Disaggregating and Consolidating Network Functionalities with SuperNIC

Yizhou Shan[†], Will Lin[†], Ryan Kosta[†], Arvind Krishnamurthy[*], Yiying Zhang[†]
[†]*University of California San Diego,* [*]*University of Washington*

Abstract

Resource disaggregation has gained huge popularity in recent years. Existing works demonstrate how to disaggregate compute, memory, and storage resources. We, for the first time, demonstrate how to disaggregate *network resources* by proposing a new distributed hardware framework called *SuperNIC*. Each SuperNIC connects a small set of endpoints and consolidates network functionalities for these endpoints. We prototyped SuperNIC with FPGA and demonstrate its performance and cost benefits with real network functions and customized disaggregated applications.

1 Introduction

Hardware resource disaggregation is a solution that breaks full-blown, general-purpose servers into segregated, network-attached hardware resource pools, each of which can be built, managed, and scaled independently. With disaggregation, different resources can be allocated from any device in their corresponding pools, exposing vast amounts of resources to applications and at the same time improving resource utilization. Disaggregation also allows data-center providers to deploy, manage, and scale different types of resources independently. Because of these benefits, disaggregation has gained significant traction from both academia [8, 11, 12, 16, 31, 52, 59, 61, 67, 71] and industry [18, 35, 38, 50, 69]. Existing disaggregated systems have focused on separating three types of resources: compute [32, 61], memory (or persistent memory) [7, 8, 31, 35, 48, 61, 62, 67, 71], and storage [15, 17, 45, 69].

While increasing amounts of effort go into disaggregating compute, memory, and storage, the fourth major resource, *network*, has been completely left out. No work has attempted to disaggregate the network. At first glance, "network" cannot be disaggregated from either a traditional monolithic server or a disaggregated device, as they both need to be attached to the network. However, our insight is that even though each endpoint still needs its own network interface, its *network-related tasks* such as a transport layer or network virtualization do not have to run at the endpoint.

With this insight, we propose to *disaggregate* network tasks (or *NTs*[1]) from individual endpoints and *consolidate* them in a network resource pool. This pool consists of a distributed set of *SuperNIC*s (or *sNIC*s), new programmable devices we built to execute NTs for the endpoints. Each sNIC connects to a small set (*e.g.*, 4 to 8) of endpoints and executes NTs for these endpoints. All sNICs connect to the ToR switch and are connected among themselves, *e.g.*, as a ring (Figure 1). By design, an sNIC device consists of an ASIC for fixed systems logic, FPGA for running and reconfiguring NTs, and software cores for executing the control plane. In building sNIC, we answer seven key research questions.

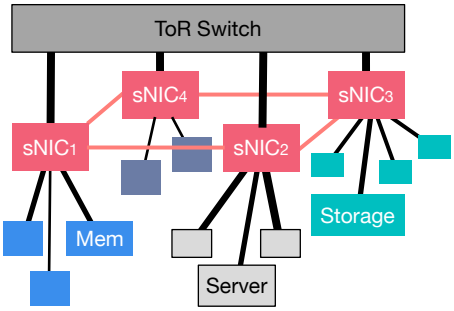

Figure 1: **Overall Architecture of SuperNIC.** *An example rack with four sNICs that connect a set of compute devices, memory devices, storage devices, and regular servers. Connection line thickness represent network link capacity.*

What is the benefit of disaggregating and consolidating network tasks? To answer it, we study network communication patterns and needs in the traditional server-based data-center environment and the disaggregated data-center environment by analyzing real-world applications and network traces in these environments. We discover three potential benefits of network disaggregation and consolidation for the disaggregated environment: 1) it enables an existing rack and ToR switch to host a large number of disaggregated devices; 2) it avoids the need to implement hardware and/or software units for NTs at each type of device; and 3) consolidation allows an sNIC to only provision hardware resources for the peak *aggregated* bandwidth instead of each endpoint provisioning for its own peak. Server-based data centers share the last benefit and, to some extent, the second benefit.

How to disaggregate network tasks? To answer this question, we explore what NTs have to remain at endpoints and what could potentially be disaggregated. To ensure reliable communication, an endpoint needs at least a physical layer and a reliable link layer that could discover and recover from packet loss and corruption. All other NTs could potentially be disaggregated, and those that have intermittent usages or use hardware resources that the endpoint lack could especially benefit from disaggregation.

[1]We call all types of network tasks NT to differentiate from traditional network function (NF). NT is a super set of NFs and also includes tasks like a transport and application-specific functionalities such as caching.

How to consolidate network tasks? We support three types of sharing: 1) sharing an sNIC's hardware resources across different NTs (*space sharing*), 2) sharing the same NT across different applications (*time sharing*), and 3) configuring the same hardware resources for different NTs at different times (*time sharing with context switching*). For space sharing, we partition the FPGA space into *region*s, with each hosting one or more NTs. Each region could be individually *reconfigured* (via FPGA partial reconfiguration, or *PR*) for starting new NTs or to context switch NTs. Different from traditional software systems, hardware context switching with PR is orders of magnitude slower, which could potentially impact application performance significantly. To solve this unique challenge, we propose a set of policies and mechanisms to reduce the need to perform PR, *e.g.*, by keeping de-scheduled NTs around like a traditional victim cache and by not over-reacting to load spikes.

How to ensure fair and safe sharing? Existing network systems consider the fair sharing of a single type of resource (*e.g.*, the total bandwidth of a NIC [49]). With sNIC, we treat each internal hardware resource separately when considering fair sharing, *e.g.*, ingress/egress bandwidth, internal bandwidth of each shared NT, payload buffer space, and on-board memory, as doing so allows a higher degree of consolidation. Our context is unique in that the packet processing system itself requires multi-dimensional resource sharing. We adopt Dominant Resource Fairness (DRF) [30] to address this. Instead of user-supplied, static per-resource demands as in traditional DRF systems, we monitor the actual load demands at run time and use them as the target in the DRF algorithm. Furthermore, we propose to use ingress bandwidth throttling to control the allocation of other types of resources. Finally, for safe sharing, we build a simple virtual memory system to isolate and protect accesses to on-board memory.

How to achieve high-throughput, low-latency, and scalable performance? We achieve high throughput using two levels of parallelism: *NT parallelism* where a packet goes through multiple NTs in parallel and *instance parallelism* where we launch multiple instances of the same NT to handle different packets in an application. sNIC automatically scales an NT out/down based on its load. To achieve low scheduling latency and scalability, we propose a scheduler that centers around a new notion, *NT chaining*. The idea is to group NTs that are likely to be executed in a sequence into a chain. Our scheduler reserves credits for the entire chain as much as possible so that packets execute the chain as a whole without involving the scheduler in between. Doing so improves both packet-processing latency and scheduler scalability.

How to build and utilize a distributed sNIC framework? When one sNIC is overloaded and needs to accommodate too much network bandwidth, on-chip hardware resources, or off-chip memory space, traditional solutions would drop packets/NTs or provision more hardware resources. The former impacts application performance, and the latter results in resource wastage when loads are not at their peaks. Our idea is to utilize other sNICs to handle load spikes, based on the observation that not all sNICs under a rack would run at their peak load at the same time. Essentially, with distributed sNICs, we provision for the maximum aggregated bandwidth in a rack instead of the sum of peak loads under each sNIC. Specifically, to support overloaded network bandwidth or on-chip hardware resources at an sNIC, we create an NT at another sNIC that has a lighter load. The current sNIC then only serves as a simple pass-through device to redirect packets to the new sNIC. To support overloaded memory space at an sNIC, our virtual memory system transparently swaps memory to/from an sNIC with less memory pressure.

How to best exploit sNICs to build disaggregated applications? Apart from cost savings, network disaggregation and consolidation have unique benefits to both disaggregated and regular distributed applications, as an sNIC is a central point for its connected endpoints and an sNIC can be attached at both the sender and the receiver side. We made an initial exploration along these directions with two case studies. First, we build a key-value store on top of real disaggregated memory devices [34] connected to an sNIC. We explore using sNICs for traditional NTs like the transport layer and customized NTs like key-value data replication and caching. Second, we build a Virtual Private Cloud application on top of regular servers by connecting sNICs at both the sender and the receiver side. We disaggregate NTs like encapsulation/decapsulation, firewall, and encryption to the sNICs.

We prototype sNIC with FPGA using the 100 Gbps, multi-port HiTech Global HTG-9200 board [2]. We demonstrate sNIC's benefits and trade-offs by disaggregating and consolidating three types of NTs: reliable transport, traditional network functions, and application-specific tasks. We evaluated these scenarios with micro- and macro-benchmarks and compared sNIC with no network disaggregation and disaggregation using a device that is adapted from a recent single-server programmable NIC [49]. Our results running a Facebook key-value trace [13] show that sNIC's consolidation of four endhosts and two NTs saves 64% costs compared to no consolidation, with only 1.3% performance overhead. Furthermore, the customized key-value store caching and replication functionalities on sNIC improves throughput by 1.31× to 3.88× and latency by 1.21× to 1.37× and compared to today's remote memory systems with no sNIC.

2 A Case for Network Disaggregation and Consolidation

2.1 for Disaggregated Datacenters

Resource disaggregation is a data-center architecture that organizes different hardware resources into separate, network-attached pools. While today's data centers use regular servers to form these pools [18, 66, 69], future data centers could benefit from using specialized *devices* to build such

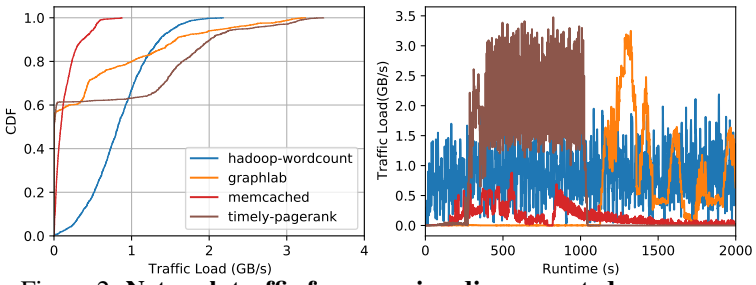

Figure 2: **Network traffic for accessing disaggregated memory.**

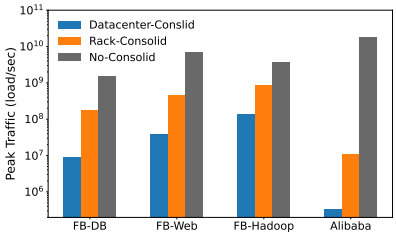

Figure 3: **Consolidation Analysis of Facebook and Alibaba Traces.** *Load represent relative amount and have different units for FB and Alibaba.*

pools, *e.g.*, a memory pool consisting of network-attached memory [34] or Optane boards [35, 67]. Three practical and key technical hurdles need to be solved before data centers can readily deploy such device-based disaggregated resource pools. Network disaggregation solves all of them.

First, when a monolithic server is replaced with multiple network-attached disaggregated devices (*e.g.*, one CPU processor, one memory device, one storage device to replace a server), the number of network endpoints could increase to hundreds per rack [64]. If all these devices directly connect to a ToR switch, the rack needs to use an expensive, high-port-count ToR switch or multiple low-port-count ToR switches (and the resulting increased scale of the entire switch hierarchy [74, 76]). With our proposed architecture, devices in a rack connect to a small set of sNICs that then can be accommodated by today's data-center ToR switch.

Second, building different types of disaggregated devices involves adding standard networking functionalities like a reliable transport layer to each of them, with many of them also desiring various customized network functions. Disaggregated devices will come in many forms, some with a software processor [61], some with only hardware units [67], and some with both software and hardware [34]. Designing and implementing network tasks in each type of disaggregated device will be a daunting job. Moreover, it likely will involve adding additional hardware units to the devices. By building NTs once for a single type of hardware (*i.e.*, sNIC), we could significantly save development and CapEx costs.

Finally, resource disaggregation introduces new types of network traffic that used to be within a server, *e.g.*, a CPU device accesses a remote memory device to read/write data. If not handled properly, such traffic could add a huge burden to the data-center network [14]. To understand this type of traffic, we analyzed a set of disaggregated-memory network traces collected by Gao et al. using five endhosts [27]. Figure 2 plots the CDF and the timeline of network traffic from four workloads. These workloads all exhibit fluctuating loads, with some having clear patterns of high and low load periods. We further compare the case where we provision network resources for the peak load of every workload and the case where we could consolidate and only provision for the aggregated network demand (*i.e.*, sum of peaks vs. peak of aggregated traffic). Consolidating just five endhosts already results in 1.1× to 2.4× savings with these traces.

2.2 for Server-Based Datacenters

To understand network behavior in real, server-based data centers, we analyze two sets of traces: a Facebook trace that consists of Web, Cache, and Hadoop workloads [58], and an Alibaba trace that hosts latency-critical and batch jobs together [33]. Both data centers exhibit high burstiness, and bursts happen at different times for different end hosts. Other prior work has also reported similar findings [27, 53]. We omit the CDF and timeline plots due to space constraints. We perform a similar consolidation analysis as the disaggregated memory traces, as shown in Figure 3. In addition to the sum of individual-endhost peaks and the peak of aggregated traffic across the entire data center (or, in Facebook's case, an entire workload), we sum the traffic under a rack and sum all the per-rack peaks. For both Facebook and Alibaba, rack-level consolidation consumes one to two orders of magnitude fewer resources than no consolidation.

In addition to the bursty traffic patterns that are conducive to the consolidation benefits of SuperNIC, the underutilization of network functions in today's network devices is another major motivation for SuperNIC. For example, existing NICs are bloated with features that are not utilized in the common case [19, 26]. Wang et al. [73] reported that state-of-the-art programmable switches have low resource utilization (common NFs consume less than 1%), and even a complex application consumes only a small fraction of resources (*e.g.*, NetChain [40] uses 3% MAT resources). Over time, vendors keep adding more features into their network device products to meet the diverse requirements from different customers, resulting in significant resource waste that could otherwise be avoided by SuperNIC's consolidation solution.

Finally, consolidating network tasks into a separate pool makes it easy for datacenter operators to manage them (*e.g.*, monitoring, reconfiguring, and upgrading).

2.3 Limitations of Alternative Solutions

The above analysis makes a case for disaggregating and consolidating network tasks from individual servers and devices. A question that follows is *where* to host these NTs and whether existing solutions could achieve the goals of network disaggregation and consolidation.

The first possibility is to host them at the **ToR switch**. This approach lets endpoints go through only one hop to the ToR switch, compared to two hops with approaches

that use a middle layer like sNIC. However, it requires ToR switches to be programmable and have more ports to connect to disaggregated devices, both of which add monetary costs and requires changes to the data-center network infrastructure [74, 76]. Moreover, existing programmable switches lack proper multi-tenancy and consolidation support [73].

Another possibility is **multi-host NICs** like NVidia BlueField [4], OCP NIC [1], and Intel Red Rock Canyon [37]. These NICs each connect to multiple servers via PCIe connections. Multi-host NICs have a similar connection topology as sNIC. However, there are several key differences that make multi-host NICs not a good choice for network disaggregation and consolidation. First, the way they support multiple hosts is by breaking out one link into several (say k) sub-links, each with a fixed portion ($1/k$) of the original link bandwidth. In contrast, sNIC uses the original link bandwidth (up to 100 Gbps in our prototype) to connect each endpoint and consolidates the *run-time* bandwidth consumption from different endpoints. Second, multi-host NICs are merely a link-layer aggregation solution; they have no support for multi-tenant resource allocation or isolation. Moreover, today's multi-host NICs either do not support NT offloading or support it in software. sNIC allows the acceleration of NTs in hardware. Finally, unlike sNIC, there is no distributed platform of multi-host NICs.

Middleboxes are a traditional way of running network functions inside the network. Traditional hardware middleboxes are specialized black-box devices that cannot be changed after deployment [60, 63, 70]. Network Function Virtualization uses regular servers to build flexible software-based middleboxes [43, 51, 54, 65, 75], but at the cost of running at lower performance [36, 55]. sNIC has the benefits of both: it is flexible as it supports offloading a wide range of NTs and can be reconfigured at run time, and it also achieves high-bandwidth line rate processing.

Increasing amount of data centers attach **SmartNICs** and **specialized ASIC/FPGA devices** like Intel IPU [39], Amazon Nitro [10], and Microsoft Catapult [6] to single servers. These devices can host offloaded network functions and other customized tasks. However, they do not support the consolidation of multiple endpoints.

Finally, there are emerging **interconnections designed for disaggregated devices** such as Gen-Z [29] and CXL [21]. These solutions mainly target the coherence problem where the same data is cached at different disaggregated devices. The transparent coherence these systems provide require new hardware units at every device, in addition to a centralized manager. sNIC supports the disaggregation and consolidation of all types of network tasks and does not require specialized units at endpoints.

3 SuperNIC Overview

As discussed in §2.3, although different existing network solutions provide some features of network disaggregation and consolidation, none of them meet all our target goals, thus necessitating the design of a new network solution. This section gives a high-level overview of the overall architecture of the sNIC platform and how to use it.

Overall Architecture. In a rack, the sNIC pool is a middle layer between endpoints (servers or devices) and the ToR switch. Each sNIC uses one port to connect to the ToR switch, and we connect all the sNICs in a rack together, currently with a ring topology to save ports on sNIC. Each sNIC connects k endpoints, and k can be different for different sNICs. The port bandwidth each endpoint has can be different as well. Our FPGA prototype now supports up to 100 Gbps. The sum of the link bandwidth at each endpoint that connects to an sNIC can and should exceed the link bandwidth between the sNIC and the ToR switch. This is because different endpoints' loads peak at different times (§2.2), and after sNIC's consolidation, the aggregated traffic would mostly fit the sNIC's uplink, as shown in Figure 3.

sNIC is a data-center-scale solution. An endpoint could either connect to an sNIC or directly to a ToR switch. A given sNIC can connect different types of endpoints. However, there is a potential benefit in connecting similar endpoints to an sNIC. Doing so offers more opportunity for resource consolidation, as similar endpoints (*e.g.*, memory devices) are likely to use the same set of NTs (*e.g.*, encryption).

When an sNIC fails, or its link to the ToR switch fails, if other links and the basic switching functionalities are still alive, the sNIC would turn into a passthrough device, forwarding NTs to other sNICs for processing. When the entire sNIC fails, the endpoints connected to it will be disconnected to the rest of the data center. This failure could be viewed as equivalent to traditional ToR switch failure but with a smaller failure domain (only the endpoints under the failed sNIC instead of the whole rack). Data centers that desire stronger reliability [28] could use a multi-homed solution by connecting each endpoint to two sNICs.

Connecting Endpoints to sNIC. To connect to an sNIC, an endpoint needs to have standard physical and link layers. For reliable transmission, the link layer needs to provide basic reliability functionality if the reliable transport is offloaded to sNIC. This is because packets could still be corrupted or dropped during the point-to-point transmission between an endpoint and its connected sNIC. Thus, the endpoint's link layer should be able to detect corrupted or dropped packets. It will either correct the corruption or treat it as a lost packet. For lost packets, the link layer needs to perform retransmission. Unlike traditional reliable link layer, our *point-to-point* reliable link layer is lightweight (only 37% more resources than an unreliable link layer with our implementation), as it only needs to maintain one logical flow and a small retransmission buffer (64 KB by default in our prototype) for the small Bandwidth-Delay Product (BDP) between the endpoint and the sNIC. This addition is the only change to today's endpoints that use an unreliable link layer, and it is

only needed when the reliable transport is offloaded to sNIC.

By design, sNIC can work with different types of physical links between endpoints and the sNIC. Our prototype uses regular Ethernet. Future extensions could use faster/tighter links like PCIe to further reduce latency overhead.

Using SuperNIC. To use the sNIC platform, users first write and deploy NTs. They specify which sNIC (sender side or receiver side) to deploy an NT. Users also specify whether an NT needs to access the packet payload and whether it needs to use on-board memory. For the latter, we provide a virtual memory interface that gives each NT its own virtual address space. Optionally, users can specify which applications share the same NT(s). Currently, our FPGA prototype only supports NTs written on FPGA (deployed as netlists). Future implementation could extend sNICs to support p4 programs running on RMT pipelines [72] and generic software programs running on a processor.

After all the NTs that a user wishes to use have been deployed, the user specifies one or multiple user-written or compiler-generated [47, 65] DAGs of the execution order of deployed NTs. Users could also add more DAGs at run time. Compared to existing works which let users specify an NF DAG when deploying NFs [24, 47, 54], we allow more flexible usages and sharing of deployed NTs. The sNIC stores user-specified DAGs in its memory and assigns a unique identifier (UID) to each DAG. At run time, each packet carries a UID, which sNIC uses to fetch the DAG.

4 SuperNIC Board Design

Traditional server SmartNICs have plenty of hardware resources when hosting network functions for applications running on the local server [19, 26]. In contrast, sNIC is anticipated to often be fully occupied or even over-committed, as it needs to host NTs from more tenants with limited hardware resources to save costs. Thus, a key and unique challenge in designing sNICs is space- and performance-efficient consolidation in a multi-tenant environment. Moreover, sNIC faces a more dynamic environment where not only the load of an application but also applications themselves could change from time to time. Thus, unlike traditional SmartNICs that focus on packet processing and packet scheduling, sNIC also needs to schedule NTs efficiently. This section first goes over the high-level architecture of sNIC, then discusses our mechanisms for efficient packet and NT scheduling, followed by the discussion of our scheduling and fairness policies, and ends with a description of sNIC's virtual memory system.

4.1 Board Architecture and Packet Flow

We design the sNIC board to simultaneously achieve several critical goals: **G1)** parsing/de-parsing and scheduling packets at line rate; **G2)** high-throughput, low-latency execution of NT DAGs; **G3)** safe and fair sharing of all on-board resources; **G4)** quick adaptation to traffic load and workload changes; **G5)** good scalability to handle many concurrent workloads and NTs; **G6)** flexible configuration and adjustment of control-plane policies; and **G7)** efficient usage of on-board hardware resources.

Figure 4 illustrates the high-level architecture of the sNIC board. By design, we envision the board to use small amounts of non-reconfigurable hardware for all sNIC functionalities, similar to the "shell" in Catapult [6] and F1 [9]. This includes fixed logic (*i.e.*, ASIC) and some software cores (SoftCores for short) (*e.g.*, a small ARM-based SoC). We dedicate all of the remaining board to be reconfigurable hardware (*e.g.*, FPGA) that run user NTs (blue parts in Figure 4). In our prototype, 90% space is left for NTs (**G7**). The NT area is further sub-divided into *region*s, each of which could be individually reconfigured. To achieve the performance that the data plane requires and the flexibility that the control plane needs, we cleanly separate these two planes. The data plane handles all packet processing on ASIC and FPGA (**G1**). The control plane is responsible for setting up policies and scheduling NTs and is handled by the SoftCores (**G6**). In our prototype, we built everything on FPGA.

When a packet arrives at an RX port, it goes through a standard physical and reliable link layer. Then our parser parses the packet's header and uses a Match-and-Action Table (MAT) to decide where to route the packet next. The parser also performs rate limiting for multi-tenancy fairness (§4.4). The parser creates a packet descriptor for each packet and attaches it to its header. The descriptor contains fields for storing metadata, such as an NT DAG UID and the address of the payload in the packet store. The SoftCores determine and install rules in the MAT, which include three cases for routing packets to the next step. First, if a packet specifies no NT information or is supposed to be handled by another sNIC (§5), the sNIC will only perform simple switching functionality and send it to the corresponding TX port (red line). Second, if a packet specifies the operation type `CTRL`, it will be routed to the SoftCores (orange line). These packets are for control tasks like adding or removing NTs, adding NT DAGs (§4.3), and control messages sent from other sNICs (§5).

Finally, all the remaining packets need to be processed on the sNIC, which is the common case. Their payloads are sent to the *packet store*, and their headers go to a central scheduler (black arrows). The scheduler determines when and which NT chain(s) will serve a packet and sends the packet to the corresponding region(s) for execution (blue arrows). If an NT needs the payload for processing, the payload is fetched from the packet store and sent to the NT. During the execution, an NT could access the on-board memory through a virtual memory interface, in addition to accessing on-chip memory. After an NT chain finishes, if there are more NTs to be executed, the packet is sent back to the scheduler to begin another round of scheduling and execution. When all NTs are done, the packet is sent to the corresponding TX port.

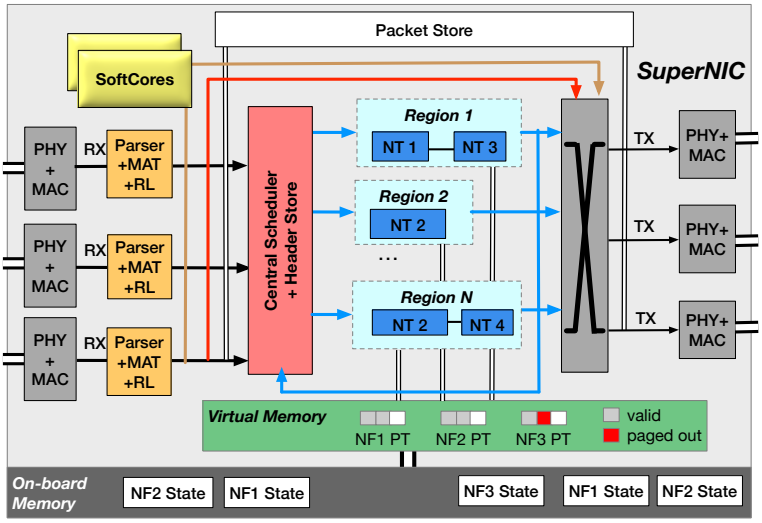

Figure 4: **sNIC On-Board Design.** *RL: Rate Limiter. PT: Page Table*

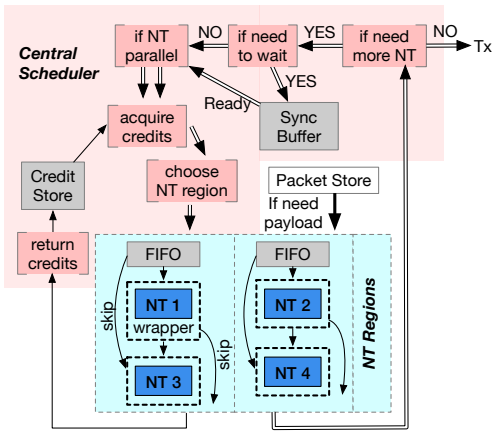

Figure 5: **sNIC Packet Scheduler and NT Region Design.** *Double arrows, single arrows, and thick arrows represent packet headers, credits, and packet payload.*

4.2 Packet Scheduling Mechanism

We now discuss the design of sNIC's packet scheduling mechanism (Figure 5). Today's network systems are seeing increasing amounts of network functions that can be executed as a DAG (*e.g.*, previous work found 53.8% NF pairs can run in parallel [65]). As sNIC enables more types of endpoints and workloads to offload their network tasks, the number of NTs and their execution flows will become even more complex. To efficiently execute such complex NT DAGs, we first propose the concept of *NT chain*, which allows one packet to be processed by a chain of NTs without the need to go through the scheduler multiple times. Second, we build a flexible run-time system that supports both *NT-level parallelism* (running different NTs in parallel) and *instance-level parallelism* (running multiple instances of the same NTs in parallel) in hardware. Existing hardware-based network function systems like Click [47], E2 [54], and PANIC [49] focus on scheduling a single NF or a simple sequence of NFs with instance-level parallelism. sNIC provides all three types of scheduling: NT chaining, NT-level parallelism, and instance-level parallelism, and in a scalable, efficient way. Doing so achieves high-throughput, low-latency NT DAG execution (**G2**), quick adaptation to traffic load changes (**G4**), and fast and scalable scheduling (**G5**).

NT chain architecture. We started our design by considering PANIC [49], a recent single-server SmartNIC solution that supports running sequences of NFs. It connects each NF to a crossbar, which is then connected to its central scheduler. PANIC's architecture falls short for an environment like sNIC that hosts many NTs, as PANIC would require a huge crossbar to connect to all of them.

To solve this problem, we observe that the same application usually has a fixed set of *NT chains* to execute. For example, a secure remote data storage application needs each of its packets to go through a transport layer, be decrypted, and be checked for integrity. We thus propose to put such a fixed NT chain in one NT region (*e.g.*, NT1→NT3 and NT2→NT4 in Figure 4). A packet that uses a chain goes through all the NTs in the chain without the need to involve the scheduler in between. Since each region has only one point of connection to the scheduler, our architecture largely reduces the port count of a crossbar (**G5**).

On top of the fixed chain design, we propose an optimization to enable efficient NT time sharing across multiple users and to accommodate cases where some packets of an application only access a part of a chain (**G4**, **G6**). Our idea is to support the *skipping* of arbitrary NT(s) in a chain. For example, a user can access NT1 and NT4 by first skipping NT3 in Region-1 and then skipping NT2 in Region-2 in Figure 5.

Scheduling packets over NT chains. Different from scheduling a packet for a single NT, scheduling it for an NT chain requires all the NTs in the chain to be able to accommodate the packet. PANIC [49] uses an optimistic way to schedule packets. Its scheduler pushes a packet to the first NF in an NF sequence if/when that NF has an available *credit*. After the first NF finishes its execution, this NF pushes the packet to the next NF regardless of the state of the next NF. If the next NF does not have credits, the packet will go back to the central scheduler. The issue with PANIC's optimistic mechanism is that a packet can often be pushed to an unavailable NF and then back to the scheduler, incurring latency overheads and affecting the scheduler's scalability.

We propose a scheduling method to solve PANIC's drawbacks based on our NT chain architecture. Our idea is to *reserve* credits for the *entire* chain as much as possible and then execute the chain as a whole; only when that is not possible, we fall back to a mechanism that may involve the scheduler in the middle of a chain. Doing so reduces the need for a packet to go through the scheduler after every NT, thereby improving both the packet's processing latency and the cen-

tral scheduler's scalability (**G5**). Specifically, we maintain credits for each NT in a credit store. When the scheduler needs to schedule a packet that uses an NT chain, it allocates one credit from each NT in the chain. If the scheduler can successfully allocate credits from all NTs, it sends the packet to the chain. This guarantees that once this packet enters the chain, it can always finish all the NTs in the chain without leaving it. When not all NTs have available credits, for better performance, we still send the packet to the NT chain instead of waiting for all NTs to be available. The scheduler allocates credits from the front of the chain until seeing an NT with no credits. It then sends the packet to the chain for execution. When the packet reaches an NT with no credits, it returns to the scheduler, which buffers the packet until credits are available at that NT. The scheduler returns one credit to an NT after the NT finishes processing a packet.

Scheduling packets with NT-level and instance-level parallelism. sNIC supports the concurrent execution of different packets at multiple instances of the same NT (instance-level parallelism) and the concurrent execution of the same packet at multiple different NTs (NT-level parallelism). Instance-level parallelism is an automatic scaling decision sNIC makes based on load (see 4.4 for autoscaling policy).

NT-level parallelism can be implied from NT DAGs (*e.g.*, in Figure 6, NT1 and NT2 can run in parallel with NT3 for user1). To execute a packet at several NTs concurrently, the scheduler makes copies of the packet header and sends them to these NTs concurrently. To obey the order of NTs that users specify, we maintain a *synchronization buffer* to store packet headers after they return from an NT's execution and before they could go to the next stage of NTs (Figure 5). For example, in Figure 6, NT1→NT2 and NT3 must be executed before NT4. Thus, a forked packet that enters the second region should only execute NT3 and then be buffered at the synchronization buffer. Only when the other forked packet finishes executing NT1→NT2 can we send the packet to the second region for NT4's execution.

4.3 NT (De-)Launching Mechanism

sNIC's SoftCore handles NT deployment, launch, and scheduling tasks, as a part of the control path. A new challenge specific to sNIC comes from the need to do more frequent NT reconfigurations than traditional programmable network devices. To enable more consolidation, we allow multiple NTs to *time share* an FPGA space, and we autoscale NTs. Both these cases involves the slow FPGA PR process. We propose a set of new mechanisms and policies to reduce or hide the PR costs. We now discuss the mechanisms, delaying policies to §4.4.

NT deployment. Users deploy NTs to the sNIC platform ahead of time as FPGA netlists (which can be thought of as Intermediate Representation in software). When receiving newly deployed NT netlists for an application, we first generate a set of FPGA bitstreams (which can be thought of as executable binaries in software). Specifically, we enumerate all possible combinations of NTs under user-specified NT DAG ordering requirements. This is because bitstream generation is a slow process that usually takes a few hours or even longer. Generating more bitstreams at deployment time gives the sNIC more flexibility to choose different NT combinations at the run time. Figure 6 shows an example of generated bitstreams based on two DAGs of two users. Note that we do not generate bitstreams for three-NT chains in this example, as that exceeds what a region can hold.

When generating bitstreams, we attach a small sNIC wrapper to each NT (Figure 5). This wrapper is essential: it enables skipping an NT in a chain (§4.2), monitors the runtime load of the NT (§4.4), ensures signal integrity during PR, and provides a set of virtual interfaces for NTs to access other board resources like on-board memory (§4.5). We store pre-generated bitstreams in the sNIC's on-board memory; each bitstream is small, normally less than 5 MB.

NT region and region reconfiguration. An *NT region* is the unit to launch an NT chain. Each region can be independently re-programmed via FPGA *partial reconfiguration (PR)*. Since the FPGA areas for PR needs to be predetermined before launching the FPGA, the region size also needs to be pre-configured. A larger region could fit more NTs in a chain but would waste FPGA space when NTs cannot fill the whole region. Our advised heuristic is to choose a relatively small region size, because smaller regions enable more efficient usage of FPGA space. A tradeoff of smaller regions is that NT chains that occupy more space will need to be broken up into sub-chains in multiple regions. However, a smaller region could still host long chains of *small NT*s. We observe that it is more beneficial to chain smaller NTs together, as smaller NTs are likely to run shorter, and the overhead we save for avoiding going through the central scheduler would be relatively larger. Thus, smaller region size achieves both efficient space utilization and good application performance. Note that it is also possible to divide an FPGA into regions of different sizes, as shown by previous work [44]. We leave this exploration to future work.

We start a new NT chain when an application is deployed (pre-launch), when the chain is first accessed by a packet in an application (on-demand), or when we scale out an existing NT chain. For the first and third cases, we start the new NT only when there is a free region (see §4.4 for detail). For the on-demand launching case, when all regions are full, we still need to launch the new chain to be able to serve the application. In this case, we need to de-schedule a current NT chain to launch the new chain (see §4.4 for how we pick the region). The sNIC SoftCore handles this context switching with a *stop-and-launch* process. Specifically, the SoftCore sends a signal to the current NTs to let them "stop". These NTs then store their states in on-board memory to prepare for the stop. At the same time, the SoftCore informs the scheduler to stop accepting new packets. The scheduler will

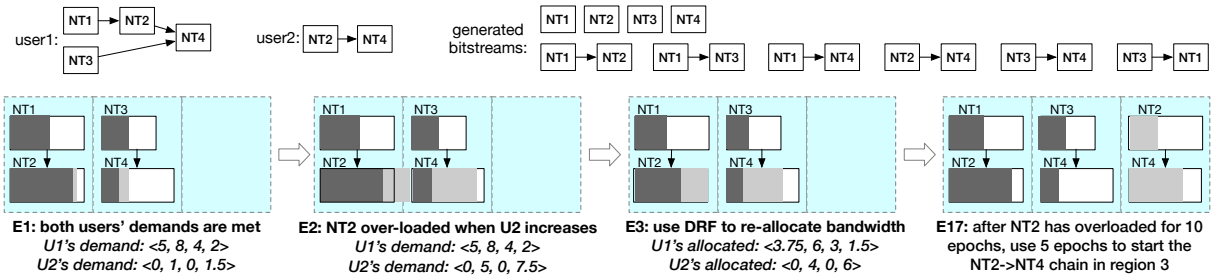

Figure 6: **An Example of NT chaining and scheduling.** *Top: user1 and user2's NT DAGs and sNIC's generated bitstreams for them. Bottom: timeline of NT bandwidth allocation change. Dark grey and light grey represent user1 and user2's load. The launched chains are NT1→NT2 and NT3→NT4, with NT2 and NT4 being shared by the two users. The maximum throughput of NT1, NT2, and NT4 are 10 units each, and NT3's is 7 units. NT2 is the dominant resource for user1, and NT4 is the dominant for user2.*

buffer packets received after this point. After the above *step*s finish, the SoftCore reads the new bitstream from the on-board memory via DMA and starts the FPGA PR process (*launch*). This process is the slowest step, as the maximum reported achievable PR throughput is around 800 MB/s [46], or about 5 ms for our default region size. Afterwards, the newly launched chain can start serving packets, and it will first serve previously buffered packets, if any.

To reduce the overhead caused by NT reconfiguration, we use a technique similar to the traditional victim cache design. We keep a de-scheduled NT chain in a region around for a while unless the region is needed to launch a new chain. If the de-scheduled NT chain is accessed again during this time, we can directly use it in that region, reducing the need to do PR at that time.

4.4 Packet and NT Scheduling Policy

We now discuss our packet and NT scheduling policies. Five major considerations separate us from previous scheduling policies: 1) as FPGA PR is slow, our policy avoids NT reconfiguration as much as possible; 2) we consider every NT, port bandwidth, memory on an sNIC board as a different resource type; 3) we monitor the actual demand at every resource type for all users and use these run-time demands to fairly allocate resources among sharing users; 4) we throttle the ingress bandwidth assignment for different users to in turn control their per-NT bandwidth assignment; and 5) we identify bottleneck/under-utilized resources and automatically scale them out/down. Figure 6 shows an example of how an sNIC with three regions evolves as load changes.

Overall NT scheduling strategy. Our overall strategy is to avoid FPGA PR as much as possible and treat NT context switching (*i.e.*, replacing a current NT chain with a new one through FPGA PR) as a last resort, since context switching prevents the old NT from running altogether and could result in thrashing in the worst case. We also try to hide PR latency behind the performance-critical path as much as possible. Specifically, when a new application is deployed, we check if any of its NTs is missing on an sNIC. If there are any and if there are free regions on the sNIC for these NTs, we *pre-launch* them at the free regions, instead of launching them *on-demand* when the first packet accesses the NTs, as the latter would require waiting for the slow PR and slow down the first few packets. These pre-launched NTs are the first batch of victims we choose to de-schedule if free regions are needed for other NTs.

For on-demand NT launching, we first check if the NT is the same as any existing NT on the sNIC. If so, and if the existing NT still has available bandwidth, we time share the existing NT with the new application. In this case, new traffic can be served immediately. Otherwise, we check if there is a free region or a region that hosts pre-launched NTs. If so, we launch the NT at this region, and new traffic can be served after FPGA PR finishes. Otherwise, we reach out to the distributed sNIC platform and check if any other sNIC has the same NT with available bandwidth or a free region. If so, we route traffic to that sNIC (to be discussed in §5). Otherwise, we inquire the user if they have an alternative way of launching the NT, *e.g.*, running in hardware or software at the endpoint. If all the above fail, we resort to context switching by picking the region that is least loaded and using stop-and-launch to start the NT.

Scheduling with fairness. As we target a multi-tenant environment, sNIC needs to fairly allocate its resources to different applications (**G3**). Different from existing fairness solutions, we treat every NT as a separate type of resource, in addition to ingress bandwidth, egress bandwidth, packet store, and on-board memory space. This is because we support the time sharing of an NT, and different NTs can be shared by different sets of users. Our fairness policy follows Dominant Resource Fairness (DRF) [30], where we identify the *dominant* resource type for each application and seek a fair share for each application's dominant type. We also support weighted DRF [30, 56] for users with different priorities.

Another novelty is our usage of *dynamically monitored resource demand* instead of a user-supplied static resource demand vector as the target used in the DRF algorithm. Specifically, at each *epoch*, we use the ingress parser, egress deparser, the central scheduler, and our virtual memory system to monitor the actual load demand before requests are dispatched to a particular type of resource. For example, for each user, the central scheduler measures the rate of packets

that should be sent next to an NT before assigning credits; *i.e.*, even if there is no credit for the NT, we still capture the intended load it should handle. Based on the measured load at every type of resource for an application, we determine the dominant type of resource and use DRF to allocate resources after considering all applications' measured load vectors. At the end of every epoch, our DRF algorithm outputs a new vector of resource allocation for each application, which the next epoch will use. Compared to static resource demand vectors, our run-time monitoring and dynamic resource vectors can promptly adapt to load changes to maximize resource utilization.

Our final novelty is in how we achieve the assigned allocation. Instead of throttling an application's packets at each NT and every type of resource to match the DRF allocation, we only control the application's ingress bandwidth allocation. Our observation is that since each NT's throughput for an application, its packet buffer space consumption, and egress bandwidth are all proportional to its ingress bandwidth, we could effectively control these allocations through the ingress bandwidth allocation. Doing so avoids the complexity of throttling management at every type of resource. Moreover, throttling traffic early on at the ingress ports helps reduce the load going to the central scheduler and the amount of payload going to the packet store. Our current implementation controls ingress bandwidth through rate limiting. Future work could also use other mechanisms like Weighted Fair Queuing. The only resource that is not proportional to ingress bandwidth is on-board memory space. We control it through our virtual memory system (§4.5).

Finally, the length of an epoch, `EPOCH_LEN`, is a configurable parameter. At every epoch, we need to run the DRF algorithm and possibly change the bandwidth and memory allocation. Thus, `EPOCH_LEN` should be longer than the time taken to perform these operations. Our measured time is $3\mu s$ for running the DRF algorithm, negligible for changing bandwidth, and 15-20μs for swapping out a 2 MB page. Note that swapping out memory can be done in a lazy fashion and does not complete in an epoch. Meanwhile, it is desirable to set a short `EPOCH_LEN` to quickly adapt to load variations and to update rate allocations approximately once per average RTT [23, 42]. Thus, we set the default value of `EPOCH_LEN` to $20\mu s$.

NT auto-scaling. To adapt to load changes, sNIC automatically scales out/down instances of the same NT (instance-level parallelism) (**G2, G4**). Specifically, we use our per-NT monitoring module to identify bottleneck NTs and the load that they are expected to handle. If there are free regions, we add more instances of these NTs by performing PR on the free regions. Since PR is slow, we should scale out an NT only if there is a persistent load increase instead of just occasional load spikes. To do so, we only scale out an NT if the past `MONITOR_PERIOD` time has overloaded the NT. `MONITOR_PERIOD` should be at least longer than the PR latency to avoid thrashing. Since our measured PR latency is $5ms$, we set `MONITOR_PERIOD` to be $10ms$ by default. This length is short enough to capture most real-world traffic peaks [13, 58]. When the load to an NT reduces, we scale down the NT by de-scheduling some of its instances and migrate the traffic of the de-scheduled instances to other running instances of it. We rerun the DRF algorithm right after scaling out/down an NT, since scaling essentially changes the "cap" of the NT's resource amount.

4.5 Virtual Memory System

sNIC's allow NTs to use off-chip, on-board memory. To isolate different applications' memory spaces and to allow the over-subscription of physical memory space in an sNIC, we build a simple page-based virtual memory system. NTs access on-board memory via a virtual memory interface, where each NT has its own virtual address space. Our virtual memory system translates virtual memory addresses into physical ones and checks access permissions with a single-level page table. We use huge pages (2 MB size) to reduce the amount of on-chip memory to store the page table. Physical pages are allocated on demand; when a virtual page is first accessed, sNIC allocates a physical page from a free list.

We further support the over-subscription of an sNIC's on-board memory, *i.e.*, an sNIC can allocate more virtual memory space than its physical memory space. When the physical memory is full, adding more NT would require shrinking memory already assigned to existing applications (§4.3). In this case, we reduce already assigned memory spaces by migrating memory pages to a remote sNIC, *i.e.*, swapping out pages. To decide what pages to swap out, we first use the DRF algorithm to identify what NT(s) should shrink their memory space. Within such an NT, we pick the least recently accessed physical page to swap out. Our virtual memory system tracks virtual memory accesses to capture per-page access frequency. It also transparently swaps in a page when it is accessed. If no other sNIC has free memory space when the sNIC needs to grow its virtual memory space, we reject requests to add new NTs or to enlarge existing NT's memory.

5 Distributed SuperNIC

When we use a single sNIC to consolidate NTs of its connected endpoints, the sNIC needs to be provisioned with the aggregated peak load of these endpoints. To further reduce cost, we build a rack-scale distributed sNIC platform that enables one sNIC to use other sNICs' resources. With this platform, a rack's sNICs only collectively provision for the maximum aggregated load of all the endpoints in the rack.

Distributed Control Plane. SoftCores on the sNICs in a rack form a distributed control plane. They communicate with each other to exchange metadata and cooperate in performing distributed tasks like NT migration and memory swapping. We choose this peer-to-peer design instead of a centralized one, because the latter requires another global

manager and adds complexity and cost to the rack architecture. Every sNIC collects its FPGA space, on-board memory, and port bandwidth consumption, and it periodically sends this information to all the other sNICs in the rack. Each sNIC thus has a global view of the rack and can make decisions like NT migration independently.

NT Migration. As discussed in §4.3, an sNIC tries to allocate an NT chain on another sNIC before resorting to the slower NT context switching. Specifically, the sNIC's SoftCore first identifies the set of sNICs in the same rack that have available resources to host the NT chain. Among them, it picks one that is closest in distance to it. The sNIC's SoftCore then sends the bitstreams of the NT chain to this picked remote sNIC, which launches the chain in one of its own free regions. The original sNIC's SoftCore then sets a rule in the parser MAT to directly forward packets that access this NT chain to the remote sNIC. When the original sNIC has a free region, it moves back the migrated NT chain. It does so by first launching the NT chain locally, then removing the MAT tunneling rule, and finally instructing the remote sNIC to remove its NT chain. If the NT chain is stateful, then the SoftCore manages a state migration process after launching the NT chain locally, by first pausing new traffic, then migrating the NT's states (if any) from the remote sNIC to the local sNIC, and finally removing the MAT rule.

6 Case Studies

We now present two use cases of sNIC that we implemented, one for disaggregated memory and one for regular servers.

6.1 Disaggregated Key-Value Store

We first demonstrate the usage of sNIC in a disaggregated environment by adapting a recent FPGA-based disaggregated memory device called *Clio* [34]. The original Clio device hosts standard physical and link layers, a Go-Back-N reliable transport, and a system that maps keys to physical addresses of the corresponding values. Clients running at regular servers send key-value load/store/delete requests to Clio devices over the network. In our setting, we connect several Clio devices with one sNIC and do not change the client-side or Clio's core key-value mapping functionality.

Disaggregating transport. The Go-Back-N transport consumes a fair amount of on-chip resources (roughly the same amount as Clio's core key-value functionality [34]). We move the Go-Back-N stack from multiple Clio devices to an sNIC and consolidate them by handling the aggregated load. After moving the Go-Back-N stack, we extend each Clio device's link layer to a reliable one (§3).

Disaggregating KV-store-specific functionalities. A unique opportunity that sNIC offers is its centralized position when connecting a set of endpoints, which users could potentially use to more efficiently coordinate the endpoints. We explore this opportunity by building a replication service and a caching service as two NTs in the sNIC.

For **replication**, the client sends a replicated write request with a replication degree K, which the sNIC handles by replicating the data and sending them to K Clio devices. In comparison, the original Clio client needs to send K copies of data to K Clio devices or send one copy to a primary device, which then sends copies to the secondary device(s). The former increases the bandwidth consumption at the client side, and the latter increases end-to-end latency.

For **caching**, the sNIC maintains recently written/read key-value pairs in a small buffer. It checks this cache on every read request. If there is a cache hit, the sNIC directly returns the value to the client, avoiding the cost of accessing Clio devices. Our current implementation that uses simple FIFO replacement already yields good results. Future improvements like LRU could perform even better.

6.2 Virtual Private Cloud

Cloud vendors offer Virtual Private Cloud (VPC) for customers to have an isolated network environment where their traffic is not affected by others and where they can deploy their own network functions such as firewall, network address translation (NAT), and encryption. Today's VPC functionalities are implemented either in software [22, 57, 68] or offloaded to specialized hardware at the server [10, 25, 26]. As cloud workloads experience dynamic loads and do not always use all the network functions (§2), VPC functionalities are a good fit for offloading to sNIC. Our baseline here is regular servers running Open vSwitch (OVS) with three NFs, firewall, NAT, and AES encryption/decryption. We connect sNICs to both sender and receiver servers and then offload these three NFs to each sNIC as one NT chain.

7 Evaluation Results

We implemented sNIC on the HiTech Global HTG-9200 board [2]. Each board has nine 100 Gbps ports, 10 GB on-board memory, and a Xilinx VU9P chip with 2,586K LUTs and 43 MB BRAM. We implemented most of sNIC's data path in SpinalHDL [5] and sNIC's control path in C (running in a MicroBlaze SoftCore [3] on the FPGA). Most data path modules run at 250 MHz. In total, sNIC consists of 8.5K SLOC (excluding any NT code). Figure 7 shows the FPGA resource consumption of different modules in sNIC. The core sNIC modules consume less than 5% resources of the Xilinx VU9P chip, leaving most of it for NTs.

7.1 End-to-End Application Performance

Environment. Our testbed is a rack with a 32-port 100 Gbps Ethernet switch, two HTG-9200 boards acting as two sNICs, eight Dell PowerEdge R740 servers, each equipped with a Xeon Gold 5128 CPU and an NVidia 100 Gbps ConnectX-4 NIC, and two Xilinx 10 Gbps ZCU106 boards running as Clio [34] disaggregated memory devices. Each sNIC uses one port to connect to the ToR switch and one port to connect to the other sNIC. Depending

Module	Logic (LUT)	Memory (BRAM)
sNIC Core	4.36%	4.74%
Packet Store	0.91%	9.17%
PHY+MAC	0.72%	0.35%
DDR4Controller	1.57%	0.29%
MicroBlaze	0.25%	1.81%
Misc	1.52%	0.75%
Total	**9.33%**	**17.11%**

Figure 7: **FPGA Utilization.**

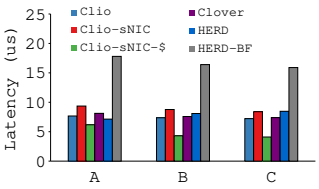

Figure 8: **YCSB Latency.**

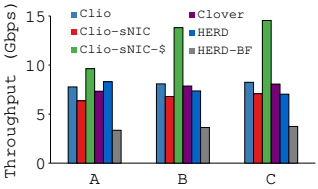

Figure 9: **YCSB Throughput.**

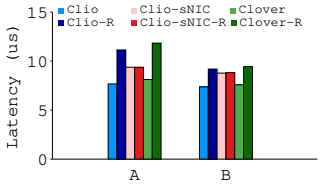

Figure 10: **Replicated YCSB.**

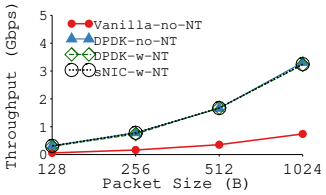

Figure 11: **VPC Performance.**

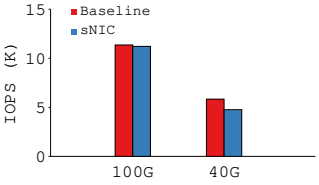

Figure 12: **Consolidation Performance w/ FB Key-Value.**

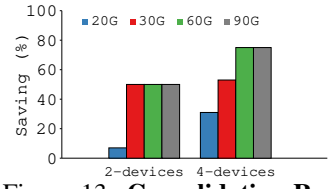

Figure 13: **Consolidation Resource Usage w/ FB KV.**

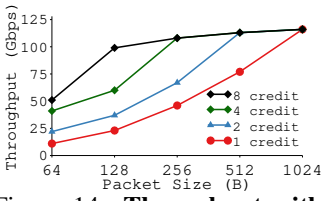

Figure 14: **Throughput with different credits.**

on different evaluation settings, an sNIC's downlinks connect to two servers or two Clio devices.

7.1.1 Disaggregated Key-Value Store

In this set of experiments, we use one client server and two Clio devices. The Clio devices connect to one sNIC which connects to the ToR switch. We run YCSB's workloads A (50% set, 50% get), B (5% set, 95% get), and C (100% get) [20] for these experiments. We use 100K key-value entries and run 100K operations per test, with YCSB's default key-value size of 1 KB and Zipf accesses ($\theta = 0.99$).

Non-replicated YCSB performance and caching. We first evaluate the performance of running YCSB without replication using one client server and one Clio memory device. Figure 8 and 9 plot the average end-to-end latency and throughput of running the YCSB workloads with (1) the original Clio, (2) Clio's Go-Back-N transport offloaded to sNIC (Clio-sNIC), (3) adding caching on top of Clio-sNIC (Clio-sNIC-$), (4) Clover [67], a *passive* disaggregated memory system where all processing happens at the client side and a global metadata server, (5) HERD [41], a two-sided RDMA system where both the client and memory sides are regular servers, and (6) HERD running on the NVidia BlueField SmartNIC [4] (HERD-BF). sNIC's performance is on par with Clio, Clover, and HERD, as it only adds a small overhead to the baseline Clio. With caching NT, sNIC achieves the best performance among all systems, esp. on throughput. This is because all links in our testbed are 100 Gbps except for the link to the 10 Gbps Clio boards. When there is a cache hit at the sNIC, we avoid going to the 10 Gbps Clio boards. HERD-BF performs the worst because of the slow link between its NIC and the ARM processor.

Replicated YCSB performance. We then test Clio, Clover, and Clio with sNIC with replicated write to two Clio devices. HERD does not support replication, and we do not include it here. Clover performs replicated write in a similar way as the baseline Clio, but with a more complex protocol. Figure 10 plots the average end-to-end latency with and without replicated writes using the YCSB A and B workloads. With sNIC's replication NT, the overhead that replication adds is negligible, while both Clio and Clover incur significant overheads when performing replication.

7.1.2 Virtual Private Cloud

We use one sender server and one receiver server, both running Open vSwitch (OVS) [57], to evaluate VPC. Our baseline is the default Open vSwitch that runs firewall, NAT, and AES. We further improve the baseline by running DPDK to bypass the kernel. In the sNIC setup, we connect the sender to an sNIC and the receiver to another sNIC. Each sNIC runs the three NFs as a chain. Figure 11 shows the throughput results. Overall, we find OVS to be a major performance bottleneck in all the settings. Using DPDK improves OVS performance to some extent. Compare to running NTs at servers, offloading them to the sNIC improves throughput, but is still bounded by the OVS running at the endhosts.

7.1.3 Consolidation across Multiple Endhosts

To evaluate the benefit and tradeoff of consolidation, we deploy a testbed with four sender and four receiving servers with four setups: each endhost connects to a ToR switch with 100 Gbps or 40 Gbps link (baseline, no consolidation), and four endhosts connect to an sNIC, each with 100 Gbps or 40 Gbps link, and the sNIC connects to the ToR switch with a 100 Gbps or 40 Gbps link (sNIC consolidation). For both settings, we execute two NTs, firewall and NAT, in FPGA. For the baseline, each endhost has its own set of NTs, while sNIC autoscales NTs as described in §4.4. On each server, we generate traffic to follow inter-arrival and size distribution reported in the Facebook 2012 key-value store trace [13].

Figure 12 reports the throughput comparison of sNIC and the baseline. sNIC only adds 1.3% performance overhead to the baseline under 100 Gbps network and 18% overhead under 40 Gbps network. We further analyze the workload and found its median and 95-percentile loads to be 24 Gbps and 32 Gbps. With four senders/receivers, the aggregated load is mostly under 100 Gbps but often exceeds 40 Gbps. Note that a multi-host NIC would not be able to achieve sNIC's per-

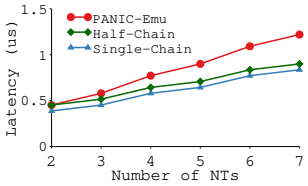

Figure 15: **NT Chain.**

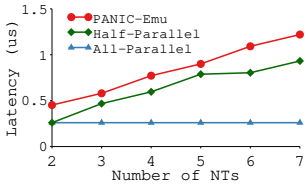

Figure 16: **NT Parallelism.**

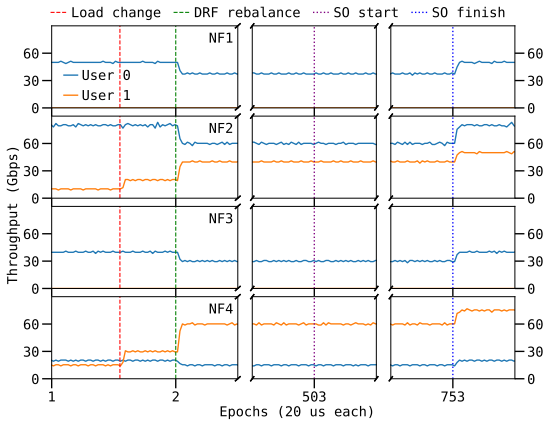

Figure 17: **NT Scheduling.** *SO: Scaling Out.*

formance, as it subdivides the 100 Gbps or 40 Gbps into four 25 Gbps or 10 Gbps sub-links, which would result in each endhost exceeding its sub-link capacity.

We then calculate the amount of FPGA used for running the NTs multiplied by the duration they are used for, to capture the run-time resource consumption with sNIC's autoscaling mechanism. The baseline has one set of NTs per endhost for the whole duration. Figure 13 shows this comparison when consolidating two and four endhosts to an sNIC and using NTs of different performance metrics. For a slower NT (*e.g.*, one that can only sustain 20 Gbps max load), the sNIC auto-scales more instances of it, resulting in less cost saving. Our implementation of firewall NT reaches 100 Gbps, while the AES NT is 30 Gbps, resulting in a 64% cost saving when deploying both of them.

7.1.4 Distributed sNIC

To run an NT at a remote sNIC, an sNIC's SoftCore first sends a control message to the remote sNIC to launch the NT and then installs forwarding rules to its parser. This process takes 2.3μs in our testbed. Afterwards, packets are forwarded to the remote sNIC. We observe an addition of 1.3μs latency when packets go through the remote sNIC.

7.2 Microbenchmark Results

Our microbenchmark tests evaluate various sNIC techniques in a controlled manner. Similar to PANIC [49], for these tests, we implement a packet generator to generate traffic on the FPGA and a delay unit to emulate NTs and other tasks like PR by delaying packets in a controlled way.

7.2.1 sNIC System Module Performance

System throughput. We first test the throughput sNIC can achieve without the bottleneck of any NTs. Specifically, we send packets to an sNIC that uses a dummy NT. These packets go through every functional module of the sNIC, including the central scheduler and the packet store. We change the number of initial credits and packet size to evaluate their effect on throughput, as shown in Figure 14. These results demonstrate that our FPGA prototype of sNIC could reach more than 100 Gbps throughput. With higher frequency, future ASIC implementation could reach even higher throughput. Similar to PANIC [49], we find that having more initial credits achieves higher throughput, and 8 credits are enough for 100 Gbps network.

System latency. We then evaluate the latency overhead an sNIC adds. It takes 1.3μs for a packet to go through the entire sNIC data path (PHY, MAC, sNIC core, MAC, and PHY). Most of the latency is introduced by the third-party PHY and MAC modules, which could potentially be improved with real ASIC implementation and/or a PCIe link. The sNIC core only takes 196 ns, or 14.8% of the total latency. Our scheduler has a fixed delay of 16 cycles, or 64 ns with the FPGA frequency. The synchronization buffer has an overhead of 4 cycles, or 16 ns. To put things into perspective, commodity switch's latency is around 0.8 to 1μs.

7.2.2 NT Chaining and NT-Level Parallelism

NT chaining. To evaluate the effect of sNIC's NT-chaining technique, we change the length of NT sequence from 2 to 7 (as prior work found real NFs are usually less than 7 in sequence [65]). In comparison, we implemented PANIC's scheduling mechanism on our platform, so that everything else is the same as sNIC. We also evaluate the case where sNIC splits the chain into two sub-chains. Figure 15 shows the total latency of running the NT sequence with these schemes. sNIC outperforms PANIC because it avoids going through the scheduler during the sequence for single-chain and only goes through the scheduler once for half-chain.

NT-level parallelism. We then evaluate the effect of sNIC's NT-level parallelism by increasing the number of NTs that could run in parallel. We compare with PANIC (on our platform), which does not support NT-level parallelism. We also show a case where we split NTs into two groups and run these groups as two parallel NT-chains (half-parallel). Figure 16 shows the total latency of these schemes. As expected, running all NTs in parallel achieves the best performance. The tradeoff is more NT region consumption. Half-parallel only uses two regions and still outperforms the baseline.

7.2.3 DRF Fairness and NT Auto-Scaling

To evaluate the effectiveness of our scheduling policy, we ran the synthetic workloads as described in Figure 6 and use

the default EPOCH_LEN of 20μs and MONITOR_PERIOD of 10ms. Figure 17 shows the resulting throughput timeline for different NTs of the two users. In between epoch 1 and 2, the loads of user2 increased to the second step in Figure 6. At the next epoch, we run DRF and adjust the allocation. After the DRF algorithm finishes (in around 3μs), user2 gets a higher (and fairer) allocation of NT2 and NT4, while user1's allocation decreases. After observing NT2 being overloaded for 10ms, the sNIC scales out NT2 by adding one more instance of it at time epoch-503. After PR is done (in 5ms), both user1 and user2's throughput increase.

8 Conclusion

We propose network disaggregation and consolidation by building SuperNIC, a new networking device specifically for a disaggregated datacenter. Our FPGA prototype demonstrates the performance and cost benefits of sNIC. Our experience also reveals many new challenges in a new networking design space that could guide future researchers.

Acknowledgments

We are very grateful to Alex Forencich. He sponsored the FPGA boards and helped us on numerous low-level FPGA debugging sessions.

References

[1] Facebook Multi-Node Server Platform: Yosemite Design Specification. https://www.opencompute.org/documents/multi-node-server-platform-yosemite-v05.

[2] HTG-9200: Xilinx Virtex UltraScale+™ Optical Networking Development Platform. http://www.hitechglobal.com/Boards/UltraScale+_X9QSFP28.htm.

[3] MicroBlaze. https://en.wikipedia.org/wiki/MicroBlaze.

[4] NVIDIA BLUEFIELD DATA PROCESSING UNITS. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[5] SpinalHDL. https://github.com/SpinalHDL.

[6] Adrian M. Caulfield et. al. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*.

[7] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (ATC '18)*.

[8] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[9] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

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

[11] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[12] Krste Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers, February 2014. Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST '14).

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, United Kingdom, June 2012.

[14] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Benn Thomsen, Kai Shi, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *SIGCOMM 2020*. ACM, August 2020.

[15] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[16] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. ASPLOS 2021, 2021.

[17] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment (VLDB '18)*.

[18] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*, 2021.

[19] A. Caulfield, P. Costa, and M. Ghobadi. Beyond smartnics: Towards a fully programmable cloud: Invited paper. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[21] CXL Consortium. https://www.computeexpresslink.org/.

[22] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[23] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control.

In *Proceedings of the ACM SIGCOMM 2006 Conference on SIGCOMM (SIGCOMM '06)*, SIGCOMM '06, 2006.

[24] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[25] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[26] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[27] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[28] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[29] GenZ Consortium. http://genzconsortium.org/.

[30] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[31] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[32] Anubhav Guleria, J. Lakshmi, and Chakri Padala. Emf: Disaggregated gpus in datacenters for efficiency, modularity and flexibility. In *2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2019.

[33] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, 2019.

[34] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. https://arxiv.org/abs/2108.03492.

[35] Hewlett-Packard. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/.

[36] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[37] Intel. https://ark.intel.com/content/www/us/en/ark/products/codename/63546/red-rock-canyon.html.

[38] Intel. Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-rack-scale-architecture.html.

[39] Intel. Intel Unveils Infrastructure Processing Unit. https://www.intel.com/content/www/us/en/newsroom/news/infrastructure-processing-unit-data-center.html.

[40] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[41] Kalia, Anuj and Kaminsky, Michael and Andersen, David G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*.

[42] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the ACM SIGCOMM 2002 Conference on SIGCOMM (SIGCOMM '02)*, SIGCOMM '02, 2002.

[43] 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)*, 2018.

[44] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[45] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[46] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on fpgas? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[47] 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, 2016.

[48] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Sym-*

posium on Computer Architecture (ISCA '09).

[49] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[50] LISA'17. Disaggregating the Network: Switching as a Service. https://www.usenix.org/conference/lisa17/conference-program/presentation/schiff.

[51] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[52] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[53] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. Netkernel: Making network stack part of the virtualized infrastructure. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[54] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[55] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[56] David C. Parkes, Ariel D. Procaccia, and Nisarg Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *ACM Trans. Econ. Comput.*, 2015.

[57] 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 *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[58] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.

[59] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[60] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[61] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*.

[62] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*.

[63] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. *SIGCOMM Comput. Commun. Rev.*, 2012.

[64] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*.

[65] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. SIGCOMM '17, 2017.

[66] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *EuroSys'20*, Heraklion, Crete, 2020.

[67] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[68] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 2021.

[69] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[70] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI'04, 2004.

[71] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[72] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 122–135, 2017.

[73] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan R. K. Ports, and Aurojit Panda. Multitenancy for fast and programmable networks in the cloud. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[74] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding lifecycle management complexity of datacenter topologies. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[75] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. Parabox: Exploiting parallelism for virtual network functions in service chaining. In *Proceedings of the Symposium on SDN Research*, SOSR '17, 2017.

[76] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.