (minutes) which report high NF load; the x axes is a logarithmic bin, that is $x = 18$ denotes that the numbers of new flows in a (VM, minute) was in the range of $[2^{17.5}, 2^{18.5})$. The bottom-most line on the figure reports the fractions of distinct containers which exhibit high NF load. If the high-load containers were concentrated into a few nodes and racks, then the fraction of nodes and racks which show high load will be no larger than the fraction of containers. However, the figure shows that highly-loaded containers are spread across many more nodes and racks. The case for other NF load metrics is similar. About 10% of the racks have at least one container which reported over 50000 new connections in a minute.

Variation in NF load: At the granularity of individual VMs, we observe sizable temporal variations in NF load of up to

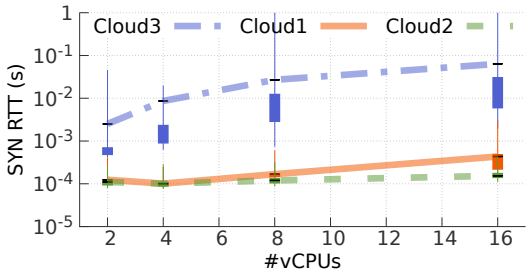
one order of magnitude larger than the median (see Figure 15). However, the variability appears uncorrelated spatially. That is, the sum of the load of all the VMs in a server or a rack has smaller coefficient-of-variation (see Table 2). Shared pools of NF processing capability thus can be provisioned with smaller peak to average ratios and will be more cost-efficient.

2.3 Characterizing NF Performance

To measure the state-of-art stateful NF performance of public clouds today, we deploy pairs of VMs of different sizes, add a stateful ACL to the *client* VM and initiate TCP connections in an open loop to the other *server VM*. We progressively increase the connection initiation rate and measure the maximum rate that was achieved and the connection establishment latency. Our tool is multi-threaded and asynchronous. We appropriately change various configuration variables and achieve better results than listed in public datasheets [13, 17–19, 34]. Figure 3a shows the maximum number of new connections per second (CPS) achieved by VMs of different sizes on the three largest public clouds today. All VMs are identically configured Ubuntu Linux instances. The variation is over experiment runs likely due to performance interference on the VMs or on the network path; we repeat each point at least ten times. The figure shows that increasing the VM size tends to increase the CPS perhaps because the per-VM networking limits improve [3, 21, 28]. However, the highest CPS across all public clouds and experimented VMs is 0.3M. Figure 3b shows the latency between sending a SYN and receiving a SYN-ACK. In the latency plot, we only use trials where most of the connections succeed to avoid latency cliffs. The figures show that processing the stateful NFs which are deployed in public clouds today represents a sizable bottleneck– there is a sizable latency when establishing new connections and VMs are limited in the number of new connections per second that they can sustain.



(a) Maximum Connections Per Second achieved.



(b) Latency between sending a SYN and getting a SYN-ACK.

Figure 3: Benchmarking connection establishment rate and latency at different VM sizes on three public clouds. The lines connect the average value, whiskers go from 1st to 99th and boxes go from 25th to 75th percentiles.

3 Disaggregating NF processing in Sirius

Sirius offers a new API to offload network function processing into pools of appliances that contain custom programmable cards. Each VM or container can specify a floating NIC (fNIC) with requirements on the following dimensions that relate to processing network functions:

- The number of new flows per second (CPS)
- The maximal number of concurrent flows
- Network capacity
- Feature, capability selection and ruleset size

Values for some of these dimensions are already in cloud provider and NVA vendor datasheets [3, 13, 17–19, 21, 28, 34]; Sirius also allows off-the-shelf fNIC sizes that users can pick from (e.g., a *small* fNIC) to help with configuration.

We extend our cloud VM allocator [65, 93] to provision fNICs using either resources on the smartNICs that are attached to servers or one or more Sirius appliances. Disaggregation lets VMs be placed on servers that may not locally satisfy fNIC requirements. Allocating fNICs to cards is an instance of the multi-dimensional bin packing problem [61, 86]. A better packing will map more fNICs onto fewer cards. We considered different heuristics and found that heuristic choice improves efficiency only when the fraction of large fNICs (whose resource needs are a substantial fraction of the card capacity) is high. In the rest of this section, we discuss the disaggregated datapath, *inline* state replication and methods to split or move an fNIC's load between multiple cards. Our design is modular and can work with different implementations of the shared processing pool; in §4, we discuss our P4 programmable cards which in our tests can serve over 3M new connections per second and 16M concurrent connections while processing a rich set of stateful NFs.

3.1 Connectivity and availability

The NF processing pool in Sirius is a collection of appliances. In our prototype, an appliance is a pair of 3U servers with six programmable cards each in PCIe slots. Each card has two 100Gbps QSFP+ connectors and 32GB DRAM.

Reliability, efficiency and flexibility were our key considerations when deciding how to connect Sirius appliances within a datacenter. Adding an appliance to each rack may lead to under-utilization (and fragmentation) since not every rack may have enough demand for NF processing. We also aim for

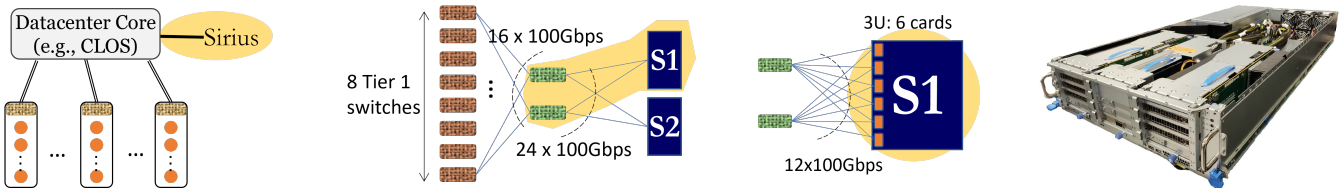


Figure 4: Connectivity diagram. Please read the figures left to right with each subsequent figure fleshing out the portion that is highlighted in the preceding figure. A Sirius appliance can be thought off as a pair of servers labeled S1 and S2 respectively that are connected as shown. Sirius assigns offloaded floating NICs of VMs to the programmable cards in the servers. We deploy more appliances to keep up with the total NF load.

a high level of reliability since NF processing is crucial for security and reachability in public clouds, e.g., for ACLs and private address spaces. Our goal is to ensure that the failure of any one server, card, link or switch must not degrade NF processing capability by a substantial amount. We also aim to independently scale out the NF processing capability as demand increases, e.g., by adding more appliances. Finally, we want VM placement to not be constrained by the availability or lack thereof of NF processing capability; that is, VMs located anywhere in a datacenter should be able to, when needed, use NF processing from the Sirius pool.

We choose to connect Sirius appliances as shown in Figure 4. Our minimum deployment unit is one appliance beneath two top-of-rack (ToR) equivalent switches that connect to the rest of the datacenter network similarly to other ToR switches. Such connectivity also ensures that any VM in any of the racks connected underneath the same CLOS will have equivalent access to Sirius's appliances thus realizing a large shared NF processing pool. In our experience with Sirius, the primary bottleneck is the NF processing capability and the state on the cards. That is, the number of new flows arriving per second which must be validated and the number of concurrent connections for whom state must be maintained (e.g., in a stateful load balancer). Our measurements show that the added latency incurred by traffic passing through an appliance is small relatively because traffic between randomly placed VMs in the public cloud almost always bounces off a switch in the CLOS tier; in particular, the increase is negligible for north-south traffic (which enters or leaves the datacenter). Finally, the connectivity diagram in Figure 4 preserves access to the NF processing capability under the following conditions.

1. At most one of the two green switches in front of a Sirius server fails.
2. At most one of the two links that connect a given card to the switches fails.
3. At most half of the links that connect the green switches to the red Tier-1 switches fail.
4. At most half of the red switches fail.

The last two conditions above ensure that other racks in the CLOS will have at least one valid path to the green switches. By preserving access to the bottleneck resource, NF processing remains unimpeded and Sirius will still be able to support high rates of new flows per second and concurrent flows.

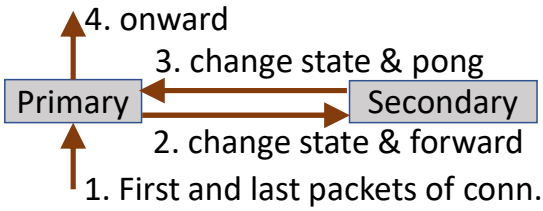


Figure 5: In-line replication of connection state in Sirius by ping-pong'ing packets that change state to both the primary and secondary cards.

3.2 In-line Connection State Replication

To avoid individual card failures from affecting ongoing connections, we duplicate connection state across two programmable cards. A key novel aspect here is that we do so without buffering packets. Due to the very high packet rates that these cards handle, holding packets in the primary card until state is established on the secondary card, as is done typically to replicate state [46, 48, 50, 81], will require very large buffers. We discuss a method that replicates state without any additional buffering by ping-pong'ing the packets of each connection that change state. As shown in Figure 5, for example, SYNs of a TCP connection which will establish state on the primary card are also forwarded to the secondary card. The secondary card also establishes state for this connection in its local memory and forwards the packet back to the primary. The primary card then transmits the packet to the destination in the usual way. Both cards independently delete the state of connections which remain idle beyond an age out threshold. Besides avoiding additional buffering, such *inline* state replication requires only a small code change to send and process ping-pong messages since the code to check rules and update state can be reused.

Each card pair (primary and secondary) exchanges heartbeats and fails over independently. That is, if the primary misses several heartbeats, the secondary card will receive all of the traffic on fNICs that were assigned to the pair. To achieve such failover, both cards announce BGP routes for the fNICs' virtual IPs; the primary card announces a shorter AS path than the secondary.[1] At a slower timescale, a different software controller provisions new replicas (e.g., pairs a newly promoted primary with a new secondary card) and schedules bulk state replication (which we describe below).

[1] We discuss corner cases in failover in §A.1.

The controller also reduces allocatable appliance capacity if necessary based on the number of cards that are operational. Two control loops at different timescales are commonly used to react to faults [67, 80]; to our knowledge, we are not aware of its use in replicating the state of network functions.

We observe that most of the connection state turns over quickly. For example, a usage stream that has 4M new connections per second and 32M concurrent connections has the average connection lasting 8s. Thus, waiting a bit will allow us to move much less state, which belongs to the long-lived connections, with the trade-off being a small increase in the period for which state is present on only one card.

The goal of our bulk synchronization is to replicate checkpointed state from one card to another quickly. There are multiple ways to implement checkpoints; we append an epoch value to each record in the state and atomically increment the value of the current epoch to take a checkpoint since then all records with a smaller epoch value will belong to the checkpoint.[2] To copy a checkpoint between paired cards, the ARM cores on the cards move state in batches over a reliable transport. We tradeoff the overhead of copying checkpoints with an increase in the period wherein only one copy of the state exists in the following additional ways: we prioritize moving the state of long-lived connections since other connections may close before needing to be moved and we pace the copy messages so that resource contention (on the memory bus and network) does not adversely affect normal activity.

To sum, replicating connection state between a pair of cards has the following costs and benefits. On the costs, storing each record at two cards halves the total available state that a Sirius applicance can maintain. The NF capacity, say in terms of connections per second that can be handled by an appliance, also halves for the same reason. The connection setup latency increases due to the ping-pong. Also, bulk synchronization, when triggered, uses memory and network bandwidth. On the benefits, the failure of a single card only impacts ongoing connections for the period before traffic failovers onto the secondary card. In-flight connections, that is, connections whose state is not yet present on both cards may only have to retransmit some SYNs (and FINs). To see why, observe that at any of the four steps for a new connection in Figure 5, the failure of either cards at best requires a retransmission.[3] Finally, planned card failures can be handled without any impact as so: (X1) promote the secondary and pick a third card to be the new secondary, (X2) take a checkpoint and (X3) initiate bulk synchronization. Upon completion of the bulk synchronization, the old primary card can be taken offline.[4]

[2] We use a small circular counter to track epoch values.

[3] We use a poison bit on the record written to the primary card which will be deleted only after the packet pongs back from the secondary to handle failures that may happen after step 2 in Figure 5.

[4] As a proof sketch, note that any new connection that reaches the new primary (old secondary) after X1 will reach the new secondary via the ping-pong method. Furthermore, all state at the time of the checkpoint, X2, will have been reliably copied to the new secondary.

3.3 Dividing NF load appropriately

So far, we have shown that the state for NFs can be maintained in a disaggregated resource pool with high availability. Here, we discuss different design points which divide the NF load between smart NICs that are directly attached to servers and one or more cards in the disaggregated Sirius pool.

3.3.1 Pin fNIC locally or to one card pair

Here, the load of each fNIC is assigned either to the on-server smart NIC or to a pair of cards as discussed in §3.2.

To realize pinning to a card pair, the outgoing packets of an fNIC are encapped in an NVGRE tunnel and sent to the chosen primary card in the Sirius pool which applies NFs on the packets and forwards them on to the destination. Traffic in the reverse direction takes an analogous path, first reaching the appliance/card which applies NFs and then forwarded to the VM if appropriate. We implement the encap and decap logic at the smart NICs on the servers.

3.3.2 Disaggregation Cost/ Benefit Analysis

The above design point already leads to substantial cost savings from disaggregation because one appliance can handle the NF load of over 24000 VMs on average. We compute this number as follows. In §6, we will show that each card used by Sirius can process over 16M new connections per second (CPS) with an extensive set of NFs. There are 12 cards per appliance. We assume that each VM has an average CPS load of $4K$ which is $400\times$ the current median load per Figure 2a and we halve the NF capacity to replicate state as discussed in §3.2. Hence, the cost for the additional switches, cables and the appliance in Figure 4 amortize well. Moreover, regarding peak load and temporal variations, note that these 24000 VMs may be distributed over hundreds of racks and, as we saw in Table 2, the total load over many rack has much lower variability. Thus, Sirius can meet SLOs with much smaller surplus capacity in its disaggregated pools.

3.3.3 Split the load of an fNIC across multiple cards

With the previous design point, the maximum size of an fNIC is limited by the capacity of one card in the Sirius pool. Moreover, as we will show, packing VMs into appliances is less efficient when the size of the balls (i.e., the fNIC size of a VM) becomes close to the size of the bins (i.e., NF capacity in one card). Sirius appliances can also be implemented using diverse hardware and different NFs may be better suited to different hardware. To this end, we aim to split the load of an fNIC across multiple cards or appliances. That is, different portions of the traffic entering or leaving one VM can receive their NF processing at different cards.

Consider splitting the load using a hash function–$\mathsf{hash}(\text{local IP}, \text{remote IP}) \bmod n$, in the encapper, to pick

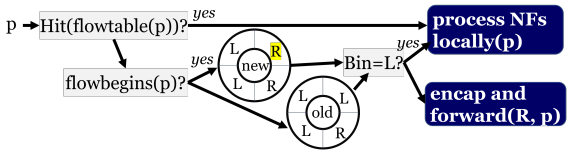


Figure 6: When spilling over NF load, as discussed in §3.3.4, we hash packets into bins and assign the bins to either local (L) or remote (R) NF processing. The figure shows two assignments new and old where the proportion of load that is processed locally is 50% and 75% respectively. When bins are re-assigned, the figure shows how we reduce the state that must be moved by using both bin assignments for a short duration.

from among *n* different cards. Such a *symmetric hash* ensures that the traffic of a flow in both directions will be processed at the same location which is required by some NFs (e.g., NATs [55]). While this requirement can be met in other ways, symmetric hashing requires no additional state at the encapper and decappers and we use hash functions that are easily implementable in NICs. Next, some NFs require groups of flows to be analyzed at one location. For example, a usage meter or a DDoS detector may want to count all bytes from a VM that leave the datacenter. Our experience is that most such NFs have mergeable actions [39], for example, to compute the total byte count, we can add up the partial sums from different processors. Many sketches (such as hyperloglog [54], count-min [49]) are mergeable with small reduction in accuracy [39, 40]. Finally, when an fNICs traffic is split across multiple NF processors, the ruleset corresponding to the fNIC must be installed at all of the corresponding processors; in practice, doing so adds overhead but is tenable because the total state at the NF processors is dominated by the per-flow state, counters and sketches rather than ruleset size; similarly processing the ruleset dominates the computation at the NF processor over the one-off installation of the ruleset.

3.3.4 Use Sirius as a load spillover

Thus far, all our load allocations have been static. That is, the whole or a portion of an fNICs traffic was allocated statically to the server's smart NIC or to a Sirius pool. An alternative is to move NF load that cannot be processed locally *dynamically* into a Sirius pool. For example, we may start processing all of the NF load locally on the server's smart NIC and when the total load nears smart NIC capacity, shed the excess load into the Sirius pool. Doing so will allow cloud providers to offer burstable SLOs on NF processing.[5] One Sirius appliance can scale to even more VMs compared to the pinned design-point above because only the excess the load of fNICs will be steered to the appliance.

Naïvely supporting such dynamism would require *moving* the state of NFs. For example, if a portion of the traffic that was to be processed on the smart NIC must now spill over into a Sirius pool then the corresponding state of all NFs must move. Intuitively, doing so is complex and our key contribution is to do so efficiently and correctly. First, our design aims to reduce the amount of state that must move to the extent possible. We hash packet headers, partition the resulting hash value into a fixed number of buckets (say 32), and assign different buckets to be processed for NFs at different locations. To move load, we change the bucket assignments; that is, to move 25% of the load from the smart NIC to a Sirius pool, we would reassign a quarter of the buckets that were being processed at the former location to the latter. Instead of moving all of the NF state that corresponds to a moving bucket we move *lazily* as shown in Figure 6. Effectively, newly created state (e.g., state for new connections) immediately reflects the current bucket assignment but for the previously established state, we delay movement by a short period (τ). Connections with duration below τ will not move and we observe that long-lived connections comprise a small fraction of all connections. The trade-off here is that we can rebalance load less frequently (once per τ). Our second idea is that many kinds of state can be re-created at the new NF processing location by just processing packets. For example, stateful ACLs insert the five tuples into a dictionary. The necessary information to create such state – the five tuple – is present in every packet of a flow and so, instead of moving state, we mark and steer packets to their new NF processing location. For state that cannot be recreated in this way, we craft new packets that include the packet header of the original flow and the state and transmit these packet to the new NF processing location. When the new location acknowledges creating the requisite state, the previous processing location deletes its state and load steering will exclusively use the new bucket assignment.

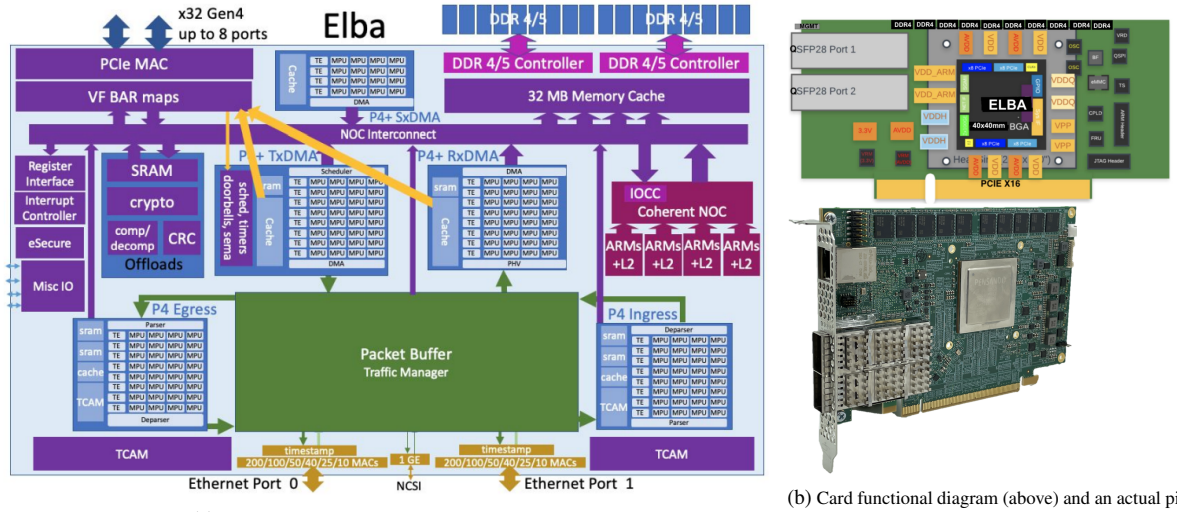
4 Efficient and high-rate NF processing

Thus far, we have discussed how to disaggregate the processing of stateful network functions in public clouds by using cards that (1) support inline replication of state (§3.2), (2) support various disaggregation design points including load splits and state movement (§3.3), and (3) implement a rich set of network functions (Table 1 and §2.1). Any implementation that satisfies these requirements can be used in this design including, for example, software-only or switch-only implementations. Here, we discuss our implementation which uses a specific kind of programmable NIC and compares favorably on functionality, performance and cost.

To process stateful network functions efficiently and at a high rate, we use the P4 programmable card shown in Figure 7 which has two 100 (or 200)GbE QSPF+ connectors, multiple pipelines that are programmable in P4, coherent shared memory, ARM cores for the more complex data plane processing and specialized logic for encryption and compression.

Relative to FPGA-based smart NICs [58], conjoining match-process-units (MPUs) that are programmable in P4

[5] Burstable allocations are already available for CPU and memory. They allow short-duration bursts or price differently the average and peak usage.



(a) Architecture diagram of our programmable ASIC.

(b) Card functional diagram (above) and an actual picture (below).

Figure 7: Hardware used to enable efficient and high-rate NF processing.

Metric	Sirius DSC-200	Other Packet Processors Tofino	Tofino2
Bandwidth (Tbps)	0.4	6.5	12.8
Memory	32GB	0.48GB	0.8GB
# Match Action Pipelines	5	4	4
# Stages/ pipeline	4	12	20
Packet Buffer	up to mem.	22MB	64MB
Integrated general-purpose cores	16	0	0

Table 3: Salient differences between packet processing hardware; Sirius uses the DSC-200 card [4] to support stateful network functions at a high scale and performance.

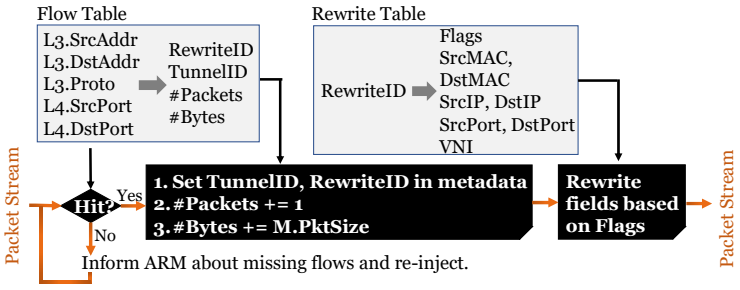


Figure 8: Stateful Load Balancer with NAT as implemented by Sirius.

with general purpose ARM cores gives us better programmability and performance at a lower power cost. Intuitively, power usage decreases because unlike FPGAs which expose general *gate-level* programmability, our card only exposes programmability in P4 that is needed to process protocols and stateful NFs efficiently. We use the ARM cores to handle programs that may be challenging to implement in P4 [72] such as reliably exchanging state migration messages (see §3.2).

Packets flow through one or more ingress and egress P4 pipelines and go through ARM cores only if needed. Each pipeline operates at 400Gbps (over 50M packets per second) thus ensuring line rate on both interfaces. We parse a packet once and populate a packet header vector (PHV) which is used by later stages. Each pipeline has local SRAM and TCAM to store high bandwidth tables and can also access the shared DRAM through a coherent shared memory which hides memory latency. A table engine at the beginning of each stage protects against stalls by processing multiple PHVs, issuing high latency reads in advance (e.g., to the DRAM), and moving to an MPU the next PHV for which all data is available. Each stage has multiple match-process-units (MPUs) which never stall and have dedicated write paths to the stage data buffer wherein writes are merged at a bit level to allow multiple MPUs to update different fields of a PHV. The MPUs implement a novel domain-specific instruction set architecture with an emphasis on bit field manipulations and fast header updates. We also use wide instructions (e.g., 64bit wide) which lets us use richer encoding and fewer instructions.

Coupling ARM cores and MPUs: Our card connects the P4 pipelines via a high speed network-on-chip (NOC) to a full system-on-chip (SOC) subsystem with multicore ARM A-72 CPUs. P4 programming determines which portions of packet data, headers, or metadata should be delivered to the DRAM and ARM on a per-application, per-packet basis. To support chained operations which may combine a P4 control operation with non-P4 operations, such as encryption or data integrity checksum verification, we attach a chaining buffer directly to the NOC to support high-bandwidth multi-hop chaining.

Illustrative Example: Using the case of a stateful load-balancer, we call out key aspects of how our hardware implementation improves upon the state of the art. Figure 8 shows a functional view of our stateful load balancer implementation. The relevant state (table shown with light background) is stored in DDR memory on the card. The logic boxes (shown in dark background) are implemented in the MPU pipelines and the exception path (for a new flow which does not have a

hit in the flow table) is handled by ARM cores. When load balancers are implemented without per-flow state (e.g., using a stateless hash function), any change to the pool of targets will disrupt ongoing flows; for example, a failure in one of the targets will cause the hash function to change from mod-n to mod-$n-1$ and all flows whose targets change will be disrupted [83]. Recognizing this issue, several large enterprises deploy stateful load balancers which remember per flow the target that the flow was assigned to [56, 83, 85, 87]. Prior work that proposes to accelerate stateful load balancers is limited by on-switch memory, for example, Sailfish [85] uses the Tofino chipset to support a few thousand stateful connections per switch, while the other flows are processed in software and receive no benefits. In our tests, one card can support over 16M concurrent connections and 3M new connections per second. We note a few aspects that help us achieve such performance:

- Although the flow table (in grey on the left in Figure 8) has one entry per ongoing connection, the rewrite table uses indirection and can be significantly smaller in size.
- Our table datastructures allow for more expressive rewrites including changes to the MAC addresses. Thus, we can use a single rewrite table for multiple NFs beyond load balancing (e.g., NVGRE encap [30]).
- We allow partitioning the MPU programs (shown in dark in Figure 8) among multiple pipelines so as to leverage data proximity.
- We divide the table ownership between ARM cores and MPUs to avoid coordinating multiple writers.
- When a new flow arrives for load balance, an ARM core installs entries in the flow and rewrite tables and reinjects the first packet of that flow into the MPU pipelines.

Comparing with recent works [42, 47, 79, 94], two of the P4 pipelines in the DSC (the Ingress and Egress pipelines at the bottom of Figure 7a) resemble reconfigurable match tables (RMT) [42] except that the DSC also has pipeline-local SRAM and not just stage-local SRAM. However, unlike RMT, all of the DSC pipelines can access shared DRAM through coherent caches. The DMA pipelines ({Tx-, Rx-, Sx-}DMA in Figure 7a) are novel and are triggered by timers and doorbells from a programmable scheduler. PANIC [79] addresses chaining offloads and is similar to the DSC which also uses specialized offloads (for crypto, compression and others, see Offloads in Figure 7a). However, while the DSC chains offloads, offloads are not central to the use of DSC in Sirius. FlexCore [94] discusses runtime re-programmability of switches; they add and remove P4 functions on an SN3000 [31] switch with minimal disruption to ongoing activity. We do not discuss re-programmability of the DSC cards in this paper. dRMT [47] pools all of the per-stage memory into shared memory that is accessible to any stage and uses a run-to-completion model wherein a packet is fully handled at one processor (and not in a sequence of match-action stages as in RMT). Our card offers larger shared DRAM instead. While it is unclear how dRMT's scheduler, which calculates a static schedule at compile time to guarantee deterministic throughput and latency, generalizes to the case of stateful NFs, the DSC supports stateful NFs more simply by dividing the work between P4 pipelines and ARM cores.

5 Implementation

We have implemented several stateful network functions (including those in Table 1) on the programmable NIC shown in Figure 7 from AMD Pensando. We have also added new code to the smartNICs attached to the hosts in Azure to steer traffic to and from the disaggregated Sirius pool. The resulting system, alongside software controllers to provision and monitor the fNICs, is in public preview at Azure [1] .

6 Evaluation

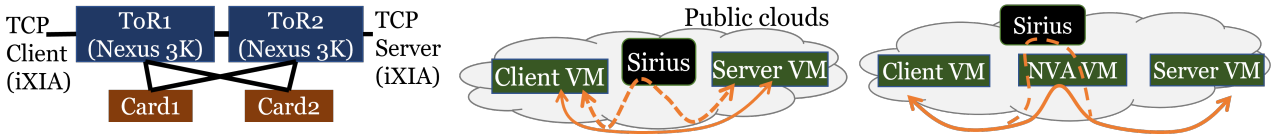
First, in a lab setting using full line-rate traffic generators, we show results for how the programmable NICs used in Sirius handle stateful network functions. We also evaluate key failure scenarios. Next, we report results from Azure wherein fNICs of virtual machines and network virtual appliances are offloaded to Sirius.

6.1 Methodology

Figure 9 shows our three experimental setups. On the left, in a lab, we use a traffic generator that sends and receives packets at hundreds of Gbps. We also mimic failures of the ToR switches, links, and cards to evaluate our state replication.

The other two setups use Sirius's production deployment in Azure. Figure 9b measures the performance between virtual machines (VMs) and Figure 9c measures the performance of network virtual appliances (NVAs) [13, 18, 33] when deployed on VMs. Here, we compare the default method that the public cloud uses to process stateful NFs versus offloading those NFs onto Sirius. In Figure 9b, we offload the floating NICs of both the VMs onto Sirius. In Figure 9c, we offload the floating NICs of the middlebox VM onto Sirius.

Stateful NFs: For the setup in Figure 9a, each card enforces a large prioritized set of stateful ACLs. As shown in Table 4, each ACL rule is a conjunction of predicates on sets or ranges of source and destination addresses, ports and protocol. Rules apply in priority order and may either accept or deny a connection. Some rules are specific to individual VMs whereas others apply to all VMs in a vnet or subscription. Recall from §2.1 that stateful firewalls maintain per-flow state of all ongoing connections so as to admit traffic in the reverse direction. As discussed in §3.3, the cards also encap and decap the packets to intercede on traffic transparently. For the VM-to-VM setup in Figure 9b, VMs in the public cloud already run many stateful NFs by default, for example, to virtualize



(a) Testbed (b) VM to VM: Default vs. offloading to Sirius (c) NVAs with fNICs offloaded to Sirius

Figure 9: Evaluation setups. On the left is a testbed where we use iXIA breakingpoint [27] to generate traffic of up to 100Gbps. The other two figures depict our experiment setups in Azure. We deploy VMs of different sizes and compare the performance when the Azure-specific stateful NFs are offloaded onto Sirius. On the right we measure the performance of network virtual appliances (such as the Palo Alto VM-series firewall) when using floating NICs that offload onto Sirius.

Granularity of ACL set	#Rules	Total #Conjuncts (prefixes or ranges)			
		Src IP	Dest. IP	Src port	Dest port
fNIC level	202	5102	5102	1021	1021
Subnet level	26	1168	1168	141	141
Subscription	8	394	394	57	57

Table 4: The ACLs for a stateful firewall deployed on the cards in Figure 9a; note: some ACLs are unique per floating NIC whereas others are common across all fNICs in a subnet or an entire subscription.

Resource type	Resource Capacity					
#Cores	2	4	8	16	32	64
Mem. (GB)	8	16	32	64	128	256
NIC Capacity (Gbps)	1	2	4	8	16	30

Table 5: The capacity of various resources for the SKUs used in Figure 9.

their network [2, 10, 22] or to estimate traffic bills [5, 9, 20]. In addition, we insert 1000 prioritized stateful ACLs on the client VM; these ACLs are similar to those in the testbed experiment. For the middlebox experiment in Figure 9c, we configure each middlebox with the reference load specified by the middlebox vendor.

VMs: We evaluate popularly-used Linux Ubuntu SKUs at various public clouds as shown in Table 5. We choose VMs with varying numbers of cores, from 2 to 64 vcpus; the other resources vary roughly proportionally as shown.

Traffic: In Figure 9a, we generate UDP and TCP flows of different sizes at different rates in an open loop using a synthetic traffic generator [27]. This appliance must be physically connected to switches and so, in the public cloud experiments, we use VM-based traffic generators. The Linux network stack cannot generate small TCP connections at high rates, e.g., fewer than 50K zero-byte flows per core [82]. We use the TREX tool instead which, using DPDK, can generate TCP-like connections at much higher rates [37].

6.2 Processing Stateful NFs in Sirius

To sum, the experiment here will show that when supporting a rich set of stateful ACLs (Table 4) the programmable NIC used by Sirius can support up to 3M new TCP connections per second (Figure 10b) and over 50M UDP packets-per-second (Figure 10a). The latency to ping-pong state messages between a card pair is less than 40μs (Figure 10c).

In more detail, Figure 10 shows the thruput and latency when the two cards in Figure 9a are set up as a state replicating pair; that is, all state changes for SYNs and FINs are ping-ponged between the cards as discussed in §3.2. Each datapoint is a several minute experiment and the errorbars show the range of measured values.

Since some stateful NFs are evaluated on every packet, we first measure the maximum number of packets per second (PPS) that our card can support by having the traffic generator send the smallest possible UDP packets at the highest possible rate.[6] Figure 10a shows that our card supports over 50M packets per second.[7] Typical packets are larger, e.g., many are MTU-sized, and so our card can process complex stateful NFs at line-rate with a fair amount of headroom.

The end-to-end latency through the card for 64B and 1500B packets is 2.36μs and 3.14μs respectively.

We also vary the number of concurrent flows which increases the state on the card and can make NF processing more challenging. Figure 10a shows only a modest decrease in PPS up to 64M concurrent flows; state for these many flows uses up most of the 32GB of DRAM on each card.

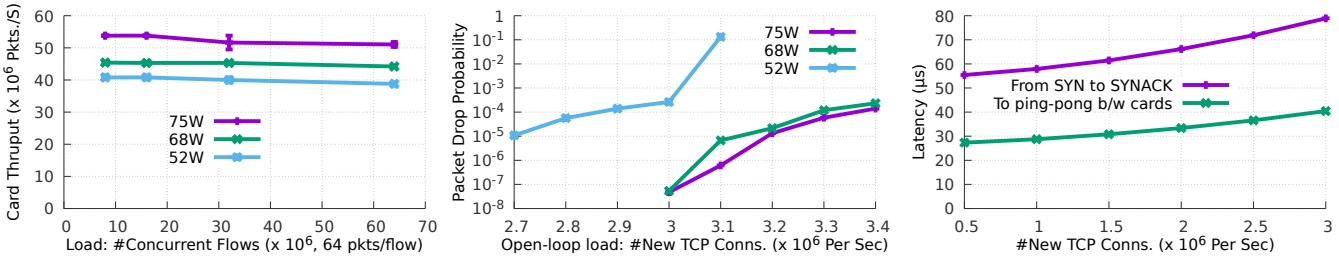
Finally, Figure 10a shows results for low power states of our card wherein we decrease the frequency of the MPU pipelines from their baseline value of 1.5GHz. Observe that we achieve 33% lower power draw with only a 25% drop in PPS. Thus, dynamic power cycling appears viable.

Next, when new connections arrive (or old connections finish) the state maintained in a stateful NF processor must change. To measure the maximum number of new connections per second (CPS) that one card pair can support, we have the traffic generator issue TCP connections in an open-loop with no payload.[8] Figure 10b shows the packet drop probability (y axes is in log scale) near our desired operating point of 3M CPS. Lower values to the right are better. Since SYN and FIN packets ping-pong between the cards, each card effectively processes twice as many state changes. The remaining packets of the connection, however, only go through the primary card. Also, recall from §4 that only the ARM cores change state and SYNs are reinjected into the MPUs after the ARM cores apply the ruleset. Figure 10b shows that while the 68W power state has very little effect on CPS, the lowest power state (52W) reduces the CPS to about 2.5M. We are not yet sure why and

[6] Each packet is 118B due to VxLAN tunneling with an interframe gap of 12B and the ethernet preamble of 8B [38]. Thus, on a 100Gbps link, the generator issues roughly 90M packets/s.

[7] Per previous calculation, this amounts to 60Gbps.

[8] 6 packets per connection: SYN, SYN-ACK, ACK, FIN, FIN-ACK, ACK



(a) The packets-per-second through our card when maintaining different numbers of concurrent flows.

(b) The fraction of packets dropped when our card is subject to open loop load, a line for each power state.

(c) Latency from sending a SYN to receiving a SYNACK and to ping-pong the SYN between cards.

Figure 10: Card measurements in the lab setup shown in Figure 9a: when applying per connection the thousands of stateful ACLs shown in Table 4, the figures show the thruput in PPS, packet drop probability and latency at different card power levels.

are looking into this issue.

For the CPS test above, Figure 10c measures the latency between sending a TCP SYN and receiving a SYN-ACK and the latency portion that is attributed to ping-pong between cards. The latency is flat at lower CPS load but grows super-linearly at higher demands likely due to queuing at the ARM cores or at reinjection. We note that RPCs can achieve a smaller latency [70, 84] by reusing connections and the latency here is better than that measured at the three clouds in Figure 3b.

6.3 Stateful NFs under faults

For the setup in Figure 9a, we have the traffic generator issue small TCP flows open-loop at the rate of 3M per second. The flows are spread over 16 floating NICs evenly allocated to the two Sirius cards. We examine the impact of three changes:

(a) planned switchover from Card1 to Card2,
(b) links between ToR1 and both cards go down and
(c) Card1 goes down.

For each scenario, we conduct three different experiments each lasting 60s and report average values. Each experiment comprises roughly 180M TCP connections and 1.08B packets. Table 6 shows that none of the flows *broke* as in there were no RSTs or connection time-outs in all three scenarios.

During planned switchover of load, as discussed in §3.2, Card2 advertises itself as the new destination for all of the floating NICs that were mapped to Card1. During the ensuing route reconvergence, the ToR switches drop 0.00316% of the packets and there are no drops at either of the cards. A naïve switchover would cause RSTs on half of the ongoing connections (all conns with state on Card1).

In scenario (b), where ToR1's links to both cards are down, the net available network capacity in/out of the cards halves but the CPS remains unaffected because, as noted in §3.1, Sirius retains large network capacity to the cards even when half of the connecting links fail. Table 6 shows that recovery here is slower and there are more drops because more routes must reconverge. The cards also see transient drops while their paths move over to ToR2.

Change	#Flow breaks	% of pkts dropped All	% of pkts dropped At Cards	Recovery Latency
(a) Planned switchover	0	0.00316%	0	1.89ms
(b) ToR1's links to both cards are down	0	0.00929%	0.0000227%	5.75ms
(c) Card1's links to both ToRs are down	0	0.00835%	0.0000201%	5.01ms

Table 6: Testing state replication under different fault scenarios in Figure 9a with 3M new TCP flows/s. Note, recovery in milliseconds and the extremely small fraction of packets that were dropped most of which are due to route reconvergence at the ToR switches. The drops at the Sirius cards are all packets that cannot be transmitted because the link to the next hop is down.

Scenario (c) mimics the failure of Card1. Here, the ToRs detect Card1 as being down and route all fNIC traffic on the backup BGP route to Card2. Contemporaneously, Card2 recognizes the failing peer, promotes itself to be the primary, and notifies the Sirius controller asking for a new secondary card. If card state was not replicated, half of all ongoing connections will receive RSTs in this scenario. Table 6 shows that no connections break. Instead, only a few packets are dropped most of which are at the ToRs. A few flows retransmit SYNs and FINs[9] which may have been lost without completing the pingpong in Figure 5 but none of the connections timeout.

6.4 VM-to-VM: Offloading fNICs to Sirius

Figure 11 shows the maximum connections per second achieved between pairs of VMs for the scenario in Figure 9b. Recall from §6.1 that we use TREX, a DPDK based generator on the VMs to create small TCP connections as many and as quickly as possible. These VMs have the default stateful NFs from public cloud and we add roughly 1000 randomly generated stateful ACLs (see §6.1). The figure shows that most of the VM SKUs, when onboarded on to Sirius, are only limited by the NIC capacity. That is, our tool makes as many connections as possible given the capacity limit of the NIC.[10] The figure shows that with Sirius, one VM pair can achieve

[9] 0.0062% and 0.0126% of the TCP flows respectively

[10] 6 packets per TCP connection each of which is 118 bytes after VxLan encapsulation which translates to 176.5K CPS per Gbps of NIC capacity. NIC capacities of the VMs are as shown in Table 5.

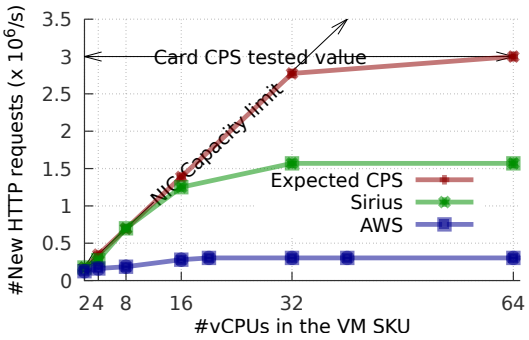


Figure 11: When the floating NIC of a VM is mapped onto the Sirius pool, showing the maximum CPS achieved between pairs of VMs.

over 1.5M connections per second. This value is below the maximum value per card – 3M CPS from Figure 10b– due to inefficiencies we believe in the code that steers fNIC traffic in the Azure smartNIC [58]. The figure also shows the CPS achieved using c5n series instances at EC2 using the same tool, the same configuration and the same guest OS. The lower CPS could be because EC2 employs different stateful NFs at each VM, uses a different NF processing system [6], applies explicit rate limits or some combination of all of the above. Comparing also with Figure 3a, we show that using Sirius a VM can achieve roughly 5× to 10× higher CPS. Further, recall from §3.3 that Sirius can split the load of a VM between multiple cards and so even higher CPS may be achievable.

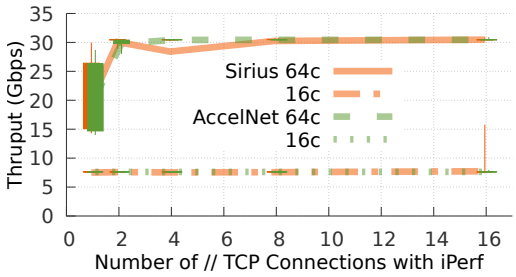
6.5 Measuring the Sirius datapath in Azure

To compare the datapath offered by the Sirius fNICs with the default datapath in Azure, we randomly and repeatedly deploy VMs and measure the latency and thruput on the two datapaths. Each VM in this experiment is equipped with three virtual NICs, one of which is used as the management interface and the other two are configured to use Sirius or AccelNet (the default in Azure) [58] respectively. The results shown are over millions of packets and tens of unique VM pairs.

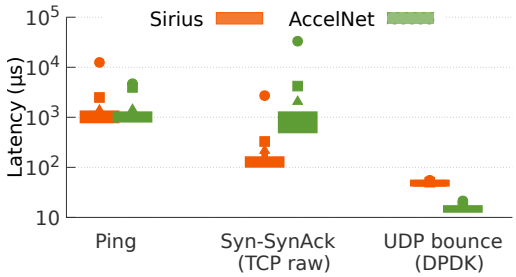
Figure 12a shows that the thruput achieved is nearly identical; with a small number of TCP flows, iPerf [26] can reach the NIC capacity on both of the datapaths.[11]

Figure 12b shows the latency for three kinds of applications. On the left are applications such as ping, tcping and hping3 which use the traditional in-kernel network stack. Such apps do not see any change in their RTT when using Sirius. Notice that with Sirius the datapath between a VM pair traverses up to two programmable NICs corresponding to the VMs' floating NICs. However, any increase in the physical length of the network path appears to be masked by the latency added by the guest kernel network stacks. In the middle of Figure 12b is the latency for the custom tool that we used in §2.3 which

[11] As noted in Table 5, the NIC capacity limits for the 16 and 64 core VMs that we used here are 8Gbps and 30Gbps respectively.



(a) Thruput achieved by iPerf with different numbers of TCP flows. The candlesticks show quartiles (bars) and the min and max values (whiskers) over many pairs of VMs and minutes.



(b) RTT using ping, TCP connections on raw sockets and a DPDK app that bounces UDP packets. In the candlesticks, the bars correspond to quartiles and the whiskers are the 10th and 90th percentiles. The triangle, square and circle above each bar show the 99th, 99.9th, and 99.99th percentile values respectively.

Figure 12: Comparing the datapath of AccelNet [58] with the disaggregated path through Sirius in Azure.

establishes TCP connections on raw sockets. As the figure shows, for such apps Sirius offers a better RTT than AccelNet because although Sirius may have a longer physical path, AccelNet takes much longer to process the stateful NFs for each new TCP connection. A third set of applications, on the right in Figure 12b, achieve very small latency by bypassing both the kernel network stacks (using an optimized DPDK app that we built) as well as the cloud's stateful NFs (by using UDP packets). As the figure shows, the typical latency for such apps is 15μs and 50μs respectively on the AccelNet [58] and Sirius datapaths. Note also the values on the tail. We conclude that any additional latency due to Sirius will only be visible to a small subset of applications and that for the vast majority of TCP-like traffic Sirius represents a clear improvement.

6.6 Offloading fNICs of middlebox NVAs

For the experiment setup shown in Figure 9c, Figure 13 shows the CPS achieved by traffic through different middlebox VMs. We generate results for Sirius using 32 core VMs as clients and servers of the traffic and offload the floating NIC of the middlebox VM onto Sirius. For all of the public clouds, we pick the best possible CPS numbers from datasheets released by the middlebox vendors [13, 17–19, 34]. The figure shows that using Sirius substantially improves the achievable throughput because the stateful network functions that cloud providers apply by default on the middlebox VM are often

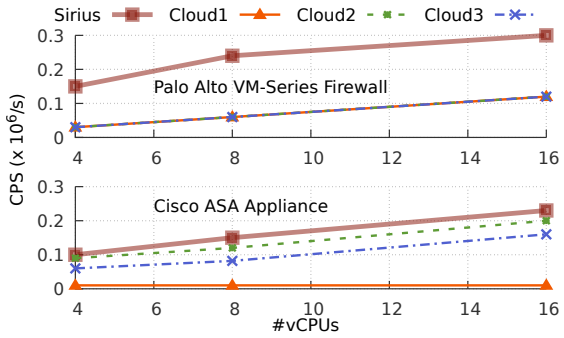


Figure 13: The CPS (# new connections per second) at which traffic can be sent through three different middlebox network virtual appliances on different public clouds and when onboarded onto Sirius.

the limiting factor to middlebox performance.

7 Related Work

The key focus of Sirius is to disaggregate stateful network functions onto pools of programmable NICs which tightly integrate P4-programmable MPUs and general purpose ARM cores with a large coherent memory system. With Sirius, we show how to replicate connection state inline so that individual card failure does not adversely impact ongoing connections (§3.2), discuss multiple design-points which split or migrate the load of a VM across different NF processors (§3.3) and offer an implementation that achieves better performance-over-cost than state-of-the-art (§4).

We are unaware of any prior characterization of the NF load at a large public cloud (§2.2). However, the case for disaggregation based on NF load being skewed across VMs and decorrelated (that is, having smaller variance when considered in aggregates such as at rack-level) is similar to the cases made to disaggregate other resources [59,66,74,78,89,91,96].

The state-of-the-art in processing stateful network functions is either in vswitch software or on programmable FPGAs that are directly connected to the host [6, 57, 58]. Andromeda [52] processes NFs at dedicated software middleboxes but explicitly states that they do not support stateful functions listing concerns such as 'state loss during upgrade or failure', 'transferring state when offloading' and 'ensuring that flows are 'sticky' to the hoverboard that has the correct state' [52]. We address some of these challenges in §3 and to the best of our knowledge are the first to disaggregate the rich class of stateful NFs listed in Table 1 and §2.1.

Offloading stateful network functions is non-trivial since a large amount of memory to maintain state must be accessible at high speeds. SRAMs support switch linerates but are expensive and so we use a bag of NICs architecture with memory coherence. Some prior works offload specific stateful NFs into programmable hardware [41, 83, 85]; however, they use switches and can only offload only a small subset of all flows, e.g., top-k by rate [41, 83, 85]. To compensate, TEA [73] pairs Tofinos with memory on remote servers and uses RPCs to access the remote state. When state is remote, it is challenging to achieve high performance and reliability. Also, Tofinos lack integrated general-purpose cores which forces TEA to build, in P4, a new RPC and a new reliable transport. With Sirius, each card has much larger memory. We replicate state between NICs on nearby servers in one pool and our general-purpose ARM cores simplify the logic. We believe that pairing cards which have tightly-integrated MPUs and ARM cores facilitates richer forms of disaggregation.

Another alternative is to use custom FPGAs with large memory (e.g., Xilinx and Altera). We are unaware of any works that match the performance and power draw of our cards using FPGAs. We believe that (1) P4 programmable MPUs are fundamentally more efficient than FPGAs [43, 76, 77], and (2) carefully dividing work between MPUs and ARM cores is key for high performance.

We discuss other related work in §C.

8 Conclusion

Stateful network functions are a key cog in today's public cloud architectures. We disaggregate their processing into a shared pool. Doing so avoids paying for smart-NICs at each server that are provisioned to support peak load, reduces constraints in VM placement and increases performance on the tail. Moreover, we attach this shared pool to the datacenter network at the layer off which most packets bounce off in the CLOS and so the latency and bandwidth overhead from packets taking a detour to the shared pool is small. We show a novel and simple solution that replicates connection state between pairs of cards without buffering packets while replicating state. We use NICs that have large memory, P4-programmable match-action pipelines and integrated general-purpose ARM cores. Our results from deployment at Azure show that network usage at VMs can reach NIC capacity even when complex stateful NFs are executed on each new connection and every packet.

Acknowledgements: We heartily appreciate the efforts of several team members whose work was crucial for the Sirius project including Aditya Baskar, Vivek Bhanu, Prachi Pravin Bhavsar, Weixi Chen, Nikita Dabir, Manasi Deval, Sumit Dhoble, Steve Espinosa, Osman Ertugay, Daniel Firestone, Shashank Gupta, Arun Jeedigunta, Sarat Kamisetti, Sam Kim, Guohan Lu, Ilias Marinos, Omar Mbarki, Kaixiang Miao, Kevin Pacella, Tommaso Pimpo, Vikas Prabhakar, Pirabhu Raman, Rohit Kumar Sharma, Yusef Skinner, Gabriel Silva, Prince Sunny, Hayden Udelson, Lihua Yuan, Zhenhua Yao, Xinyan Zan, Yuanyuan Zhou and Qi Zhang. We also thank the anonymous reviewers and our shepherd Aurojit Panda for feedback on the paper.

References

[1] Accelerated Connections and NVAs (Preview). https://bit.ly/424BkuF.

[2] Amazon EC2 instance IP addressing. https://go.aws/3qRcooQ.

[3] Amazon EC2 instance Network Bandwidth. https://go.aws/3mKS1ef.

[4] AMD DSC-200 Datasheet. https://bit.ly/3BrC0in.

[5] AWS: Data Transfer Costs for Common Architectures. https://go.aws/3cg5J3O.

[6] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[7] AWS PrivateLink. https://go.aws/3KG9xIs.

[8] AWS: VPC Peering. https://go.aws/3QcZxHI.

[9] Azure: Bandwidth Pricing. https://bit.ly/3Cou81Z.

[10] Azure: Private IP Addresses. https://bit.ly/3BqoSJ7.

[11] Azure PrivateLink. https://bit.ly/3CRXjKV.

[12] Azure: Virtual Network Peering. https://bit.ly/2AT5czqG.

[13] Cisco Adaptive Security Virtual Appliance (ASAv) Data Sheet. https://bit.ly/3UldTJq.

[14] Connection tracking: State and examples. https://bit.ly/3wrNdfP.

[15] Conntrack Tales – One thousand and one flows. https://bit.ly/3dL3eHf.

[16] Floodlight Controller. http://goo.gl/kzmC7.

[17] FortiGate-VM on Amazon Web Services. https://bit.ly/3Bp5qwb.

[18] FortiGate-VM on Google Cloud. https://bit.ly/3Sfz6CD.

[19] FortiGate-VM on Microsoft Azure. https://bit.ly/3BoXN93.

[20] Google Cloud: Bandwidth Pricing. https://bit.ly/3Cw83i9.

[21] Google cloud engine: Network bandwidth. https://bit.ly/3ywTVld.

[22] Google Cloud Platform: VPC Network Overview. https://bit.ly/3LpH4XP.

[23] Google Cloud: VPC Network Peering. https://bit.ly/3RsxiWG.

[24] Intel Tofino. https://intel.ly/3wxWT8w.

[25] Intel Tofino 2. https://intel.ly/3QTeD6F.

[26] iPerf. http://dast.nlanr.net/Projects/Iperf.

[27] IXIA BreakingPoint. https://bit.ly/3DqfOqd.

[28] Linux and Windows networking performance enhancements | Accelerated Networking. https://bit.ly/3kh7x05.

[29] New Private Service Connect simplifies secure access to services. https://bit.ly/3RyLMnN.

[30] NVGRE: Network Virtualization Using Generic Routing Encapsulation. https://www.rfc-editor.org/rfc/rfc7637.

[31] NVIDIA Spectrum SN3000 Open Ethernet Switches. https://bit.ly/3JLG9C1.

[32] OpenVSwitch: Conntrack Tutorial. https://bit.ly/3LsYtPd.

[33] Palo Alto Networks VM-Series Firewall. https://docs.paloaltonetworks.com/vm-series.

[34] Palo Alto Networks VM-Series Firewall Performance Datasheet. https://bit.ly/3f7vrJa.

[35] Palo alto networks vm-series performance & capacity. https://bit.ly/3BE8W7t.

[36] RedHat: IPTables and Connection Tracking. https://red.ht/3DAgucH.

[37] TREX: Realistic Traffic Generator. https://trex-tgn.cisco.com/.

[38] IEEE Standard for Ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, 2018.

[39] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. Mergeable Summaries. *TODS*, 2013.

[40] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *NSDI*, 2012.

[41] Manikandan Arumugam et al. Bluebird: High-performance SDN for Bare-metal Cloud Services. In *NSDI*, 2022.

[42] 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. In *SIGCOMM*, 2013.

[43] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference. *ACM Trans. Reconfigurable Technol. Syst.*, 2018.

[44] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *Dependable Computing for Critical Applications 3*. Springer, 1993.

[45] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[46] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 2002.

[47] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslass, Ariel Orda, and Tom Edsall. dRMT: Disaggregated Programmable Switching. In *SIGCOMM*, 2017.

[48] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *SOSP*, 2009.

[49] G Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[50] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *SOSP*, 2015.

[51] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[52] Michael Dalton et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*, 2018.

[53] Tobias Distler. Byzantine fault-tolerant state-machine replication from a systems perspective. *ACM Comput. Surv.*, 2021.

[54] Marianne Durand and Philippe Flajolet. Loglog Counting of Large Cardinalities. In *ESA*, 2003.

[55] K. Egevang and P. Francis. The IP Network Address Translator (NAT). In *RFC 1631*, 1994.

[56] 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 *NSDI*, 2016.

[57] Daniel Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *NSDI*, 2017.

[58] Daniel Firestone et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[59] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[60] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[61] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[62] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *SIGCOMM*, 2020.

[63] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, 2017.

[64] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In *International conference on reliable software technologies*. Springer, 1996.

[65] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *OSDI*, 2020.

[66] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, 2013.

[67] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, and Min Zhu. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[68] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical network performance isolation at the edge. In *NSDI*, 2013.

[69] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*, 2017.

[70] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *NSDI*, 2019.

[71] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *NSDI*, 2019.

[72] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: Enabling Fault-Tolerant Stateful In-Switch Applications. In *SIGCOMM*, 2021.

[73] Daehyeok Kim, Yibo Zhu, Zaoxing Liu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *SIGCOMM*, 2020.

[74] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *EuroSys*, 2016.

[75] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[76] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, 2006.

[77] Ian Kuon and Jonathan Rose. *Quantifying and exploring the gap between FPGAs and ASICs*. Springer Science & Business Media, 2010.

[78] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *ASPLOS*, 2020.

[79] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. Panic: A high-performance programmable nic for multi-tenant networks. In *OSDI*, 2020.

[80] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, 2014.

[81] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *OSDI*, 2016.

[82] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

[83] Rui Miao et al. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *SIGCOMM*, 2017.

[84] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[85] Tian Pan et al. Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches. In *SIGCOMM*, 2021.

[86] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. In *ESA*, 2011.

[87] 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. In *SIGCOMM*, 2013.

[88] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vSwitch. In *NSDI*, 2015.

[89] Pramod Subba Rao and George Porter. Is memory disaggregation feasible? A case study with Spark SQL. In *ANCS*, 2016.

[90] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*, 2017.

[91] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[92] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for Multi-Tenant cloud storage. In *OSDI*, 2012.

[93] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[94] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *NSDI*, 2022.

[95] 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 *NSDI*, 2022.

[96] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB Endow.*, 2021.

A Discussion

A.1 Complications in failover

It is possible for the ToR switches (shown in green in Figure 4), which probe the cards for liveness, to reach different liveness estimates for a card pair. That is, one of the switches can conclude a card is down while the other switch concludes otherwise. Similarly, even though we use multiple heartbeats and there are multiple network paths between a card pair, it is possible that so many consecutive heartbeats are lost allowing one of the cards in a card pair to conclude that the peer is down even though the peer is alive. A *split-brain* happens when different parts of the network assume that different cards (in a card pair) are responsible for an fNIC. Recall that the primary card announces a BGP route with a smaller AS path which helps resolve some of these complications. In addition, we notify all card role changes to a logically centralized Sirius controller which helps to ensure that *split-brain* cases, were they to happen, do not persist for very long.

A.2 fNIC abstraction guarantees

We statically partition each card's *capacity*, on the dimensions listed in §3, among the fNICs that are mapped to a card. The capacity values that we use for apportioning (e.g., the last row of Table 7) are slightly smaller than the maximum values that we see in experiments using iXIA traffic generators and a rich variety of network functions in subsection 6.2 and so we do not anticipate performance interference issues at the card.

A.3 Encryption, Traffic QoS

Some aspects such as end-to-end encryption and traffic prioritization can require on-host support. With Sirius, we are exploring how to divide such functions between the disaggregated pool and the smart NIC that is directly attached to the host. For example, the disaggregated pool can perform the more stateful processing such as exchanging keys or determining which queue to assign a flow to so as to meet its priority class or bandwidth limit while the on-host FPGA performs tasks that require less state such as marking or rate limiting queues [90].

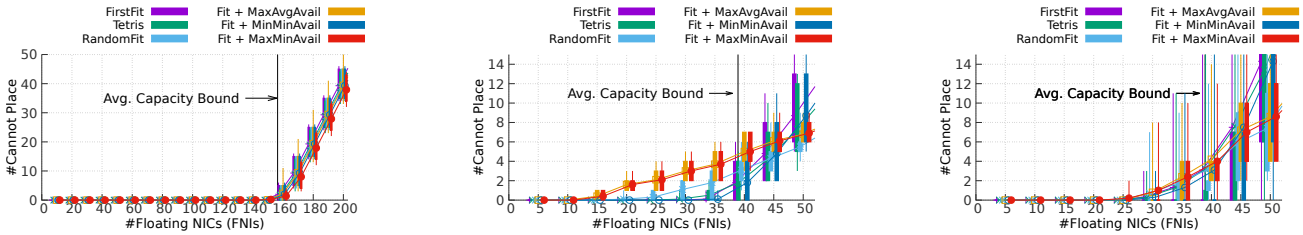
B Additional Results

B.1 Packing floating NICs into Sirius cards

Table 7 shows the sizes of the floating NICs that Sirius offers and the card capacity. A Sirius appliance has 12 cards. We evaluate several vector bin packing heuristics [61, 86] to pack fNICs onto Sirius cards. Our results in Figure 14 show that:

- The optimal choice of a packing heuristic, in terms of packing efficiency, depends on the size distribution of the fNICs and whether or not we split load of an fNIC across multiple cards.
- In some cases, such as when small fNICs dominate the workload, any heuristic can achieve the average capacity bound[12] which assumes that there are no card boundaries and that all resources are in one large pool.
- Splitting the load of an fNIC across cards substantially improves efficiency but also increases state that must be maintained on cards since rulesets belonging to split fNICs must now be deployed on multiple cards. The per flow and per endpoint state substantially dominates the ruleset size however. Nevertheless, we aim to split only fNICs that have large resource needs.

[12] $\text{bound} = \min_{\text{resource r}} \frac{\text{total capacity on r}}{\text{avg. FNI load on r}}$



(a) Packing efficiency when the fNICs listed in Table 7 arrive as per the production distribution from §2.2.

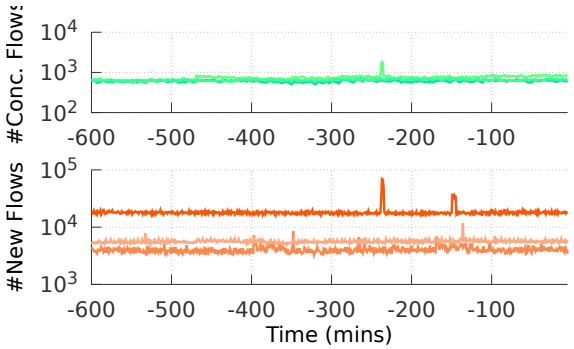
(b) Comparing the packing efficiency when the fNICs listed in Table 7 arrive with an equal probability.

(c) The case of Figure 14b except with the load from each fNIC split evenly across two cards.

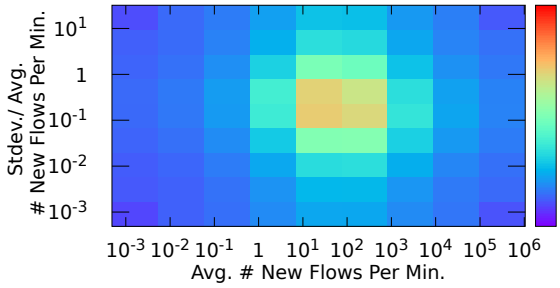
Figure 14: Comparing different vector bin packing strategies when mapping floating NICs (granularity of resource allocation for network functions supported by Sirius as listed in Table 7) to a Sirius appliance with 12 cards.

Term	Resource Sizing		
	#New Flows (Millions Per Sec.)	# Concurrent Flows (M)	Throughput (Gbps)
FNI_XXS	0.05	1	5.0
FNI_XS	0.10	1	10.0
FNI_S	0.25	2	12.5
FNI_M	1.00	2	12.5
FNI_L	2.00	4	12.5
FNI_XL	3.00	16	50.0
Sirius card	3.00	16	200.0

Table 7: The resource sizes, along multiple dimensions, that Sirius associates with different kinds of floating NICs. Cloud customers can choose which floating NIC to associate with their VM.



(a) Temporal changes in NF load; showing the total load at three randomly chosen nodes; note: y axes is in log scale.



(b) Clustering containers based on their average NF load and the coefficient of variability (=stdev/ avg) of their NF load.

Figure 15: Additional characterization results from the dataset in §2.2.

B.2 Variation in the load for NFs at Azure

We analyze load variability across all containers in Azure using the same dataset described in §2.2.

Figure 15a shows the temporal variation in the cumulative NF load. The figure shows many short-lived spikes, some of which are larger than 4×; note y axes is in log-scale. We also see some innate variability in the steady-state load across these nodes.

Figure 15b is a 2D matrix where each entry represents the number of containers that have the corresponding (x, y) value. The x axes is the average numbers of new flows in each minute. The y axes is the coefficient of variation in the same metric (=stdev./ avg.). Both axes are in log-scale. As well, the number of containers which is shown as a heat plot on the right is also in log scale. The figure shows that most of the containers have between 10 to 1000 new flows per minute and the coefficient of variation is typically between 0.1 and 1. While the variability is high for many containers, containers with more average load appear to only have slightly higher variability and there are no unexpected patterns.

C Additional Related Work

State replication for fault tolerance has received much attention; see [53, 64] for a review. Our method in §3.2 is different from the primary-backup style replication [44, 51] wherein the primary processes requests, forwards state changes to the backup and emits responses after receiving an acknowledgement from the backup. Our method is also different from Paxos-like protocols [46, 48, 50, 81] where replicas first agree on an order in which to process requests and then process the requests. The key difference is that both alternatives hold requests while replication is underway. In the case of stateful NFs, requests are packets that change state and so holding requests at speeds of hundreds of Gbps will require a very large packet buffer. To our knowledge, we are unaware of any prior work that ping-pong's packets between replicas which effectively holds the requests on the network wire. Redplane [72] replicates the state between programmable switches and a server-based remote state store but must cope in P4 with route changes that happen between the switches and the server-based store. Our method is simpler and more performant.

For the case of moving state on-the-fly between multiple NF processors, OpenNF [60] is perhaps the first to describe in detail the multiple issues that arise. Their solution however buffers all the packets that arrive while the state is being moved at an SDN controller (e.g., Floodlight [16]) which becomes a scaling bottleneck. OpenNF [60] reports results for O(100) flows (e.g., "a loss-free move involving state for 500 flows takes only 215ms") whereas each fNIC in Sirius can have many millions of ongoing flows. Similar to RedPlane [72], StatelessNF [69] and [71] store relevant NF state in an external state store (e.g., in a RAMCloud [69]). As noted above, relative to Sirius (which stores state in a nearby secondary card), we believe that storing state in an external state store has higher intrinsic overheads.