SuperNIC: An FPGA-Based, Cloud-Oriented SmartNIC

Will Lin*
University of California, San Diego
San Diego, California, USA
w5lin@ucsd.edu

Yizhou Shan*
University of California, San Diego
San Diego, California, USA
ys@ucsd.edu

Ryan Kosta
University of California, San Diego
San Diego, California, USA
rkosta@ucsd.edu

Arvind Krishnamurthy
University of Washington
Seattle, Washington, USA
arvind@cs.washington.edu

Yiying Zhang
University of California, San Diego
San Diego, California, USA
yiying@ucsd.edu

ABSTRACT

With CPU scaling slowing down in today's data centers, more functionalities are being offloaded from the CPU to auxiliary devices. One such device is the SmartNIC, which is being increasingly adopted in data centers. In today's cloud environment, VMs on the same server can each have their own network computation (or *network tasks*) or workflows of network tasks to offload to a SmartNIC. These network tasks can be dynamically added/removed as VMs come and go and can be shared across VMs. Such dynamism demands that a SmartNIC not only schedules and processes packets but also manages and executes offloaded network tasks for different users. Although software solutions like an OS exist for managing software-based network tasks, such software-based SmartNICs cannot keep up with the quickly increasing data-center network speed.

This paper proposes a new SmartNIC platform called *SuperNIC* that allows multiple tenants to efficiently and safely offload FPGA-based network computation DAGs. For efficiency and scalability, our core idea is to group network tasks into virtual chains that are dynamically mapped to different forms of physical chains depending on load and FPGA space availability. We further propose techniques to automatically scale network task chains with different types of parallelism. Moreover, we propose a fair sharing mechanism that considers both fair space sharing and fair time sharing of different types of hardware resources. Our FPGA prototype of SuperNIC achieves high bandwidth and low latency performance whilst efficiently utilizing and fairly sharing resources.

CCS CONCEPTS

• **Networks** → **Network adapters**.

KEYWORDS

SmartNIC, multi-tenancy, network programmability

ACM Reference Format:
Will Lin, Yizhou Shan, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. 2024. SuperNIC: An FPGA-Based, Cloud-Oriented SmartNIC. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626202.3637564

*Both authors contributed equally to this research.

FPGA '24, March 3–5, 2024, Monterey, CA, USA

ACM ISBN 979-8-4007-0418-5/24/03
https://doi.org/10.1145/3626202.3637564

1 INTRODUCTION

Data-center networking is seeing three trends recently. First, with the slowdown of Moore's Law and Denard's Scaling [16], more network functionalities are offloaded from the CPU to network devices like RDMA NICs. Computation offloading cannot only benefit from data centers' network infrastructure functionalities but also cloud users' network computing needs. Second, in addition to fixed-logic network functionalities, there is an increasing need for offloading application-specific, customized functionalities [12, 27, 37, 42], especially for cloud users. Third, network speed in the data center is increasing fast. Today, 40 Gbps and 100 Gbps are the norm, with 200 Gbps [43] available and 400 Gbps [44] on the horizon.

These trends call for a hardware-based programmable SmartNIC that can adapt to data-center workloads and cloud environments. Among the potential candidates for future SmartNICs, FPGA stands out due to its programmability at hardware speed and wide adoption in data centers and clouds [6, 7, 18, 50]. However, despite the extensive use of FPGA in data centers and its availability as a service in public clouds, there are no existing solutions that offer network programmability in a cloud setting.

Enabling network programmability on an FPGA-based SmartNIC in a cloud setting carries several implications. First, network tasks are triggered by packets in a flow in a streaming manner, making existing FPGA solutions targeting generic computation unusable, as such computation is triggered by one input from a tenant's application and returns one result after the computation. Secondly, in a cloud environment where multiple tenants share a server, it is essential to *efficiently* and *fairly* share an FPGA-based SmartNIC across tenants. Existing multi-tenancy solutions that work on software-based or ASIC-based SmartNICs cannot be directly applied to FPGA-based SmartNICs as they do not work with reconfigurable hardware resources. Finally, a cloud setting causes significant dynamism due to large variations in network traffic (*i.e.*, flow size and arrival rate) and frequent tenant entry and exit [9, 48]. A SmartNIC capable of quickly adapting to load changes and frequently switching offloaded network tasks is essential in such an environment.

In this paper, we present *sNIC* (*SuperNIC*), an FPGA-based SmartNIC that offers network programmability in a dynamic, multi-tenant environment. sNIC comprises an FPGA for executing user-offloaded network computation, an ASIC for fixed systems logic that receives, schedules, and sends packets, and software cores for executing control-plane tasks, such as policy enforcement. We support offloading classical network functionalities (*e.g.*, transport layer, firewall, encryption) as well as application-specific computation (*e.g.*, key-value store operations [27, 37], real-time analytics [27], and serverless/microservice functions [12, 42]). Unlike generic accelerator computation that is traditionally deployed on FPGA, a user's network computation usually consists of a series of smaller tasks (*e.g.*, firewall, encryption). Many such network tasks are common across tenants and could be supplied by a cloud provider or a third party. Based on these observations, we design sNIC's user interface to be a DAG (Directed Acyclic Graph) of network tasks (*i.e.*, *NT*s) that a user deploys to sNIC before sending network flows designated to the DAG. NTs can be thought of as *microservices* in software, but they are FPGA netlists instead.

In the design of sNIC, we tackled a crucial research question: ***how to launch and execute offloaded NTs on FPGA in a dynamic, multi-tenancy environment?***

Our approach to this question revolves around a novel abstraction we introduce: *virtual network-task chain*, or *NT chain* for brevity. A virtual NT chain is a portion of a user-defined NT DAG that comprises sequential NTs. Depending on the network load and FPGA space availability, one virtual chain can be mapped to multiple instances of physical chains that run in parallel, a partial physical chain whose bandwidth is shared with other tenants, or several smaller physical chains, each of which is a part of other tenants' longer chains, as shown in Figure 1.

To implement NT chains, we first need a board architecture that can efficiently handle the NT computation needs of a considerable number of tenants. Prior works [39] connect each network computation unit (NT in our abstraction) to a crossbar, which cannot scale well. To reduce the number of ports required from the crossbar and achieve a more scalable board architecture, we propose to connect the crossbar to a fixed and relatively small number of FPGA *region*s, each hosting one physical NT chain. At the other end of the crossbar sits sNIC's packet scheduler, which receives incoming packets and dispatches them to designated regions. We design the packet scheduling mechanism to minimize latency overhead for packets passing through the scheduler and to enhance the scheduler's scalability.

With this board architecture, a naive implementation of a virtual NT chain would involve mapping it identically to a physical chain in one region. However, doing so could result in wasted region space for smaller virtual NT chains or cause failure to execute larger ones. For better FPGA resource utilization, we propose a feature of *skipping* NTs in a chain based on packet types, by adding a wrapper to each NT. NT skipping enables us to connect multiple small virtual NT chains into a single long physical chain to fill a region: when a flow only accesses a part of the long physical chain, it can skip the remaining NTs. We can also split a virtual chain into multiple smaller physical chains, each fitting into one region. Furthermore, we can allow a tenant to partially share an NT chain that another tenant uses (*e.g.*, $A \rightarrow C$ in $A \rightarrow B \rightarrow C$) by skipping NTs (*e.g.*, *B*). Together, these techniques enable the use of larger regions and fewer regions that the crossbar connects to while maximizing FPGA space usage across tenants.

To deliver performance under dynamic load, we exploit two types of NT-chain parallelism. The first type explores the parallelism within an NT DAG by executing multiple virtual NT chains as parallel physical chains so as to shorten the total execution time of an NT DAG. The second type increases the overall packet execution throughput by creating multiple parallel physical instances of an entire NT DAG or a smaller virtual chain inside the DAG. We determine the amount of parallelism (*i.e.*, *scaling*) for an NT DAG based on network traffic load, resource availability, and proper share of the resource that a user gets.

Finally, to maximize the usage of an sNIC, we support multiple types of resource sharing, including the *space sharing* of FPGA chip, *bandwidth sharing* of an NT chain, and *time sharing* of an FPGA area by context switching between multiple NT chains. To minimize the performance overhead of context switches, we propose several techniques to hide the overhead of FPGA partial reconfiguration. Additionally, when performing the above types of sharing, sNIC needs to ensure fairness across tenants. Traditional fairness solutions only consider space- or time-sharing. We propose a policy to jointly consider fair space and time sharing in an adaptive and fine-grained manner.

We prototype the sNIC dataplane with FPGA using a 100 Gbps HTG-9200 board [1]. We simulate fairness (scaling) policies in software. We build or port twelve NTs in three types to run on sNIC: a reliable transport, traditional network functions like firewall and encryption, and application-specific tasks such as key-value data replication and caching. We evaluate sNIC with micro- and macro-benchmarks and compare sNIC with PANIC [39], a multi-tenant SmartNIC that supports ASIC-based and CPU-based offloads. Our results show that sNIC delivers 100 Gbps throughput with only 196 ns scheduling overhead. Our real NT-DAG experiments reveal that our NT-chain-based scheduling system can largely reduce the crossbar size while reducing NT-DAG latency by up to 40% compared to PANIC. Furthermore, sNIC improves performance per FPGA area by up to 2.81x, and sNIC achieves up to 3.83x aggregated utilization than prior works while guaranteeing fairness.

2 MOTIVATION AND RELATED WORKS

This section presents related works and motivates sNIC. Table 1 summarizes how sNIC compares to existing systems.

2.1 Network Task Offloading in Data Centers

While the CPU's frequency scaling is slowing down, network speed is increasing much faster. Today, most data centers are running at 40 Gbps or 100 Gbps [21, 38]. Soon, 200 Gbps [43] and 400 Gbps [44] networks will arrive. As a result, the CPU consumption of software network stacks becomes increasingly prohibitive. Network stacks tend to consume 30-40% of CPU cycles [8]. As such, more network functionalities are being offloaded from the CPU to various networking devices. For example, RDMA NICs execute a transport layer in hardware and allow the full bypass of the CPU. Apart from network stacks, today's datacenter users also have the need to perform other higher-level network tasks. The first type of tasks

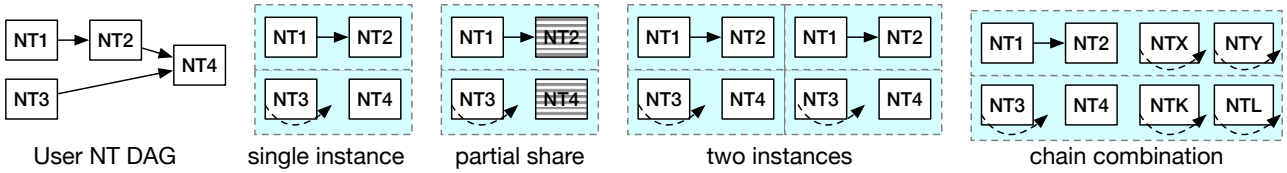

Figure 1: Different Ways of Mapping a DAG to Physical Chains. *Each blue box is a region. Dash arrows represent NT skipping.*

	Multi-Tenancy	Programmable	Hardware Acceleration	Network-Oriented	DAG-Support	Fairness
ARM-based [3]	Y	Y	N	Y	N	Y
ASIC-based [39]	Y	N	Y	Y	P^1	P^2
pipeline-based [57]	Y	P^3	Y	Y	P^1	N
FPGA-based [18]	N	Y	Y	Y	N	N
non-NIC FPGA [31]	Y	Y	Y	N	N	N
SuperNIC	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**

Table 1: SmartNIC Comparison. *P^1: only support sequential chains, not generic DAG. P^3: only primitive Weighted-Fair-Quequing support. P^3: restricted to the pipeline capability.*

is traditional network problems like packet scheduling [53], congestion control [51], and load balancing [45]. The second broad type is applications-specific computation such as accelerating consensus [36], storage [24, 25], databases [32, 35, 61], and machine learning [49]. In a cloud environment, a physical machine can host hundreds or even thousands of (lightweight) VMs or containers, each of which could have its own network offloading need. As the cloud keeps adopting more lightweight virtualization environments, we expect this need to grow even more in the future. These application trends call for SmartNICs that can support multi-tenancy and user-defined computation offloading, while the network-speed trend calls for hardware acceleration of the offloaded computation. Together, they justify the need for a device like sNIC.

2.2 Existing SmartNIC Solutions

SmartNICs are NICs that can perform certain computations. Depending on the hardware providing the computation power, SmartNICs can be categorized into four types.

The first type is ARM-based SmartNICs that run a Linux-like operating system to host user software programs on the ARM processor [3, 34]. Software is flexible but cannot sustain the high processing speed needs with today's 100 Gbps or higher line rate [15].

The second type is ASIC-based SmartNICs, which include specialized network-function accelerators such as AES, compression, regular expression matching, and flow steering/filtering, which users can choose for processing their packets. Although ASICs often offer excellent performance, they only offer fixed sets of functionalities and cannot meet the needs of users who desire to offload customized functionalities. Moreover, ASIC makes it hard and expensive to iterate over versions and updates of deployed network functionalities. Because of the ASIC limitation, many recent SmartNICs combine general-purpose processors with ASIC accelerators [2, 3, 46]. For example, NVIDIA BlueField SmartNICs [3] use general-purpose cores and several fixed-logic network function accelerators together with an RDMA NIC to support network processing offloading. Although when only using the fixed-logic accelerators, BlueField can achieve high throughput, when software offloading is added, the performance drops dramatically [40].

The third type is network-specific pipeline-based SmartNICs that allow users to program on a fixed packet-processing pipeline. For example, Menshen [57] allows for multi-tenant programmability on a pipeline consisting of reconfigurable match-and-action tables. The main limitation with this type of SmartNIC is its restricted programmability, *e.g.*, limited types of actions, a limited number of stages to finish processing a packet, etc. sNIC provides generic hardware programmability by being FPGA-native.

The last type is FPGA-based SmartNICs. Unlike software-based or ASIC-based SmartNICs, FPGA-based ones fall at a middle point between software's ease of programming but limited processing speeds and ASIC's full hardware speed but inflexible fixed-logic functions. FPGA is able to provide orders of magnitude speed improvements compared to software-based solutions while maintaining flexibility at the cost of being harder to program. Because of this benefit and with FPGA development tool chains becoming mature, FPGA-based SmartNICs have been deployed at scale inside Microsoft [18]. However, Microsoft's FPGA-based SmartNICs are for internal infrastructure usages, so they do not have multi-tenancy or cloud support.

Among all prior SmartNIC solutions, PANIC [39] is the most relevant to sNIC. PANIC is a SmartNIC platform that schedules and executes chains of network functionalities for multiple tenants. There are four main differences between PANIC and sNIC. First, PANIC's design is for fixed-logic network function accelerators and CPU-based compute units. In contrast, sNIC is designed for FPGA-based SmartNICs and solves unique challenges related to FPGA reconfiguration and space sharing.Second, PANIC does not adapt to dynamic traffic or allow for NT changes. It focuses on scheduling

packets to static ASIC/software NTs. sNIC adapts to both traffic and NT changes. Third, PANIC does not have sNIC's virtual NT chain abstraction and connects all network function units directly to a crossbar, thereby incurring space and/or performance overhead and scalability limitations. Finally, unlike sNIC, PANIC only has primitive fairness support (*e.g.*, Weighted Fair Queuing), not handling fair spatial and temporal allocation of different resources.

2.3 Generic Multi-Tenant FPGA Solutions

With the increasing popularity of FPGA, solutions have been proposed to provide virtualized, isolated environments for multiple tenants to perform generic computation acceleration (*i.e.*, not network targeted). The first sharing mechanism is time multiplexing, where an entire FPGA chip is dedicated to one tenant for a time period before it is reconfigured to serve the next tenant. Today's cloud FPGA services like AWS F1 [50], Alibaba Cloud [7], and Tencent Cloud [55] all take this approach. The main issue with this mechanism is that most tenants only use a small part of an FPGA, and the remaining FPGA resource is wasted.

The second type is space sharing, where different tenants' applications run on different parts of an FPGA chip. Most space-sharing FPGA solutions partition the physical FPGA into fixed-sized regions, each of which is assigned exclusively to an application [10, 11, 17, 29, 31, 58]. Another approach is exemplified by the *high-throughtput mode* of AmorphOS [28], which packs multiple FPGA applications together at compilation time that are then scheduled onto dynamically-sized regions. Yet another approach taken by ViTAL [59] is to compile and decompose an FPGA application into a set of fixed-size chunks, each of which can be mapped freely onto any of the homogenized, fixed-size slices of ViTAL in an FPGA.

Although these prior works proposed various solutions to time- and space-share an FPGA, they are not targeting network usages and are largely orthogonal to sNIC. sNIC is a multi-tenant SmartNIC that customizes the FPGA for executing network task DAGs. In addition to space-sharing and time-sharing with context switching, sNIC also allows multiple tenants to safely share the same NT's bandwidth. We further propose different types of NT parallelism and autoscaling techniques and a new fairness algorithm targeting fair-sharing of NTs in an FPGA-based SmartNIC.

3 USAGE AND BOARD OVERVIEW

Before delving into the detailed design of sNIC, this section first gives an overview of sNIC, how to use it, its high-level architecture, and the path taken by a packet through sNIC.

3.1 Using SuperNIC

To use sNIC, users need to deploy NTs as netlists, either by acquiring them from a cloud provider or third party or by writing and generating NTs netlists themselves. We assume all cloud-provider-offered NTs can be shared across users, with sNIC's guaranteed performance and memory isolation. Users can also specify other NTs that they are willing to share. We expect users who share NTs to not trust each other but trust the cloud provider and sNIC.

After deploying NTs, a user can specify one or more user-written or compiler-generated [33, 54] DAGs of the deployed NTs. Different from traditional NT execution flows that execute NTs only in sequence, we allow multiple NTs to execute in parallel. The sNIC stores user-specified DAGs in its memory and assigns a unique ID (UID) to each NT in a dag. At run time, each packet carries a UID, which sNIC uses to direct the packet to the appropriate FPGA region.

Finally, in addition to NT DAGs, users also supply their desired ingress bandwidth for each NT DAG. In a cloud setting, this desired ingress bandwidth could be viewed in the same way as how clouds today ask users to specify the size of a VM. Our fairness algorithm will guarantee that all users get at least their desired ingress bandwidth (§ 4.5).

3.2 Board Architecture and Packet Flow

Figure 2 illustrates the high-level architecture of the sNIC board. sNIC's data plane handles all packet processing. It consists of reconfigurable hardware (FPGA) for running NTs (blue parts in Figure 2) and a small amount of non-reconfigurable hardware (ASIC) for non-NT systems stacks, including ingress and egress network stack and parser/de-parser, a central packet scheduler, a virtual memory system to be used by NTs, and a global packet store. sNIC's control plane is responsible dynamically configuring NTs and sNIC itself, as well as performing PR. It runs as software on a small set of general-purpose cores (SoftCores for short) (*e.g.*, a small ARM-based SoC).

When a packet arrives at an RX port, it goes through a standard physical and reliable link layer. Currently, we utilize a user-provided packet descriptor for each packet describing which NTs each packet should run. 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.

Data-plane packet payloads are sent to the *packet store*. Their headers go to the central packet scheduler. The scheduler determines when and which NT(s) will serve a packet and sends the packet accordingly. 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. When all NTs are done, the packet is sent to the TX port.

4 SUPERNIC DESIGN

A key and unique challenge in designing sNICs is space- and performance-efficient execution of hardware-based NTs in a multi-tenant environment. Moreover, we target a dynamic environment where not only the load of an application but also the 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. We design sNIC to simultaneously achieve several critical goals:

- **(G1)** a system stack (non-NT parts) that can process packets at line rate.
- **(G2)** high-throughput, low-latency NT DAG execution.
- **(G3)** quick adaptation to workload changes.
- **(G4)** efficient usage of on-board hardware resources.
- **(G5)** safe and fair sharing of all on-board resources.

This section first discusses how sNIC organizes, deploys, launches, auto-scales, and parallelizes NTs. We then discuss how

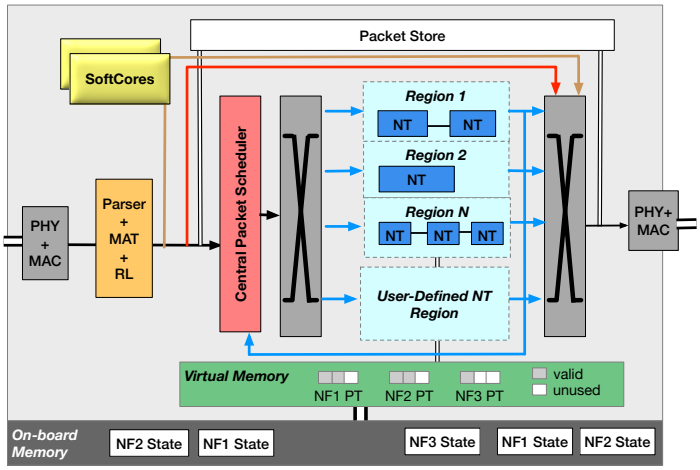

Figure 2: sNIC On-Board Design. *RL: Rate Limiter. PT: Page Table. Orange lines: control message path. Red lines: packets with no NT processing.*

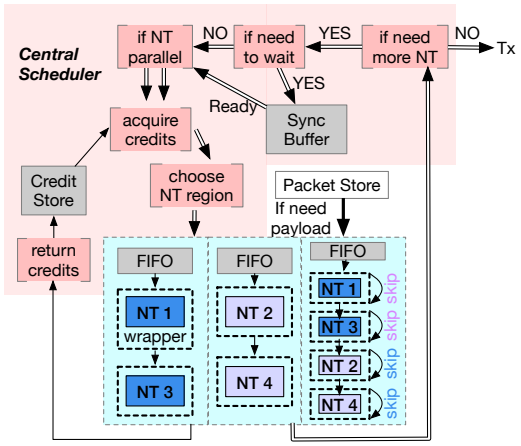

Figure 3: sNIC Packet Scheduler and NT Region Design. *Double arrows, single arrows, and thick arrows represent packet headers, credits, and packet payload.*

sNIC schedules packets, ensures fairness, and enables NTs to access memory via a virtual memory system.

4.1 NT Chain and NT Region

As data-center applications' scale increases fast but a single machine's computing resources increase slowly, it is inevitable that more data will cross the network and more computation will be offloaded to network devices. Unlike general-purpose computation, the types of network computation (*i.e.*, NTs) are usually common across users, and each network task is smaller or can be decomposed into smaller parts in a *microservice* manner. Thus, we expect increasing needs to deploy NT DAGs, many of which have common parts.

Based on the above observation, we propose a basic management and deployment unit of a virtual *NT chain* — a sequential list of NTs that are part of a user-defined NT DAG, as shown in Figure 2. We explore different ways these chains can be designed to optimize throughput, latency, and space. Currently, we let users provide physical NT chains. A compiler could potentially automate the mapping from user DAGs to physical chains, and we leave its development to future work.

Each physical chain is placed in one FPGA *NT region*, which can be independently re-programmed via FPGA *partial reconfiguration (PR)* [41]. We connect the central scheduler and all the NT regions to the two sides of a crossbar. By chaining NTs and by allowing the sharing of a full or a partial physical chain, we can largely reduce the number of ports of the crossbar, compared to prior work that connects each NT to a crossbar [39]. This helps reduce the hardware complexity and area cost of sNIC (**G4**).

NT skipping. To further improve the FPGA space utilization (**G4**), our idea is to allow the *skipping* of arbitrary NT(s) in a physical chain (Figure 3). To achieve skipping, we add wrapper logic around each NT in a DAG, which determines whether a packet is sent to the NT or is skipped to the wrapper of the next NT. The wrapper makes this decision based on a packet header field inserted by the scheduler based on user-specified NT DAGs.

NT skipping enables two designs in sNIC, both for better resource utilization. The first design is to share a partial physical chain among tenants who may have different parts of a deployed NT chain overlapping with their NT DAGs. For example, in Figure 4, when deploying DAG-b, sNIC can directly use NT2 and NT4 in deployed chains of DAG-a by skipping NT1 and NT3 in these chains. The second design is to combine multiple virtual NT chains into a longer physical chain in a region. As a region size can be larger than any user NT chain, being able to form physical chains that are longer than any virtual chain can best use all the space in a region. For example, in Figure 3, user-A's chain-A (NT1→NT3) is concatenated with user-B's chain-B (NT2→NT4) in a region, user-A's packets skip chain-B, and user-B's packets skip chain-A.

NT region size. Since the FPGA areas for PR need to be predetermined before launching the FPGA, the region size also needs to be pre-configured. As we support provider/third-party supplied NTs whose sizes are known to the provider (*i.e.*, known NTs) and user-defined NTs whose sizes are unknown, we also support two types of regions. We use a fixed but configurable size for known-NT regions and dedicate most crossbar ports to them. We use the remaining ports and remaining FPGA space for user-defined NT regions. The size of the known-NT regions should be able to at least fit the largest known NT and can be larger based on the FPGA size. We currently use a fixed size for all known NT regions. Prior FPGA works have proposed splitting FPGA into different sizes [28]. We leave the adoption of such techniques to future work.

4.2 NT Pipelining and Parallelism

When executing an NT DAG, we exploit various forms of pipelining and parallelism, as illustrated in Figure 4. First, we pipeline a physical chain of NTs by dividing it into individual NT stages and sending a new packet to a given stage after it is done executing, as in **S1** of Figure 4. In S1, the two DAGs share a single chain (flattened DAG-a), and to execute DAG-b, NT1 and NT3 are skipped. Second, we execute parallel paths in a DAG as separate physical chains that can run in parallel to reduce the total time needed to process a

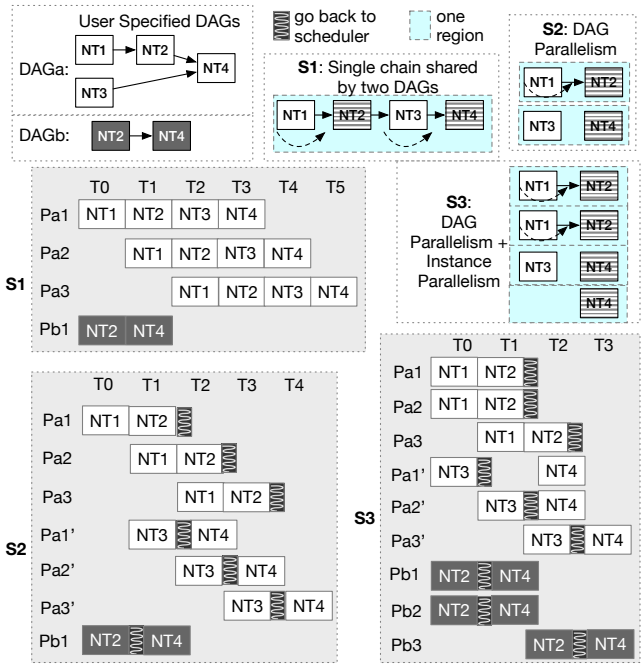

Figure 4: sNIC NT Pipeline. *Two deployed DAGs, a and b. S1, S2, and S3 are three ways of executing them. Pa_i/Pb_i refer to the ith packet targeting the first/second DAG, P'_i refers to a forked packet. T_i refers to a time unit in the timeline.*

packet (which we refer to as *DAG parallelism*). We provide a lightweight splitter NT which sends the *same packet* to different NTs in parralel by duplicating the packet. For example, to reduce the execution time of DAG-a, we can run NT1→NT2 and NT3 in parallel, as in **S2**, which reduces DAG-a's execution time from four units to three units. Note that after executing NT3, the packet goes to a synchronization NT, which waits for the completion of NT1→NT2 before executing NT4 (§4.4). Finally, we create multiple physical instances of the same virtual NT chain to further increase packet-processing throughput (which we refer to as *instance parallelism*). The scheduler sends different packets in a round-robin way to the parallel instances of an NT DAG (§4.4). For example, we create two instances of NT1→NT2 and two instances of NT4 in **S3** to improve **S2**'s overall throughput.

We dynamically calculate recommendations for the target amount of parallelism for a given physical chain. Based on request load to an NT DAG and the fair share we assign to the user (§4.5), our policies make recommendations of which DAGs (or subsets of DAGs) to scale up or down. The number here is not necessarily an integer and can be less than one, as an NT can be shared by multiple users (*e.g.*, NT2 and NT4 in Figure 4) and multiple chains can be placed in one region.

4.3 NT Deployment

Users provide NTs to the sNIC platform ahead of time as FPGA netlists and specify their desired DAGs (virtual chains) of these NTs. If the DAG contains a user-defined NT, we place it in the user-defined-NT region. The user provides FPGA bitstreams for various configurations of their physical chains. Each NT utilizes a small sNIC wrapper (Figure 3) for monitoring the runtime load of the NT (§4.5), ensuring signal integrity during PR, and providing a set of virtual interfaces for NTs to access other board resources like on-board memory. In the above process, if we find a generated bitstream for a virtual chain that leaves a region largely unused (*e.g.*, smaller than half of the region), we recommend consolidating virtual NT chains, creating new bitstreams and deploying them to the sNIC. We store pre-generated bitstreams in the sNIC's on-board memory; each bitstream is small, normally less than 5 MB. We expect a compiler like ViTAL[59] to assist with bitstream generation and leave this exploration to future work.

4.4 Packet Scheduling Mechanism

We now discuss the design of sNIC's packet scheduling mechanism. Figure 3 illustrates the overall flow of sNIC's packet scheduling and execution. Based on the NT-chain architecture, we propose a scheduling mechanism that reduces scheduling overhead and increases the scalability of the scheduler (**G1**, **G2**). Our idea is to *reserve* credits for an *entire* NT chain in a region and then execute the chain as a whole without involving the scheduler in the middle.

Executing chains in their entirety improves both the packet's processing latency and the central scheduler's scalability (**G5**). If the physical NT chain has available credits when a packet is about to be scheduled, the scheduler reserves a credit for the NT chain.

To utilize multiple instances of a virtual NT chain (§4.2), our scheduler pipelines different packets to the instances in a round-robin fashion. To achieve DAG parallelism, a lightweight splitter NT makes copies of the packet and sends them to these regions concurrently. Handling this with an NT allows us to potentially pack multiple copies of a given NT or NT chain into a single region. We can even parallelize only a specific bottleneck NT in the middle of a given NT chain. To obey the order of NTs that users specify, we maintain a *synchronization buffer* to store packet headers after they return from an NT chain's execution and before they can go to the next stage of NTs (Figure 3).

4.5 Fairness Policy

As we target a multi-tenant environment, sNIC needs to fairly allocate its resources to different users (**G5**). Traditional network devices provide fairness by the fair share of the link bandwidth [14, 22, 47, 52], essentially time-sharing network hardware resources. Traditional fairness solutions that target server environments fairly allocate different portions (*i.e., space shares*) of each type of resource among multiple jobs, but they do not time-share or incorporate the networking nature when sharing. Different from prior sharing and fairness solutions, we integrate both space and time sharing on an sNIC for more efficient consolidation, via a two-step approach for achieving fairness.

In the first step, we provide fair space-sharing by determining how many instances of an NT chain (*i.e.*, number of regions) to launch and how much onboard memory to assign to each user. We represent the demanded FPGA size to be the FPGA area multiplied by the ratio of user-required bandwidth to the NT's maximum bandwidth. For example, if a user requires 10 Gbps and uses an

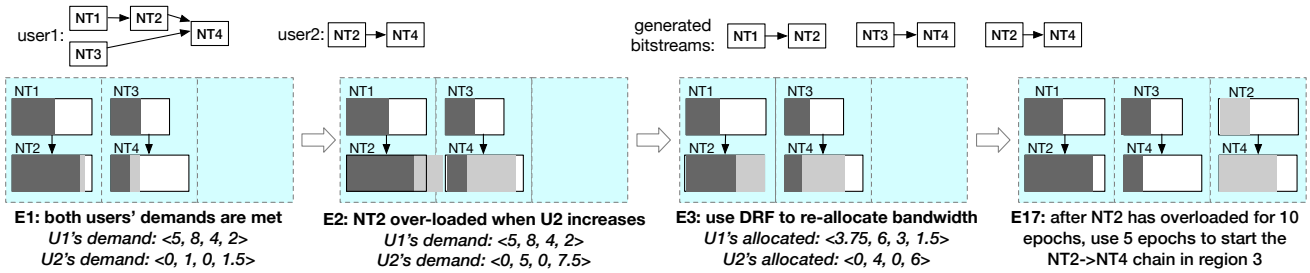

Figure 5: An Example of NT chaining and scheduling. *Top: user1 and user2's NT DAGs and sNIC's generated bitstreams for them. Bottom: timeline of NT bandwidth allocation change. Dark grey and light grey represent user1 and user2's load. The launched chains are NT1⟶NT2, NT3⟶NT4, and NT2⟶NT4 with the first two chain's NT2 and NT4 being shared by both users. The maximum throughput of each NT is 10 Gbps. U1's requested ingress bandwidth is 8 Gbps, and U2's requested ingress bandwidth is 14 Gbps.*

NT DAG that occupies 4 units of FPGA area and supports a maximum of 40 Gbps, we will calculate the user's demand for FPGA as 1 unit ($4 * 10/40$). Afterward, we leverage DRF [20]'s approach by finding each user's dominant type of resources (the most demanding type) and allocating resources proportionally so that all users get equalized dominant shares. This allocation algorithm provides recommendations for FPGA area and on-board memory space.

After the first step, the space allocation is fixed, and we can treat each NT DAG as a fixed-size resource type. We then perform time sharing on these fixed resources. Specifically, our fairness considers the load that is *needed* by each user at every type of resource. These loads are the user-intended loads, not loads that are actually handled. The fairness system fairly assigns a virtual start and a virtual finish time [19] to each packet, allowing users to time-share each NT region, the ingress bandwidth, the egress bandwidth, and the packet store buffer, all based on the monitored intended load.

To enforce the above fairness assignment, the algorithm provides per user's *ingress* bandwidth allocations, instead of limiting a user's bandwidth at each NT and every type of resource. 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 potential complexity of enforcing limits at every type of resource. Moreover, imposing limits early on at the ingress ports could reduce the load going to the central scheduler and the amount of payload going to the packet store.

5 EVALUATION RESULTS

Implementation. Although our design includes ASIC, FPGA, and SoftCore on the sNIC board, for ease of implementation, we build everything on FPGA. We implement most of sNIC's data path in SpinalHDL [4] and sNIC's control path in C. Most data path modules run at 250 MHz. In total, sNIC consists of 23.8k SLOC. Figure 6 shows the FPGA resource consumption of different modules in sNIC and our implemented NTs. The core sNIC modules consume less than 5% resources of the FPGA chip, leaving most of it for NTs. We implement and test our fairness/autoscaling algorithm (§4.5) only in software. We leave integrating fairness/autoscaling with the rest of the sNIC system for future work.

We project sNIC's latency in a potential ASIC implementation in a similar way as previous work [60]. We collect the latency breakdown of time spent in third-party IPs and cycles spent in sNIC components. We then scale the frequency of sNIC component to 2 GHz while maintaining the amount of time spent in third-party IPs. This estimate is conservative as most of the latency is introduced in the third-party MAC and PHY modules. Real ASIC implementations of these IPs would lower overall latency further.

Environment and baseline. We build sNIC on an HiTech Global HTG-9200 board [1], which has 100 Gbps ports, 10 GB on-board memory, and a Xilinx VU9P chip with 2,586K LUTs and 43 MB BRAM. We perform cycle-accurate simulations with Verilator [5] for most experiments, aside from Figures 19 and 20 which are performed on real end-to-end deployments. For the end-to-end deployment, we use a cluster with a 100 Gbps Ethernet switch, an HTG-9200 board, two Dell PowerEdge R740 servers each equipped with a Xeon Gold 5128 CPU and an NVidia 100 Gbps ConnectX-4 NIC, and a Xilinx 10 Gbps ZCU106 board running as the Clio [23] disaggregated memory device.

For most experiments, we use PANIC as a baseline. As PANIC's open-source code is specific to their FPGA setup and cannot run on our FPGA board, we re-implemented PANIC's core scheduling mechanism on our FPGA platform. It uses the same other on-board components like MAC and PHY as sNIC.

5.1 Overall Performance

We first evaluate the throughput an sNIC board can achieve with a dummy NT, *i.e.*, testing the performance of all non-NT parts of sNIC, including the central scheduler. We change the number of initial credits our scheduler sets and packet size to evaluate their effect on throughput, as shown in Figure 7.

With more initial credits, sNIC more packets can be enqueued at the NTs, thus reaching full bandwidth with smaller packet sizes. Similar to PANIC [39], we find that having more initial credits achieves higher throughput, and 8 credits are enough for 100 Gbps network.

Next, we evaluate the latency overhead an sNIC FPGA board adds. It takes 1.3μs for a packet to traverse the entire sNIC data path, from the ingress port to the egress port. Most of the latency is introduced by the third-party PHY and MAC modules, which could potentially be improved with ASIC implementation. The sNIC core only takes 196 ns (or 25 ns with ASIC projection). Our scheduler

Module	LUT	BRAM (KB)
sNIC Core	51.5K	102
Packet Store	10.8K	198
PHY+MAC	8.5K	8
DDR4Controller	18.5K	6
Go-back-N/LB	4900/4533	0
FW/NAT	468/864	0
KV Rep/Cache	458/2452	96/48
dfadd/AES	4266/535	70/0

Figure 6: FPGA Utilization.

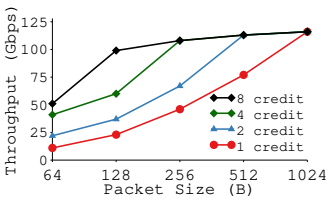

Figure 7: Throughput with different credits.

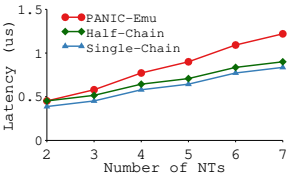

Figure 8: Dummy NT Chain Latency.

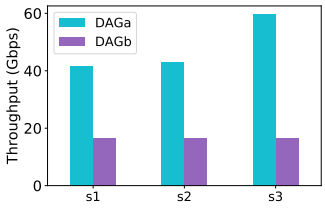

Figure 9: NT DAG Throughput. *S1: sequential. S2: DAG parallel. S3: DAG+instance parallel (Fig. 4)*

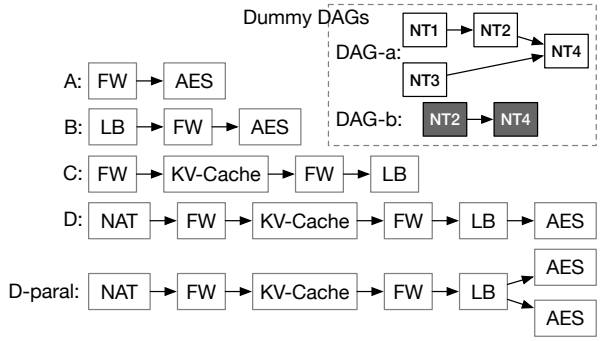

Figure 10: NT DAGs Implemented. *LB: Load Balancer; FW: Firewall.*

achieves a small, fixed delay of 16 cycles, or 64 ns with the FPGA frequency (8 ns with ASIC projection). To put things into perspective, a commodity switch's latency is ~0.8 to 1μs. These results demonstrate that with higher frequency, future ASIC implementation could reach even higher throughput.

5.2 NT and NT DAG Implementation

We implemented three types of NTs and several realistic NT DAGs.

Transport and traditional NTs. To demonstrate sNIC's ability of supporting transport offloading, we implemented a simple reliable transport using the go-back-N protocol on top of a lossless network. When the receiver receives an out-of-order packet, it simply discards it and sends a NACK to the sender. When the sender sees a NACK, it will retransmit all packets that were sent after the last acknowledged packet. We then implement a set of NTs that represent what cloud users use in a Virtual Private Cloud (VPC) setting. VPC allows users to have an isolated network environment. We implemented four NTs on sNIC for VPC: network address translation (NAT), firewall, AES encrypt, and load balancer.

Application-specific NTs. To demonstrate how users can offload application-specific tasks to sNIC, we build an NT for key-value stores. The NT performs key-value pair caching, where the NT maintains recently written/read key-value pairs in a small buffer. If there is a cache hit, the NT directly returns the value to the client.

NT DAGs and NT sharing. We deploy several NT DAGs as illustrated in Figure 10, which are adapted from a prior network-function-chaining work [30]. We further deploy several NT sharing and skipping cases. Here, we assume a scenario where DAG-D already runs on an sNIC, and one of the DAGs, A, B, and C, is then triggered. Because DAG-A/B/C's NTs all exist in the DAG-D, we can leverage partial NT-chain sharing to execute them without launching new DAGs. For example, to execute DAG-A, sNIC skips NAT in DAG-D, execute FW, skips the next three NTs, and finally executes AES.

As before sNIC, there was no support for DAGs, the above NT DAGs that we acquire from the real world are simple chains of NTs. To explore more complex DAGs, we also evaluated two DAGs with dummy NTs, following the example in Figure 4.

5.3 Deep Dive into sNIC Designs

We now perform a set of experiments to understand the implications of sNIC's various designs. For these experiments, we generate traffic load using the Facebook distribution [48], which captures various traffic in the Facebook datacenter.

NT chaining. To evaluate the effect of sNIC's NT-chaining technique and compare it with PANIC, we use an artificial sequence of dummy NTs with length from 2 to 7 (as prior work found real NT chains are usually less than 7 NTs [54]). We also evaluate a case where sNIC splits the chain into two sub-chains. Figure 8 shows the total latency of running the NT sequence with these schemes. sNIC outperforms PANIC because it only goes through the scheduler once (for Single-Chain) or twice (for Half-Chain) for the entire chain, whilst Panic may go through the scheduler after every single NT.

DAG parallelism and instance parallelism. We evaluate the different parallelism mechanisms introduced in §4.2 by measuring the throughput and latency of the three schemes (S1, S2, S3) in Figure 4 for the two DAGs: DAGa and DAGb. Here, we treat all NTs as dummy ones that simply spins for 10 or 50 cycles for each packet; we set each NT's max processing bandwidth to 64 Gbps. Figure 9 plots the throughput of packets going to the two DAGs under the three schemes. For this experiment, we disable the credit system to focus the evaluation on NTs. As can be seen, S3 improves the throughput of DAGa as we launch two instances of NT1, NT2, and NT4. The throughput is less than doubles of S1/S2's because the two instances of NT2 and NT4 are shared by DAGb.

Figure 11 plots the average execution time of the two DAGs under the three schemes and two different NT processing latencies. Adding DAG parallelism (S2) largely reduces the total execution time for DAGa when each NT runs for 50 cycles. However, when each NT runs for 10 cycles, S2 has no execution-time improvement. This is because S2 requires a packet to go back to the scheduler after processing NT3, which adds 16 cycles and is relatively large for

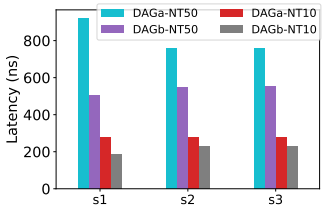

Figure 11: NT DAG Latency. *S1-S3 same as Fig.10. NT50 and NT10 represent 50- and 10-cycle NTs.*

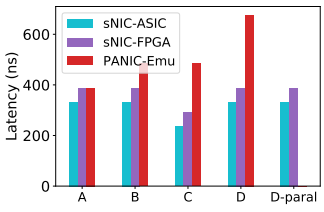

Figure 12: Real NT DAG Latency. *sNIC-ASIC shows projected ASIC performance.*

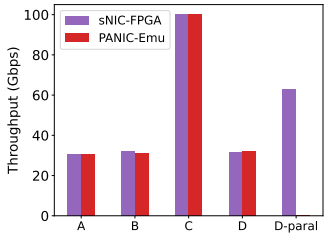

Figure 13: Real NT DAG Throughput.

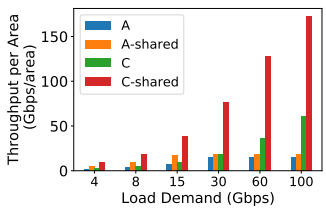

Figure 14: NT sharing *A and C (foreground) sharing chain D (background).*

short-running NTs. This indicates that when NTs are short running, it is more beneficial to chain them in a single region.

Real NT DAGs. We now present real NT DAG evaluation results (DAGs in Figure 10). Figures 12 and 13 plot the latency and throughput of running these DAGs on sNIC and on our emulated PANIC via cycle accurate simulation. Similar to the artificial workload, real NT DAGs also benefit from sNICin latency while maintaining throughput, especially when the DAG is long. PANIC sends packets out as soon as the *first* NT is available. Thus, packets in PANIC can go back to the scheduler after each NT in a chain when the next NT is unavailable. sNIC avoids this overhead by always trying to reserve credits from the entire chain. As our scheduler can handle full line rate for every region, PANIC has no effect on the throughput. We expect similar performance from the ASIC implementation.

sNIC's throughput is similar to PANIC. For both settings, we disable their credit systems to only evaluate NT throughput.

We also evaluate the impact of instance parallelism with a real NT DAG (DAG-D). In this DAG, AES is the bottleneck NT, which has the lowest computation throughput. Thus, by launching two parallel AES NTs in DAG-D-paral, throughput roughly doubles.

NT sharing. We evaluate the effect of sharing NTs by running DAG-D in Figure 10 as a background workload, with bandwidth consumption of 15 Gbps. We then add one of the DAG-A/C as the foreground workload and vary the foreground workload's traffic load. We compare the foreground workload's throughput and FPGA area consumption when enabling and disabling NT sharing. We calculate FPGA area consumption by measuring FPGA LUTs and BRAM each NT consumes, normalized to DAG-D-paral's usage. Figure 14 plots the throughput of DAG-A and DAG-C divided by their area consumption (we disable the credit system for this test). Overall, sharing improves throughput per area for both DAGs, since without sharing, we would need to launch each DAG in its own region. DAG-A's throughput keeps increasing as its load increases until 15 Gbps. Afterwards, its throughput saturates at 15 Gbps. This is because the AES NT can only achieve 30 Gbps maximum throughput, and the background DAG-D consumes 15 Gbps. Yet, we still see a small gain in throughput per area. DAG-C does not utilize AES and can fully utilize the unused bandwidth of the remaining NTs in DAG-D. Thus, it can scale to 60 Gbps, providing a large benefit in throughput per area with sharing.

Autoscaling with pre-launch. We consider how well sNIC adapts to changing traffic with a synthetic workload that starts with having 1x traffic load (10 Gbps), increases to 2x load at time 20ms and to 3x at time 30ms, decreases to 1.5x at time 70ms and increases back to 3x at time 90ms. We use a dummy NT chain whose space fits one region. We model this scenario based on our expected PR time (10ms) and our fairness epoch (20ms). Figure 15 shows the throughput and the number of regions used in a timeline for this workload calculated by our model. Before the workload starts, sNIC has pre-launched two instances of the NT chain (when the chain is deployed). Thus, no PR is needed in the first 30ms. At time 30ms, the increased load requires 3 regions, and sNIC starts the PR of a new region. As the PR finishes at around 35ms, the workload's throughput increases to 30 Gbps. When the load shrinks at 70ms, sNIC does not explicitly remove the launched-NT-chain, as evicting deployed chains does not improve the performance of future PRs. Thus, at time 90ms when the load increases to 3x, no PR is needed.

Fair resource sharing. To evaluate the effectiveness of our fairness policy, we ran a synthetic workload that includes two users in a multi-resource environment. User 1 runs 4 dummy NTs in a chain, and user 2 runs 2 dummy NTs in a chain. User 2's chain is a subset of user 1's. Their user-supplied load requirement is the same. Thus, a good fairness policy should ensure that they each get half of their dominant resource. We run the two workloads for 100 seconds. At 50 seconds, user 1's load increases. We evaluate this workload on three different schemes. *Static* is the baseline, where each user gets assigned an equal number of NT regions. The *DRF* scheme uses DRF to space share but does not allow the time-sharing of NT DAGs amongst different users. *sNIC* is our complete sNIC fairness policy.

Figure 16 shows the resulting dominant share timeline for the two users. *sNIC* consistently delivers a fair share for both users even when one user's load changes. In contrast, the *DRF* scheme cannot adjust to the load change, because it statically decides resource allocation. Figure 17 shows the aggregated throughput of the two users. By allowing the time sharing of common NT DAGs, *sNIC* allows for underutilized DAGs to process other users' flows. Thus, we achieve higher aggregated throughput than DRF. Compared to *Static*, *DRF* can fully use all the regions, resulting in a slightly higher aggregated throughput *Static*.

NT region size and utilization. To evaluate the effect of NT region size, we collect 12 NTs, six from AmorphOS [28] and six implemented by us. We combine them into physical chains of different lengths and deploy them to different areas of our FPGA board. Figure 18 shows two region sizes, one set to be the same as the largest NT among the 12, and one twice the size of that NT. For each region size, we choose the largest NTs and smallest NTs to form chains of different lengths. For example, for 2x region size, when chain lengths are 2, the largest two NTs can fill 79% of the

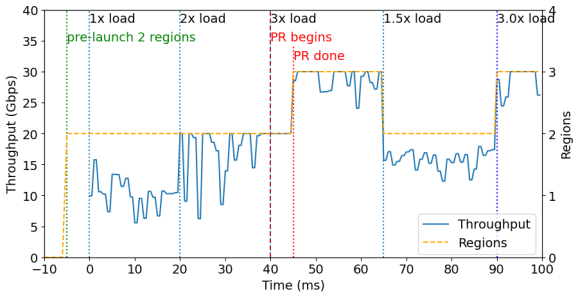

Figure 15: sNIC Adapting to Load Changes.

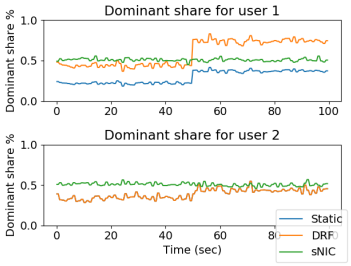

Figure 16: Dominant Resource Share. *The Static line overlaps with DRF for user 2.*

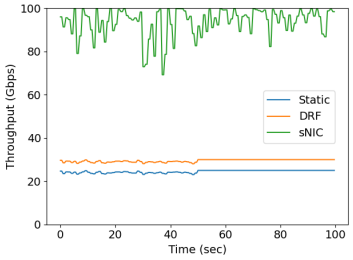

Figure 17: Aggregated Throughput

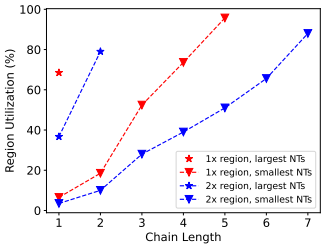

Figure 18: Region Size and Utilization.

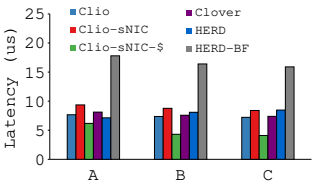

Figure 19: YCSB Latency.

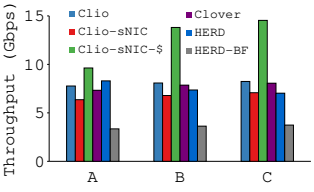

Figure 20: YCSB Throughput.

region, while the smallest two NTs fit 10%, implying that longer chains or combination of chains of smaller NTs are needed to fill the region.

5.4 End-to-End Application Performance

We now present our end-to-end application performance. For this experiment, we deployed sNIC with Clio [60], our recently developed disaggregated memory system. Clio includes a client-side server that issues remote-memory access requests such as key-value get and put. It also includes a network-attached hardware-based memory device that hosts the user data and performs the accesses. When deploying sNIC, we connect the sNIC board to the Clio memory board via Ethernet. We then connect the client-side server and the sNIC board to a 100 Gbps Ethernet switch. After the connection, we offload Clio's transport (go-back-N) to sNIC. We further deploy a key-value cache NT in the sNIC.

We run YCSB's workloads A (50% set, 50% get), B (5% set, 95% get), and C (100% get) [13] for this experiment. 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 ($\theta = 0.99$). We compare sNIC to several baselines: the original Clio, a one-sided RDMA-based key-value store, Clover [56], an RPC-like RDMA-based key-value store, HERD [26]. We run Clover and HERD with NVidia 100 Gbps ConnectX-4 RDMA NIC. We also run HERD on the NVidia 100 Gbps BlueField-Gen1 SmartNIC.

We evaluate the sNIC performance when we only run the Go-back-N transport NT at it and when we run both the transport NT and the key-value caching NT. Figures 19 and 20 show the latency and throughput of sNIC and other baseline systems. 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 Clio board's 10 Gbps link, which connects to the sNIC board. 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. Newer generations of BlueField are more powerful than BlueField-1. Unfortunately, we do not have access to the newer generations.

6 CONCLUSION

We presented an FPGA-based, cloud-oriented SmartNIC, sNIC, that allows for user network task offloading in a dynamic, cloud environment. Our contributions include the proposal of a virtual NT-chain abstraction, its load- and space-based dynamic mapping to physical chains, a packet scheduler designed for NT chains, various parallelism and autoscaling techniques, and a full set of fair sharing mechanisms.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank members of WukLab, including Zhiyuan Guo and Yutong Huang for their feedback to earlier versions of this paper.

This material is based upon work supported by the National Science Foundation under the grant NSF 2016262, the Superconductor Research Corporation under the grant 2023-JU-3135, and gifts from Google, VMware, Meta, and AWS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these institutions.

REFERENCES

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

[2] [n. d.]. LiquidIO II 10/25GbE Adapter family. https://goo.gl/tZPD6c.

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

[4] [n. d.]. SpinalHDL. https://github.com/SpinalHDL.

[5] [n. d.]. Verilator, the fastest Verilog/SystemVerilog simulator. https://www.veripool.org/verilator/.

[6] Adrian M. Caulfield et. al. [n. d.]. A Cloud-scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*.

[7] Alibaba. [n. d.]. Alibaba Cloud FPGA-Accelerated Instances. https://www.aliyun.com/product/ecs/fpga/.

[8] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 93–109. https://www.usenix.org/conference/nsdi20/presentation/arashloo

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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.

[10] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 109–116. https://doi.org/10.1109/FCCM.2014.42

[11] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (Cagliari, Italy) *(CF '14)*. Article 3, 10 pages. https://doi.org/10.1145/2597917.2597929

[12] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2019. λ-NIC: Interactive Serverless Compute on Programmable SmartNICs. http://arxiv.org/abs/1909.11958.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*.

[14] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures Protocols* (Austin, Texas, USA) *(SIGCOMM '89)*. 1–12. https://doi.org/10.1145/75246.75248

[15] Noah Diamond, Scott Graham, and Gilbert Clark. 2022. Securing InfiniBand Networks with the Bluefield-2 Data Processing Unit. In *17th International Conference on Cyber Warfare and Security (ICIWS '22, Vol. 17)*. 459–468. https://doi.org/10.34190/iccws.17.1.58

[16] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 365–376.

[17] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 430–435. https://doi.org/10.1109/CloudCom.2015.60

[18] 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. [n. d.]. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[19] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-Resource Fair Queueing for Packet Processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. Helsinki, Finland.

[20] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*.

[21] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. 2022. Aquila: A Unified, Low-Latency Fabric for Datacenter Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*. Renton, WA.

[22] Pawan Goyal, Harrick M. Vin, and Haichen Chen. 1996. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Palo Alto, California, USA) *(SIGCOMM '96)*. 157–168. https://doi.org/10.1145/248156.248171

[23] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. [n. d.]. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. https://arxiv.org/abs/2108.03492.

[24] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. 121–136.

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

[27] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[28] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[29] Oliver Knodel, Patrick Lehmann, and Rainer G. Spallek. 2016. RC3E: Reconfigurable Accelerators in Data Centres and Their Provision by Adapted Service Models. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 19–26. https://doi.org/10.1109/CLOUD.2016.0013

[30] Nodir Kodirov, Sam Bayless, Fabian Ruffy, Ivan Beschastnikh, Holger H. Hoos, and Alan J. Hu. 2018. VNF Chain Allocation and Management at Data Center Scale. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems (ANCS '18)*.

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

[32] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. 2019. The Case for Network Accelerated Query Processing. In *CIDR*.

[33] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*.

[34] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[35] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3132747.3132751

[36] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 467–483.

[37] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[38] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*.

[39] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[40] Jianshen Liu, Carlos Maltzahn, Craig D. Ulmer, and Matthew Leon Curry. [n. d.]. Performance Characteristics of the BlueField-2 SmartNIC. https://doi.org/10.2172/1783736

[41] Ming Liu, Wolfgang Kuehn, Zhonghai Lu, and Axel Jantsch. 2009. Run-time Partial Reconfiguration speed investigation and architectural design space exploration. In *2009 International Conference on Field Programmable Logic and Applications*. 498–502. https://doi.org/10.1109/FPL.2009.5272463

[42] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. Renton, WA.

[43] Mellanox. [n. d.]. ConnectX-6 EN Single/Dual-Port Adapter ASIC Supporting 200GbE. http://www.mellanox.com/page/products_dyn?product_family=268&mtag=connectx_6_en_ic.

[44] Mellanox. [n. d.]. ConnectX-7 1,2,4-Port Adapter supporting up to 400Gb/s. https://nvdam.widen.net/s/srdqzxgdr5/connectx-7-datasheet.

[45] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA.

[46] Netronome. [n. d.]. Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/.

[47] A.K. Parekh and R.G. Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking* 1, 3 (1993), 344–357. https://doi.org/10.1109/90.234856

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

[49] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[50] Amazon Web Services. [n. d.]. Amazon EC2 F1 Instances - Enable faster FPGA accelerator development and deployment in the cloud. https://aws.amazon.com/ec2/instance-types/f1/.

[51] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation *(NSDI'17)*. USENIX Association, USA.

[52] M. Shreedhar and George Varghese. 1995. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Cambridge, Massachusetts, USA) *(SIGCOMM '95)*. 231–242. https://doi.org/10.1145/217382.217453

[53] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA.

[54] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV *(SIGCOMM '17)*.

[55] Tencent. [n. d.]. Tencent Cloud FPGA Instances. https://cloud.tencent.com/product/fpga.

[56] Shin-Yeh Tsai, Yizhou Shan, , and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them from Remote: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC '20)*. Boston, MA, USA.

[57] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. 2022. Isolation Mechanisms for High-Speed Packet-Processing Pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA.

[58] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. 1078–1086. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199

[59] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*.

[60] Zhiyuan Guo and Yizhou Shan and Xuhao Luo and Yutong Huang and Yiying Zhang. 2022. 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.

[61] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.* 13, 3 (nov 2019), 376–389.