# Quadrant: A Cloud-Deployable NF Virtualization Platform

Jianfeng Wang
University of Southern California
USA
jianfenw@usc.edu

Tamás Lévai
Budapest University of Technology and Economics
Hungary
levait@tmit.bme.hu

Zhuojin Li
University of Southern California
USA
zhuojinl@usc.edu

Marcos A. M. Vieira
Universidade Federal de Minas Gerais
Brazil
mmvieira@dcc.ufmg.br

Ramesh Govindan
University of Southern California
USA
ramesh@usc.edu

Barath Raghavan
University of Southern California
USA
barathra@usc.edu

## Abstract

Network Functions (NFs) now process a significant fraction of Internet traffic. Software-based NF Virtualization (NFV) promised to enable rapid development of new NFs by vendors and leverage the power and economics of commodity computing infrastructure for NF deployment. To date, no cloud NFV systems achieve NF chaining, isolation, SLO-adherence, and scaling together with existing cloud computing infrastructure and abstractions, all while achieving generality, speed, and ease of deployment. These properties are taken for granted in other cloud contexts but unavailable for NF processing.

We present Quadrant, an efficient and secure cloud-deployable NFV system, and show that Quadrant's approach of adapting existing cloud infrastructure to support packet processing can achieve NF chaining, isolation, generality, and performance in NFV. Quadrant reuses common cloud infrastructure such as Kubernetes, serverless, the Linux kernel, NIC hardware, and switches. It enables easy NFV deployment while delivering up to double the performance per core compared to the state of the art.

## CCS Concepts

• **Networks → Middle boxes / network appliances**.

## Keywords

Network Functions, service chain, virtualization

**ACM Reference Format:**
Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. 2022. Quadrant: A Cloud-Deployable NF Virtualization Platform. In *Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3542929.3563471

## 1 Introduction

Network Function Virtualization (NFV) enables both simple (*e.g.,* VLAN tunneling) and complex (*e.g.,* traffic inference) packet processing using software-based Network Functions (NFs). Over the last decade, much research has explored the design of NFV platforms or components thereof (*e.g.,* [14, 37, 40, 44, 49, 55, 56] among others). Despite this, we know of no widely deployed NFV platforms that have achieved the original goal of NFV: making hardware middleboxes "someone else's problem" [48]. Instead, industry has doubled down on custom hardware solutions [38] and complex and bespoke NFV frameworks [33].

We posit that NFV can achieve its original goals using an NFV platform architected as a cloud service. In fact, cloud-deployability of NFV is fast becoming a necessity, driven by the move to cloud-hosted 5G cellular function virtualization [30]. A cloud-deployable NFV platform must *concurrently* support several functional and performance requirements identified by prior work: (1) chaining multiple, possibly-stateful third-party NFs to achieve operator objectives [37, 55, 56]; (2) NF-state and traffic isolation between mutually-untrusted, third-party NFs [40, 44]; (3) near-line-rate, high-throughput packet processing [14]; and (4) latency and throughput SLO-adherence [56]. In addition, it must

| Key Property | Quadrant | NetBricks [40] | EdgeOS [44] | Metron [14] | SNF [49] |
|---|---|---|---|---|---|
| Performance | High | Medium | Medium | High | Low |
| Isolation | ✓ | ✓ | ✓ | ✗ | ✓ |
| Stateful-NF Support | ✓ | ✗ | ✗ | ✓ | ✓ |
| Third-party Compatibility | ✓ | ✗ | ✓ | ✗ | ✓ |
| SLO-aware Chaining | ✓ | ✗ | ✗ | ✗ | ✗ |
| Failure Resilience | ✓ | ✗ | ✗ | ✗ | ✓ |

**Table 1:** A comparison of NFV platforms' properties that are key for being production-ready.

substantially reuse cloud computing infrastructure and abstractions [49]. To our knowledge, no prior work satisfies *all* these objectives (Table 1).

Recent work employs clean-slate custom interfaces, runtimes, and control planes [14, 19, 37, 40, 44], but has not achieved NF chaining, isolation, and scaling without losing generality, performance, or ease of deployment. Many approaches break layering and isolation for performance [14, 19, 37], or leverage specialized hardware [17, 22, 56] at the cost of poor deployability. To support untrusted third-party NFs, other solutions either use language-based isolation [40, 42], losing generality, or require expensive per-hop packet copying [44], sacrificing performance (§3.1). Most relevant is SNF [49], a recent effort on cloud-based NFV. It sheds significant insight on distributing traffic among NF instances, but does not support SLO-aware chaining and ignores optimization opportunities available in today's cloud infrastructure, such as using hardware and OS kernel features.

In this paper, we describe the design and implementation of Quadrant, an NFV platform that achieves key functional and performance requirements (Table 1), while significantly reusing cloud infrastructure and abstractions. It uses containers to run NFs, NIC virtualization and software-based packet steering to balance load among NFs, extends a standard cluster management system (Kubernetes) to auto-scale NF processing and ensure failure resilience of NF chains, and standard OS kernel mechanisms to achieve isolation without sacrificing performance.

**Contributions.** We make the following contributions:

***High-performance spatiotemporal packet isolation.*** Quadrant's use of containerized NFs, together with NIC virtualization, ensures that an NF chain can only see its own traffic. Quadrant also ensures a stronger form of packet isolation (§4.2): an NF in a chain can access a packet only after its predecessor NF. Quadrant achieves this by spatially isolating the first NF from the others in the chain using a packet copy. Subsequent NFs can process packets in a zero-copy fashion, with temporal isolation enforced by CPU scheduling. This approach is general and transparent to NF implementations, and requires no language support.

***Performance-aware scheduling.*** For performance, Quadrant dedicates cores to NF chains and uses kernel bypass to deliver packets to NFs, and uses standard OS interfaces to *cooperatively schedule* NF threads from different NFs to mimic *run-to-completion* [14], proven to be essential for high NFV performance (§4). Run-to-completion processes a batch of packets; Quadrant selects batch sizes that satisfy SLOs while minimizing context-switch overhead.

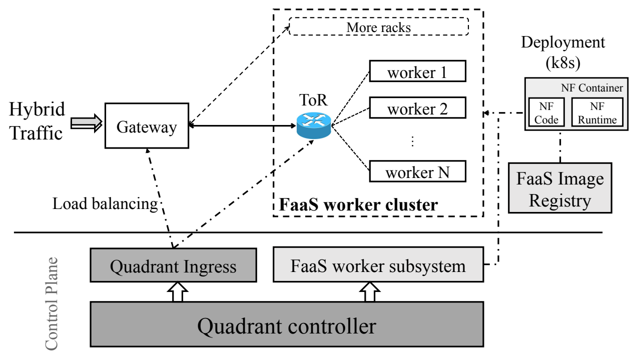
***SLO-aware auto-scaling.*** In response to changes in traffic, Quadrant auto-scales NF chains by dynamically adjusting the number of NF chain instances to minimize CPU core usage while preserving latency SLOs (§5). This flexibility allows tenants to trade-off latency for lower cost, a capability present in a few bespoke NFV systems [52, 56].

We find (§6) Quadrant achieves up to 2.31× the per-core throughput when compared against state-of-the-art NFV systems [40, 44] that use alternative isolation mechanisms. Under dynamic traffic loads, Quadrant achieves zero packet losses and is able to satisfy tail-latency SLOs. Compared to a highly-optimized NFV system that does not provide packet isolation and is not designed to satisfy latency SLOs (but is designed to minimize latency) [14], Quadrant uses slightly more CPU cores (12–32%) while achieving isolation *and* satisfying latency SLOs. Quadrant's total code base is less than half the size of existing NFV platforms [14, 40, 49], and just 3% of an existing open-source Function-as-a-Service (FaaS) platform [34].

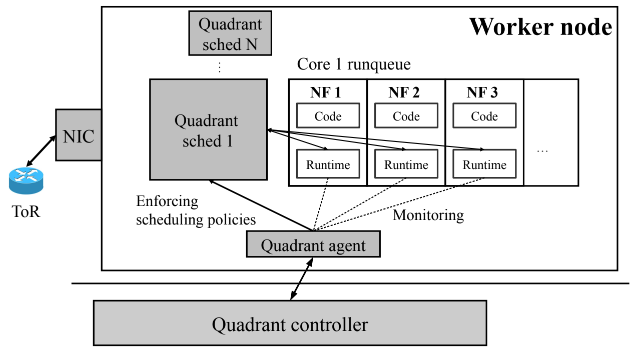
## 2 Quadrant Overview

Here we sketch Quadrant's architecture (Figure 1), which we detail in later sections.

**Quadrant Interface.** Clients access today's cloud services via REST APIs, and front-ends access back-ends via RPC. Both these abstractions work well for normal web requests but can be inappropriate and heavyweight for NFs that process traffic at the packet level, and can introduce significant overhead in the form of unnecessary network headers and additional protocol processing [49]. As a cloud service for deploying custom NFs, Quadrant needs to have an efficient programming model that allows developers to easily create custom NF logic to support packet processing. Quadrant adopts an event-based programming model widely used for web services, and adjusts the programming model so that NFs accept a raw packet struct (a pointer to a packet) as input: they are handler functions for raw-packet events (rather than web or RPC requests). NFs can have state, and they share state using a standard distributed key-value store (*e.g.,* Redis [46]). They invoke a Quadrant runtime that abstracts access to packets and state via library APIs (Figure 1(B)). Indeed, this abstraction is standard for commercial NFs that attach to virtual Ethernet devices. A Quadrant customer (*e.g.,* an organization or an ISP) can then assemble an NF chain of such NFs along with (a) a traffic filter specification for what traffic is to be processed by the chain and (b) a per-packet latency SLO.



(A) Quadrant's control plane.



(B) A Quadrant worker.

Figure 1: Quadrant's controller interacts with Quadrant's ingress and worker subsystem to deploy containerized NFs for packet processing. Unshaded boxes are existing cloud components that Quadrant reuses, lightly shaded ones are components that Quadrant modifies, and darker ones represent new components specific to Quadrant embedded within the infrastructure.

**Quadrant Design.** Quadrant's architecture reuses existing cloud infrastructure (Figure 1(A)). It assumes commodity servers and OpenFlow-enabled switches, and reuses cloud-native worker subsystems (*e.g.,* Kubernetes) that manage a pool of worker servers and allocate system resources (NIC, CPU core and memory) to NFs. Each worker executes NFs encapsulated in containers [1]; in Quadrant, each container hosts an NF.[2] For custom NFs, customers provide container images for each NF: they compile and containerize each NF together with the NF runtime. For third-party NFs, Quadrant's runtime offers a virtual Ethernet interface (a.k.a. *veth*) that is an API wrapper for exchanging packets, which is a standard interface for NFs in production environments. NF images are ready for deployment once uploaded to Quadrant.

For each worker pool, Quadrant requires two new components: a Quadrant *controller* and the Quadrant *ingress*. At each worker, Quadrant adds a *scheduler* and a Quadrant *agent*. The controller manages the deployment of NF instances by interacting with the worker subsystem to deploy Quadrant components (an ingress, per-worker scheduler, and agents) prior to startup. At runtime, the Quadrant controller uses the worker subsystem to deploy NFs as containers. It collects NF performance statistics from each Quadrant agent, serves queries from the ingress, or pushes load balancing decisions to the ingress which then enforces them by modifying a flow table.

Traffic enters and leaves the system at the ingress which forwards traffic using flow entries that enforce the Quadrant controller's workload assignment strategies. A flow entry forwards traffic to a deployed NF chain instance. When a new flow arrives, the ingress queries the Quadrant controller (or uses prefetched queries) to instantiate a new flow entry and routes subsequent packets in the flow to the corresponding NF chain instance.

By design, this architecture is similar to that of Function-as-a-Service (FaaS), because NF chains resemble cloud functions: they are event-based and require scalability, but also have significantly different functional and performance needs. Indeed, our Quadrant implementation is built upon an open-source FaaS platform, OpenFaaS [34], that can be easily deployed in commodity clouds. Quadrant reuses its components (§6). However, it incorporates four novel features designed to address NFV requirements (§1): a novel *execution model* (§3) that permits high throughput packet processing, a *core allocation and scheduling strategy* (§4) that minimizes latency and overhead, a *packet isolation strategy* that permits third-party NFs to run fast and securely (§4.2), and an *auto-scaling technique* that minimizes CPU core usage while being able to meet latency SLOs (§5). We describe these next.

## 3 Quadrant's Execution Model

An execution model describes how an application is executed, what memory it can access, and how it accesses the NIC resource. It is critical for achieving key functional and performance requirements of NFV described in Table 1.

NFs can be seen as network applications that operate on network packets and internal state. Quadrant users write packet processing functions, and use runtime APIs to access packets and state. NFs run as processes with isolated memory for NF states. Packet memory is carefully managed by Quadrant to enable memory sharing, avoiding unnecessary packet copying. Each NF can have many instances that run as processes across multiple cores, each with a data-plane

[1]Containers and VMs are common isolation mechanisms for applications. We adopt NF containers because they are 1.5-2.3× faster than NF VMs [40].
[2]This also includes cases where NFs can be concatenated into a single NF with tools such as OpenBox [3] or SNF [16].

thread managed by Quadrant's scheduler. We argue that this design is critical for achieving both performance and isolation. To provision NFs, Quadrant acts as a cluster system manager to allocate resources, *e.g.,* CPU cores, memory, and network interfaces, to NFs. Quadrant tracks the liveness and performanace for each chain with per-worker Quadrant agents. To scale NFs, Quadrant adjusts the number of instances allocated for each chain with the goal of minimizing CPU core usage while meeting SLOs.

In this section, we describe Quadrant's NF execution model and how it differs qualitatively from prior work.

## 3.1 Existing NF Execution Models

Prior work has explored different NF execution models that dictate how NFs share packet memory, how the runtimes steer packets to NFs, and how they schedule NF execution.

**Memory model.** Prior work has explored three different models for NFs running on the same worker machine: they (1) may share NF state memory, and packet buffer memory (*e.g.,* in Metron [14] and NetBricks [40]), (2) do not share NF state memory, but share packet buffer memory globally (*e.g.,* in E2 [37] and NFVnice [19]), or (3) do not share either NF state memory or packet buffer memory (*e.g.,* in EdgeOS [44]).

**Network I/O model.** Packets must be sent to a specific NF running on a specific server core. In many NFV platforms, such as E2 [37], NFVnice [19], and EdgeOS [44], a hardware switch forwards packets to specific worker machines. Once packets arrive at the server's NIC, a virtual switch forwards traffic locally. In a multi-tenant environment, the vSwitch has read and write access to each individual NF's memory space, and copies packets when forwarding them from an upstream NF to its downstream. The vSwitch can become a bottleneck for both intra- and inter-machine traffic. To scale it up, a runtime can add CPU cores for vSwitches, but is a waste of otherwise productive cores. On our test machine, a CPU core can achieve a 6.9 Gbps throughput when forwarding 64-byte packets (or 13.5 Mpps). Consider a chain with 4 NFs running on a server with a 10 Gbps NIC. The aggregate traffic volume can reach 40 Gbps at peak on the vSwitch, which requires at least 7 CPU cores to run vSwitches (more if traffic is not evenly distributed across the vSwitches).

An alternative approach is to offload packet switching to the ToR switch and the NIC's internal switch. Both switches coordinate to ensure packets arrive at the target machine's/target process's memory. When a packet hits the ToR, the switch not only forwards the packet to a dedicated machine, but also facilitates intra-machine forwarding via L2 tagging. This approach eliminates the need to run a vSwitch. However, it can only ensure that packets are received by the first NF in a chain. Metron [14] and NetBricks [40] take this approach, but rely on a strong assumption: that all NFs can be compiled and run in a single process. However, many popular NFs are commercially available only as containers or VMs and cannot be compiled with other NFs to form a single binary that runs the NF chain. Even if that were possible, the packet isolation requirement constrains flexibility significantly, since it can only then be achieved using language-based memory isolation (*e.g.,* by using Rust [40, 42]).

**CPU scheduling model.** Memory and network I/O models also impact CPU resource allocation and scheduling of NFs and NF chains. When NF chains run in a single process (as in Metron and NetBricks), those runtimes can dedicate a core to an entire chain. When NFs run in separate processes (as in E2 or NFVnice), runtimes must decide whether to allocate one or more cores to a chain, and how to schedule each NF.

## 3.2 NF Chain Execution Model

To ensure minimal changes to existing cloud infrastructures, Quadrant chooses an execution model that sits in a different point in the design space: ***it deploys each NF in a chain as a container***. NFs can share packet buffers (as in Metron or NetBricks), but packet isolation is enforced through OS protection, careful scheduling, and packet copying (unlike NetBricks, which relies on language-specific isolation). Quadrant uses NIC I/O virtualization and kernel bypass to reduce packet steering overhead. The rest of this subsection describes some of the details of Quadrant's packet I/O and memory models. The next section describes CPU allocation and scheduling.

**Packet I/O.** Quadrant uses DPDK for fast userspace networking to handle packet I/O for NFs. Because other cloud services may use the kernel networking stack and run on the same worker, Quadrant must use userspace networking for NFs while being compatible for kernel networking options. To do so, it uses Single-root Input/Output Virtualization [21] (SR-IOV) to virtualize the NIC hardware. SR-IOV allows a PCIe device to appear as many physical devices (vNICs). With SR-IOV, NIC hardware generates one Physical Function (PF) that controls the physical device, and many Virtual Functions (VFs) that are lightweight PCIe functions with the necessary hardware access for receiving and transmitting data.[3] On a worker, the Quadrant agent (Figure 1) manages the virtualized devices via kernel APIs through the PF.

**Flow to Chain Mapping.** Quadrant maps flows to NF chains at its ingress (§5.2). Before the Quadrant controller allocates a CPU core to a chain, the Quadrant agent sets up

[3]Using multi-queue NICs may lead to performance isolation issues that have solutions proposed by recent research to improve fairness and performance [11, 50]. In Quadrant, NICs do not involve complex packet scheduling. Instead, they just dispatch packets based on L2 headers, so simply applying a bandwidth limit to VFs is sufficient to avoid this issue.

a VF to the chain and pins the chain to its allocated core (§4).[4] Later, the hardware switch, when matching a flow, rewrites the MAC address of the packet to be the one from the corresponding VF's MAC address. This approach enables outsourcing flow dispatching and provides a flow-level granularity.

**Memory.** In Quadrant, a runtime on behalf of an NF chain initializes a file-backed dedicated memory region that holds fixed-size packet structures for incoming packets. It also creates a ring buffer that holds packet descriptors that point to these packet structures. To receive packets from the virtualized NIC, the NF runtime passes this ring buffer to its associated VF so that the NIC hardware can perform DMA directly to the NF runtime's memory.

**NF State Management.** Stateful NF (*e.g.,* IDS) packet processing depends on both the packet itself and the NF's current state. Prior work (*e.g.,* statelessNF [13], S6 [55], SNF [49]) has demonstrated that it is feasible to efficiently decouple NF processing from state, because most stateful NFs only have to access remote state 1–5 times per connection/flow [13].

In Quadrant, we leverage this observation to maintain per-NF global state remotely in Redis, while providing efficient caching to mitigate the latency overhead of pulling state from the external store. Quadrant's programming model exposes a set of simple APIs for writing a stateful NF: `update(flow, val)` and `read(flow, val)`, where `flow` corresponds to a BPF matching rule. Besides global NF state, Quadrant's NF runtime maintains general NF state in a hash table locally so that the user-defined NF can process most packets with state present in its local memory. The runtime makes the state synchronization transparent to the NF by interacting with the external Redis service, and ensures that each NF can only access its own state. It processes packets in batches (§4), and for each packet batch, the runtime batches all state accesses required by all packets prior to processing. It pulls state from Redis with a batched read request to amortize the per-packet state access delay.

Once an NF calls `update`, its runtime issues a request to the local Quadrant agent to update global state in the Redis service and the packet triggering the state update. The agent releases the packet once the global state has been updated. This is necessary to keep NF state consistent: the packet won't reach its destination unless the NF's global state has been updated. This design also avoids doing state synchronization operations in the data plane, and minimizes Quadrant's state synchronization's impacts on the overall end-to-end latency.

[4]Mellanox ConnectX-5 100 GbE NICs and Intel XL710 40 GbE NICs support up to 128 VFs, while Intel E810 100 GbE NICs can support up to 256 VFs. With a large number of VFs, Quadrant can saturate all cores on modern platforms, even for a hundred cores.

In Quadrant, each NF is associated with a unique hash key, which is used to tag NF states in the Redis service. This is useful to recover the state of a single NF instance when migrating flows from it or recovering from failure (§4.3).

Quadrant's state consistency mechanism builds on Redis's consistency guarantee. In Redis, acknowledged writes are committed and never lost and reads return the most-recent committed write [46]. In Quadrant, an NF emits a packet only after receiving a state update acknowledgement, and starts processing a migrated flow only after emitting packets from the original core. When an NF update *per-flow* state (see also §7), this ensures state consistency. This can add some delay, but our experiments demonstrate that, despite this, Quadrant can achieve its performance goals.

## 4 Core Allocation and Scheduling

In Quadrant, each NF is deployed as an individual container in a Kubernetes cluster. Quadrant dedicates a core to all NFs in a chain; that core serves a traffic aggregate assigned to that chain. When the total traffic exceeds the capacity of a single core, Quadrant spins up another chain instance on another core, and splits incoming traffic between NF chain instances (§5). The Quadrant controller manages all NFs via Kubernetes APIs to control the allocation of memory, CPU share, and disk space.

### 4.1 Controlling Chain Execution

Userspace I/O and shared memory can reduce overhead, but to be able to process packets at high throughput and low latency, Quadrant must have tight control over NF chain execution. As discussed earlier, custom NF platforms use two different approaches. One approach bundles NFs in an NF chain into a single process to *run to completion* in which each NF in the chain processes a batch of packets before moving onto the next batch (as in Metron or NetBricks). This approach ensures high performance and predictability by amortizing overhead over a packet batch. To achieve packet isolation, NetBricks relies on language isolation, so cannot support third-party NFs. The second approach, used by NFVnice and others, is to run each NF in a separate process and use vSwitches for packet forwarding, which ensures isolation but incurs high overhead by copying packets, requiring careful CPU allocation and scheduling (*e.g.,* tuning CFS and using ECN for backpressure in NFVnice).

Instead, Quadrant aims for the best of both worlds: it does not force developers to write and release NF code in a specific programming language; it also avoids overheads and complexity brought by approaches that use vSwitches. Quadrant introduces *spatiotemporal packet isolation* in which NF chains operate on 1) spatially-isolated packet memory regions (as opposed to the typical model in run-to-completion software switches such as BESS, in which all NF chains on a machine run in the same memory) and 2) are temporally isolated through careful sequencing of their execution, which proceeds in a run-to-completion fashion *across processes* and uses *cooperative scheduling* mechanisms to hand off control at the natural execution boundary of packet batch handoff (§4.2). This isolation ensures that NF chains (which may process different customers' traffic) cannot see each others' packet streams or state, and even within a chain each NF maintains private state and only gets to execute (and thus access packet memory) when it is expected to perform packet processing in the chain.

**Enforcing run-to-completion scheduling.** Quadrant uses a per-core NF *Cooperative Scheduler.* All NF containers in a chain are assigned to a single core; each runs two processes. The NF process is single threaded and processes traffic. The NF runtime process has an RPC server to control the NF and a monitoring thread to collect statistics (§5). To avoid interfering with packet processing, the monitoring thread runs on a separate core. The runtime is invisible to NF authors.

To tightly coordinate NF chain execution, Quadrant uses Linux's real-time (RT) scheduling support, and manages NF threads' real-time priorities and schedules them using a FIFO policy. We use this policy to emulate, as described below, NF chain run-to-completion execution in which each NF in the chain processes a batch of packets in sequence.

**Scheduling model.** In Quadrant's cooperative scheduling, an upstream NF runs in a loop to process individual packets of a given batch, and then yields the core to its downstream NF. This is transparent to the NFs: once the user-defined NF finishes processing, the NF runtime determines whether to transmit the packet batch to the downstream NF; if yes, the runtime invokes yield.[5]

For this, the Cooperative Scheduler has to bypass the underlying scheduler (CFS in our implementation) and take full control of a core. Internally, the scheduler maintains two FIFO queues: a run queue that contains runnable NFs, and a wait queue that contains all idle NFs. It offers a set of APIs that the NF runtime can use to transfer the ownership of NF processes of a chain from CFS to the Cooperative Scheduler. These APIs are used by the Quadrant agent, which runs as a privileged process. NFs themselves cannot access these APIs, so cannot change scheduling priorities or core affinity.

Once a chain is deployed, all NFs are managed by the Cooperative Scheduler, and are placed in the scheduler's wait queue as ***detached***. Once an NF chain switches into the ***attached*** state (see below), the Cooperative Scheduler pushes NFs of this chain into its run queue and ensures that the original NF dependencies are preserved in the run queue. To detach a chain, the Cooperative Scheduler waits for the chain to finish processing a batch of packets, if any, and then moves these NF processes back to the wait queue.

[5]To deal with non-responsive NFs, the runtime terminates chain execution if an NF fails to yield after a conservative timeout.

**How scheduling works.** Once an NF starts, Quadrant's NF runtime reports its thread ID (tid) to the Quadrant agent running on the same worker. Once all NFs are ready, the Quadrant agent *registers* their tids as a scheduling group (called a sgroup) to the Cooperative Scheduler. Thereafter, the cooperative scheduler takes full control of NFs. An NF chain starts in the ***detached*** state. When the Quadrant controller assigns flows to the chain (§5), the Cooperative Scheduler *attaches* the chain to the core. When the monitoring thread sees no traffic has arrived for the chain, the scheduler *detaches* the chain, so the Quadrant controller can re-assign the core.

For attach and detach operations, and to schedule NF chain execution, the Cooperative Scheduler has a master thread to serve scheduling requests and runs one enforcer thread on each managed core. The scheduler uses features of Linux FIFO thread scheduling: 1) high-priority threads preempt low-priority threads and 2) a thread is executed once it is at the head of the run queue, and is moved to the tail after it finishes. An enforcer thread is raised to the highest priority when enforcing scheduling decisions. When an NF chain is instantiated on a core, the enforcer thread registers the corresponding NF processes as low-priority FIFO threads so that they are appended to the wait queue. When attaching the NF chain, it moves NF processes to the run queue by assigning them a higher priority, and vice versa when detaching a sgroup. Operations are done in the sequence that NFs are positioned in the NF chain, so when an NF yields, the CPU scheduler automatically schedules the next NF in the chain.

In this model, each worker machine splits CPU cores into two groups. One group is managed by the Cooperative Scheduler, while the other runs with normal threads managed by CFS. We use a standard kernel and support different schedulers on different cores. This enables running NF and non-NF workloads on the same machine. Recent research [8, 35] shows that this is critical for achieving high CPU core efficiency for latency-sensitive applications.

**Estimating minimum batch size.** The Cooperative Scheduler introduces $N$ context switches for a chain with $N$ NFs. Without packet batching, a core may incur significant context switch overhead. Quadrant estimates the minimum batch size required to bound the context switch overhead within a fraction $p$ (which is configurable).

Let $\tilde{r}$ is the packet rate when running the NF chain in a single thread, the maximum achievable rate. Then if $F$ is the processor clock frequency, and $S_i$ is the cycle count of the $i$-th NF in a chain needed to process a packet, $\tilde{r} = \frac{F}{\sum_{i=1}^{N} S_i}$.

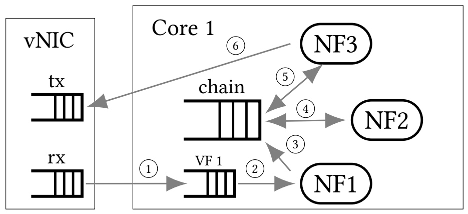


**Figure 2:** Timeline of packets on a Quadrant worker. A packet is tagged at the ingress. 1) NIC's L2 switch sends it to NIC VF associated with the destined chain. NIC VF DMAs packets to the first NF's memory space. 2) NF 1 processes the packet. 3) After NF 1's packet processing function returns, the packet is copied to the chain's pktbuf by the NF runtime if there are other NFs. This is necessary to ensure packet isolation as the NIC's pktbuf should only been seen by NF 1. 3)–5) A per-core cooperative scheduler controls the execution sequence of NFs to ensure temporal packet isolation. 6) Final NF asks VF to send the packet out.

The actual packet rate is given by:

$$r = \frac{F}{\sum_{i=1}^{N} S_i + \frac{N \cdot C_{ctx}}{B}} \tag{1}$$

where $C_{ctx}$ is the context switch cost, $B$ is the batch size.

To bound the overhead to a fraction $p$, we simply solve for the smallest $B$ that satisfies the inequality $\tilde{r}/r \leq (1 + p)$.

## 4.2 Spatiotemporal Packet Isolation

**What is packet isolation?** Quadrant targets support for third-party NFs (*e.g.,* a Palo Alto Networks firewall [39]) in multi-tenant settings where each chain may consist of NFs from multiple vendors, and each chain may be responsible for processing a specific customer's traffic. For this, Quadrant must ensure (1) memory isolation: each NF must have its own private memory for maintaining NF state; (2) packet isolation: within an NF-chain, an NF should not be able to access a packet until its predecessor NF has finished processing the packet, and across chains, an NF should not be able to access packets not destined to its own chain.

**Achieving Isolation in Quadrant.** Since each NF is encapsulated in a container, memory isolation for NF state is trivially ensured. Quadrant uses shared memory to effect zero-copy packet transfers. Figure 2 describes how Quadrant achieves packet isolation while permitting (near) zero-copy transfers. The key idea is to use shared packet memory for NFs to avoid packet copying whenever possible, and control the access to the shared memory via cooperative scheduling to provide lightweight isolation.

Quadrant allocates each NF chain a separate virtual NIC with SR-IOV, each initialized with a separate ring buffer queue that holds packets for the chain. Upon packet arrival, the NIC hardware directly DMAs packets to this queue. Ideally, NFs within the chain access the queue directly in the shared memory region, avoiding copying. However, this can violate packet isolation because a downstream NF could access shared memory while the NIC hardware writes to it.

To avoid this, Quadrant gives only the first NF in the chain access to the NIC packet queue, and also allocates a second packet queue for each NF chain. This second queue holds packets for downstream NFs in the chain, and is shared among those NFs. Thus, the first NF can access the NIC packet queue and is *spatially* isolated from other chains and from downstream NFs. It processes each batch of packets and copies it to the second packet queue.

Quadrant then *temporally* isolates the second packet queue across all downstream NFs through cooperative scheduling. Cooperative scheduling ensures NFs run in the order they appear in the chain, so even though it has access to shared memory, a downstream NF cannot access a batch that has not been processed by an upstream NF since it will not be scheduled. This permits zero-copy packet transfer between all NFs except the first.

For a chain with only one NF, Quadrant omits the unnecessary packet copying and cooperative scheduling. The Quadrant NF runtime also applies an optimization that prefetches packet headers into the L1 cache before calling the user-defined NF for processing. This optimization can improve performance (§6.2).

Finally, Quadrant allocates each chain its own packet queues, and does not share queues across chains. This ensures *spatial* packet isolation across different chains.

## 4.3 Other Details

**Mitigating startup cost.** Auto-scaling may need to allocate a new worker to an NF chain. Cold-starts can incur significant delay, especially since Quadrant uses user-space networking libraries that can incur 500 ms or more to set up memory buffers. This delay can result in SLO violations. Like prior work [29, 32] on reducing serverless startup time, Quadrant keeps a pool of pre-deployed NF chains that start in the detached state and do not consume CPU resources.

**Failure resilience.** Quadrant is resilient to NF failures. Each NF monitor tracks liveness of each NF in a chain by tracking the progress of per-NF packet counters. Other Quadrant components like the controller, the agents, and the ingress are instantiated by Kubernetes, which manages their recovery. Once it detects a failed NF, the controller must migrate flows assigned to it to another worker. This is conceptually identical to the flow migration (§5.3) discussed above.

## 5 Auto-scaling in Quadrant

Quadrant *auto-scales* (adapts resources allocated to) NF chains in response traffic volume changes. Quadrant uses an architecture (Figure 1) similar to other cloud services [34, 54]: its controller coordinates with the global ingress and worker machines for auto-scaling. The ingress forwards requests to idle worker instances. The controller manages the pool of instances to handle dynamic traffic while achieving cost efficiency. The controller is aided by a per-worker Quadrant agent that monitors NF performance and works with the cooperative scheduler to enforce scheduling policies.

## 5.1 Monitoring and scaling signals

**The NF monitor.** Monitoring is critical for scaling NF chains. At each NF, the NF monitor collects performance statistics, including NIC queue length, the instantaneous packet rate, and the per-batch execution time. The packet rate is measured as the average processing rate of the whole NF chain and NIC queue length is as reported by the NIC hardware. It also estimates per-batch execution time by recording the global CPU cycle counter at the beginning and the end of sampled executions. A chain's latency SLO is the upper-bound for the tail (defined as the 99th percentile) end-to-end latency.[6]

To avoid interfering with data-plane processing, the NF monitor runs in a separate thread and is not scheduled on a core running NFs. Each NF monitor maintains statistics and sends updates to the Quadrant controller only when significant events occur (to minimize control overhead), such as when queue lengths or packet rates exceed a threshold.

**Signals used by auto-scaling algorithms.** Quadrant's scaling algorithm estimates *end-to-end tail latency* and the *packet load* (defined below) to determine when to scale up or down. To estimate the end-to-end tail latency, Quadrant estimates the p99th duration that a packet spends on a worker (the worker latency), and the p99th network transmission latency. It estimates the worker latency as 2× the p99 per-batch execution time acquired from the monitoring service, as a packet may have to wait for the previous and current batch.We use a function of the link's throughput for the network transmission delay and use offline profiling to map a worker's throughput to the p99 network transmission latency.

Our end-to-end latency estimation is conservative because 1) the worker latency is the worst-case latency and 2) the p99th end-to-end latency is less than or equal to the sum of the p99th worker latency and network transmission latency. Quadrant also measures the *packet load* as the ratio between the current packet rate and the maximum packet rate.[7]

[6]End-to-end packet latency measures the time a packet spends in Quadrant, including both packet processing and network transmission.

[7]Queuing theory notes that the delay can skyrocket as the arrival rate nears the service rate. Quadrant avoids scheduling a chain close to its maximum rate because a small rate increase can significantly increase the latency; it stops assigning more flows to a chain above a given load (*e.g.*, 90%).

## 5.2 Quadrant Ingress

Quadrant's ingress implements its controller's load balancing decisions. It adapts existing load-balancers to ensure flow-consistent forwarding decisions. To do this, it pre-fetches from the controller a list of (worker, core) pairs, and their associated load, to assign to new flows. When a new flow arrives, it assigns it to a worker based on the associated load and installs a flow entry. These actions can be implemented either in hardware or software (we have implemented both).

## 5.3 Scaling of NF Chains

Quadrant tries to schedule chains on the fewest CPU cores that can serve traffic while meeting SLOs. It does so by (a) carefully managing flow-to-worker mappings, and (b) monitoring SLOs and migrating flows to avoid SLO violations.

**Managing flow-to-worker mappings.** The Quadrant controller uses the per-chain end-to-end latency (§5.1) estimation as the primary scaling signal to balance loads among workers to avoid SLO violations. It uses a hysteresis-based approach to control the end-to-end latency under a given latency SLO, while maximizing core utilization. Suppose $T_{slo}$ is the target chain's SLO. Quadrant uses two thresholds: a *lower* threshold $\alpha T_{slo}$; the *upper* threshold is $\beta T_{slo}$ $(0 < \alpha < \beta < 1)$. The Quadrant controller only assigns new flows to chains whose estimated end-to-end latency is less than the lower threshold. Of these, it selects the chain with the highest packet load (§5.1), thereby ensuring that Quadrant uses the fewest cores. Finally, it stops assigning new flows to a chain whose estimated p99 latency is between the two thresholds.

**Migrating flows to meet SLOs.** Due to traffic dynamics, a chain's estimated end-to-end latency can exceed the upper threshold; then, the controller moves flows from this chain until its end-to-end latency falls below the lower threshold.

Migrating flows reduces the queuing delay. According to Little's law [27], the average packet queuing delay is $d = {}^{1}/_{(r_{max}-r)}$, where $r_{max}$ is the maximum packet rate that a chain runs on a core, and $r$ is the chain's current packet rate. To compute the slope of the queuing delay curve:

$$\frac{\delta d}{\delta r} = \frac{1}{(r_{max} - r)^2} \tag{2}$$

Translate (2) into the following form:

$$\frac{\delta r}{r} = \frac{\delta d}{d}\left(\frac{r_{max}}{r} - 1\right) \tag{3}$$

where ${}^{\delta r}/_{r}$ is the packet rate change ratio; ${}^{\delta d}/_{d}$ is the latency change ratio; the rate-adapting term $(\frac{r_{max}}{r} - 1)$ indicates that decreasing packet rate more is necessary for decreasing the latency by the same ratio when the packet rate $r$ is low.

With the above intuition, we decide the sum of packet rates $\Delta r$ for migrated flows as a function of the chain's current packet rate $r_{curr}$ and its estimated latency $t_{curr}$. Note that, $t_{curr} > \beta T_{slo}$, where $T_{slo}$ is the latency SLO and $\beta T_{slo}$ is the *upper* latency threshold (§5.3). Quadrant uses the *lower* threshold $\alpha T_{slo}$ as the target latency for the migration, and calculates the sum of migrated flows' packet rates as[8]:

$$\Delta r = r_{curr} \frac{t_{curr} - \alpha T_{slo}}{t_{curr}} \left(\frac{r_{max}}{r_{curr}} - 1\right) \tag{4}$$

Alternatively, Quadrant can migrate flows so that the aggregated packet rate is proportional to the latency change ratio w/o the rate-adjusting term. We evaluate these in Appendix.

Quadrant's runtime manages migration of stateful NFs to a new worker. The runtime on the old worker synchronizes state with Redis before emitting packets in a batch. When a flow migrates to another worker, that worker's runtime fetches related state from Redis before processing packets.

**Reclaiming idle cores.** Finally, when an NF thread becomes idle (all flows previously assigned to it have completed), Quadrant reclaims the assigned core. Quadrant could have, instead, migrated flows away from underutilized NF chains, but this would have complicated state management for stateful NFs. We have left this optimization to future work.

Listing 1 in Appendix shows the ingress algorithm to assign flows to NF chain instances, and the algorithm used at workers' cooperative schedulers to dynamically attach and detach cores to/from chains in response to traffic dynamics.

## 6 Evaluation

Next we substantiate claims listed in Table 1: Quadrant ensures high performance and meets SLOs, provides NF isolation, supports stateful NFs, and is robust to NF failure, while reusing existing cloud components.

**Implementation.** Quadrant is built upon OpenFaaS [34], an open-source FaaS platform for hosting serverless functions. OpenFaaS consists of infrastructure and application layers, and uses Kubernetes, Docker, and the Container Registry. Quadrant reuses these APIs to manage and deploy NFs. OpenFaaS uses its gateway to trigger functions, and Quadrant adds an ingress. Incoming traffic is split at the system gateway; normal application requests are forwarded to OpenFaaS's gateway, while NFV traffic is forwarded to the Quadrant ingress. OpenFaaS uses a function runtime that maintains a tunnel to the FaaS gateway, and hands off requests to user-defined functions; instead, Quadrant uses the above mechanisms to receive traffic from its ingress (§5). Quadrant reuses OpenFaaS's general framework and relies on a per-worker agent for NF performance monitoring and its cooperative scheduler for enforcing scheduling policies. We quantify Quadrant's additions in §6.1.

**Experiment setup.** We use Cloudlab [5] and run experiments on a cluster of 10 servers, and configure both DPDK and SR-IOV. Each server has dual-CPU 16-core 2.4 GHz Intel Xeon E5-2630 (Haswell) CPUs with 128 GB memory (DDR4 1866 MHz). To reduce jitter, we disable hyperthreading and CPU frequency scaling. Each server has one dual-port 10 GbE Intel X520-DA2 NIC. Both are connected to an experimental LAN for data-plane traffic. Each machine has one 1 GbE Intel NIC for control and management traffic. Servers connect to a Cisco C3172PQs ToR switch with 48 10 GbE[9] ports and Openflow v1.3 support. The traffic generator and the Quadrant ingress run on dedicated machines.

**Methodology and Metrics.** Our experiments use end-to-end traffic with 3 canonical chains from light to heavy CPU cycle cost, from documented use cases [20]. Chain 1 is a L2/L3 tunneling pipeline: Tunnel⟶IPForward; Chain 2 is an expensive chain with DPI and encryption NFs: ACL⟶UrlFilter⟶Encrypt; Chain 3 is a state-heavy chain that requires connection consistency: ACL⟶NAT. Tunnel parses a packet's header, determines its VLAN TCI value and appends a VLAN tag to the packet. ACL enforces 1500 access control rules. UrlFilter performs TCP reconstruction over flows, and applies complex string matching rules (e.g., Snort [45] rules) to block connections mentioning banned URLs. Encrypt encrypts each packet payload with 128-bit ChaCha. NAT maintains a list of available L4 ports and performs address translation for connections, assigning a free port and maintaining this port mapping for a connection's lifetime.

Key performance metrics include: end-to-end latency distribution and packet loss rate and time-average and max CPU core usage for the test duration. The traffic generator uses BESS [2] to generate flows with synthetic test traffic.

### 6.1 Quantifying Reuse of Abstractions

Quadrant's deployability stems from its reuse of existing cloud frameworks and its limited new code. Quadrant adds code in three categories. The first is code for NFV at the (edge) cloud (independent of Quadrant), 4150 LOC, including for packet processing, monitoring, isolation, SLO scaling, and core reclaiming. The second category contains 1210 LOC to support Quadrant's specific mechanisms, including isolation with shared memory and SLO-adherent chaining. The third category is 4200 LOC to leverage standard APIs, including run-to-completion scheduling, supporting statefulness

[8]Due to measurement errors, $r_{curr}$ samples (calculated by using the chain's packet counter) may be higher than $r_{max}$ (calculated by using the amortized per-packet cycle cost). To avoid a negative value, we apply a hard lower bound 0.25 for the rate-adapting term, when $r_{curr} \geq 0.8 r_{max}$.

[9]We also conducted one experiment (§6.6) using 40/100 GbE NICs on our own testbed. (Our experiments use servers w/ Inter CPUs and an OpenFlow-enabled network. Cloudlab does not support these for 40/100 GbE.)

and packet processing interfaces, and cooperative scheduling. The rest is for CLI and debugging tools, which are nice to have but not necessary. By comparison, OpenFaaS [34] is 345k, OpenLambda [12] is 217k, NetBricks [40] is 31k, Metron [15] is 30k for its control plane, and SNF [49] is 20k.

In summary, Quadrant adds a small fraction to existing FaaS systems (2.7% of OpenFaaS). Further, Quadrant uses far fewer lines of code versus custom NFV systems because it reuses existing abstractions judiciously, and only requires about 1k lines of custom code (the second category above).

## 6.2 Performance Comparisons: Isolation

We compare Quadrant against other NFV systems that make different isolation choices. For this experiment, we use chains of many instances of a canonical Berkeley Packet Filter (BPF) [26, 53] NF that parses packet headers and performs 200 longest-prefix matches on packet 5-tuples.[10] Our evaluations vary NF chain lengths as in prior work [14, 23].

**Isolation via copying.** EdgeOS [44] supports isolation via data copying. We emulate EdgeOS on top of a reimplementation of NFVnice [19] with the same set of mechanisms for packet copying, scheduling notifications, and cache-line optimizations. We use NFVnice's master module to move packets between NF processes. The master module runs as a multi-threaded process with one RX thread for receiving packets from the NIC, one TX thread for transmitting packets among NFs, and one wake-up thread for notifying a NF that a message has arrived at its message buffer. All three threads run on dedicated cores to maximize its performance.

**Isolation via safe languages.** NetBricks [40] uses compile-time language support from Rust to ensure isolation among NFs plus a run-time array bounds check. We reuse NetBricks's open-source implementation.

**Results.** Figure 3 shows the throughput of different isolation approaches. Quadrant outperforms NetBricks (1.21-1.51×) and NFVnice w/ packet copying (1.61-2.31×).

NFVnice with packet copying achieves 62% throughput relative to Quadrant with a single-NF chain. As chain length increases its throughput decreases despite its 3 extra CPU cores for transmitting packets among NFs because: (a) cross-core packet copy overheads and (b) load imbalance across NFs since NFVnice tunes scheduling shares for NFs on a single core using Linux's `cgroup` mechanism.

NetBricks suffers from memory access overheads due to array bounds checks; in our experiments, memory accesses are incurred during longest prefix matches. These overheads become significant when packets trigger complex computations, which explains its drop in performance. To validate this

[10]Fixing the number of per-packet memory accesses is important for a reason described later. We have also experimented with different values of the number of matches, and omit results for brevity.

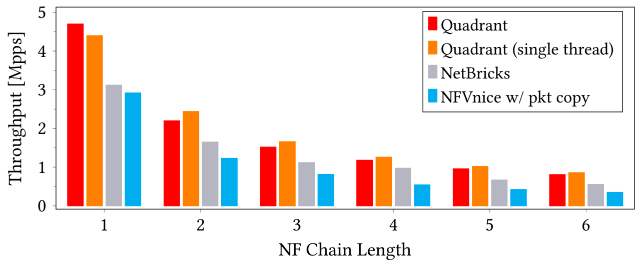


**FIGURE 3:** Throughput with increasing chain length for running an NF chain on a single core.

assertion, we ran NetBricks with dummy NFs (that use an equivalent number of CPU cycles with *no* per-packet memory accesses), and found that it can achieve 94-99% of Quadrant's performance. By contrast, Quadrant's lightweight isolation does not incur per-memory-access overheads, so it has higher throughput.

To understand the overhead in Quadrant imposed by isolation, we implemented a no-isolation variant labeled Quadrant (single thread) that runs all NFs in a single thread. Compared to this unsafe-but-fast variant, Quadrant has an overhead that remains at the same level *regardless of the chain length*: Quadrant achieves a 90.2%-94.2% per-core throughput when deploying a multi-NF chain while providing isolation. Thus, Quadrant pays a 6-10% penalty for achieving isolation. For single-NF chains, we turned off the prefetch-into-L1 optimization described in §4.2 in Quadrant's variant, and found that Quadrant achieves slightly better performance.

## 6.3 Performance Comparisons: Scaling

Quadrant scales chains to meet their latency SLOs. We quantify CPU core usage when deploying chains. Here, we compare Quadrant against Metron [14], a high-performance NFV platform, in the same end-to-end deployment setting. Metron auto-scales core usage, but does not support SLO-adherence. E2 [37] and OpenBox [3] also have the same property, but Metron outperforms them, so we compare only against Metron. Metron does not provide packet isolation, so we do not include it in isolation comparisons.

Before each experiment, an NF chain specification is passed to both systems' controllers to deploy NFs in the test cluster. Metron also uses a hardware switch to dispatch traffic, and has its own CPU scaling mechanism. Unlike Quadrant, it compiles NFs into a single process, and runs-to-completion each chain as a thread. Each Metron runtime is a multi-threaded process that takes all resources on a worker machine to execute chains with no isolation.

***Results.*** Across all experiments, both systems achieve a zero loss rate. Thus, we compare two systems by looking at the tail latency, and the CPU core usage when they serve the test traffic (100 million packets). Quadrant can meet the tail

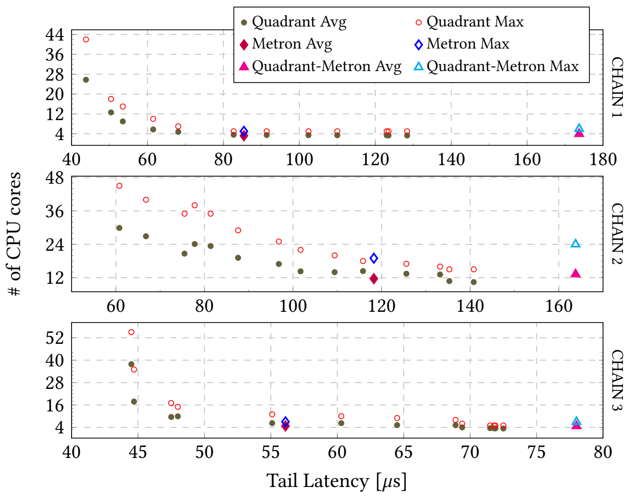


**FIGURE 4:** Core usage of NF chains implemented in Quadrant and Metron as a function of achieved tail latency.

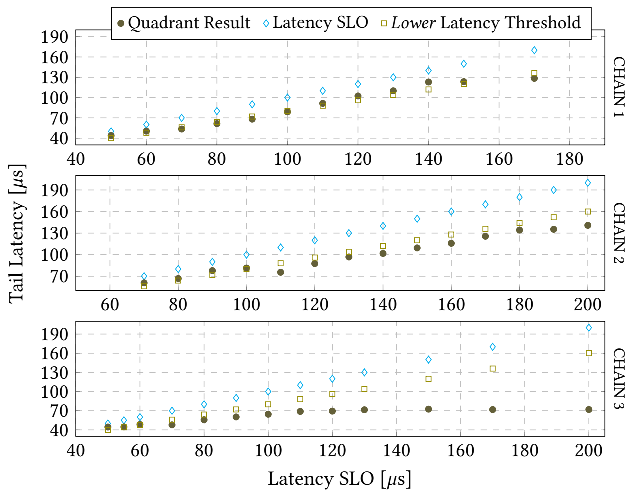


**FIGURE 5:** End-to-end tail latency achieved by NF chains deployed in Quadrant as a function of latency SLO.

latency SLO for all chains. Metron targets zero loss, not SLO adherence. Figure 4 plots the CPU core usage as a function of achieved tail latency by both systems. Metron does not adjust its CPU core usage for different latency SLOs, while Quadrant is able to adjust the number of cores used to serve traffic under different SLOs, to trade off latency and efficiency; it dedicates more cores for a stringent SLO.

To fairly compare CPU core usage, we select Quadrant's samples whose tail latency are *smaller but closest* to Metron's achieved latency, and compare the time-averaged CPU cores again Metron. For Chain 1, Quadrant achieves 82.7 $\mu$s latency using 3.61 cores on average, about 12% more than Metron (they both use the same number of max cores), while Metron achieves 85.4 $\mu$s latency. For Chain 2, Quadrant achieves comparable latency, uses about 23% more cores on average (14.38 vs. Metron's 11.66). Results are similar for Chain 3.

Quadrant's higher core usage results from its support for isolation, its SLO-adherence (both of which Metron lack), and its scaling algorithm (different from Metron's, §5.3). Quadrant incurs multiple context switches in scheduling a chain. With a tight latency SLO, Quadrant uses smaller batch sizes, resulting in a higher amortized per-packet overhead; this is more significant for light chains (*e.g.,* Chain 3). However, the absolute number of extra cores remains small because such chains run at high per-core throughput. Quadrant's monitoring may notify users if chains have small batch size due to a stringent SLO; they can relax the latency SLO or proceed with higher overhead.

To understand the impact of the scaling algorithm by itself, we port Metron's scaling algorithm to Quadrant, and implement a variant, called Quadrant-Metron. Figure 4 shows the achieved latency and CPU core usage for this variant. Like Metron, Quadrant-Metron does not adjust CPU core usage for different latency SLOs; for Chain 1, Quadrant achieves 128.4 $\mu$s, but Quadrant-Metron achieves 173.7 $\mu$s latency and uses 16% more cores on average. Similar results hold for other chains, and validate our decision to design a new scaling algorithm instead of using Metron's.

## 6.4 Validating SLO-adherence with Scaling

**Methodology.** We evaluate Quadrant's SLO-adherence in scaling different chains under traffic dynamics. For each experiment, we run a DPDK-based flow generator to generate traffic at 10 flows/s with a median packet size of 1024-byte, which we selected through trace analysis [4]. The traffic generator gradually increases the number of flows and reaches the maximum throughput after 60 seconds, with a peak load of 18 Gbps.[11] Then the traffic generator stays steady at the maximum rate until 100 million packets are sent. All traffic enters the system through a switch. We evaluate end-to-end metrics, including the tail latency, and the time-averaged number of cores for deploying chains.

**SLO-adherence for different NF chains.** Quadrant scales chains to meet latency SLOs. Quadrant estimates a chain's tail latency, and uses it as a knob to control the end-to-end delay for packets being processed by the chains. We evaluate Quadrant's ability of controlling the end-to-end tail latency under different SLOs with all test chains.

***Results.*** Figure 5 shows the end-to-end tail (p99) latency achieved by Quadrant as a function of latency SLOs. For each chain, Quadrant meets the tail latency SLO for all tested SLOs. At a higher latency SLO, both the *lower* latency threshold and the tail latency are higher. We see the cause of this behaviour: Quadrant's controller migrates flows from a chain when its

[11]This traffic volume is similar to that used by prior work [13].

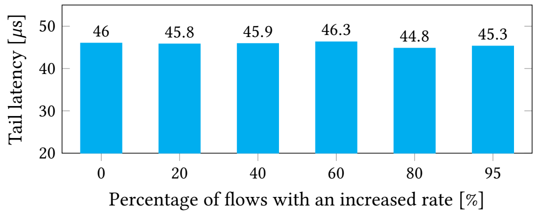


**Figure 6:** End-to-end tail latency achieved under different levels of traffic dynamics. Latency SLO is 70 μs for all groups.

estimated latency exceeds the *upper* latency threshold, and it sets the *lower* threshold as the latency target (§5.3).

This feature aligns with the trade-off between latency and efficiency: for a traffic input, achieving a higher tail latency results in a higher per-core throughput, which means Quadrant can devote fewer CPU cores to serve traffic. This feature is important in the cloud context: Quadrant can use the right level of system resources to meet the latency SLO.

We note that Chain 1 and Chain 2 have tail latency close to *lower* latency thresholds. Chain 3 behaves differently: its tail latency stops increasing after its latency SLO is greater than 130 μs, because Chain 3 deployments have reached the per-core packet load limit. As we described in §5, Quadrant avoids executing chains close to its max per-core packet rate. For these cases, the per-core rate is high enough so that it is less beneficial to pursue a higher per-core efficiency at the cost of making the end-to-end latency unstable.

**SLO-adherence under traffic dynamics.** It is important that Quadrant works for different traffic inputs so we verify Quadrant's ability to control latency with such inputs. To do so, we deploy chains with a fixed latency SLO to see whether Quadrant can control latency with traffic dynamics. We gradually increase traffic by randomly accelerating a subset of flows by 30% of their packet rates for half of a flow duration. We vary the percentage of flows with an increased packet rate, and measure Quadrant's latency performance.

***Results.*** We show the tail latency under traffic inputs with different subsets of flows with an increased packet rate. (Figure 6) For all these cases, Quadrant is able to meet the tail end-to-end latency SLO; in fact, all groups achieve similar latency results regardless of the input.

## 6.5 Quantifying Isolation Overhead

**Spatial isolation overhead.** Spatial isolation overhead results from SR-IOV; we compare running a test NF with and without SR-IOV enabled. Our test NF is an Empty module so that it only involves swapping the dst and src Ethernet addresses of a packet to send it back.

***Results.*** Figure 7 shows running with SR-IOV adds only 0.1 us latency for both 80-byte and 1500-byte packets. We

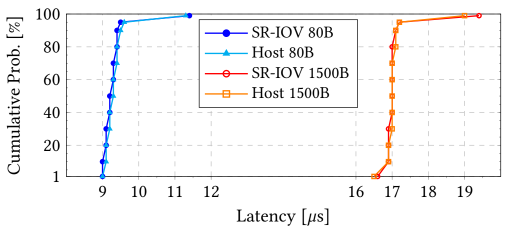


**Figure 7:** End-to-end latency CDF with SR-IOV on and off.

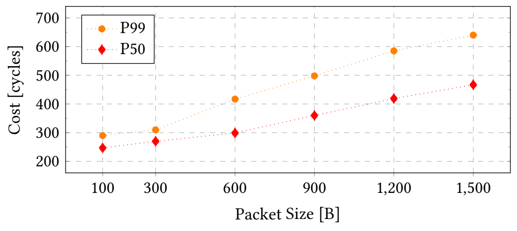


**Figure 8:** Per-packet cost of copying packets of different sizes

also find that the maximum throughput achieved by an SR-IOV enabled NIC ≥ 99.6% of the throughput achieved by a NIC running in a non-virtualized mode.

**Temporal isolation overhead.** NFs in a chain hand off packet ownership. Packet isolation requires that NFs in the same chain can only acquire packet ownership after its predecessor finishes processing it (§4.2). To quantify this temporal isolation overhead, we evaluate using a multi-NF chain.

***Results.*** Figure 8 shows the p50 and p99 CPU cycle cost for copying one packet of different sizes. The median cost to copy a 100-byte packet is 247 cycles and, for a 1500-byte packet, 467 cycles. This small difference is due to the cost of allocating a packet struct. Scheduling NFs cooperatively involves context switches between NF threads that belong to different NF processes. We profile the average cost of context switches between NFs: 2143 cycles per context switch. Note that this context switch cost is amortized among the batch of packets in each execution. For a default 32-packet batch, the amortized cost is only 67 cycles per packet. This cost is 27 | 14% of the cost for copying a 64 | 1500-byte packet respectively. Further, it is only 31% of the cost for forwarding a packet via a vSwitch with packet copying, as in EdgeOS.[12]

**Using `munmap`/`mmap` for transferring packet ownership.** For isolation we could have used `munmap` and `mmap` to explicitly manage the ownership of the shared packet buffer. `munmap` requires 4083 cycles, and `mmap` 8495 cycles. With all packets placed in the same memory page, we need one `munmap` and `mmap` to transfer the page to a different process. This costs significantly more (5.87×) than the context switch, justifying our approach to isolation.

[12]Quadrant has zero software packet switching cost because it uses the ToR switch and the NIC's L2 to dispatch packets to different chains.

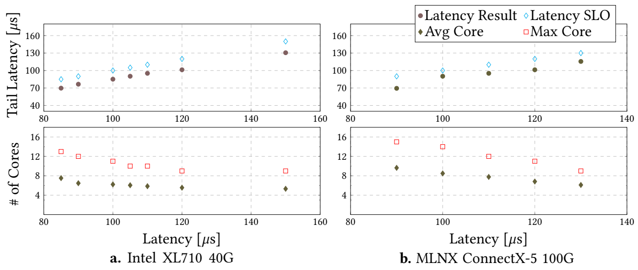


**Figure 9:** End-to-end tail latency and CPU core usage achieved by Quadrant (Chain 1) as a function of latency SLO.

## 6.6 Scaling to 40 and 100 GbE NIC

Cloudlab only supports OpenFlow for 10 GbE NICs, so most of our experiments use those. To show that Quadrant scales to 40/100 GbE NICs, we set up a separate two-node cluster: one node as the traffic generator and the other one as the Quadrant worker. The traffic generator is a dual-socket 20-core 2.2GHz Xeon E5-2630. The Quadrant worker is a dual-socket 16-core 1.7GHz Xeon Bronze 3106. Both servers have one 100Gbps single-port Mellanox ConnectX-5 NIC. The worker has one additional 40Gbps single-port Intel XL710 NIC. They connect to an Edgecore 100BF-32X (32x100G) switch. Experiments in §6.3 and §6.4 use this setup.

***Results.*** Figure 9 shows that, for Chain 1, Quadrant is able, as before, to adjust the number of cores used to serve traffic for different latency SLOs, and utilize all available cores on the worker to meet stringent SLOs, both for 40 GbE and 100 GbE. Chain 3 behaves similarly as Chain 1, but Chain 2, because it is CPU heavy, needs more cores than we have to saturate the NICs, so we have omitted the experiment. Overall, these experiments show that Quadrant's design scales seamlessly at higher NIC speeds.

## 6.7 Cooperative Scheduling

**Do we need the Cooperative Scheduler?** Quadrant's cooperative scheduler enables packet isolation, even for third-party NFs. A weaker form of isolation, assuming that NFs can be trusted, can be achieved using the Linux CFS scheduler, together with explicit handoff from one NF to another using shared memory (an NF sets a flag to indicate packets are ready to be processed by the next downstream NF). Unfortunately, this weaker alternative is also *slower* (Table 2); Quadrant w/ Cooperative Scheduler outperforms Quadrant w/ CFS by 40.7-95.2%. Note that the latter still outperforms NFVNice w/ pkt copy, as Quadrant does not require expensive cross-core packet copying for each inter-NF hop.

**Cache and TLB effects.** Cooperative scheduling involves context switches between NF processes in a chain. It can also flush caches and TLBs; we conduct an experiment to quantify these. We run the same test chain, with 5 BPF modules, as in §6.2 with four experimental groups: 1) Quadrant: the vanilla deployment w/o adaptive batch optimization; 2) Local mem: the vanilla deployment that operates on one dummy packet in the shared memory region; 3) Dummy NF: a chain of dummy NFs that do not process packets, but simulate NF cycle costs; 4) Single thread: a chain that runs in a single thread. The traffic generator produces traffic (1024B packets) to saturate the chain's NIC queue so that each chain runs at a batch size of 32, the NIC's default batch size. The TLB and cache misses are measured as the average value for a 15-second execution duration for 5 measurements.

| NF Chain Length | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Quadrant (coopsched) | 4700 | 2200 | 1520 | 1180 | 960 | 810 |
| Quadrant (CFS) | 3340 | 1530 | 980 | 680 | 520 | 415 |
| NFVNice w/ pkt copy | 2920 | 1230 | 815 | 545 | 425 | 350 |

**Table 2:** Per-core NF chain throughput (kpps) w/ and w/o coopsched.

| Metric | | Quadrant | Local mem | Dummy NF | Single thread |
|---|---|---|---|---|---|
| Per packet cycles | | 2846 | 2746 | 2730 | 2592 |
| Chain delay in cycles | p50th | 121536 | 116232 | 116008 | 85980 |
| | p99th | 128116 | 122324 | 117892 | 87492 |
| Misses | dTLB [count] | 72,864,218 | 68,259,827 | 61,251,661 | 1,185,591 |
| | dTLB [%] | 0.52% | 0.48% | 4.07% | 0.00% |
| | iTLB [count] | 11,542,564 | 12,325,256 | 9,615,381 | 591,737 |
| | iTLB [%] | 478.18% | 488.28% | 379.31% | 182.39% |
| Misses | LLC cache | 18,923,855 | 6,710,255 | 6,699,896 | 12,963,766 |
| | L1 dcache | 508,578,460 | 417,298,551 | 333,262,806 | 327,837,034 |
| | L1 icache | 41,272,281 | 37,127,027 | 30,856,178 | 17,568,605 |

**Table 3:** Overheads under isolation variants.

***Results.*** Table 3 shows NF runtime statistics. For all multiple-process groups, we see higher iTLB and dTLB misses. As shown, the number of dTLB misses is less than 1% of dTLB hits for cases that run a non-dummy NF, though dTLB misses are less important for an NF's performance. (Quadrant uses 2 MB hugepages for packet memory for each NF chain. Increasing this to 1 GB does not alter Quadrant's max throughput as there are few dTLB misses.)

All multi-process groups see higher iTLB misses compared to the single thread case because NF processes do not share code in memory. Local mem and dummy NF perform similarly in terms of per-packet cycle cost (and the number of cache misses) because Local mem processes one packet that resides in the chain's local memory and is likely to benefit from the L3 cache. Quadrant has a slightly higher per-packet cost compared to the other two multi-process cases. We find that the extra cost only comes from the first NF that copies incoming packets. The 2nd-4th NFs in 1)-3) have the same cycle cost (509 cycles / packet). These NFs benefit from L3 caching as the first NF's runtime loads when copying packets from the NIC's buffer to the per-chain packet buffer.

In the above four cases, two major differences explain the per-chain cycle cost: a) iTLB misses when deploying as a multi-process chain and b) L3 cache misses when processing network traffic. Quadrant incurs both; Local mem and Dummy NF only the former, and Single thread only the latter.

| NF Chain | Quadrant | Fixed-small (batch=32) | Fixed-medium (batch=128) | Fixed-large (batch=512) |
|---|---|---|---|---|
| Chain 1 | 306 | 246 | 305 | 260 |
| Chain 2 | 116 | 106 | 116 | 62 |
| Chain 3 | 322 | 260 | 313 | 142 |

**Table 4:** Per-core chain throughput (kpps) under different batch settings.

We calculate cycle cost for each for the 5-NF chain: for iTLB misses it is 254 cycles / packet (or 50.8 cycles / packet / hop), and cache misses add 100 cycles / packet. The former is extra overhead of a context switch, which could be reduced by tagged TLBs, while cache misses are unavoidable. Overall, the amortized TLB overhead is relatively small compared to the context switch itself.

**Batching to reduce overhead.** Quadrant uses batching to amortize context switching overheads, and estimates an appropriate batch size for an NF chain (§4). Here, we show Quadrant's batching by comparing the maximum per-core throughput produced under Quadrant's batching and other schemes that use a fixed batch size for different chains.

***Results.*** Table 4 demonstrates that Quadrant's batch setting performs significantly better than `Fixed-small` (which uses a small batch size of 32) and `Fixed-large` (batch size of 512) batch settings, and always produces a throughput that matches the highest throughput among all experimental groups. Surprisingly, using a large batch size decreases per-core throughput of NF chains.

## 7 Discussion

Quadrant can leverage prior work to scale and better support stateful NFs. We have left this to future work.

**Consistency model.** While Quadrant is consistent for per-flow state, it needs to be extended to ensure consistent updates to global state (*e.g.,* per-device packet counts in 4G/5G Evolved Packet Core (EPC)). Prior work (S6 [55]) has explored mechanisms to ensure eventual consistency of global state, and can be incorporated into Quadrant.

**Ingress scalability.** Quadrant's ingress runs as software, and installs rules in harware switches, for when OpenFlow becomes available. Fastpass [41] has demonstrated that software-based per-flow routing is feasible and efficient even at datacenter scale. SNF [49] used software ingress, and it also showed that its implementation incurs negligible latency and can scale-out dynamically to adapt to traffic volume. Quadrant can incorporate these to scale better.

**Hardware ingress.** Modern hardware switches have enough resources for per-flow rules (e.g., NoviFlow switches have 225K entries [15]). Finally, in a multi-tenant cloud environment, Genesis [51] has explored managing flow space of hardware switches for isolation.

## 8 Related Work

FaaS is widely available on cloud platforms [7, 10, 31, 47], and as open-source [12, 34]. They are designed for latency-insensitive applications, not for NFV. Research on FaaS has two styles. The first improves aspects of FaaS. Sock [32] and LightVM [29] optimize sandbox startup time. SAND [1] optimizes IPC performance. E3 [28] accelerates FaaS execution with SmartNICs. The second explores new applications made possible by FaaS, such as real-time big data analysis [18, 43] and video encoding [6].

On NFV/FaaS, SNF [49] proposed FaaS to execute NFs. It isolates NFs via LXC containers and stores NF state via a key-value store. However, SNF does not support NF chains, nor does it ensure latency SLOs (its 99%-ile latency in some cases is 2.8 ms [49], an order of magnitude more than Quadrant).

NFV frameworks [14, 19, 37, 40, 56] deploy and orchestrate fast NF chain execution. E2 [37] deployed NF chains using commodity servers without isolation. NetBricks [40] isolates NFs with a safe Rust runtime, requiring NF vendors to use Rust. In contrast, most commercial NFs are packaged containers or VMs without source [39]. EdgeOS [44] employs an expensive isolation via packet copying for each NF. Some work leverages specialized hardware [14, 17, 22, 24, 56, 57]. Quadrant sees NFV's fundamental motivation as reducing deployment cost and complexity, and does not use specialized hardware. Some work [19, 23, 25, 36, 52] examines dirty-slate solutions to optimize NFV's performance. They often work in a restricted setting without considering multi-tenant cluster deployments; that said, this line of work is largely compatible with Quadrant and these insights have been and could be further integrated. Another line of research [13, 49, 55] considers optimizations for stateful NFs. Quadrant supports stateful NFs; such prior work can also be integrated.

## 9 Conclusions

Quadrant supports NFV in cloud computing environments using commodity software and hardware, fulfilling NFV's original ambitions. It extends cloud abstractions, and eases the deployment of third-party NFs. With its spatiotemporal packet isolation, it outperforms state-of-the-art NFV platforms that use alternative isolation mechanisms, and performs as well as custom NFV systems that do not provide NF isolation.

**Acknowledgements.** This work was supported in part by US NSF grant CNS-1705086, the CONIX Research Center (one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA), the CNPq 309496/2019-5, FAPESP 2020/05131-0, FAPEMIG APQ-02400-18, the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund (project 2019-2.1.11-TÉT-2020-00183 under the 2019-2.1.11-TÉT funding scheme), USC's Annenberg Fellowship, Intel, and Cisco.

## References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[2] BESS (Berkeley Extensible Software Switch). 2022. The official BESS Github repository: https://github.com/NetSys/bess.

[3] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/2934872.2934875

[4] Digital Corpora Enterprise Network Traces. 2009-2022. Available at https://downloads.digitalcorpora.org/corpora/scenarios/2009-m57-patents/net.

[5] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[6] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[7] The Apache Software Foundation. 2017. Apache OpenWhisk. http://openwhisk.org/. Accessed on May 2020.

[8] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Online, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[9] Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali José Mashtizadeh. 2020. Fault Tolerant Service Function Chaining. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 198–210. https://doi.org/10.1145/3387514.3405863

[10] Google. 2017. Google Cloud Functions. https://cloud.google.com/functions/. Accessed on May 2020.

[11] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 301–314. http://www.usenix.org/conference/atc19/presentation/hedayati-queue

[12] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO, 1–6. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[13] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 97–112. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[14] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, JR. Gerald Q. Maguire, and Rebecca Steinert. 2021. Metron: High-Performance NFV Service Chaining Even in the Presence of Blackboxes. *ACM Trans. Comput. Syst.* 38, 1–2, Article 3 (July 2021), 45 pages. https://doi.org/10.1145/3465628

[15] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/conference/nsdi18/presentation/katsikas

[16] Georgios P Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q Maguire Jr, and Dejan Kostić. 2016. SNF: Synthesizing high performance NFV service chains. *PeerJ Computer Science* 2 (2016), e98.

[17] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/2741948.2741969

[18] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[19] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. 2017. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/3098822.3098828

[20] Surendra Kumar, Mudassir Tufail, Sumandra Majee, Claudiu Captari, and Shunsuke Homma. 2017. *Service Function Chaining Use Cases In Data Centers*. Internet-Draft draft-ietf-sfc-dc-use-cases-06. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-sfc-dc-use-cases-06 Work in Progress.

[21] Patrick Kutch. 2011. PCI-SIG SR-IOV Primer an Introduction to SR-IOV Technology Intel® LAN Access Division, Jan. 2011, 321211-002.

[22] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T. V. Lakshman. 2017. *UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing*. Association for Computing Machinery, New York, NY, USA, 506–519. https://doi.org/10.1145/3127479.3132252

[23] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor Retvari. 2020. Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 633–649. https://www.usenix.org/conference/nsdi20/presentation/levai

[24] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016.

ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2934872.2934897

[25] Yang Li, Linh Thi Xuan Phan, and Boon Thau Loo. 2016. Network functions virtualization with soft real-time guarantees. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. IEEE, San Francisco, CA, USA, 1–9. https://doi.org/10.1109/INFOCOM.2016.7524563

[26] Linux. 2021. BPF Documentation. https://www.kernel.org/doc/html/latest/bpf/index.html.

[27] John DC Little. 1961. A proof for the queuing formula: L= $\lambda$ W. *Operations research* 9, 3 (1961), 383–387.

[28] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 363–378. https://www.usenix.org/conference/atc19/presentation/liu-ming

[29] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[30] D. Meyer. 2021. MWC Barcelona Shows the Cloud Has Eaten Telecom. https://www.sdxcentral.com/articles/news/op-ed-mwc-barcelona-shows-the-cloud-has-eaten-telecom/2021/07/.

[31] Microsoft. 2017. Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed on May 2020.

[32] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[33] ONAP Platform Architecture. 2018. https://www.onap.org/architecture.

[34] OpenFaas. 2021. https://www.openfaas.com/.

[35] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[36] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[37] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/2815400.2815423

[38] Palo Alto Networks. 2018. Disable Firewall offloading traffic. https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000Cm8cCAC.

[39] Palo Alto Networks. 2021. Tomorrow's Network Security Stops Threads at Scale. https://www.paloaltonetworks.com/prisma/vm-series.

[40] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 203–216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[41] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 307–318. https://doi.org/10.1145/2619239.2626309

[42] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 201–216. https://www.usenix.org/conference/nsdi18/presentation/poddar

[43] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[44] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. 2020. Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Online, 927–942. https://www.usenix.org/conference/atc20/presentation/ren

[45] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Seattle, Washington) *(LISA '99)*. USENIX Association, USA, 229–238.

[46] Salvatore Sanfilippo. 2009. Redis. https://github.com/redis/redis.

[47] Amazon Web Services. 2017. AWS Lambda. https://aws.amazon.com/lambda/. Accessed on May 2020.

[48] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 13–24.

[49] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. 2020. SNF: Serverless Network Functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 296–310. https://doi.org/10.1145/3419111.3421295

[50] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 33–46. https://www.usenix.org/conference/nsdi19/presentation/stephens

[51] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: Synthesizing Forwarding Tables in Multi-Tenant Networks. *SIGPLAN Not.* 52, 1 (jan 2017), 572–585. https://doi.org/10.1145/3093333.3009845

[52] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 283–297. https://www.usenix.org/conference/nsdi18/presentation/tootoonchian

[53] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with EBPF and XDP: Concepts,

Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (Feb. 2020), 36 pages. https://doi.org/10.1145/3371038

[54] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[55] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 299–312. https://www.usenix.org/conference/nsdi18/presentation/woo

[56] Jane Yen, Jianfeng Wang, Sucha Supittayapornpong, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. 2020. Meeting SLOs in Cross-Platform NFV. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 509–523. https://doi.org/10.1145/3386367.3431292

[57] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Online, 1083–1100. https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng