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 network resource pool that consists of a new hardware-based network device called *SuperNIC*. Each SuperNIC consolidates network functionalities from multiple endpoints by fairly sharing limited hardware resources, and it achieves its performance goals by an auto-scaled, highly parallel data plane and a scalable control plane. 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 decomposes 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 independently deploy, manage, and scale different types of resources. Because of these benefits, disaggregation has gained significant traction from both academia [9, 12, 13, 17, 32, 54, 60, 62, 66, 70] and industry [19, 36, 39, 49, 68].

While increasing amounts of effort go into disaggregating compute [33, 62], memory (or persistent memory) [8, 9, 32, 36, 47, 62, 63, 66, 70], and storage [16, 18, 44, 52, 68], the fourth major resource, *network*, has been completely left out. At first glance, "network" cannot be disaggregated from either a traditional monolithic server or a disaggregated device (in this paper collectively called *endpoints*), as they both need to be attached to the network. However, we observe that even though endpoints need basic connectivity, it is not necessary to run *network-related tasks* at the endpoints. These network tasks, or *NT*s, include the transport layer and all high-level layers such as network virtualization, packet filtering and encryption, and application-specific functions.

This paper, for the first time, proposes the concept of *network disaggregation* and builds a real disaggregated network system to segregate NTs from endpoints.

At the core of our network-disaggregation proposal is the concept of a rack-scale disaggregated *network resource pool*, which consists of a set of hardware devices that can execute NTs and collectively provide "network" as a service (Figure 1), similar to how today's disaggregated storage pool provides data storage service to compute nodes. Endpoints can offload (*i.e.*, disaggregate) part or all of their NTs to the network resource pool. After NTs are disaggregated, we further propose to *consolidate* them by aggregating a rack's endpoint NTs onto a small set of network devices.

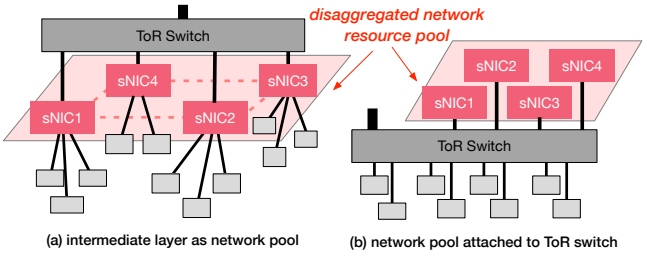

Figure 1: **Overall Architectures of SuperNIC.** *Two ways of connecting sNICs to form a disaggregated network resource pool. In (a), dashed lines represent links that are optional.*

We foresee two architectures of the network resource pool within a rack. The first architecture inserts a network pool between endpoints and the ToR switch by attaching a small set of endpoints to one network device, which is then connected to the ToR switch (Figure 1 (a)). The second architecture attaches the pool of network devices to the ToR switch, which then connects to all the endpoints (Figure 1 (b)).

Network disaggregation and consolidation have several key benefits. (1) Disaggregating NTs into a separate pool allows data center providers to build and manage network functionalities only at one place instead of at each endpoint. This is especially helpful for heterogeneous disaggregated clusters where a full network stack would otherwise need to be developed and customized for each type of endpoint. (2) Disaggregating NTs into a separate pool allows the *independent scaling* of hardware resources used for network functionalities without the need to change endpoints. (3) Each endpoint can use more network resources than what can traditionally fit in a single NIC. (4) With NT consolidation, the total number of network devices can be reduced, allowing a rack to host more endpoints. (5) The network pool only needs to provision hardware resources for the peak *aggregated* bandwidth in a rack instead of each endpoint provisioning for its own peak, reducing the overall CapEx cost.

Before these benefits can be exploited in a real data center, network disaggregation needs to first meet several goals, which no existing solutions fully support (see §2.2).

First, each disaggregated network device should meet endpoints' original performance goals even when handling a much larger (aggregated) load than what each endpoint traditionally handles. The aggregated load will also likely require

many different NTs, ranging from transports to application-specific functionalities. Moreover, after aggregating traffic, there are likely more load spikes (each coming from a different endpoint) that the device needs to handle.

Second, using a disaggregated network pool should reduce the total cost of a cluster. This means that each disaggregated network device should provision the right amount of hardware resources (CapEx) and use as little of them as needed at run time (OpEx). At the same time, the remaining part of a rack (*e.g.*, endpoints, ToR switch, cables) needs to be carefully designed to be low cost.

Third, as we are consolidating NTs from multiple endpoints, in a multi-tenant environment, there would be more entities that need to be isolated. We should ensure that they fairly and safely share various hardware resources in a disaggregated network pool.

Finally, network devices in a pool need to work together so that lightly loaded devices can handle traffic for other devices that are overloaded. This load balancing would allow each device to provision less hardware resources as long as the entire pool can handle the peak aggregated load of the rack.

Meeting these requirements together is not easy as they imply that the disaggregated network devices need to use minimal and properly isolated hardware resources to handle large loads with high variation, while achieving application performance as if there is no disaggregation.

To tackle these challenges and to demonstrate the feasibility of network disaggregation, we built ***SuperNIC*** (or *sNIC* for short), a new hardware-based programmable network device designed for network disaggregation. 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. We further built a distributed sNIC platform that serves as a disaggregated network pool. Users can deploy a single NT written for FPGA or a directed acyclic graph (DAG) execution plan of NTs to the pool.

To tightly **consolidate** NTs within an sNIC, we support three types of resource sharing: (1) splitting an sNIC's hardware resources across different NTs (*space sharing*), (2) allowing multiple applications to use the same NT at different times (*time sharing*), and (3) configuring the same hardware resources to run 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 or to move it off the performance-critical path, *e.g.*, by keeping de-scheduled NTs around like a traditional victim cache, by not over-reacting to load spikes, and by utilizing other sNICs when one sNIC is overloaded.

To achieve high **performance** under large, varying load with minimal cost, we automatically scale (auto-scale) an NT by adding/removing instances of it and sending different flows in an application across these instances. We further launch different NTs belonging to the same application in parallel and send forked packets to them in parallel for faster processing. To achieve low scheduling latency and improve scalability, we 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.

To provide **fairness**, we adopt a fine-grained approach that treats each internal hardware resource separately, *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. We adopt Dominant Resource Fairness (DRF) [31] for this multi-dimensional resource sharing. 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. We also build a simple virtual memory system to **isolate and protect** accesses to on-board memory.

Finally, for **distributed sNICs**, we automatically scale out NTs beyond a single sNIC when load increases and support different mechanisms for balancing loads across sNICs depending on the network pool architectures. For example, with the switch-attached pool architecture, we use the ToR switch to balance all traffic across sNICs. With the intermediate pool architecture, we further support a peer-to-peer, sNIC-initiated load migration when one sNIC is overloaded.

We prototype sNIC with FPGA using two 100 Gbps, multi-port HiTech Global HTG-9200 boards [2]. We build three types of NTs to run on sNIC: reliable transport, traditional network functions, and application-specific tasks, and port two end-to-end use cases to sNIC. The first use case is a key-value store we built on top of real disaggregated memory devices [74]. We explore using sNICs for traditional NTs like the transport layer and customized NTs like key-value data replication and caching. The second use case is a Virtual Private Cloud application we built on top of regular servers by connecting sNICs at both the sender and the receiver side. We disaggregate NTs like encapsulation, firewall, and encryption to the sNICs. We evaluate sNIC and the ported applications with micro- and macro-benchmarks and compare sNIC with no network disaggregation and disaggregation using alternative solutions such as multi-host NICs and a recent multi-tenant SmartNIC [48]. Overall, sNIC achieves 52% to 56% CapEx and OpEx cost savings with only 4% performance overhead compared to a traditional non-disaggregated per-endpoint SmartNIC scenario. 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× when

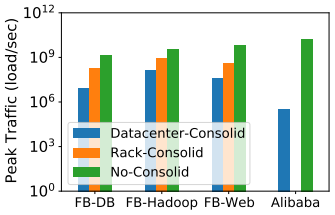

Figure 2: **Consolidation Analysis of Facebook and Alibaba Traces.** *Load represent relative amount and have different units for FB and Alibaba.*

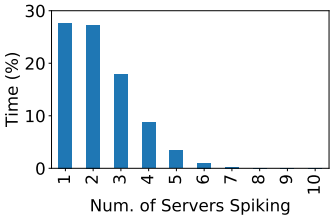

Figure 3: **Load Spike Variation across Endhosts in FB.**

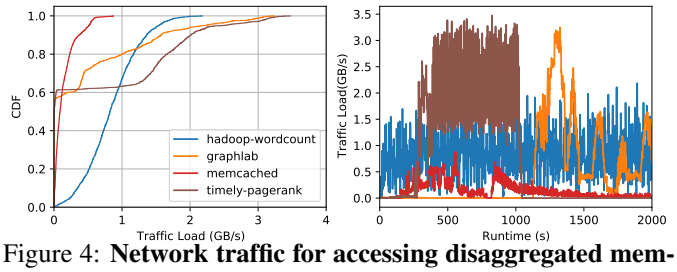

Figure 4: **Network traffic for accessing disaggregated memory.**

compared to today's remote memory systems with no sNIC.

2 Motivation and Related Works

2.1 Benefits of Network Disaggregation

As discussed in §1, disaggregating network functionalities into a separate pool has several key benefits for data centers, some of which are especially acute for future disaggregated, heterogeneous data centers [50, 53, 62].

Flexible management and low development cost. Modern data centers are deploying an increasing variety of network tasks to endpoints, usually in different forms (*e.g.*, software running on a CPU, fixed-function tasks built as ASIC in a NIC, software and programmable hardware deployed in a SmartNIC). It requires significant efforts to build and deploy them to different network devices on regular servers and to different types of disaggregated hardware devices. After deployment, configuring, monitoring, and managing them on all the endpoints is also hard. In contrast, developing, deploying, and managing network tasks in a disaggregated network pool with homogeneous devices is easy and flexible.

Independent scaling. It is easy to increase/decrease network hardware resources in a disaggregated network pool by adding/removing network devices in the pool. Without disaggregation, changing network resources would involve changing endpoints (*e.g.*, upgrading or adding NICs).

Access to large network resources. With disaggregation, an endpoint can potentially use the entire network pool's resources, far beyond what a single NIC or server can offer. This is especially useful when there are occasional huge traffic spikes or peak usages of many NTs. Without network disaggregation, the endpoint would need to install a large NIC/SmartNIC that is bloated with features and resources not commonly used [20, 27].

Beside the above benefits, a major benefit of network disaggregation is cost savings. A consolidated network pool only needs to collectively provision for the peak aggregated traffic and the maximum total NTs used by the whole rack at any single time. In contrast, today's non-disaggregated network systems require each endpoint to provision for its individual peak traffic and maximum NTs. To understand how significant this cost is in the real world, we analyze a set of traces from both traditional server-based production data centers and disaggregated clusters.

Server-based data center traffic analysis. To understand network behavior in server-based data centers, we analyze two sets of traces: a Facebook trace that consists of Web, Cache, and Hadoop workloads [59], and an Alibaba trace that hosts latency-critical and batch jobs together [34].

We first perform a consolidation analysis where we calculate the sum of peaks in each individual endhost's traffic (sum of peak) and the peak of aggregated traffic within a rack and across the entire data center (peak of sum). These calculations model the cases of no disaggregation, disaggregation and consolidation at the rack level and at the data-center level. Figure 2 shows this result for the two data centers. For both of them, a rack-level consolidation consumes one magnitude fewer resources than no consolidation.

We then analyze the load spikes in these traces by comparing different endhosts' spikes and analyzing whether they spike at similar or different times, which will imply how much chance there is for efficient consolidation. Specifically, we count how much time in the entire 1-day trace X number of endhosts spike together. Figure 3 shows that 55% of the time only one or two servers spike together, and only 14% of the time four or more servers spike together. This result shows that servers mostly spike at different times, confirming the large potential of consolidation benefits.

Disaggregated cluster traffic analysis. Resource disaggregation introduces new types of network traffic that used to be within a server, *e.g.*, a CPU device accesses data in a remote memory device. If not handled properly, such traffic could add a huge burden to the data-center network [15]. To understand this type of traffic, we analyzed a set of disaggregated-memory network traces collected by Gao et al. using five endhosts [29]. Figure 4 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 perform a similar analysis using sum of peaks vs. peak of aggregated traffic as our server-based trace analysis. Consolidating just five endhosts already results in 1.1× to 2.4× savings with these traces.

2.2 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 a **programmable**

ToR switch. Programmable switches allow for configurable data planes, but they typically support only a small amount of computation at high line rates. SmartNICs, on the other hand, handle more stateful and complex computations but at lower rates. Transport protocol processing and encrypted communications are examples of complex network tasks better supported by a SmartNIC than a programmable switch. Moreover, existing programmable switches lack proper multi-tenancy and consolidation support [72]. As a consequence, most data center designs require the use of SmartNICs even in the presence of programmable switches, and our proposal simply disaggregates SmartNIC-like capabilities into a consolidated tier.

Another possibility is upcoming *multi-host SmartNICs* (e.g., Mellanox BlueField3) that are enhancements of today's **multi-host NICs** [1, 38]. These NICs connect to multiple servers via PCIe connections and provide general-purpose programmable cores. Our work identifies three key extensions to such devices. (1) Our approach enables a greater degree of aggregation as we enable coordinated management of a distributed pool of network devices. (2) Moreover, in the case of these multi-host SmartNICs, NTs that cannot be mapped to the NIC's fixed functions have to be offloaded as software. In contrast, sNIC allows the acceleration of NTs in hardware, enabling higher performance while tackling issues related to runtime reconfigurability of hardware. (3) Our approach provides adaptive mechanisms for adjusting to workloads and providing fair allocation of resources across applications. It is also worth noting that (1) and (3) can by themselves be used in conjunction with commercial multi-host SmartNICs to achieve a different software-based instantiation of our vision.

Middleboxes are a traditional way of running network functions inside the network either through hardware black-box devices that cannot be changed after deployment [61, 64, 69] or through server-based Network Function Virtualization (NFV) that enables flexible software-based middleboxes [42, 51, 55, 65, 73], but at the cost of lower performance [37, 56]. Our deployment setting differs from traditional datacenter middleboxes: we target "nearby" disaggregation, as in the endhost or SmartNIC tasks are disaggregated to a nearby entity typically located on the same rack. Consequently, our mechanisms are co-designed to take advantage of this locality (e.g., we use simple hardware mechanisms for flow control between the end-host and the disaggregated networking unit). Further, we target network functionality that is expected either at the endhost itself or at the edge of the network, such as endhost transport protocols, applying network virtualization, enhancing security, which all require nearby disaggregation and are also not typically targeted by middleboxes. We do note that our dynamic resource allocation mechanisms are inspired by related NFV techniques, but we apply them in the context of reconfigurable hardware devices.

Finally, there are emerging **interconnections designed for disaggregated devices** such as Gen-Z [30] and CXL [22]. These solutions mainly target the coherence problem where the same data is cached at different disaggregated devices. The transparent coherence these systems provide requires 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

This section gives a high-level overview of the overall architecture of the sNIC platform and how to use it.

Overall Architectures. We support two ways of attaching an sNIC pool in a rack (Figure 1). In the first architecture, the sNIC pool is an intermediate layer between endpoints (servers or devices) and the ToR switch. Each sNIC uses one port to connect to the ToR switch. Optionally, all the sNICs can be directly connected to each other, *e.g.*, with a ring topology. All remaining ports in the sNIC connect endpoints. We expect each of these endpoint-connecting links to have high bandwidth (*e.g.*, 100 Gbps) and the uplink to the switch to have the same or slightly higher bandwidth (*e.g.*, 100 Gbps or 200 Gbps). The second architecture attaches sNICs to the ToR switch, and endpoints directly attach to the ToR switch. In this architecture, the ToR switch re-directs incoming or outgoing traffic to one or more sNICs. Note that for both architectures, the actual implementation could either package the network pool with the ToR switch to form a new "switch box" or be separated out as an pluggable pool.

Requirements for endpoints and the last hop. For basic connectivity, an endpoint needs to have the equivalence of 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/switch (the last hop). 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 and retransmit it. The link layer also requires a simple flow control to slow down packet sending when the sNIC pool is overloaded or the application's fair share is exceeded.

Any interconnect fabric that meets the above requirements can be used as the last-hop link. PCIe is one such example, as it supports reliable data transfer and flow control. Our sNIC prototype uses Ethernet as it is more flexible. We use Priority Flow Control (PFC) for the one-hop flow control and add simple retransmission support. Unlike a traditional reliable link layer, our *point-to-point* reliable link layer is lightweight, as it only needs to maintain one logical flow and a small retransmission buffer for the small Bandwidth-Delay Product (BDP) of the last hop (64 KB in our prototype).

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 [71] and generic software programs running on a processor.

After all the NTs that a user desires have been deployed, the user specifies one or multiple user-written or compiler-generated [46, 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 [25, 46, 55], 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 [20, 27]. 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 5 illustrates the high-level architecture of the sNIC board.

sNIC's data plane consists of reconfigurable hardware (*e.g.*, FPGA) for running user NTs (blue parts in Figure 5) and a small amount of non-reconfigurable hardware (ASIC) for non-user functionalities, similar to the "shell" or "OS" concept [7, 10, 43, 45]. We choose a hardware-based data-plane design because NTs like transports demand high-speed, parallel processing, and a fully reconfigurable hardware allows the maximum flexibility in NT hardware designs. Many of our design ideas can potentially be applied to other types of hardware and software NT implementations, such as PISA pipelines and ARM cores.

We divide the NT area into *region*s, each of which could be individually reprogrammed to run different NTs. Different NT regions can be executed in parallel.

The control plane runs as software on a small set of general-purpose cores (SoftCores for short) (*e.g.*, a small ARM-based SoC). 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.

4.2 Packet Scheduling Mechanism

We now discuss the design of sNIC's packet scheduling mechanism. Figure 6 illustrates the overall flow of sNIC's packet scheduling and execution.

NT-chain-based FPGA architecture and scheduling. As sNIC enables more types of endpoints and workloads to offload their network tasks, the number of NTs and their

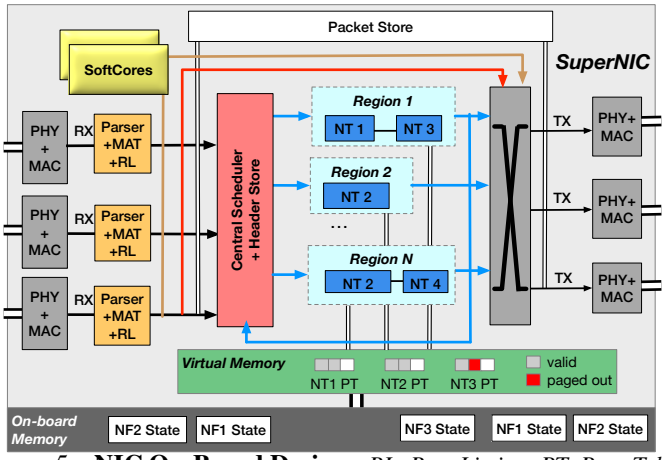

Figure 5: **sNIC On-Board Design.** *RL: Rate Limiter. PT: Page Table*

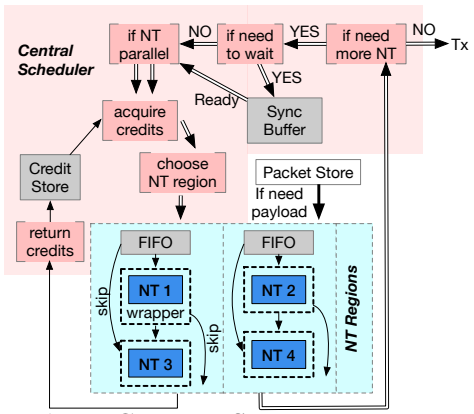

Figure 6: **sNIC Packet Scheduler and NT Region Design.** *Double arrows, single arrows, and thick arrows represent packet headers, credits, and packet payload.*

execution flows will become even more complex, which could impact both the complexity of board design and the performance of packet scheduling. Our idea to confront these challenges is to execute as many NTs as possible in one go, by chaining NTs together. We put chained NTs (called an *NT chain*) in one NT region (*e.g.*, NT1→NT3 and NT2→NT4 in Figure 5). Instead of connecting each NT to a central scheduler (as what PANIC [48] does), we connect each region to the scheduler. Doing so allows us to use a much smaller crossbar between the NT regions and the central scheduler, thereby reducing hardware complexity and area cost (**G7**).

Furthermore, we leverage NT chains to reduce the scheduling overhead and improve the scalability of the central scheduler. Our idea is to *reserve* credits for an *entire* NT 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 central scheduler's scalability (**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 6.

Scheduling packets with NT-level and instance-level parallelism. At an sNIC, different regions run in parallel. We exploit two types of parallelism by controlling what NTs to put in parallel regions. The first type concurrently executes *different packets* at multiple instances of the *same NT chain* (what we call *instance-level parallelism*). We automatically create more/less instances of an NT chain based on load and send different packets in a round-robin way to the parallel instances. We will discuss our NT autoscaling policy in 4.4.

The second type concurrently executes the *same packet* at multiple *different NTs* (what we call *NT-level parallelism*). We infer what NTs can run in parallel in an NT DAG (*e.g.*, in Figure 7, NT1 and NT2 can run in parallel with NT3 for user1). We expect a fair amount of opportunities to explore NT-level parallelism, as previous work found that 53.8% NF pairs can run in parallel [65]. 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 6).

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 involve 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 and defer the policy discussions 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 Representations 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). We enumerate all possible combinations of NTs under user-specified NT DAG ordering requirements when generating bitstreams. 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 7 shows an example of generated bitstreams based on two DAGs of two users. We store pre-generated bitstreams in the sNIC's on-board memory; each bitstream is small, normally less than 5 MB.

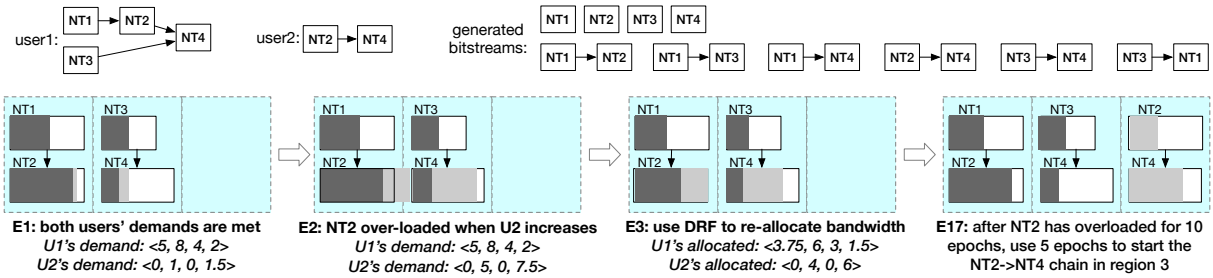

Figure 7: **An Example of NT chaining and scheduling.** *Top: user1 and user2's NT DAGs and sNIC's generated bitstreams for them. Bottom: timeline of NT bandwidth allocation change. Dark grey and light grey represent user1 and user2's load. The launched chains are NT1→NT2 and NT3→NT4, with NT2 and NT4 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.*

When generating bitstreams, we attach a small sNIC wrapper to each NT (Figure 6). 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).

NT chain launching. 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 buffer packets received after this point. After the above *stop 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 [45], 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. Figure 7 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.

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. 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 run the NT at the endpoint if users provide the alternative way of executing it there. If all of 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.

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.

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. When the load to an NT reduces to what can be handled with one instance less, we stop one of its instances and migrate the traffic of the stopped instance to other running instances. Since PR is slow, we should scale out/down an NT only if there is a persistent load increase/decrease instead of just occasional load spikes. To do so, we only scale out/down an NT if the past `MONITOR_PERIOD` time has overloaded/underloaded 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 $5ms$ by default. After experimenting other length, we find this length to work the best with most real-world traffic [14, 59].

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) [31], 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 [31, 57] for users with different priorities.

Instead of a user-supplied static resource demand vector used in traditional DRF systems, we use *dynamically monitored resource demands* as the target in the DRF algorithm. Specifically, at each *epoch*, we use the ingress parser, egress de-parser, 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.

Another 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 (around $3\mu s$ with our implementation). 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 [24, 41]. Thus, we set the default value of `EPOCH_LEN` to $20\mu s$.

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

The design discussion so far focused on a single sNIC. To enable better consolidation and network as a service, we build a rack-scale distributed sNIC platform that enables one sNIC to use other sNICs' resources. With this platform, a rack's sNICs can collectively provision for the maximum aggregated load of all the endpoints in the rack.

As discussed in §7.1, we support two types of sNIC pool topology. For the switch-attached topology, the ToR switch serves as the load balancer across different sNICs. It also decides which sNIC to launch a new instance of an NT with the goal of balancing traffic and efficiently utilizing sNIC hardware resources. Supporting the intermediate-pool topology where the ToR switch cannot perform the above tasks is more complex. Below we discuss our design for it.

SoftCores on the sNICs in the intermediate pool form

a distributed control plane. They communicate with each other to exchange metadata and cooperate in performing distributed tasks. 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 redirect traffic to other sNICs if it is overloaded. To redirect traffic, the sNIC's SoftCore sets a rule in the parser MAT to forward certain packets (*e.g.*, ones to be processed by an NT chain on another sNIC) to the remote sNIC.

If an sNIC is overloaded and no other sNICs currently have the NT chain that needs to be launched, the sNIC tries to launch the chain at another sNIC. 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 (*i.e.*, fewest hops). 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. When the original sNIC has a free region, it moves back the migrated 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.

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 open-source FPGA-based disaggregated memory device called *Clio* [74]. 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. When porting to sNIC, we 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 [35]). 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 (§7.1).

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 [23, 58, 67] or offloaded to specialized hardware at the server [11, 26, 27]. 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). The core sNIC modules consume less than 5% resources of the Xilinx VU9P chip, leaving most of it for NTs (see Appendix for full report). To put it in perspective, the Go-back-N reliable transport we implement consumes 1.3% LUTs and 0.4% BRAM.

Environment. We perform both cycle-accurate simulation (with Verilator [6]) and real end-to-end deployment. Our deployment 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 [35] disaggregated memory devices. Each sNIC uses one port to connect to the ToR switch and one port to connect to the other sNIC. Depending on different evaluation settings, an sNIC's downlinks connect to two servers or two Clio devices.

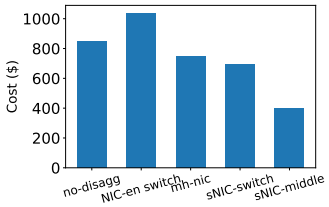

Figure 8: **Per-Endpoint CapEx.** *A rack's network cost divided by endpoint count.*

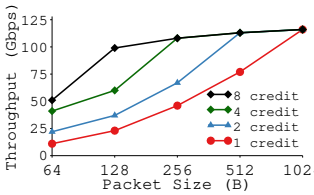

Figure 9: **Throughput with different credits.**

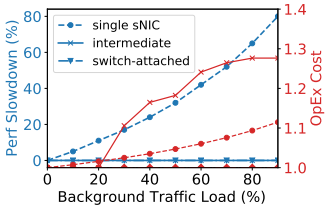

Figure 10: **Distributed sNIC.**

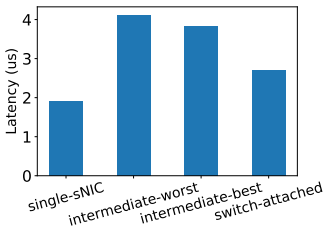

Figure 11: **Topology Comparison.**

7.1 Overall Performance and Costs

CapEx cost saving. We compare the CapEx cost of sNIC's two architectures with no network disaggregation, traditional multi-host NIC, and traditional NIC-enhanced switches. All calculations are based on a single rack and include 1) endpoint NICs, 2) cables between endpoints, sNICs, and the ToR switch, 3) the ToR switch, and 4) cost of sNICs or multi-host NICs. We use market price when calculating the price of regular endpoint NICs and cables. With sNIC, the endpoint NICs can be shrunken down to physical and link layers (based on our prototype estimation, it is roughly 20% of the original NIC cost of $500), and the cables connecting endpoints and sNICs in the middle-layer-pool architecture can be shortened (roughly 60% of the original cable cost of $100 [75]). We estimate the ToR-switch cost based on the number of ports it needs to support and a constant per-port cost of $250 [28].

To estimate the cost of an sNIC, we separate the non-NT parts and the NT regions. The former has a constant hardware cost, while the latter can be provisioned to the peak of aggregated traffic and NT usages, both of which are workload dependent. We use the peak-of-sum to sum-of-peak ratios obtained from the Facebook traces (§2.1). Today's multi-host NIC and NIC-enhanced switches do not consolidate traffic, and we assume that they will be provisioned for the sum-of-peak. See Appendix for detailed calculation.

Figure 8 plots the per-endpoint dollar CapEx cost. Overall, sNIC achieves **52% and 18% CapEx savings** for the middle-layer and switch-attached pool architecture compared to no disaggregation. Multi-host NIC and NIC-enhanced switches both have higher CapEx costs than sNIC, because of its high provisioning without auto-scaling. The NIC-enhanced switches are even more costly than traditional racks mainly because of the added switch ports and NICs.

OpEx saving and single-sNIC performance. We compare a single sNIC connecting four endpoints with the baseline of no disaggregation when these endpoints each run its NTs on its own SmartNIC. We generate workloads for each endpoint based on the Facebook memcached dataset distributions [14]. For the per-endpoint SmartNIC, we statically allocate the hardware resources that can cover the peak load. Overall, we found that sNIC achieves **56% OpEx saving**, because sNIC dynamically scales the right amount of hardware resources for the aggregated load. sNIC only adds **only 4% performance overhead** over the optimal performance that the baseline gets with maximum allocation.

We then test the throughput a real sNIC board can achieve with 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 9. 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.

Next, we evaluate the latency overhead a real sNIC board adds. It takes 1.3μs for a packet to go through the entire sNIC data path. 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. Our scheduler achieves a small, fixed delay of 16 cycles, or 64 ns with the FPGA frequency. To put things into perspective, commodity switch's latency is around 0.8 to 1μs.

Distributed sNICs. To understand the effect of distributed sNIC pool, we compare the two pool topology with a single sNIC (no distributed support). Figure 10 shows the performance and OpEx cost. Here, we use the workload generated from the Facebook distribution as the foreground traffic and vary the load of the background traffic. As background load increases, a single sNIC gets more over-committed and its performance becomes worse. With distributed sNIC, we use an alternative sNIC to handle the load, thus not impacting the foreground workload's performance. Note that the background workload's performance is not impacted either, as long as the entire sNIC pool can handle both workloads' traffic. Furthermore, these workloads are throughput oriented, and we achieve max throughput with distributed sNICs. As for OpEx, the intermediate-pool topology uses one sNIC to redirect traffic to the other sNIC. As the passthrough traffic also consumes energy, its total OpEx increases as the total traffic increases. The switch-attached topology does not have the redirecting overhead. The single sNIC also sees a higher OpEx as load increases because the workload runs longer and consumers more energy.

We then compare the two topologies of sNIC pool. Figure 11 shows the latency comparison. The intermediate-pool topology where we connect the sNICs using a ring has a range of latency depending on whether the traffic only goes through the immediately connected sNIC (single-sNIC) or needs to be redirected to another sNIC when the immedi-

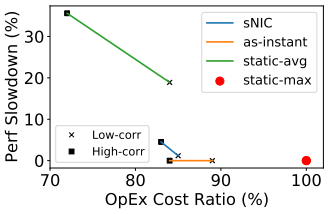

Figure 12: **Performance and OpEx Overview.** *Lower is better for both axis.*

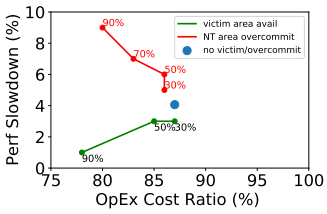

Figure 13: **Single sNIC Sensitivity.** *Each lines is running a different set of experiments.*

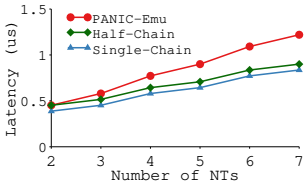

Figure 14: **NT Chain.**

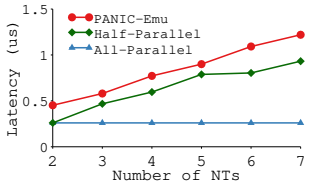

Figure 15: **NT Parallelism.**

ate sNIC is overloaded. Because of the ring connection, this other sNIC's distance to the immediately connected sNIC determines the additional latency incurred (intermediate-best and worst). In contrast, the switch-attached topology has a constant latency, even when one or multiple sNICs are overloaded. This is because the traffic always goes through the switch which directs it to the right sNIC.

7.2 Deep Dive into sNIC Designs

We now perform a set of experiments to understand the implications of sNIC's various designs in terms of performance and OpEx cost, also with the Facebook distribution.

Effect of auto-scaling. We compare our auto-scaled implementation of sNIC with two types of static allocations (*i.e.*, no load-based scaling): allocating for the highest load needs (*static-max*) and allocating for the average load needs (*static-avg*), and an unrealistic auto-scaled scheme which instantly scales the right amount of NT instances as load changes and incurs zero context switching overhead (*as-instant*). We generate two workloads using the Facebook distributions, one where different endpoints spike at similar time (high correlation) and one where they spike at different times (low correlation). Figure 12 shows the performance slowdown (compared to no network disaggregation) and OpEx costs (compared to static-max).

sNIC is at the Pareto frontier compared to the two static allocation schemes. Static-max has the best performance but the worst OpEx cost, as it pays for the peak hardware resources for the entire duration. In contrast, static-avg has the worst performance but best OpEx cost, since it only allocates the resource for the average usage for the entire duration. Compared to sNIC, as-instant only achieves slightly better performance with slightly more OpEx spending, as it tightly matches resources to the load which is unrealistic.

Comparing the two workloads, the low-correlation one always has better performance and more OpEx spending than the high-correlation one (except for as-instant which always yields best performance and static-max which always yields best performance and worst OpEx). This is because with low correlation, the aggregated traffic is more flattened out, which gives sNIC better chance to properly handle. As a result, sNIC scales the right amount of resources to satisfy the load's performance needs. When correlation is high (which is unlikely from our trace analysis in §2.1), there will be fewer but more intensive spikes. When sNIC is not fast or powerful enough to handle some of them, less resources is used but the performance goes down.

Effect of victim cache. To evaluate the effect of our victim-cache design, we set the baseline to be disabling victim cache (blue dot in Figure 13). We then change how often a de-scheduled NT can be kept around as a victim instead of being completely deleted (shown as percentage on the green line). This configuration models how often an sNIC's area is free to host victim NTs. As expected, the more de-scheduled NTs we can keep around, the better performance we achieve, with no victim cache (baseline) having the worst performance. The OpEx implication is less intuitive. Here, we only count the time and amount of NT regions that are actually accessed, as only those will cause the dynamic power (when idle, FPGA has a static power consumption regardless of how it is programmed). With fewer de-scheduled NTs kept around, more NTs need to be re-launched (through FPGA PR) when the workload demands them. These re-launching overhead causes the OpEx to also go up.

Effect of area over-commitment. We change the degree of area over-commitment by limiting how much hardware resources (*i.e.*, NT regions) the workload can use compared to the ideal amount of resources needed to fully execute it. As Figure 13 shows, as we increase the area over-commitment rate, we see worse performance and less resources (OpEx) used. Thus, our design uses distributed sNICs to avoid the over-commitment of a single sNIC.

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 14 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 15 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

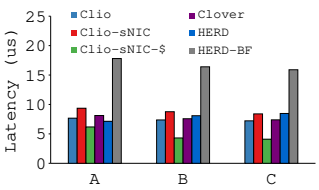

Figure 16: **YCSB Latency.**

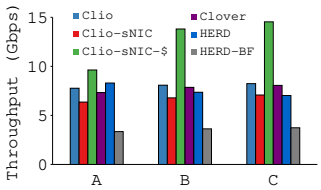

Figure 17: **YCSB Throughput.**

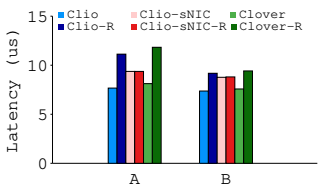

Figure 18: **Replicated YCSB.**

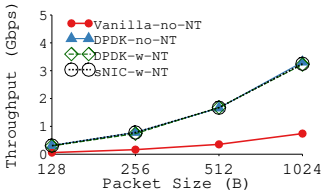

Figure 19: **VPC Performance.**

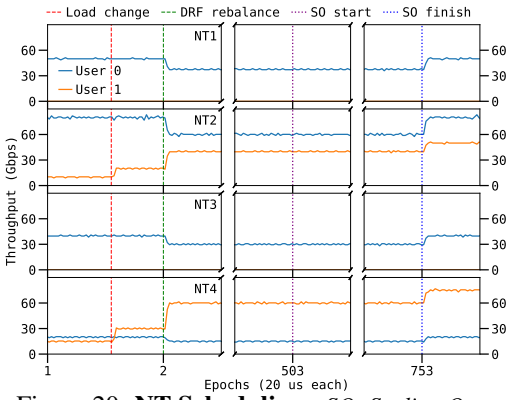

Figure 20: **NT Scheduling.** *SO: Scaling Out.*

only uses two regions and still outperforms the baseline.

DRF Fairness. To evaluate the effectiveness of our scheduling policy, we ran the synthetic workloads as described in Figure 7 and use the default `EPOCH_LEN` of $20\mu s$ and `MONITOR_PERIOD` of $10ms$. Figure 20 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. At the next epoch, we run DRF and adjust the allocation. After the DRF algorithm finishes (in around $3\mu s$), 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.

7.3 End-to-End Application Performance

We now present our end-to-end application performance conducted on our rack testbed. Because of space constraint, *we move the consolidation experiments of these applications to the Appendix*.

7.3.1 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) [21] 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 16 and 17 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 [66], a *passive* disaggregated memory system where all processing happens at the client side and a global metadata server, (5) HERD [40], 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 18 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.

7.3.2 Virtual Private Cloud

We use one sender server and one receiver server, both running Open vSwitch (OVS) [58], 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 19 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.

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.

This work does not raise any ethical issues.

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] Verilator, the fastest verilog/systemverilog simulator.

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

[8] 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)*.

[9] 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.

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

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

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

[13] 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).

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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)*.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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)*.

[28] FS. N8560-64C, 64-Port Ethernet L3 Data Center Switch, 64 x 100Gb QSFP28. https://www.fs.com/products/110481.html.

[29] 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)*.

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

[31] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. 2011.

[32] 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)*.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

[39] 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.

[40] 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)*.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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 Symposium on Computer Architecture (ISCA '09)*.

[48] 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.

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

[50] Shai Bergman Matthias Hille Till Miemietz Nils Asmussen Michael Roitzsch Hermann Härtig Mark Silberstein Lluis Vilanova, Lina Maudlej. Slashing the disaggregation tax in heterogeneous data centers with fractos. EuroSys' 22, 2022.

[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] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 2021.

[53] Joel Nider and Alexandra (Sasha) Fedorova. The last cpu. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, 2021.

[54] 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)*.

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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)*.

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

[64] 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.

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

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] Zhiyuan Guo and Yizhou Shan and Xuhao Luo and Yutong Huang and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, Lausanne, Switzerland, March 2022.

[75] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. RAIL: A case for redundant arrays of inexpensive links in data center networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

A Appendix

A.1 FPGA Resource Utilization

The following table shows the FPGA resources used by sNIC shell. Most of the resources are left for running NTs.

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%**

A.2 Cost Calculation

We explain the different deployment models and the cost calculation formulas behind our CapEx comparisons. We limit our scope to rack-scale as the higher-level network hierarchies are orthogonal to the resource pool deployment models. We calculate that, to deploy a certain number of endpoints, what's the network cost (i.e., the network interface card, cable, and switch port costs).

We compare the following models: 1) Non-disaggregation model, or the traditional model, termed `traditional`. 2) Disaggregation model, in which we insert the network pool between endpoints and the ToR switch (Figure 1 (a)), termed `ring`. 3) Disaggregattion model, in which we connect the pool of network devices directly to the ToR switch (Figure 1 (b)), termed `direct`. For both disaggregation models, we further compare two type of devices: sNIC which has auto-scaling capability and multi-host NIC which can only provision for max resource usage. With runtime dynamic scaling and load balancing features, sNICs can provision for less than the max required resource , the specific ratio is calculated by comparing a particular workload's the sum-of-peak versus the peak-of-sum.

In all, we have the following models under comparison: `traditionl`, `sNIC-direct`, `sNIC-ring`, `mhnic-direct`, `mhnic-ring`.

We now detail the cost calculations. In the traditional non-disaggregation model, each endpoint has a full-fledged NIC and a normal high-speed cable for connection to the ToR switch. In both disaggregation models, since most network tasks are offloaded to the network resource pool, each endpoint can uses a down-scaled NIC. Furthermore, the last hop link layer between endpoints and the network resource pool is reliable, we can leverage down-scaled, cheaper and less reliable physical cable [75].

We use the following parameters in our calculation:

- Deploy `N` devices.
- Each switch port has a cost of `costSwitchPort`
- A full-fledged NIC's cost is `costNIC`. A down-scaled NIC cost is `costDSNIC`.
- A normal high-speed cable cost is `costCable`. A down-scaled less reliable physical cable cost is `costDSCable`.
- A consolidation ratio `consolidRatio` determines how many endpoints are sharing one network resource pool device. We can calculate the number of network pool devices by `M = N / consolidRatio`.
- For a network device, only a certain portion is dedicated to running network task, other parts are used as shell. We define the cost ratio used by network task to be `NTCostRatio`.
- The peak-of-sum versus the sum-of-peak yields the auto-scaling potentials. A multi-host NIC (mhnic) provisions for the sum-of-peak while an sNIC provisions for the peak-of-sum. We call this ratio `capExConsolidRatio`.
- The multi-host NIC's cost can be calculated as `costMHNIC = costNIC * N`.
- The sNIC's cost can be calculated as `costsNIC = costMHNIC * capExRatio`, in which `capExRatio = (1 - NTCostRatio) + NTCostRatio * capExConsolidRatio`.

We now define each model's cost.

The traditional deployment model's cost is straightforward, it includes NIC, cable and switch ports:

$$N * (costNIC + costCable + costSwitchPort) \quad (1)$$

The disaggregation models' cost has more moving parts than the traditional. It includes the down-scaled NICs and cables, network pool devices, the cables to the ToR switch, and switch ports.

The first disaggregation model (Figure 1 (a)) can be calculated as follows (for both `sNIC-ring`, `mhnic-ring`).

$$N * (costDSNIC + costDSCable) + \quad (2)$$

$$M * (costsNIC + costCable + costSwitchPort) \quad (3)$$

The second disaggregation model (Figure 1 (b)) can be calculated as follows (for both `sNIC-direct`, `mhnic-direct`).

$$N * (costDSNIC + costCable + costSwitchPort) + \quad (4)$$

$$M * (costsNIC + costCable + costSwitchPort) \quad (5)$$

This tables shows the real-world numbers we use.

Parameters	Value	Note
costSwitchPort	$250	FS 100Gbps switch [28]
costNIC	$500	Mellanox Connect-X5
costCable	$100	FS DAC 100Gbps cable
costDSNIC	costNIC * 0.2	Numbers from our prototpe
costDSCable	costCable * 0.6	[75]
consolidRatio	4	Current model
NTCostRatio	0.9	Numbers from our prototype
capExConslidRatio	0.23	Facebook Hadoop trace [59]

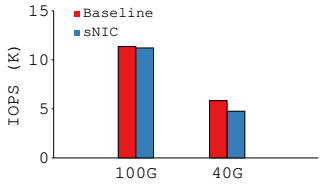

Figure 21: **Consolidation Performance w/ FB Key-Value.**

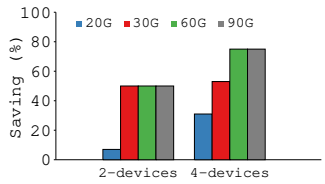

Figure 22: **Consolidation Resource Usage w/ FB KV.**

A.3 End-to-End Application Performance and Cost with Consolidation

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 [14].

Figure 21 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 performance, 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 22 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.