PANIC: A High-Performance Programmable NIC for Multi-tenant Networks

Jiaxin Lin
University of Wisconsin-Madison

Kiran Patel
University of Illinois at Chicago

Brent E. Stephens
University of Illinois at Chicago

Anirudh Sivaraman
New York University (NYU)

Aditya Akella
University of Wisconsin-Madison

Abstract

Programmable NICs have diverse uses, and there is a need for a NIC platform that can offload computation from multiple co-resident applications to many different types of substrates, including hardware accelerators, embedded FPGAs, and embedded processor cores. Unfortunately, there is no existing NIC design that can simultaneously support a large number of diverse offloads while ensuring high throughput/low latency, multi-tenant isolation, flexible offload chaining, and support for offloads with variable performance.

This paper presents PANIC, a new programmable NIC. There are two new key components of the PANIC design that enable it to overcome the limitations of existing NICs: 1) A high-performance switching interconnect that scalably connects independent engines into offload chains, and 2) A new hybrid push/pull packet scheduler that provides cross-tenant performance isolation and low-latency load-balancing across parallel offload engines. From experiments performed on an 100 Gbps FPGA-based prototype, we find that this design overcomes the limitations of state-of-the-art programmable NICs.

1 Introduction

The gap between network line-rates and the rate at which a CPU can produce and consume data is widening rapidly [71, 66]. Emerging *programmable ("smart") NICs* can help overcome this problem [32]. There are many different types of offloads that can be implemented on a programmable NIC. These offloads, which accelerate computation across all of the different layers of the network stack, can reduce load on the general purpose CPU, reduce latency, and increase throughput [32, 48, 59, 69, 13].

Many different cloud and datacenter applications and use cases have been shown to benefit from offloading computation to programmable NICs [13, 48, 59, 42, 32, 37, 49, 46, 62, 47, 30, 36, 70, 69, 35, 55, 45]. However, there is no single "silver bullet" offload that can improve performance in all cases. Instead, we anticipate that different applications will specify their own chains of offloads, and that the operator will then merge these chains with infrastructure-related offloads and run them on her programmable NICs. To realize this vision, this paper presents PANIC, a new scalable and high-performance programmable NIC for multi-tenant networks that supports a wide variety of different types of offloads and composes them into isolated offload chains.

To enable cloud operators to provide NIC offload chains as a service to tenants, a programmable NIC must support: 1) Offload variety: some offloads like cryptography are best suited for hardware implementations, while an offload providing a low-latency bypass for RPCs in an application is better suited for an embedded core [51]; 2) Offload chaining: to minimize wasted chip area on redundant functions, the NIC should facilitate composing independent hardware offload units into a chain as needed, with commonly-needed offloads shared across tenants; 3) Multi-tenant isolation: tenants should not be able to consume more than their allocation of a shared offload; 4) Variable-performance offloads: there are useful offloads that are not guaranteed to run at line-rate, as well as important offloads that run with low latency and at line-rate.

There exist many different programmable NICs [32, 12, 75, 31, 58, 72, 23, 24, 11, 57, 53, 54, 52, 76], but, there is no programmable NIC that is currently able to provide all of the above properties. Existing NIC designs can be categorized as follows, with each category imposing key limitations:

- **Pipeline-of-Offloads NICs** place multiple offloads in a pipeline to enable packets to be processed by a chain of functions [52, 32]. Chaining can be modified in these NICs today but requires a significant amount of time and developer effort for FPGA synthesis, and slow offloads cause packet loss or head-of-line (HOL) blocking.
- **Manycore NICs** load balance packets across many embedded CPU cores, with the CPU core then controlling the processing of packets as needed for different offloads [23, 24, 53, 54, 57, 72, 58]. These designs suffer from performance issues because embedded CPU cores add tens of microseconds of additional latency [32]. Also, no existing manycore NICs provide performant mechanisms to isolate competing tenants. Further, performance on manycore NICs can degrade significantly if the working set does not fit within the core's cache.
- **RMT NICs** use on-NIC reconfigurable match+action (RMT) pipeline to implement NIC offloads. The types of offloads that can be supported by RMT pipelines are limited because each pipeline stage must be able to handle processing a new packet every single clock cycle.

This paper presents the design, implementation and evaluation of PANIC, a new NIC that overcomes the key limitations of existing NIC designs. PANIC draws inspiration from recent work on reconfigurable (RMT) switches [21, 67, 68, 27, 16]. PANIC's design leverages three key principles:

1. *Offloads should be self-contained.* The set of potentially useful offloads is diverse and vast, spanning all of the layers of the network stack. As such, a programmable NIC should be able to support both hardware IP cores and embedded CPUs as offloads.
2. *Packet scheduling, buffering, and load-balancing should be centralized* for the best performance and efficiency because decentralized decisions and per-offload queuing can lead to poor tail response latencies and poor buffer utilization due to load imbalances.
3. Because the cost of small/medium-sized non-blocking fabrics is small relative to the NIC overall, *the offloads should be connected by a non-blocking/low-oversubscribed switching fabric* to enable flexible chaining of offloads.

Following these design principles, this paper makes three key contributions: 1) A novel programmable NIC design where diverse offloads are connected to a non-blocking switching fabric, with chains orchestrated by a programmable RMT pipeline, 2) A new hybrid push/pull scheduler-and-load balancer with priority-aware packet dropping, and 3) An analysis of the costs of on-NIC programmable switching and scheduling that finds them to be low relative to the NIC as a whole.

The PANIC NIC has four components: 1) an RMT switch pipeline, 2) a switching fabric, 3) a central scheduler, and 4) self-contained compute units. The RMT pipeline provides programmable chain orchestration. A high performance interconnect enables programmable chaining at line-rate. The central scheduler provides isolation, buffer management, and load-balancing. Self-contained compute units may be either hardware accelerators or embedded cores and are not required to run at line-rate.

To evaluate the feasibility of PANIC, we have performed both ASIC analysis and experiments with an FPGA prototype. Our ASIC analysis demonstrates the feasibility of the PANIC architecture and shows that the crossbar interconnect topology scales well up to 32 total attached compute units. Our FPGA prototype can perform dynamic offload chaining at 100 Gbps, and achieves nanosecond-level ($<$0.8 μs) packet scheduling and load-balancing under a variety of chaining configurations. We empirically show that PANIC can handle multi-tenant isolation and below line-rate offloads better than a state-of-the-art pipeline-based design. Our end-to-end experiments in a small scale testbed demonstrate that PANIC can achieve dynamic bandwidth allocation and prioritized packet scheduling at 100 Gbps. In total, the components of PANIC, which includes an 8 * 8 crossbar, only consume a total of 11.27% of the total logic area (LUTs) available on the Xilinx UltraScale Plus FPGA that we used. The Verilog code for our FPGA prototype is publicly available [1].

Offload	Config	Tput (Gbps)	Delay
Data Processing			
Compression (Lzrw1)	HW@300MHz	3.6	0.05-3.3μs
Cryptography (AES-256)	HW@300MHz	38.4	407ns
Cryptography (AES-256)	CPU@1.5GHz	0.154	–
Network Processing			
Authentication (SHA1)	HW@220MHz	113.0	0.47-10.8μs
Authentication (SHA1)	CPU@1.5GHz	0.192	–
Application Processing			
Inference (3-layer-NN)	HW@200MHz	120	66ns

Table 1: **A breakdown of the performance of different offloads when implemented in either hardware or software.**

2 Motivation

We discuss in detail the requirements that we envision programmable NICs in multi-tenant networks ought to meet. We then explain why existing NICs designs fail to meet them.

2.1 Requirements

1. Offload Variety: There are a large variety of network offloads, and different types of offloads have different needs. Not all offloads are best implemented on the same type of underlying engine. For example, a cryptography offload can provide much better performance if implemented with a hardware accelerator built from a custom IP core instead of an embedded processor core. To shed light on this, we experimented with a few different types of offloads using an Alpha Data ADM-PCIE-9V3 Programmable NIC [12] to evaluate the behavior of different hardware IP cores that could be used as on-NIC accelerators, and the Rocket Chip Generator [14] to perform cycle-accurate performance measurements of a RISC V CPU to understand the costs of running these offload with an on-NIC embedded processor. Our results in Table 1 indeed show that offloads for encryption/decryption and authentication are a poor fit for embedded CPU designs and should be implemented in hardware.

In contrast, an application-specific offload to walk a hash table that is resident in main memory is better suited for an embedded processor core because a hardware offload may not provide enough flexibility [51]. Thus, a programmable NIC should ideally provide support for both hardware and software offloads.

2. Dynamic Offload Chaining: In the case of hardware accelerators, it is important to be able to compose independent offload functionality into a chain/pipeline to avoid wasted area on redundant functionality. For example, using a programmable NIC to implement a secure remote memory access for a tenant may require the tenant to compose cryptography, congestion control, and RDMA offload engines.

[1] PANIC artifact: https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact

Further, as tenants come and go, and as a given application's traffic patterns change, the on-NIC offload chains will also need to be dynamically updated. This is because different network transfers benefit from different sets of offloads. Further, not every application packet needs every offload. For example, for a key-value store that serves requests from both within-DC and WAN-distributed clients, IPSec and/or compression could be offloaded, but only the packets sent over the WAN may need IPSec authentication and/or compression.

Thus, an ideal programmable NIC should not restrict the type of offloads that may be simultaneously used, and should instead support dynamic offload chaining, i.e., switching and scheduling packets as needed between independent offloads.

3. Dynamic Isolation: Today's data center servers colocate applications from different competing tenants [50, 39, 15, 61, 32]. Each tenant may have its own offload chains that may need to run on a programmable NIC, so it is necessary for a programmable NIC to provide performant low-level isolation mechanisms. For example, consider the case that two tenants A and B are running offload chains where packets are first uncompressed and then sent to an embedded CPU for further processing, and packet contents are such that the workload for tenant B runs at half the rate of that of tenant A. To support this, the NIC's mechanisms must ensure fair packet scheduling at the shared compression offload and that the slow chain does not cause head-of-line (HOL) blocking for the other chain. Further, if a third tenant C were to start, packet processing load across chains may shift. To handle this, the scheduling policy may need to be reprogrammed.

4. Support for offloads with variable and below line-rate performance: Some offloads may not run at line-rate. Of the compression, cryptography, authentication, and inference offloads that we ran on hardware, only inference was able to run at 100 Gbps (Table 1), and others ran well below line-rate. Also, offload performance is variable and sometimes workload-dependent, incurring significant delay for certain requests; see, for example, compression and authentication, whose performance depends on packet size.

These results also show the need for an approach to load-balancing that can accommodate offloads with variable performance. Slow offloads can be duplicated across multiple engines (e.g., 3 AES-256 engines) for line-rate operation.

5. High-Performance Interconnect: It is important for a programmable NIC to be able to provide high throughput for line-rate offloads. In the case where no offloads or only low-latency offloads are used, a programmable NIC should not incur any additional latency. Achieving high performance is complicated by bidirectional communication, multi-port NICs, and chaining. An offload that is used for TX and RX on a dual port NIC needs to operate at four times line-rate to prevent becoming a bottleneck. When offloads are chained, a single packet may traverse the on-NIC network multiple times. Effectively, the NIC must be able to emulate creating a line-rate connection between each hop in an offload chain.

NIC Design	Offload Chaining	Multi-Tenant Isolation	Variable Perf	High Perf	Offload Variety
Pipeline	✗	✗	✗	✓	✗
Manycore	✓	✗	✓	✗	✓
RMT	✗	✓	✗	✓	✗

Table 2: **Programmable NIC designs compared w.r.t. the requirements in Section 2.1.**

2.2 Limitations of Existing Designs

We argue below that programmable NIC designs today (Figure 1) lag behind these requirements (Table 2).

2.2.1 Pipeline Designs

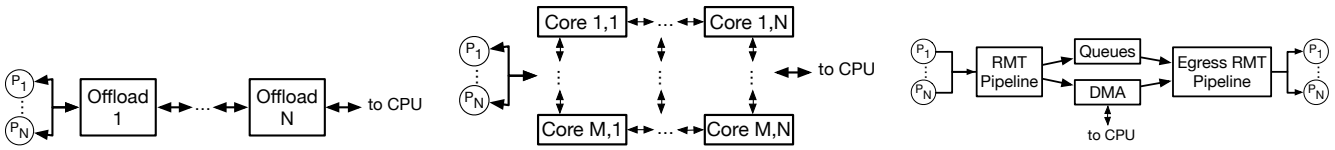
Figure 1a illustrates the pipelined programmable NIC design. In this design, the offloads are arranged in a linear sequence, *i.e.*, a pipeline. Effectively, each offload looks as though it is an independent device attached in the middle of the wire connecting the NIC to a TOR switch. Most existing NICs with on-board FPGAs located as a "bump-in-the-wire" use this design [52, 32, 31], and other NICs use this design for fixed function offloads for TCP checksums and IPSec [6, 38].

Chaining: Chaining offloads is difficult in pipelined designs because of their static offload topology; the offloads are arranged in a line. Although packets can be *recirculated* through the pipeline as needed, this wastes on-NIC bandwidth and hurts line-rate performance.

Variable Performance Offloads: A slow offload that does not run at line-rate can cause HOL blocking in the pipeline of offloads if the pipeline is stalled, and packet loss if the pipeline is not. This can be avoided with routing logic to bypass offloads, but this requires additional buffer memory at each offload: packet arrivals in Ethernet are bursty [41, 17], and it common for tens of packets to arrive back-to-back at line-rate. There would be significant packet loss if offloads that are not guaranteed to run at line-rate are not allocated buffer resources. For offloads where running at line-rate is workload or configuration dependent, the chip area allocated to per-offload buffers would be wasted under some traffic patterns.

Multi-tenant Isolation: In a pipeline, packets are forwarded through offloads that do not need to process the packet. Even if every offload runs at full line-rate, a high latency offload used by Tenant A but not by Tenant B will unnecessarily lead to increased latency for Tenant B. This can be avoided with routing logic to bypass offloads, but this also requires additional buffer memory at each offload to avoid pipeline stalls or packet drops. It is only possible to bypass an offload without stalling the pipeline if there is somewhere else to put the packets that it is currently processing.

Multi-tenant isolation is more problematic if not all offloads are guaranteed to run at line-rate. In this case, if tenant A has already consumed all of the packet buffers allocated for an offload, then tenant B will experience HOL-blocking and possibly packet loss. Although per-offload scheduling



(a) **A pipeline-of-offloads architecture** (b) **A tiled manycore NIC architecture** (c) **A NIC with RMT pipeline [42]**

Figure 1: **Illustrations of existing programmable NIC architectures.**

logic could be used to overcome this limitation, this has area overheads, and, as with per-offload packet buffers, this logic may be unutilized in some workloads.

Offload Variety: Pipeline-of-offloads designs are typically used for programmable NICs that only support hardware offloads. The limitations of pipeline designs are best avoided with low-latency offloads that run at line-rate. Because embedded CPU cores may not run at full line-rate and can incur high processing latency, this makes them a poor fit for pipeline designs. To overcome this limitation, the Azure SmartNIC [32] onloads computation from the programmable NIC to a core on the main CPU for certain tasks. This approach is costly, especially in cloud environments where servers are leased to customers on a per-core basis.

Some FPGA NICs implement all NIC functionality on an FPGA, including the Ethernet MAC and PCIe engines [12, 75, 76, 31]. Such NICs do not have many inherent limitations as a platform. With the right design, such NICs can meet all of our requirements, but no such design currently exists.

2.2.2 Manycore Designs

Figure 1b illustrates a manycore programmable NIC design [24, 53, 72, 73]. These designs implement network offloads by parallelizing flow processing across a large number of embedded processors that are arranged into a multi-hop on-chip tiled topology. Some manycore NICs additionally contain hardware engines for cryptography and compression [72, 58]. This supports chaining and a variety of different offloads, but performance and isolation are poor.

Performance: Manycore NICs use an embedded CPU core to orchestrate the processing of a packet across offloaded functions [34]. This is because the on-chip network cannot parse complex packet headers to determine the appropriate on-NIC addresses for the packet's destination. As a result of this design choice, manycore NICs have throughput and latency problems that prevent high-performance chaining. Further, manycore NICs even struggle to drive 100 Gbps and faster line-rates [32]. Because a single embedded processor is not enough to saturate line-rate, manycore NICs require packet load-balancing and buffering to scale performance.

Manycore NICs struggle to provide high-throughput chaining because manycore interconnects typically only provide enough throughput for a received packet to be sent to one embedded core before being sent via DMA to main memory. As applications become complex, state and caching limitations can require that different offloads be implemented as microservices distributed across cores instead of parallel monoliths. Current manycore NIC designs are not able to provide high performance for such a usecase.

Similarly, involving a CPU in a manycore NIC adds significant packet processing latency that otherwise could be avoided for packets that only need to be processed by a hardware accelerator. For example, Firestone *et al.* [32] report that processing a packet in one of the cores on a manycore NIC adds a latency of 10 μs or more.

Multi-Tenant Isolation: Because manycore NICs must buffer packets and load-balance them across parallel embedded cores [48], the extent to which tenants are isolated is determined by how buffer resources are managed, and how packets are load balanced. Unfortunately, existing manycore NIC designs do not provide explicit control over packet scheduling and buffering [48]. They use FIFO packet queuing and drop-tail buffer management; without any other isolation mechanisms, this can lead to HOL blocking, and drop-tail packet buffers can allow one tenant to unfairly consume buffers.

However, some level of isolation is possible in manycore NICs by (1) statically partitioning CPU resources across different tenants [48], which is inefficient, and (2) then using NIC-provided SDN mechanisms for steering tenants' flows to different cores. Additionally, some NICs such as the Broadcom Stingray allow running an OS to provide software-based isolation through a Linux operating system [22], but this can exacerbate the NICs' performance issues.

2.2.3 Reconfigurable Match+Action (P4) Designs

Figure 1c shows an RMT NIC design; these are built using an ASIC substrate with a programmable match+action (RMT) pipeline [42, 60]. In this model, incoming packets are first parsed by a programmable parser and then sent through a pipeline of M+A tables. Unfortunately, RMT NICs cannot support many interesting offloads (e.g., compression, encryption, or any offload that must wait on the completion of a DMA from main memory) because the actions that are possible at each stage of the pipeline are limited to relatively simple *atoms* that can execute within 1-2 clock cycles [67, 60, 21]. However, RMT NICs do not suffer from multi-tenant performance isolation problems because each offload runs at line-rate with extremely low latency.

3 PANIC Overview

PANIC is a new programmable NIC design that meets the aforementioned requirements (Section 2.1). The core idea be-

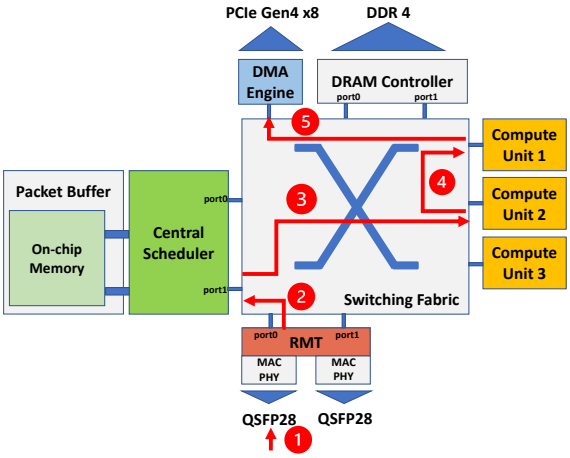


Figure 2: **PANIC Architecture**

hind the design of PANIC is that programmable NICs should be implemented as four logical components (Figure 2): 1) A programmable RMT pipeline, which provides programmable offload chaining on a per-packet basis; 2) A switching fabric, which interconnects all other components in PANIC and enables dynamic chaining at line-rate; 3) A central scheduler, which achieves high-performance packet scheduling, traffic prioritization and traffic isolation; 4) Compute units, each of them running a single offload. This system architecture is shown in Figure 2. We show that this design is suitable for both ASIC and FPGA implementations.

Operational overview: Figure 2 illustrates how PANIC operates when packets are received. In this example, there are three compute units 1, 2, and 3 running services A, B, and C respectively. When packets are received in PANIC in Step 1, they are first processed by the RMT pipeline. The RMT pipeline parses the packet headers and matches on them to identify the chain of offloads that the packet should be forwarded to, and then it generates a PANIC descriptor that contains this offload chain information. In this example, the offload chain that is found will first send the packet to service B and then to service A.

Next, the packet is injected into the switching fabric. If the packet does not need to be processed by any offloads, it will be forwarded straight to the DMA engine of the NIC, which is connected to the interconnect in the same manner as all of the compute units used to implement offloads. Otherwise, it is sent to the central packet scheduler (Step 2).

The scheduler then buffers the packet and orchestrates scheduling and load-balancing the request across its offload chain. When there is no load, packets are chained with a source route that takes them from offload to offload without stopping at the packet scheduler. In this example, the scheduler first buffers this packet until Unit 2 is idle. Then, in Step 3, it steers this packet to Unit 2, and, in Step 4, the packet is directly *pushed* to Unit 1. Finally, in Step 5, after Unit 1 finishes the computation of service A, the source route steers this packet to the DMA engine, which is responsible for transferring packets over the PCIe bus into main memory on the host.

When load is high (not shown), the loaded unit (say Unit 1) detours a packet that was pushed to it (by Unit 2) off to the buffer in the central scheduler. From there, the packet can be *pulled* later for processing by either Unit 1 when it has finished processing a packet or by another parallel unit running the same logic as Unit 1 entirely.

Transmitting packets is similar to receiving packets in reverse, except that the main CPU can associate offload chains with transmit queues beforehand so that the RMT pipeline does not need to process packets before they can be sent to offloads. After the main CPU enqueues packet descriptors, they will be read by the DMA engine, forwarded through an offload chain and managed by the central packet buffer as needed, and then forwarded to the appropriate Ethernet MAC.

PANIC makes it possible to meet all of our requirements:

1. Offload Variety: Each offload in PANIC is an independent tile attached to the high-performance interconnect, and the RMT pipeline builds the packet headers necessary to enable hardware offloads to process packet streams without any additional routing or packet handling logic. This allows for a large variety of different types of computation to be performed by the offload engine, including hardware IP cores, embedded processors, and even embedded FPGAs [74].

2. Dynamic Offload Chaining: Installing a new chain in the RMT pipeline for received packets involves programming lookup tables, and installing a new chain for a transmit queue can be done by issuing MMIO writes from the main CPU.

3. Policies for Dynamic Multi-Tenant Isolation: Performance isolation is provided by the central packet scheduler, which performs packet scheduling across the packets buffered for groups of parallel offloads that provide the same service. The scheduling algorithm determines both how chains competing for a service are isolated and how chains share packet buffers. Similar to prior work [68, 70], packet scheduling policies in PANIC are programmable. Further, PANIC improves upon prior work by also providing policy-aware packet dropping to enable cross-tenant memory isolation.

PANIC supports any scheduling algorithm that can be implemented by assigning an integer priority to a packet, and this includes a wide range of different policies, including strict priority, weighted fair queuing (WFQ), least slack time first (LSTF), and leaky bucket rate limiting [56, 68]. Although strict priorities lead to starvation, this is intended—if there is enough mission-critical traffic to consume all available resources, then it is acceptable for competing best-effort traffic to starve. If starvation is undesirable, it can be avoided by using WFQ and rate-limiting.

PANIC can support a hierarchical composition of different scheduling algorithms, *e.g.*, fair sharing across tenants with prioritization of flows for each tenant, although this comes with additional hardware costs. More complex scheduling algorithms are also possible in PANIC because priorities for

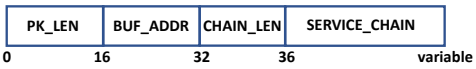


Figure 3: **PANIC Descriptor**

later services in a chain can be dynamically computed by an earlier chain stage. Similarly, each group of offloads that form a service can have its own custom scheduling algorithm, which is useful when different chains start with different offloads and then converge and share the same service.

4. Support for offloads with variable and below line-rate performance: The central packet scheduler supports offloads that have variable performance. Packets for slow offloads will be buffered at the central scheduler. As loads shift, packet buffers can be dynamically allocated to different offload groups. PANIC's hybrid push/pull load balancing scheme outlined in the example above load-balances packets across parallel offloads, ensuring precise load control, low tail latency, and minimal and efficient on-chip network use. Similar to the packet scheduler, the load balancer is also programmable.

5. High Performance: PANIC uses an on-chip network inspired by network routers to provide a high-performance interconnect between different offload tiles and the tiles for DMA and the Ethernet MACs. PANIC uses non-blocking high-bisection topologies like the crossbar making it possible to guarantee line-rate performance even if every offload in a chain sends/receives at line-rate over the on-chip network.

4 Design

This section discusses the design of the individual components of PANIC in more detail.

4.1 RMT Pipeline

The RMT pipeline in PANIC is used to provide programmable chaining and to look up scheduling metadata as part of providing programmable scheduling. The design of the RMT pipelines is borrowed from the design used in programmable switches [21, 67]. When a packet is received by the NIC, the RMT pipeline first parses incoming packets and then processes them with a sequence of match-action tables (MA tables). Each MA table matches specified packet header fields and then performs a sequence of actions to modify or forward the packet. Via these actions, the RMT pipeline 1) performs simple, line-rate packet processing (*e.g.*, IP checksum calculation) and 2) generates a PANIC descriptor for each packet that contains the appropriate chaining and scheduling metadata given the configuration that was programmed by the operator/user. Additionally, the RMT pipeline can maintain state on a per-traffic-class or per-flow basis if needed to support programmable scheduling or flow affinity.

Figure 3 shows the PANIC descriptor added by the RMT pipeline. It includes the packet length, the allocated buffer address, and the service chain for this packet, which is a list of services to send the packet to along with per-service metadata from the RMT. Because multiple compute units

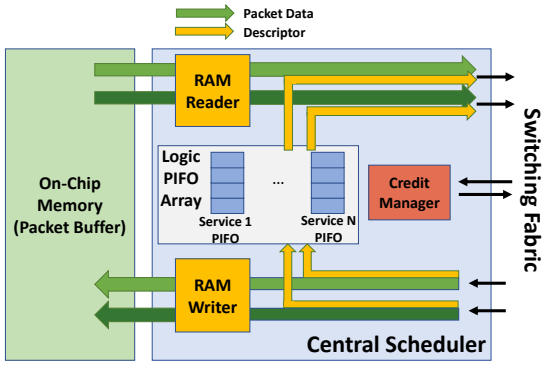


Figure 4: **Architecture of the multi-ported central scheduler**

may implement the same service (offload), this means that the RMT pipeline does not specify the exact unit a packet will be sent to in advance. This enables the scheduler to perform dynamic load balancing across multiple computation units implementing the same service in parallel.

In addition to specifying a list of services, the offload chain also contains metadata. For example, per-hop scheduling metadata like traffic class and priority allows a chain to have different priorities and weights across different services. Similarly, the descriptor may also contain service-specific metadata to allow the RMT pipeline to perform pre-processing to speed-up or simplify different compute units. Examples of this type of metadata include pointers to fields in a parsed packet and a pre-computed hash of an IP address.

The RMT pipeline directly connects to the switching fabric. To ensure low latency, the pipeline directly steers packets that are not processed by any service to the DMA engine.

4.2 High Performance Interconnect

To enable dynamic service chaining, PANIC use an on-chip interconnect network to switch packets, providing high-speed communication between the scheduler and on-NIC units.

Because it is necessary to forward packets between offloads at line-rate, it is important to build a high-performance network. PANIC utilizes a non-blocking, low latency, and high throughput crossbar interconnect network, which, for the scale of our design, still has a low area and power overhead. The crossbar can be configured to connect any input node to any output node with no intermediate stages, and each port runs at line-rate. As a result, every offload can simultaneously send and receive at line-rate, which enables line-rate dynamic chaining regardless of which offloads a chain uses.

Although economical in small configurations, crossbar interconnects unfortunately do not scale well with an increase in the number of cores. Most of these problems arise from the delay and area cost associated with long interior wires because the number of these wires increases significantly with the number of cores. Fortunately, with PANIC, we are able to choose between different interconnect topologies without having to change other parts of the design. If there is

a need to scale beyond the limits of a single crossbar, we can switch to a more scalable (but higher-latency) flattened butterfly topology [43].

4.3 Centralized Scheduler

The centralized scheduler buffers packets, schedules the order in which competing packets are processed by a service, and load-balances packets across the different compute units in a service. The scheduler architecture is shown in Figure 4. The scheduler uses a new hybrid scheduling algorithm to support low-latency chaining while avoiding load imbalance, and it uses a new hardware-based priority queue (*i.e.*, PIFO [68]) to schedule and drop packets according to a programmable inter-tenant isolation policy.

An overview of the operation of the central scheduler is as follows: When a packet and its descriptor arrive, the scheduler writes the packet data into high-speed on-chip memory and stores the packet descriptor into the appropriate logical PIFO queue given the next destination service of this packet. Each logical PIFO queue corresponds to a service and sorts buffered descriptors by rank, which enables the scheduler to drop packets according to the same policy as they are scheduled by dropping the lowest-rank packet currently enqueued for the service if needed. Then, whenever any of the parallel compute units for a service have available "credits" at the scheduler, the credit manager (Figure 4) chooses the compute unit with most credits, dequeues the head element of the corresponding logical PIFO queue, and sends the packet data and descriptor along with the packet data across the on-chip interconnect to the chosen unit.

4.3.1 Hybrid Push/Pull Scheduling and Load Balancing

When one service cannot achieve line-rate with a single unit, PANIC uses multiple parallel units to improve bandwidth. To support load-balancing across variable performance offloads, PANIC provides *load-aware* steering. Specifically, PANIC introduces a new hybrid pull/push scheduler and load balancer that overcomes the limitations of either push or pull scheduling to provide both precise request scheduling and high utilization.

Pull-based scheduling provides the most precise control over scheduling because decisions are delayed until each unit is able to perform work. However, pull-based scheduling can lead to utilization inefficiencies because each unit must wait for a pull to complete before it can start work on a new packet, and busy-polling can lead to increased interconnect load. In contrast, push-based scheduling can lead to load-imbalance and increased tail latencies when packets have variable processing times. In this scenario, it is not possible to know how much work is enqueued at each unit at the time that load-balancing decisions must be made.

The hybrid scheduler used in PANIC provides the best properties of both pull and push scheduling. In this scheduler, during periods of *high* load, the central scheduler uses *pull*-based load balancing to provide effective load balancing and packet scheduling. During *low* load, the scheduler *pushes* packets to all of the units in a service chain with low latency. To accomplish this, the scheduler uses credits to monitor the load at different units. Next, we describe the two modes of operation, pull and push, and the use of credits.

Credit Management: The credit manager tracks credits to measure load and dynamically switch between push-based and pull-based scheduling. The credit manager initially stores C credits for each compute unit. After sending a packet out, it decreases the credit number for that unit by one. When a compute unit is done processing a packet, it returns credit back to the credit manager.

The central scheduler operates in push mode as long as any of the parallel compute units in a service have credits available. If flow affinity is not needed, the scheduler steers packets to the unit which has the maximum number of pull credits to avoid load imbalance.

In contrast, if no unit has credit when a packet arrives, the scheduler buffers packets until credit is available. In this case, the central scheduler provides pull scheduling. Because the decision on which replica to use is made lazily, the number of packets queued at each unit will never exceed C.

By default, the number of initial credits C is set to two to avoid a stop-and-wait problem. However, it is possible to configure different credit numbers for each unit if needed. For example, ClickNP [47] uses a SHA1 engine that can process 64 packets in parallel, and PANIC can support this level of parallelism by giving 64 or more credits to the SHA1 engine.

Push-based Chaining: Push scheduling provides low-latency offload chaining. When a packet needs to traverse multiple offloads (*e.g.*, from A to B to C), the packet will be directly pushed to B when it finishes the computation in A rather than going back to the central scheduler. If B accepts the pushed packets, it will send a *cancel* message to the central scheduler to decrease its credit by one. In the case that there are multiple parallel units providing a service, the push destination is precalculated in the central scheduler. By pushing the packet directly to the next destination unit, PANIC reduces interconnect traversal latency and reduces on-chip network bandwidth demands. Furthermore, this reduces the load on the central scheduler as chain lengths grow.

Detour Routing: Push mode chaining may cause a packet to be pushed to a busy downstream unit that has no buffer space to accept packets. In this case, we use *detour* routing: when local buffer is occupied, the downstream unit redirects the packet back to the central scheduler, where it is buffered until it can be scheduled to another idle unit.

4.3.2 Packet Scheduling

To achieve priority scheduling and performance isolation, every packet stored in the packet buffer has its descriptor enqueued in a PIFO [68]. PANIC uses this PIFO-based priority

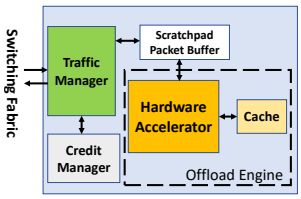


Figure 5: **Accelerator-Based Compute Unit Design**

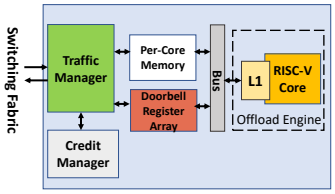


Figure 6: **Core-Based Compute Unit Design**

scheduler to provide both performance and buffer isolation across tenants.

Isolation Policy and Rank Computation: When a packet arrives, the central scheduler uses stateful atoms (ALUs) [67] to take metadata about the packet, look it up in the RMT pipeline, and compute an integer priority that is used to enqueue a descriptor for the packet into a PIFO block. This enables PANIC to provide multi-tenant isolation, ensuring traffic from high-priority tenants has low latency. Additionally, if multiple PIFO blocks are used inside the scheduler, it is possible to support hierarchical policies. Because the on-chip network ports are bidirectional, there is enough network throughput to forward incoming packets back out regardless of which logical queue the packets use.

Prioritized Dropping: PANIC's PIFO scheduler performs *prioritized packet dropping*. Specifically, PANIC ensures that when the NIC is overloaded, the lowest priority packets will be dropped. To achieve prioritized dropping, PANIC reuses the priority-sorted descriptor queue already used for scheduling in the PIFO. When the free space in the packet buffer for a logical PIFO is smaller than a threshold, the scheduler will remove the least-priority descriptor from the logical PIFO and drop this packet.

4.3.3 Performance Provisioning:

It is important to ensure that the central scheduler does not become a performance bottleneck and can forward packets across chains at full line-rate. To ensure that the scheduler has sufficient throughput, PANIC uses multiple ports to attach the scheduler to the on-chip network. Because the switching fabric is designed to forward between arbitrary ports at line-rate, increasing the number of ports used by the scheduler is sufficient to scale the network performance of the scheduler.

The speed of the PIFO block used to schedule packets can also become a performance bottleneck. The PIFO block that we use can schedule one packet per cycle, *e.g.*, 1000Mpps when operating at a 1GHz frequency when implemented in an ASIC design. Although this is sufficient to schedule packets in both transmit and receive directions in our current design, in the case that this number is greater than the performance of a single PIFO block, multiple parallel PIFO blocks need to be used to scale up performance.

Provisioning for Compute Unit Performance: The design of the on-chip network and the scheduler can also ensure that offloads may be fully utilized despite complications from chaining. Specifically, when an offload O1 in a chain (Chain A=O1–O2–O3) runs at a slower rate than the rest of the offloads (O2–O3), it will become a bottleneck and cause O2–O3 to be not fully utilized. However, this does not lead to resource stranding. A second chain B that does not use O1 can still use O2–O3 and benefit from the remaining capacity of these offloads. The scheduler can ensure that the contending chains fairly share capacity.

4.4 Compute Unit

To support offload variety, PANIC utilizes compute units to attach offloads to the switching fabric. These compute units are self-contained, meaning that hardware offloads can be designed without needing to understand the packet switching fabric and without having to issue pull requests to the hybrid scheduler. The interfacing with the switching fabric is handled by the traffic manager (Figures 5 and 6). This both reduces offload complexity by avoiding duplicating packet processing functionality and reduces packet processing latency by avoiding incurring the overheads of processing a packet with a CPU.

An offload engine in PANIC can either be a hardware accelerator or a core, and Figure 5 and Figure 6 presents the design of an ASIC accelerator-based and CPU-based compute unit in PANIC, respectively. In both of these designs, the offload functionality is encapsulated as an offload engine. Both perform packet processing by reading a packet once it arrives from the network and has been placed in a local scratchpad buffer. The traffic manager is responsible for communicating with the switching fabric. This component includes logic for sending and receiving packets as well as logic for updating PANIC descriptors as needed for push-based chaining. The compute unit's local credit manager interfaces with the central scheduler and is responsible for returning credits (when a packet's processing is done) and sending cancel messages that decrement credits (when accepting a pushed packet).

The primary difference between an accelerator-based design and CPU-based design is that there is additional logic in the CPU-based design that is used to interface with the memory subsystem of the embedded CPU core. As Figure 6 shows, the compute unit utilizes memory-mapped I/O (MMIO) to connect an embedded CPU core as follows: 1) The traffic manager (TM) writes network data directly to a pinned region of the per-core memory. 2) Then the TM writes to an input doorbell register to notify the core that data is ready. 3) After the core finishes processing, it writes data back to the pinned memory region if needed and then writes to an output doorbell register that is used to notify the TM of a new outgoing packet. To make sure the packet data is written back to memory, the core needs to flush the cache lines for the pinned memory region. 4) The TM collects the output data and sends it back to the switching fabric.

5 ASIC Analysis

We expect an eventual implementation of PANIC to use an application-specific integrated circuit (ASIC), although we have also prototyped PANIC in the context of an FPGA platform for expediency. This is because relative to an FPGA, an ASIC provides higher performance, consumes less power and area, and is cheaper when produced in large volumes [44]. While an ASIC implementation is beyond the scope of this work, in this section we briefly discuss the feasibility of implementing different components of PANIC in an ASIC.

RMT: The implementation of an RMT pipeline in ASIC has already been proven feasible [21]. The Barefoot Tofino chip [16] is a concrete realization of the RMT architecture.

PIFO: PANIC uses a hardware priority queue to provide programmable scheduling. Our current design was borrowed from the ASIC-based flow scheduler design of the PIFO paper [68]. While this design is conceptually simple and easy to implement because it maintains a priority queue as a sorted array, it is less scalable relative to other priority queue designs, e.g., PIEO [64], which uses two levels of memory or pHeap [20], which uses a pipelined heap. For better scalability, we can replace our current design with such scalable hardware designs of priority queues at the expense of greater design and verification effort.

Interconnect: One of the biggest potential scalability limitations of a PANIC implementation is the on-chip switching fabric. While crossbar interconnects are conceptually simple, the sheer number of wires in a crossbar might become a physical design and routing bottleneck, causing both an increase in area as well as an inability to meet timing beyond a certain scale. Fortunately, prior work has already demonstrated that it is feasible to build crossbars on an ASIC that are larger than are needed in PANIC. Specifically, Chole *et al.* built a 32 * 32 crossbar with a bit width of 640 bits [28, Appendix C] at a 1 GHz clock rate. As another data point, the Swizzle-Switch supports a 64 * 64 crossbar with a bit width of 128 bits using specialized circuit design techniques at 559 MHz in a relatively old 45 nm technology node [63]. For comparison, to provide 32 compute units each a 128 Gbps connection to the switching interconnect, PANIC only needs a 32 * 32 crossbar with a bit width of 128 bits and 256 bits at 1 GHz and 500 MHz clock frequencies, respectively.

At the same time, we anticipate a crossbar becoming no longer viable at some point as the number of offloads continues to increase. At this point, we anticipate switching to other more scalable topologies such as a flattened butterfly at the cost of increased latency and reduced bisection bandwidth.

Compute Units: The PANIC offload engine can be a hardware accelerator or a CPU core. There are several ASIC-based RISC-V implementations which can used as a CPU core for the offload engine [25, 26, 8]. Several functions important to networking, such as compression, encryption, and checksums are available as hardware accelerators, which can be reused for PANIC. Our own AES and SHA implementations (Section 6) are based on open-source hardware accelerator blocks [1, 7].

6 FPGA Prototype

We implement an FPGA prototype of PANIC in ∼6K lines of Verilog code, including a single-stage RMT pipeline, the central scheduler, the crossbar, the packet buffer, and compute units. Also, we built a NIC driver, DMA Engine, Ethernet MAC, and physical layer (PHY) using Corundum [33]. Although the PANIC architecture supports both the sending path and receiving path, in our current implementation, we mainly focus on the receive path.

RMT pipeline: We implemented a single-stage RMT pipeline in our FPGA prototype. We configure the RMT pipeline in our prototype by programming the FPGA. The RMT pipeline maintains a flow table, in which each flow is assigned an offload chain and scheduling metadata. In the match stage, the RMT module matches the flow table with the IP address fields and port fields in the packet header. In the action stage, the RMT module calculates scheduling metadata and generates the PANIC descriptor (Figure 3). The frequency of the RMT pipeline is 250 MHz.

FPGA-based Crossbar: We have implemented an 8 * 8 fully connected crossbar in our FPGA prototype. The frequency for this crossbar is 250 MHz, and the data width is 512 bits. This leads to a per-port throughput of 128 Gbps.

Central Scheduler and Packet Buffer: The architecture of the scheduler is shown in Figure 4. The scheduler is connected with two crossbar ports to ensure a sufficiently high throughput connection to the on-chip network. In our implementation, the PIFO block runs at 125 MHz frequency with a queue size of 256 packets; all other components in the scheduler run at 250 MHz, with a 512 bit data width, and we add a cross-domain clocking module between other components and the PIFO. We use lower frequency for the PIFO because it suffers from a scalability issue when implemented on the FPGA (we explain this further in Section 7.5). The packet buffer is implemented with dual-channel high-speed BRAM, where each BRAM channel supports concurrent reads and writes. The packet buffer size in our implementation is 256 KB with a 512 bit data width and a 250 MHz frequency. For ease of implementation, our current prototype uses a separate physical PIFO for each logical PIFO at the cost of increasing the relative resource consumption of PIFOs.

Compute Units: As Figure 5 shows, our implementation of a compute unit includes a traffic manager, a credit manager, and a scratchpad packet buffer. We choose the AXI4-Stream interface [2] as the common interface between the offload engine and scratchpad buffer. We have included two types of accelerator-based offload engines in our FPGA prototype. One is the AES-256-CTR encryption engine [1], and the other is the SHA-3-512 hash engine [7].

We have also implemented a RISC-V core engine based on the open-source CPU core generator [9]. Figure 6 shows how the RISC-V core is connected to PANIC's traffic manager, credit manager, and per-core memory. The RV32I RISC-V core we use has a five-stage pipeline with a single level of cache. The data cache and instruction cache are 2 KB each, and the local memory size is 32 KB. The frequency for this CPU is 250 MHz, and the per-core memory data width is 512 bits.

7 Evaluation

This section evaluates our FPGA prototype to show that it meets the design requirements put forth in Section 2.1. In Section 7.2, we use microbenchmarks to show that PANIC achieves high throughput and low latency under different offload chaining models. In Section 7.3, we compare PANIC's performance with a pipeline-of-offloads NIC. Section 7.4 measures the I/O performance of a RISC-V core in PANIC, and Section 7.5 measures the hardware resource usage of our FPGA prototype. Finally, in Section 7.6, we implement different offload engines, and test PANIC end-to-end; these results demonstrate that PANIC can isolate and prioritize traffic efficiently under multi-tenant settings.

7.1 Testbed and methodology

For our microbenchmarks, we implemented our FPGA prototype in the ADM-PCIE-9V3 network accelerator [12], which contains a Xilinx Virtex UltraScale Plus VU3P-2 FPGA. For this evaluation, we also implemented a delay unit, a packet generator, and a packet capture agent on the FPGA. The delay unit emulates various compute units by delaying packets in a programmable fashion, which allows us to flexibly control per-packet service time and chaining models. This enables us to run microbenchmarks that systematically study PANIC's performance limits. The packet generator generates traffic of various packet sizes at different rates. The packet capture agent receives packets and calculates different flows' throughput and latency. We calculate packet processing latency by embedding a send timestamp in every generated packet.

For our end-to-end experiments, we evaluate PANIC in a small testbed of 2 Dell PowerEdge R640 servers. One server is equipped with a Mellanox Connectx-5 NIC, and the other server is equipped with the ADM-PCIE-9V3 network accelerator carrying our PANIC prototype. The Mellanox NIC and ADM-PCIE-9V3 card are directly connected. We program the VU3P-2 FPGA on the network accelerator using our Verilog implementation of PANIC. We use DPDK to send customized network packets and use PANIC to receive packets and run offloads. Because of a performance bottleneck in the kernel-based FPGA NIC driver [33], we use the packet capture agent on the FPGA to report PANIC's receive throughput instead of capturing packets on the host machine.

7.2 PANIC System Microbenchmarks

We microbenchmark PANIC's performance using the different chaining models shown in Figure 7. Our results demonstrate that PANIC can both achieve high throughput and low latency for various common offload chaining models.

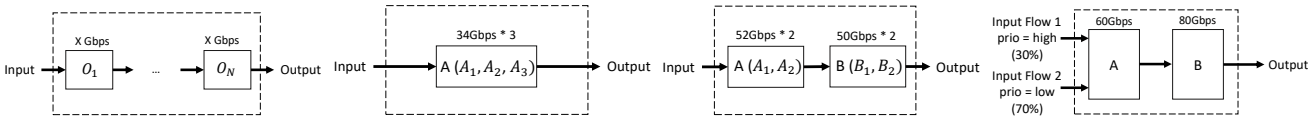
Model 1 ("Pipelined Chain"): In model 1 (Figure 7a), we attach N delay units in sequence. Each unit emulates a different service, and all of them process packets at X Gbps with fixed delay. We then configure a service chain that sends packets through all N units in numerical order.

First, we measure the throughput and latency overhead of PANIC when N = 3 and X = 100. Figure 8a shows the overall throughput under different packet sizes. With MTU-sized packets, PANIC can schedule packets at full line-rate. When the initial number of credits in PANIC is small, we see a nonlinear performance downgrade with small packets. This is because throughput for small packets is bounded by the scheduling round trip time in PANIC, which is 14 clock cycles. If we increase the initial credit number for each unit, we see a performance increase for small packets. When the credit number is greater than 8, the small packet performance is no longer bounded by the scheduling round trip, instead, it is bounded by the small packet performance of our delay unit. These results also demonstrate the benefits of PANIC's flexibility in per-unit credit allocation. Setting different credit numbers for each unit can improve performance.

Next, Figure 8b shows the latency of different packet sizes in the same experiment. In this pipelined chain, packets can be scheduled through three units within 0.5 microseconds. This low latency performance also arises from PANIC's push scheduling which helps PANIC avoid extra packet traversals between units and the scheduler.

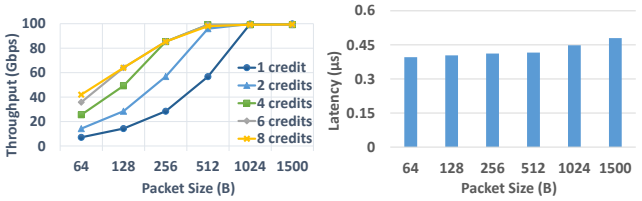
Next, Figure 9 shows PANIC's throughput as a function of the chain length when push scheduling is disabled. Without push scheduling, the packet needs to go back to the scheduler at every hop, and the total traffic that goes into the scheduler is the ingress traffic from the RMT pipeline plus the detoured traffic from units, which is: $T_{total} = T_{RMT} + T_{detour} = T_{RMT} + (N-1) * X = N * X$. As Figure 9 shows, when T_{total} exceeds the dual-ported scheduler's maximum bandwidth (250 MHz * 512 bits * 2 ports = 256 Gbps), the chaining throughput downgrades. For example, when N = 3, X = 70, $T_{total} = 210\ Gbps < 256\ Gbps$, thus PANIC can schedule this chain at full speed even when push scheduling is disabled. When N = 3, X = 90, $T_{total} = 270\ Gbps > 256\ Gbps$, the chaining throughput downgrades since the scheduler bandwidth becomes the bottleneck.

Model 2 ("Parallelized Chain"): In model 2 (Figure 7b), we attach three delay units running in parallel. These three units run the same service, and each unit has an average 34 Gbps throughput but *variable latency*. PANIC load-balances packets across these units. Figure 8c shows the throughput under different packet sizes and service time variance (the service time follows uniform distribution). We see that even when

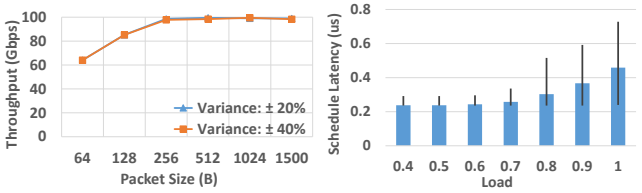


(a) **Chaining Model 1** (b) **Chaining Model 2** (c) **Chaining Model 3** (d) **Chaining Model 4**

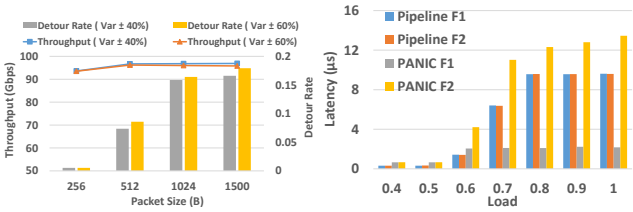
Figure 7: **The different chaining models used in experiments.**



(a) **Model 1 throughput** (b) **Model 1 latency**



(c) **Model 2 throughput** (d) **Model 2 latency**



(e) **Model 3 detour rate and throughput.** (f) **Different flow's latency in the pipeline design and PANIC**

Figure 8: **PANIC performance under different chaining models**

the processing latency variance increases from 20% to 40%, PANIC can still efficiently load-balance packets between the parallel units without impacting throughput. In this experiment, small packet performance is better than model 1 because model 2 has multiple units running in parallel. Overall throughput is no longer bounded by the delay of a single unit.

Figure 8d shows PANIC scheduling latency under different loads with MTU sized packets and 40% service time variance. The error bars in this figure represent 5%-ile and 99%-ile latency. Scheduling latency reveals how long the incoming packets wait before being processed by an idle unit; we calculate it by subtracting the unit processing time from the total latency. When the NIC load is much smaller than 1, scheduling tail latency grows slowly, and is under 0.4 μs. When the load approaches 1, queueing occurs in the packet buffer, which causes tail latency to grow, but it still stays < 0.8 μs. This shows that our credit-based scheme keeps latency low even at high load, and most latency is due to queueing.

Model 3 ("Hybrid Chain"): Model 3 (Figure 7c) is a hybrid chaining model where packets are not only load-balanced between parallel units but also go through multiple services.

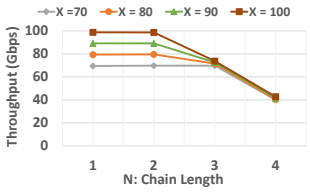


Figure 9: **Model 1 throughput when push is disabled.**

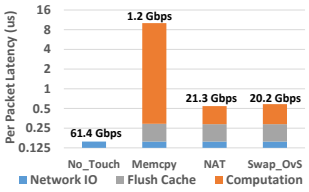


Figure 10: **Performance of a single RISC-V core with MTU-sized packets.**

	Throughput	Total
Flow 1 (Pipeline Design)	18.7 Gbps	60.6 Gbps
Flow 2 (Pipeline Design)	41.9 Gbps	
Flow 1 (PANIC)	30.7 Gbps	59.8 Gbps
Flow 2 (PANIC)	29.1 Gbps	

Table 3: **Throughput of the pipeline design and PANIC.**

Packets need to be processed first by service A and then by service B, and both services have multiple parallel compute units. Each compute unit for service A has an average throughput of 52 Gbps, while each compute unit for service B has an average throughput of 50 Gbps. Compute units for both service A and B have variable latency.

Figure 8e shows the throughput and detour rate in this hybrid model. When the packet size is bigger than 256 bytes, the detour rate is high. This is because the downstream B units have lower throughput than the upstream A units. As a result, the B units are always busy because they are the throughput bottleneck in this system. Busy units are likely to have no space to accept pushed packets: if A unit tries to push packets to a busy downstream B unit, then the B units will more often than not detour the pushed packets back to the central scheduler.

Figure 8e also shows that detour routing does not degrade throughput. This is because the maximum bandwidth of our dual-ported scheduler is 256 Gbps, and in this hybrid model, the ingress traffic from the RMT pipeline will take up 100 Gbps bandwidth in the scheduler, thus there is more than 100 Gbps bandwidth left for the detoured traffic.

However, detour routing can increase packet latency. In order to mitigate this, the central scheduler increases the priority for each detoured packet, to help them get rescheduled first. Thus, the latency incurred by detoured packets is the RTT between the compute unit and the scheduler, which is < 0.5 μs.

7.3 Comparison with the Pipeline Design

To demonstrate that PANIC handles multi-tenant isolation and below line-rate offloads better than state-of-the-art, we build and compared against the pipeline-of-offloads NIC as our baseline. We choose model 4 (Figure 7c) as the offload chain for this comparison. The difference between model 4 and model 1 is that the delay unit emulates a below-line-rate offload in model 4. We assume two flows are competing: Flow 1 has a higher priority, and takes up 30% of the total traffic. Flow 2 has a lower priority, and takes up 70% of the total traffic.

We implemented a pipeline-of-offloads NIC in the ADM-PCIE-9V3 network accelerator. In this NIC, all incoming packets are first buffered in a FIFO (First-In-First-Out) queue before entering unit A. Unit A and unit B are directly connected using the AXI4-stream interface [2]. We configured the pipeline-of-offloads NIC and PANIC to have the same buffer size (64 KB), same frequency (250 MHz), and same bit-width (512 bits).

Figure 8f presents a comparison of the latency of both the pipeline design and PANIC in this experiment. When the NIC load is low, Unit A is not the bottleneck, and both NICs have low latency. The pipeline design has slightly better latency since units are directly connected in it, while scheduling packets in PANIC has some overhead. When load increases, Unit A becomes the bottleneck, and both NICs start to buffer and drop packets. With high load, Flows 1 and 2 have the same latency in the pipeline design, since packets are scheduled in First-Come-First-Served order and can experience HOL blocking. In PANIC, the high priority packets have fixed low latency due to the central scheduler sorting buffered packet descriptors and serving high priority packets first.

We compare the throughput between the pipeline design and PANIC in Table 3. The total throughput is bounded by Unit A (60 Gbps). In the pipeline design, the low priority flow 2 has a higher throughput than flow 1, because the high-volume flow 2 steals on-chip bandwidth by taking up most of the on-chip buffer. PANIC preferably allocates buffer to high priority packets and always drops the lowest priority ones. Thus flow 1 can always achieve full throughput in PANIC.

Overall, PANIC achieves good isolation: 1) PANIC achieves comparable throughput and latency with the pipeline design when there is no HOL blocking. 2) When HOL blocking occurs, PANIC ensures that the high priority flows have a fixed low latency. 3) PANIC allocates bandwidth according to a flow's priority.

7.4 RISC-V Core Performance

To investigate the I/O overhead of using an embedded NIC core to send/receive network packets from PANIC, we performed experiments with a single RISC-V CPU core as the only offload engine in a chain. We measure the system throughput and per-packet latency using four example C programs:

No-Touch: After receiving the packet from PANIC, this program will send the packet back to PANIC immediately. This program does not make any changes to the packet data.

Memcpy: This program will copy the received packet to another memory address and then send the copied packet back to PANIC.

NAT: The Network Address Translation (NAT) program uses the embedded CPU core to lookup a <Translated IP, Port>pair for a given 5-tuple, and then replace the IP address and port header fields using the lookup results. The lookup table is stored in the local memory inside the offload engine. The RMT pipeline will pre-calculate the hash value for each packet, and the hash value is stored as per-service metadata in the PANIC descriptor. Thus the CPU core can directly read the pre-calculated hash value from the descriptor.

Swap OvS: This program swaps the Ethernet and IP source and destination addresses.

Figure 10 shows the RISC-V core throughput and per-packet latency with MTU sized packets. We breakdown the latency number into three different parts: 1) Network I/O: the time that is spent on pulling/writing the input/output doorbell register, 2) Flush Cache: the time spent on flushing the L1 cache, 3) Computation: the time spent on computation, including the data exchange time between the L1 cache and the per-core memory. The results of this experiment show that the overhead of the NIC to CPU core interface is low, and, for those low-throughput applications, the I/O time introduced by PANIC is negligible.

For example, the throughput of the No-Touch program is 61.4 Gbps, and all the time is spent in network I/O. The throughput of the NAT and Swap OvS programs is 21.3 Gbps and 20.2 Gbps, respectively. $\sim$20% of the time is spent in flushing the cache, $\sim$27% in network I/O, and $\sim$50% in computation. Cache flushing is costly in our current prototype: to synchronize the data between the cache and memory, the whole L1 cache is flushed before processing the next packet. If needed, this performance could be improved by modifying the CPU to support an instruction that only flushes the cache lines for the pinned memory region used by the packet.

The throughput of Memcpy is only 1.2 Gbps, and 97% of the time is spent in computation. This is due to the limitations of the performance of the FPGA based RISC-V core. With a faster core and a higher clock frequency, this performance can be improved.

7.5 Hardware Resource Usage

Our UltraScale VU3P-2 FPGA has 3 MB BRAM, and 394k LUTs in total. Table 4 shows different components' resource usage under different settings. In our end-to-end experiments (Section 7.6), the crossbar has 8 ports, total queue size in the PIFO array is 256 packets, and packet buffer size is 256 KB. Under this setting, we find that PANIC's design will only

Module	Setting	LUTs(%)	BRAM(%)
Crossbar	8 ports	5.5	0.00
	16 ports	13.64	0.00
Scheduler (PIFO)	PIFO = 256	5.18 (4.9)	0.07 (0.01)
	PIFO = 512	9.95 (9.42)	0.07 (0.01)
Packet Buffer	256 KB	0.16	8.94
Simple RMT	/	0.27	0.00

Table 4: **FPGA resource usage for different components.**

cost 11.27% logic area (LUTs) in our middle-end FPGA. Total BRAM usage is 8.94% due to the limited BRAM in our FPGA.

The crossbar and PIFO occupy most of the on-chip logic resources in PANIC. When the crossbar uses 8 ports, it costs around 5.5% logic area, and for 16 ports, the logic area cost is 13.64%. When the total PIFO size is 256, it will cost 4.9% logic area, and when the size is 512, it will cost 9.42%. PIFO suffers from high logic area cost because its hardware design does not access BRAM at all; it only uses the logic unit to compare and shift elements. This design causes PIFO to be less scalable in the FPGA since it cannot benefit from the FPGA's memory hierarchy to efficiently distribute storage and processing across SRAM and LUTs. Recent advancements [64] can be used to address this (Section 5). Overall, we find that PANIC can easily fit on any middle-end FPGA without utilization or timing issues.

7.6 End-to-End Performance

In this section, we measure PANIC's end-to-end performance in our cluster. Because of the performance bottleneck of the kernel-based FPGA NIC driver, we use hardware counters to measure PANIC's receiving throughput. We implement two FPGA-based offload engines in PANIC: a SHA-3-512 engine and an AES-256 engine. Our end-to-end experiment demonstrates that: 1) PANIC can schedule network traffic at full line-rate, 2) PANIC can precisely prioritize traffic when different flows are competing for computation resources at the offloads, and 3) PANIC can support different isolation policies, including strict priority and weighted fair queueing.

The AES-256-CTR encryption engine [29] encrypts input plain text into ciphered text or decrypts ciphered text to yield plain text. The fully pipelined AES-256 engine can accept 128-bit input per cycle, and it can run at 250 MHz frequency with 32 Gbps throughput. The SHA-3-512 engine [19] performs SHA-3, a newest cryptographic hash which uses permutation as a building block [18]. The FPGA-based SHA-3-512 engine that we use runs at 150 MHz with 6 Gbps throughput.

Since the throughput of a single SHA engine is low, we put 4 SHA engines into a single hash unit, and set the initial credit number for the hash unit to 4. Thus, the hash unit can use 4 SHA engines to process these packets in parallel. We connect two decryption units and two hash units with PANIC. Thus, the bandwidth of hash computation is (6 * 4) * 2 = 24 * 2 = 48 Gbps, and that for decryption is 32 * 2 = 64 Gbps.

In our experiment, we assume there are two types of traffic

Traffic	IPSec	Video	Background
Drop Rate	0%	33.1%	0%

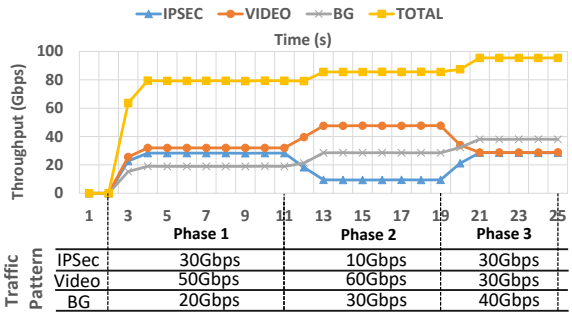
Table 5: **Packet dropping rate in phase 1 in Figure 11a.**

competing for the computation resource in PANIC. One is high-volume multimedia traffic, which uses AES offload to decrypt video streams. Another is low-volume IPSec traffic, which first uses SHA to ensure the integrity of the data and then uses AES to decrypt IP payload. The IPSec traffic has higher priority than video stream traffic, and each of these traffic streams contains multiple flows. Also, we add background traffic that does not need to be processed by any compute unit. The offload chains are shown in Figure 12.

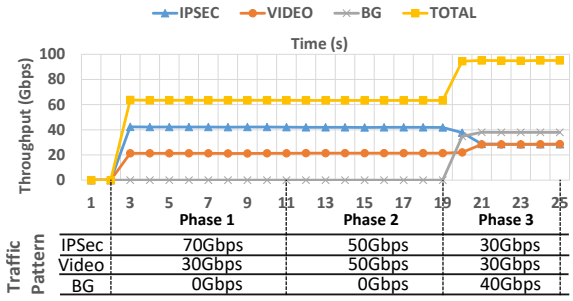
In the first experiment, we use the strict priority policy, which means all the IPSec packets have higher priority than the video packets. Figure 11a shows different traffic's receiving throughput under different traffic patterns. In phase 1, the sending throughput is 30 Gbps for IPSec and is 50 Gbps for video. We can see the receiving throughput for IPSec is 30 Gbps, which is the same as the sending throughput. The receiving throughput for the video stream is only 34 Gbps. This is because IPSec and video stream share the AES offload. However, the available peak bandwidth for the AES offload is only 64 Gbps. Thus, PANIC will first satisfy the high priority IPSec traffic requirement, which only leaves 34 Gbps (64 - 30) of bandwidth for the video stream. Table 5 shows the dropping rate under phase 1. Due to prioritized packet dropping, PANIC only drops low priority video packets. Overall, when a below-line-rate offload becomes the bottleneck, PANIC always first satisfies high priority traffic's bandwidth demands.

In phase 2, the DPDK sender switches to the next traffic pattern, in which the IPSec traffic sending rate drops to 10 Gbps, and video traffic sending rate grows to 50 Gbps. Since the IPSec sending rate drops, the video stream can get more bandwidth, but it will still lose some bandwidth and experience packet drops because of the AES bottleneck. In phase 3, the DPDK sender switches to the last pattern, in which the AES offload is no longer the bottleneck; no packet drops occur, and the total throughput can reach 100 Gbps. Another noteworthy aspect in Figure 11a is that no matter what computation happens, background traffic performance is unaffected.

In the second experiment, we use a weighted fair queueing (WFQ) scheduling policy where the AES offload's capacity is divided across IPSec traffic and video traffic in 2 : 1 ratio. Figure 11b shows the throughput of the different traffic types under the WFQ policy for different traffic patterns. In phase 1, the sending throughput for IPSec is 70 Gbps, and for the video stream is 30 Gbps. We can see the receiving throughput for IPSec is exactly twice of the video stream, and the total throughput is 64 Gbps. If the IPSec sending rate drops to 50 Gbps (phase2), the receiving throughput remains unchanged. This result proves PANIC can shape the traffic



(a) **Strict Priority Policy**



(b) **Weighted Fair Queueing Policy**

Figure 11: **Receiving throughput with different traffic patterns. Figure a uses strict priority policy: all the IPSec packets have higher priority than the video packets. Figure b uses WFQ policy: the offload capacity is divided across IPSec traffic and video traffic in the ratio 2:1. The table in Figure a and Figure b shows how the sending traffic pattern changes with time.**

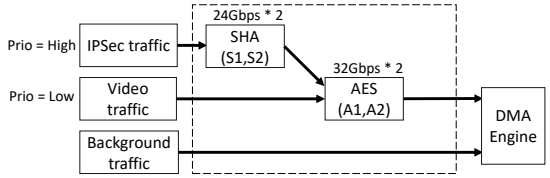


Figure 12: **Offload chains; end-to-end experiment**

precisely, regardless of the sending rate.

In phases 1 and 2, the scheduler switches to pull-based scheduling since the AES offload is always congested. As a result, the egress packet of the SHA offload goes directly back to the scheduler instead of the congested AES offload. The scheduler then shapes the video traffic and the detoured IPSec traffic into a desired rate using WFQ.

In phase 3, the AES offload is no longer the bottleneck. Thus, the central scheduler operates in push mode: the egress packet of the SHA offload can bypass the scheduler and be directly pushed to the AES offload. As shown in Figure 11b, both IPSec and video's receiving throughput can reach the sending rate, which is 30 Gbps. Overall, this shows that PANIC can shape the traffic precisely with the WFQ policy.

8 Related Work

Several projects introduce new offloads that utilize programmable NICs and new frameworks for deploying these offloads [13, 48, 59, 42, 32, 37, 49, 65, 46, 62, 47, 40, 30, 36, 70, 69, 35, 55, 45]. PANIC is orthogonal to these projects.

The Pensando DSC-100 NIC [58] is similar to PANIC in that it has an RMT pipeline and supports both hardware and software offloads. However, the DSC-100 requires cores to achieve offload chaining instead of a hardware scheduler.

The Fungible Data Processing Unit (DPU) is a NIC design that was recently announced in August 2020 [3]. Based on publicly available documents [4, 5], it has a hardware architecture that shares a few similarities with PANIC (e.g., processing cores, accelerators, a hardware work scheduler, and a customized on-chip network). A head-to-head comparison of PANIC to the Fungible DPU would be an interesting avenue for future work once the DPU is generally available.

PANIC is also similar to FairNIC [34], which improves fairness between competing applications running on a commodity manycore NIC. However, PANIC provides features not possible in FairNIC like chaining without involving a CPU. Further, FairNIC helps motivate the need for PANIC detailing the non-trivial costs of isolation on manycore NICs. Adopting PANIC's scheduler and non-blocking crossbar interconnect can solve these fundamental problems with manycore NICs.

9 Conclusions

Programmable NICs are an enticing option for bridging the widening gap between network speeds and CPU performance in multi-tenant datacenters. But, existing designs fall short of supporting the rich and high-performance offload needs of co-resident applications. To address this need, we presented the design, implementation, and evaluation of PANIC, a new programmable NIC. PANIC synthesizes a variety of high-performance hardware blocks and data structures within a simple architecture, and couples them with novel scheduling and load balancing algorithms. Our analysis shows that PANIC is amenable to an ASIC design. We also built a 100G PANIC prototype on an FPGA, and conducted detailed experiments that show that PANIC can isolate tenants effectively, ensure high throughput and low latency, and support flexible and dynamic chaining.

Acknowledgements: We thank our shepherd, Costin Raiciu, and the anonymous OSDI reviewers for their feedback that significantly improved the paper. We thank Suvinay Subramanian and Tushar Krishna for discussions on crossbar designs and Tao Wang for his assistance with the artifact evaluation. Brent E. Stephens and Kiran Patel were funded by a Google Faculty Research Award and NSF Award CNS-1942686. Aditya Akella and Jiaxin Lin were funded by NSF Awards CNS-1717039 and CNS-1838733 and a gift from Google.

References

[1] AES Hardware Accelerator. https://opencores.org/projects/tiny_aes.

[2] Axi reference guide. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[3] Fungible DPU: A New Class of Microprocessor Powering Next Generation Data Center Infrastructure. https://www.fungible.com/news/fungible-dpu-a-new-class-of-microprocessor-powering-next-generation-data-center-infrastructure/.

[4] Fungible F1 Data Processing Unit. https://www.fungible.com/wp-content/uploads/2020/08/PB0028.01.02020820-Fungible-F1-Data-Processing-Unit.pdf.

[5] Fungible S1 Data Processing Unit. https://www.fungible.com/wp-content/uploads/2020/08/PB0029.00.02020811-Fungible-S1-Data-Processing-Unit.pdf.

[6] Intel ethernet switch fm10000 datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-multi-host-controller-fm10000-family-datasheet.pdf.

[7] SHA-3 Hardware Accelerator. https://opencores.org/projects/sha3.

[8] Silicon at the speed of software. https://www.sifive.com. Accessed: 2020-05-25.

[9] Vexriscv. https://spinalhdl.github.io/SpinalDoc-RTD/SpinalHDL/Libraries/vexriscv.html.

[10] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[11] ACCOLADE TECHNOLOGY. Accolade ANIC. https://accoladetechnology.com/whitepapers/ANIC-Features-Overview.pdf.

[12] ALPHA DATA. ADM-PCIE-9V3 - High-Performance Network Accelerator. https://www.alpha-data.com/pdfs/adm-pcie-9v3.pdf.

[13] ARASHLOO, M. T., LAVROV, A., GHOBADI, M., REXFORD, J., WALKER, D., AND WENTZLAFF, D. Enabling programmable transport protocols in high-speed NICs. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2020).

[14] ASANOVIĆ, K., AVIZIENIS, R., BACHRACH, J., BEAMER, S., BIANCOLIN, D., CELIO, C., COOK, H., DABBELT, D., HAUSER, J., IZRAELEVITZ, A., KARANDIKAR, S., KELLER, B., KIM, D., KOENIG, J., LEE, Y., LOVE, E., MAAS, M., MAGYAR, A., MAO, H., MORETO, M., OU, A., PATTERSON, D. A., RICHARDS, B., SCHMIDT, C., TWIGG, S., VO, H., AND WATERMAN, A. The rocket chip generator. Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[15] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2011).

[16] BAREFOOT. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino, 2017.

[17] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[18] BERTONI, G., DAEMEN, J., PEETERS, M., AND ASSCHE, G. The keccak reference, version 3.0. *NIST SHA3 Submission Document (January 2011)* (2011).

[19] BERTONI, G., DAEMEN, J., PEETERS, M., AND VAN ASSCHE, G. Keccak sponge function family main document. *Submission to NIST (Round 2)* (2009).

[20] BHAGWAN, R., AND LIN, B. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)* (2000).

[21] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F. A., AND HOROWITZ, M. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2013).

[22] BROADCOM. Stingray SmartNIC Adapters and IC. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[23] CAVIUM CORPORATION. Cavium CN63XX-NIC10E. http://cavium.com/Intelligent_Network_Adapters_CN63XX_NIC10E.html.

[24] CAVIUM CORPORATION. Cavium LiquidIO. http://www.cavium.com/pdfFiles/LiquidIO_Server_Adapters_PB_Rev1.2.pdf.

[25] CELIO, C., CHIU, P.-F., ASANOVIĆ, K., NIKOLIĆ, B., AND PATTERSON, D. Broom: an open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos. *IEEE Micro* (2019).

[26] CELIO, C., CHIU, P.-F., NIKOLIC, B., PATTERSON, D., AND ASANOVIC, K. Boom v2, 2017.

[27] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., ORDA, A., AND EDSALL, T. dRMT: Disaggregated programmable switching. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2017).

[28] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., ORDA, A., AND EDSALL, T. dRMT: Disaggregated programmable switching - extended version. https://cs.nyu.edu/~anirudh/sigcomm17_drmt_extended.pdf, 2017.

[29] DAEMEN, J., AND RIJMEN, V. *The design of Rijndael: AES-the advanced encryption standard*. 2013.

[30] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[31] EXABLAZE. ExaNIC V5P. https://exablaze.com//exanic-v5p.

[32] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. Azure accelerated networking: SmartNICs in the public cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2018).

[33] FORENCICH, A., SNOEREN, A. C., PORTER, G., AND PAPEN, G. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines* (2020).

[34] GRANT, S., YELAM, A., BLAND, M., AND SNOEREN, A. C. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2020).

[35] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over commodity Ethernet at scale. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2016).

[36] HUMPHRIES, J. T., KAFFES, K., MAZIÈRES, D., AND KOZYRAKIS, C. Mind the Gap: A case for informed request scheduling at the NIC. In *ACM Workshop on Hot Topics in Networks (ACM HotNets)* (2019).

[37] IBANEZ, S., SHAHBAZ, M., AND MCKEOWN, N. The case for a network fast path to the CPU. In *ACM Workshop on Hot Topics in Networks (ACM HotNets)* (2019).

[38] INTEL. Intel 82599 10 GbE controller datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[39] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2015).

[40] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Datacenter rpcs can be general and fast. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[41] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., AND PORTER, G. Bullet trains: A study of NIC burst behavior at microsecond timescales. In *Conference on Emerging Networking Experiments and Technologies CoNEXT* (2013).

[42] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with FlexNIC. In *ASPLOS* (2016).

[43] KIM, J., DALLY, W. J., AND ABTS, D. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual International Symposium on Computer Architecture (ISCA)* (2007).

[44] KUON, I., AND ROSE, J. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays* (2006).

[45] LE, Y., CHANG, H., MUKHERJEE, S., WANG, L., AKELLA, A., SWIFT, M. M., AND LAKSHMAN, T. V. UNO: Uniflying host and smart NIC offload for flexible packet processing. In *SoCC* (2017).

[46] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *SOSP* (2017).

[47] LI, B., TAN, K., LUO, L., LUO, R., PENG, Y., XU, N., XIONG, Y., AND CHENG, P. ClickNP: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2016).

[48] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading distributed applications onto smartnics using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2019).

[49] LIU, M., PETER, S., KRISHNAMURTHY, A., AND PHOTHILIMTHANA, P. M. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *Usenix Annual Technical Conference (ATC)* (2019).

[50] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving resource efficiency at scale. In *International Symposium on Computer Architecture (ISCA)* (2015).

[51] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RYAN, M., RUBOW, E., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: a microkernel approach to host networking. In *SIGOPS* (2019).

[52] MELLANOX TECHNOLOGIES. Innova - 2 Flex Programmable Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf.

[53] MELLANOX TECHNOLOGIES. Mellanox BlueField Smart-NIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[54] MELLANOX TECHNOLOGIES. NVIDIA Mellanox BlueField-2 DPU. https://www.mellanox.com/products/bluefield2-overview.

[55] MELLETTE, W. M., DAS, R., GUO, Y., MCGUINNESS, R., SNOEREN, A. C., AND PORTER, G. Expanding across time to deliver bandwidth efficiency and low latency. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2020).

[56] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[57] NETRONOME. NFP-6xxx flow processor. https://netronome.com/product/nfp-6xxx/.

[58] PENSANDO. DSC-100. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-100-Product-Brief.pdf.

[59] PHOTHILIMTHANA, P. M., LIU, M., KAUFMANN, A., PETER, S., BODIK, R., AND ANDERSON, T. Floem: A programming system for NIC-accelerated network applications. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[60] PONTARELLI, S., BIFULCO, R., BONOLA, M., CASCONE, C., SPAZIANI, M., BRUSCHI, V., SANVITO, D., SIRACUSANO, G., CAPONE, A., HONDA, M., HUICI, F., AND SIRACUSANO, G. FlowBlaze: Stateful packet processing in hardware. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[61] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2012).

[62] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for end-host rate limiting. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[63] SEWELL, K., DRESLINSKI, R. G., MANVILLE, T., SATPATHY, S., PINCKNEY, N., BLAKE, G., CIESLAK, M., DAS, R., WENISCH, T. F., SYLVESTER, D., ET AL. Swizzle-switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems 2*, 2 (2012), 278–294.

[64] SHRIVASTAV, V. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2019).

[65] SHU, R., CHENG, P., CHEN, G., GUO, Z., QU, L., XIONG, Y., CHIOU, D., AND MOSCIBRODA, T. Direct universal access: Making data center resources available to FPGA. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[66] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2015).

[67] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2016).

[68] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2016).

[69] STEPHENS, B., AKELLA, A., AND SWIFT, M. Your programmable NIC should be a programmable switch. In *ACM Workshop on Hot Topics in Networks (ACM HotNets)* (2018).

[70] STEPHENS, B., AKELLA, A., AND SWIFT, M. Loom: Flexible and efficient nic packet scheduling. In *Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[71] THOMAS, S., MCGUINNESS, R., VOELKER, G. M., AND PORTER, G. Dark packets and the end of network scaling. In *ANCS* (2018).

[72] TILERA. Tile Processor Architecture Overview For the TILE-GX Series. http://www.mellanox.com/repository/solutions/tile-scm/docs/UG130-ArchOverview-TILE-Gx.pdf.

[73] WENTZLAFF, D., GRIFFIN, P., HOFFMANN, H., BAO, L., EDWARDS, B., RAMEY, C., MATTINA, M., MIAO, C.-C., BROWN III, J. F., AND AGARWAL, A. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 5 (Sept. 2007).

[74] WILTON, S. J. E., HO, C. H., LEONG, P. H. W., LUK, W., AND QUINTON, B. A synthesizable datapath-oriented embedded FPGA fabric. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA)* (2007).

[75] XILINX. Xilinx Alveo: Adaptable Accelerator Cards for Data Center Workloads. https://www.xilinx.com/products/boards-and-kits/alveo.html.

[76] ZILBERMAN, N., AUDZEVICH, Y., COVINGTON, G., AND MOORE, A. NetFPGA SUME: Toward 100 Gbps as research commodity.

A Artifact Appendix

A.1 Abstract

This artifact contains the source code and test benches for PANIC's 100Gbps FPGA-based prototype. Our FPGA prototype is implemented in pure Verilog. Features of the prototype include: the hybrid push/pull packet scheduler, the high-performance switching interconnect, self-contained compute units, and the lightweight RMT pipeline.

This artifact provides two test benches to reproduce the results in Figure 8c and Figure 11a in the Vivado HDL simulator.

A.2 Artifact check-list

- **Compilation:** Running this artifact requires Vivado Design Suite [10]. Vivado v2019.x and v2020.1 WebPack are verified.
- **Hardware:** This artifact does not requires any specific hardware.
- **Metrics:** This artifact measures PANIC's receiving throughput under different chaining models and traffic patterns.
- **Output:** The result will be printed to the console and log files.
- **Experiments:** This artifact includes testbenches and running scripts to replay Figure 8c and Figure 11a.
- **Public link:** https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact

A.3 Description

A.3.1 How to access

This artifact is publicly available at https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact.

A.3.2 Software dependencies

Running this artifact requires Vivado [10]. Vivado WebPack version is license-free, and it has simulation capabilities to recreate our results. Since installing the Vivado WebPack requires plenty of disk space (>20GB), you can choose to instance an FPGA Developer AMI in AWS (https://aws.amazon.com/marketplace/pp/B06VVYBLZZ) to run this artifact. The FPGA Developer AMI has pre-installed the required Vivado toolchain.

A.4 Experiment workflow

1. **Check Vivado is Installed Correctly**

```
$ vivado -mode tcl
// Enter the Vivado Command Palette
Vivado% version
// v2019.x and v2020.1 is verified
Vivado% quit
```

2. **Clone the Repo and Make Run**

```
$ git clone [Artifact_Repo]
$ cd panic_osdi20_artifact
$ make test_parallel
$ make test_shaaes
```

The make command first compiles the source code, then runs the simulation tasks in Vivado. The *test_parallel* test replays Figure 8c and the *test_shaaes* test replays Figure 11a.

A.5 Evaluation and expected result

The result will be printed to the console. The output will also be logged in *./build/export_sim/xsim/simulate.log*. For the expected output and analysis please reference Figure 8c and Figure 11a.

A.6 Notes

For more details about the code structure, please reference https://bitbucket.org/uw-madison-networking-research/panic_osdi20_artifact/src/master/README.md

A.7 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts