Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization

Wei Zhang
The George Washington University
District of Columbia, USA
zhangwei1984@gwu.edu

Jinho Hwang
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
jinho@us.ibm.com

Shriram Rajagopalan
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
shriram@us.ibm.com

K. K. Ramakrishnan
University of California, Riverside
California, USA
kk@cs.ucr.edu

Timothy Wood
The George Washington University
District of Columbia, USA
timwood@gwu.edu

Abstract

The combination of Network Function Virtualization (NFV) and Software Defined Networking (SDN) allows flows to be flexibly steered through efficient processing pipelines. As deployment of NFV becomes more prevalent, the need to provide fine-grained customization of service chains and flow-level performance guarantees will increase, even as the diversity of Network Functions (NFs) rises. Existing NFV approaches typically route wide classes of traffic through pre-configured service chains. While this aggregation improves efficiency, it prevents flexibly steering and managing performance of flows at a fine granularity.

To provide both efficiency and flexibility, we present Flurries, an NFV platform designed to support large numbers of short-lived lightweight NFs, potentially running a unique NF for each flow. Flurries maintains a pool of Docker container NFs–several thousand on each host–and resets NF memory state between flows for fast reuse. It is based on a hybrid of polling and interrupt driven approaches that improve throughput and latency while allowing multiple NFs to efficiently share CPU cores. By assigning each NF an individual flow or a small set of flows, it becomes possible to dynamically manage the QoS and service chain functionality for flows at a very fine granularity.

Our Flurries prototype demonstrates the potential for this approach to run as many as *80,000 Flurry NFs during a one second interval*, while forwarding over 30Gbps of traffic, dramatically increasing the customizability of the network data plane.

Keywords

Cloud Computing, Network Function Virtualization, Scheduling, Resource Management

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoNEXT '16, December 12-15, 2016, Irvine, CA, USA
© 2016 ACM. ISBN 978-1-4503-4292-6/16/12. . . $15.00
DOI: http://dx.doi.org/10.1145/2999572.2999602

Network Function Virtualization (NFV) runs in-network services in software, as virtual machines or containers on commodity servers. NFV combined with Software Defined Networking (SDN) seeks to transform networks from purpose-built hardware appliances to a predominantly software base, increasing flexibility and providing a path for the evolution of network capabilities, possibly at lower cost. NFV promises to support a rapidly evolving network with dynamic instantiation of network services as needed in the path of a flow [1].

However, existing NFV platforms do not fully bring the per-flow customization supported by SDNs to the data plane layer. NFs are still often deployed as monolithic processing entities that process a queue of packets from many different flows. This prevents per-flow customization of processing since NFV platforms must multiplex a small number of NFs for thousands of different flows. This aggregation limits flexibility and prevents the deployment of diverse processing pipelines customized for each customer [2].

A further challenge for NFV is providing flow-level performance management and isolation. Hardware-based middleboxes implement complex queueing structures to provide per class service. The holy-grail is to offer per-flow queuing and service to provide true isolation at large scale. When network functions are implemented in a software platform, providing the same level of isolation is more challenging because of dynamic bottlenecks exacerbated by complex service chains where processing times vary by function and flow. A system that provides a high degree of isolation across flows in a scalable manner, while incorporating appropriate scheduling mechanisms is very desirable. This would allow for the platform to provide in-network services dynamically placed with little concern on the impact on the flows in the network. At a minimum, this needs to be provided to each service class of flows, yet current NFV platforms take a function-centric view on scheduling and isolation–they do not focus on the flows themselves.

When we look at the application ecosystem in cloud environments, it is undergoing dramatic changes with the goal of increasing the flexibility and efficiency with which services can be deployed and isolated from one another. Technologies such as Docker and LXC containers are gaining popularity as an alternative to heavier weight server virtualization platforms. The shift towards microservices breaks applications down into modular components that can be more flexibly developed, deployed, and customized based on a user's needs. This can be taken to the extreme with cloud platforms such as Amazon Lambda, which allow microservices to be further divided into functions that are instantiated and executed on demand for each incoming request. This form of "transient computing" transforms the provisioning time of cloud services from hours to milliseconds.

These approaches are eminently suitable to be used in a NFV environment as a way to increase flexibility while providing fine grained isolation [3, 4]. Breaking network functions down into their component pieces allows for more flexible service chaining depending on the precise needs of a flow. From a network service provider or an enterprise administrator's perspective, being able to download and instantly deploy a container image of a network service dramatically simplifies manageability when compared to past approaches that relied on specialized hardware or monolithic software middlebox implementations. The elasticity of light-weight container-based NFs has the potential to extend this even further by allowing a vast array of NFs to be instantiated and customized on demand for different customer flows.

The challenge in bringing these concepts to an NFV platform is that network services have strict performance requirements so as to function essentially as a 'bump in the wire' [5–7]. Scalability is also a major challenge since current NFV platforms achieve high performance only by dedicating resources (i.e., CPU cores and memory) to a small number of functions running on each server. Current approaches are simply unable to offer high throughput, low latency, and efficient resource usage when running a large number of NFs.

We have developed a NFV platfrom called Flurries to tackle these challenges. Akin to flurries in the real world, NFs in Flurries are extremely transient. They last for the lifetime of a flow, and the platform is carefully architected to support thousands of unique Flurry NF containers executing at the same time. Flurries allows network operators to extend the flow-level routing customization provided by SDNs to the data plane layer. This provides greater flexibility in how processing pipelines are deployed and improves QoS management by making flows a first class scheduling entity. Our transient NF approach also enables entirely new functionality such as zero-downtime NF upgrades. Our contributions include:

- A Docker Container-based NFV platform that uses Intel's DPDK [8] and shared memory between NFs for efficient packet processing.
- A novel hybrid software architecture that uses polling to rapidly retrieve packets from the NIC and interrupt driven NF processing to allow a large number of functions to share each CPU core.
- Fine grained mappings of flows to NFs that allow per-flow customization of processing and scheduling.
- An adaptive scheduling scheme that dynamically adjusts batching to improve latency for interactive flows and reduces context switch overheads for throughput-based flows.

We have developed Flurries on top of our DPDK-based OpenNetVM platform [9]. Our evaluation shows how Flurries improves performance of service chains and allows customizable priority for different customer flows. Our prototype demonstrates the potential for this approach to run nearly 80,000 unique NFs over a one second interval, while maintaining 30 Gbps throughput for real web traffic.

2. MOTIVATION

In this section, we highlight the recent trends in the industry that motivated the design of Flurries. We then outline how the current advances in the NFV community is lagging behind these industry trends and is sometimes at odds with the enterprise requirements.

2.1 Industry Trends

The following trends in the industry have dramatically changed the way in which organizations design, deliver and operate their services in the cloud.

Docker. Container-based process isolation has existed for a long time in mainline Linux distributions. However, Docker and similar emerging technologies have taken the concept one level further. The application, its configurations, software dependencies, and the filesystem are packaged and distributed as one single unit. Traditional enterprises and startups alike are shifting away from using heavy-weight virtual machines and towards packaging their applications as light-weight containers that can be scaled and distributed much more efficiently than VM images.

Microservices. Driven by the need to quickly respond to market needs, enterprises and startups alike have adopted modern software practices such as DevOps and Microservices [10, 11] . Applications are no longer delivered as monolithic pieces of software that are updated once or twice a year. The microservice architecture breaks an application into a collection of small components, each implementing a single service, that communicate over the network to provide the end-to-end application functionality.

The microservices architecture approach eenables enterprises to deliver software updates to or or more components immediately and continuously. When combined with the packaging ease of Docker containers, these practices have enabled even traditional enterprises to update and deploy their cloud deployments hundreds of times a day [12–14] .

Extensive automation. The key enabling factor behind such frequent deployments is automation. The "infrastructure as code" approach to automation has enabled the entire data center deployment to be expressed and controlled through software. The goal is to create an immutable infrastructure, i.e., one where automation and modularity allow any component of the infrastructure to be fully replaced by a new version. When one or more components misbehave or fail, their associated Docker containers are simply thrown away and restarted or upgraded. This is in contrast to past approaches where systems organically evolved over time, often leading to overly complex and unwieldy components.

2.2 Making the Data Plane Agile

Given the agility and automation that is prevalent throught the industry, we posit that NFV environments should be equally agile and automated. In fact, NFV solution providers have already begun movining in this direction [3, 4]. Despite recent advances in NFV, much of the ecosystem requires careful tuning, setup and configuration and professional services to manage the systems. None of these characteristics suit the modern enterprise infrastructure.

While one might argue that container based NFVs cannot provide as strong isolation as virtual machines since they rely on OS-level process address space protection, we posit that this is a small trade-off, especially in NFV environments where applications are likely to be trusted or thoroughly vetted.

Currently, SDN enables fine grained routing by allowing traffic to be steered at the flow level. Each flow can be flexibly routed based on the particular needs of the user or the application being accessed. Unfortunately, the flexibility of flow routing cannot fully extend to the network function layer because NFV platforms are ill-equipped to support large numbers of unique functions: *they provide neither the abstractions necessary to isolate and process flows with fine granularity nor the resource efficiency to effectively run large numbers of distinct NFs*.

Many NFV platforms, such as those based on the popular DPDK library rely on polling to recieve and process packets, meaning that each NF must be dedicated CPU cores [15, 16]. Our evaluation reveals that even interrupt-driven I/O libraries such as netmap perform poorly when a modest number of NFs, e.g., five or more, share a CPU core. As a result, the number of NFs that can be safely run on a single server is approximately equal to its number of cores. This is far less than the number of flows or even traffic classes likely to be seen by a server. Therefore, NFV servers typically run a small number of general purpose functions. Each NF receives packets from many different flows, and it is typically the NF's responsibility to divide those packets into flows. This leaves the onus on the NF to prioritize processing of packets at the flow level (e.g., handling video traffic first) and to maintain a state table with data for each flow (e.g., a TCP state machine for each flow), as illustrated at the top of Figure 1.

Flurries takes the fine grained *routing* flexibility offered by SDNs and extends it to *processing* at the network function layer. This allows unique data plane processing entities to be spawned and customized for each flow, enabling far greater diversity in the number and types of NFs that can be run on each server, as shown at the bottom of Figure 1. Deploying a unique NF for each flow provides many benefits. Some of these are:

Application-specific Networking: Our approach allows for more flexible and powerful customization of flow processing, since a particular middlebox function may want to distinguish how it processes different flows based on their characteristics. For example a flow classified as malicious may be treated differently than a benign flow, or 'elephant' flows may be treated differently than 'mice'. Flurries vastly expands the number of different processing pipelines that can be run on a server, making it feasible to carefully customize processing for each destination application, customer sending traffic, or n-tuple in general. While current NFV platforms are designed to run a handful of different NFs over the time scale of hours, our evaluation shows how Flurries can run tens of thousands of different NFs *each second.*

Multi-Tenant Isolation: A cloud or network provider may wish to host network functions from a variety of vendors or for a variety of customer flows. In either case, it is important for NFs to be kept cleanly isolated since different vendors may be unwilling to have their proprietary functions be combined with code from other vendors. Similarly, clients whose flows are being processed by NFs may demand that their packet data be kept isolated from other flows and functions they do not trust. Supporting this type of isolation is impractical and inefficient unless many distinct NFs can be run on each server.

Simplified NF development: The use of single flow-specific NFs that last only as long as the flows they process can simplify their state management and application logic. This approach moves complex flow tables outside of the NF and into the management layer, allowing the NF to focus only on packet processing.

Flow-Level Performance Management: By dedicating a logical processing 'entity' to each flow, it becomes easier for the management system to prioritize and schedule NFs to achieve flow-level performance goals. For example, when service times for packets of different flows are different at each of the NFs, and the processing of a flow on the platform involves a service chain of multiple NFs, bottlenecks may dynamically form within the chain. Such dynamic variability can be difficult to provision against. This results in interference between flows, with the potential for excessive delay and potential packet loss for flows being processed in the service chain. Having isolation on a per-flow basis, provided by having flow-specific NFs can significantly mitigate the effect of such dynamic bottlenecks. Flurries changes the object being scheduled from a heavy-weight network function, to a light-weight flow at a particular stage of its execution.

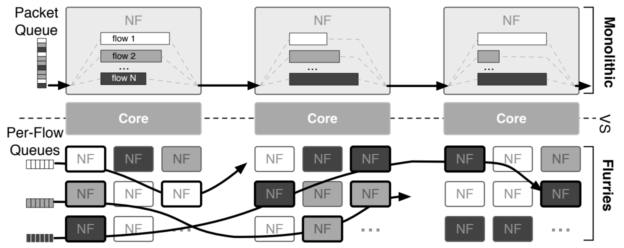


Figure 1: Flurries splits monolithic NFs (top) into light weight, flow-specific functions (bottom). This simplifies NF logic, since they don't need to differentiate flow types, and enables more flexible chaining and resource management.

3. DESIGN

To support a large number of NFs, the Flurries system design strikes a healthy balance between the two polar opposites of packet processing system designs: busy polling and interrupts. To achieve performance isolation across flows, we resort to containerization and leverage the Linux scheduler to appropriate resources across NFs. In addition, Flurries adopts a simple thread pool design pattern in order to minimize the launch time per container by "recycling" NFs. We describe each of these design concepts in detail in the following subsections.

3.1 Hybrid Polling-Interrupt Processing

Poll-based platforms like DPDK [17] provide the ability to quickly receive large numbers of packets from the wire into the system. Such platforms have been known to scale to extremely high bandwidth systems on the order of 40Gbps. Such scalability also comes at a cost: the CPU performing the polling is at 100% usage at all times. To support a large number of NFs running simultaneously, a poll-based approach would restrict the concurrency to the number of cores available in the system.

Interrupt based packet processing [18, 19] allows for a large number of NFs to be active simultaneously, while only a handful are actually processing packets. However, interrupt based approaches cannot scale to extremely high throughput environments (e.g., 40Gbps).

Flurries design marries the processing capacity of poll-based techniques and the resource efficiency of interrupt-based techniques to achieve the best of both worlds. Polling at the NIC-level lets us get large numbers of packets quickly into the system. Interrupt based NF management lets us divide up those packets and only wake up the NFs that actually need to process them.

3.2 An NF per Flow

A distinguishing feature of Flurries platform is its *NF per flow* processing model, in addition to the *NF per traffic class* model. In the *NF per flow* mode, each NF runs as a transient process managing a single flow at a time. When packets are received from the NIC, a flow table lookup determines if this is the start of a new flow. If it is, then a free NF is allocated out of an idle pool of NFs to handle the flow. If the packet matches an existing flow, then it is directed from the management layer to the correct NF.

The lifetime of an NF is that of its respective flow. For TCP-based flows, when a flow terminates, the NF is recycled back into the idle pool. To identify the NFs that can be recycled, Flurries tracks the state of TCP flow using lightweight state tracking. As flows terminate, Flurries invokes a cleanup callback handler provided by the NF, wherein the NF is expected to cleanup any residual state associated with the flow. Flurries defers responsibility of state management to the NF itself. Such functionality can be achieved using state management frameworks for middlebox applications [20].

3.3 NF Containers

In order to run a large number of NFs (on the order of 1000s), while keeping the design lightweight, we utilize process-based NFs, where each process handles a network flow. While the lifetime of a process transcends that of a flow, the processes themselves are expected to voluntarily yield the CPU between batches of packets, and to reset themselves when the flow finishes.

Flurries uses containerized NFs to support manageability. We assume that a set of trusted NF providers package NF container images with all necessary dependencies, file system layers, etc., as a Docker container that can simply be installed and run on the provider's platform with minimal configuration and management effort.

Flurries chooses to use containers because they provide NF developers a way to combine code and dependencies with a lightweight execution environment. This is in contrast to approaches such as Click [21], Cisco's VPP [22] and Bess [23] in which the NFV management framework contains the execution environment in the form of a set of worker threads, and NFs are simply code libraries that are called by the framework. We believe our approach is more flexible for NF developers since they have greater control over the environment that will process their functions and get greater isolation from other NFs.

Unlike the traditional Docker containers, our security model does not enforce complete memory isolation between NF containers. The address space within each NF process is restricted, protecting their code and private data, but all NFs map in a shared memory region from the manager that hosts packet data. This enables zero-copy I/O across NF service chains as described in the next section.

4. ARCHITECTURE & IMPLEMENTATION

The Flurries architecture is split between the Flurries Manager that interfaces with the NIC and moves packets, and the set of Flurry NFs, which wait in the idle pool until they have a flow to process, as illustrated in Figure 2. We first provide a high level overview of our architecture and then discuss the implementation details of the key components.

When packets arrive at the NIC, the RX threads in the Flurries Manager use DPDK's poll mode driver to bypass the operating system and read packets into a memory region shared with all NFs. The RX thread then performs a lookup in the Flow Table to determine if this is a new flow; if so, the packet is directed to the special Flow Director NF that is responsible for finding an Idle NF to handle processing. Once a flow is matched to an NF, packet descriptors are copied into the NF's receive ring buffer and the Wakeup subsystem decides whether to transition the NF process into the runnable state.

Each NF is compiled together with libFlurry, which handles the interface with the manager for sending and receiving packets. libFlurry also keeps track of TCP state to detect when the current flow ends so the NF can be recycled. When the NF is scheduled to run, libFlurry will check for new packets in its receive ring and call the designated handler function (e.g., an intrusion detection function) for each one. When its receive ring is empty (or it has watchdog timer expire), libFlurry puts the NF to sleep so that another process can use the CPU. When libFlurry detects the last packet in a TCP flow, it notifies the Flow Director NF that it has become idle, resets its state, and goes to sleep.

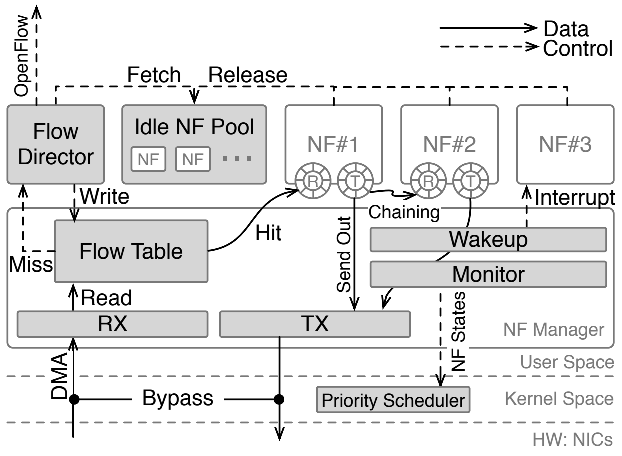


Figure 2: The Flurries architecture bypasses the OS and uses polling for high performance I/O. RX threads do Flow Table lookups to route existing flows, and new flows are assigned an NF by the Flow Director. The Wakeup system alerts NFs when they have packets available, allowing many NFs to efficiently share the CPU.

Flurries Manager Initialization: When the Flurries Manager starts, it initializes the DPDK EAL (Environment Abstraction Layer), allocates the shared memory pool used to store packets, and preallocates a set of RX and TX rings for the first batch of NFs added to the idle pool. The Manager is also responsible for configuring the NIC ports to initialize one queue for each RX and TX thread.

Wherever possible, Flurries preallocates memory buffers to avoid any dynamic memory allocations within the critical packet processing path. The memory pools used to store packets as well as the receive and transmit ring buffers created for each NF are all stored in a shared memory region backed by huge pages. This allows the Manager to provide zero-copy packet movement between NFs since only a small packet descriptor needs to be copied into the ring of an NF for it to be able to access a packet. Flurries, like the OpenNetVM platform it is based on [9], relies on lock-free ring buffers for all inter-thread and most inter-process communication. This eliminates expensive mutual exclusion and system call overheads.

Once memory is allocated and the NIC has been configured, the Manager starts the RX, TX, Monitor, and Wakeup threads, which are described in detail below. Multiple RX or TX threads can be used if necessary—we have found that one RX thread and one TX threads are sufficient to handle a 10 Gbps arrival rate that is distributed across several thousand individual NFs. When using multiple RX threads, Flurries enables Receive Side Scaling (RSS) in the NIC using a symmetric hash to ensure that all packets for a given TCP flow will be sent to the same RX thread for processing.

RX and TX Threads: RX threads use polling to pull batches of packets from the NIC into the shared memory pool. The RX thread then does a flow table lookup for each packet to determine if it is for a new or old flow. As part of this lookup, the RX thread updates

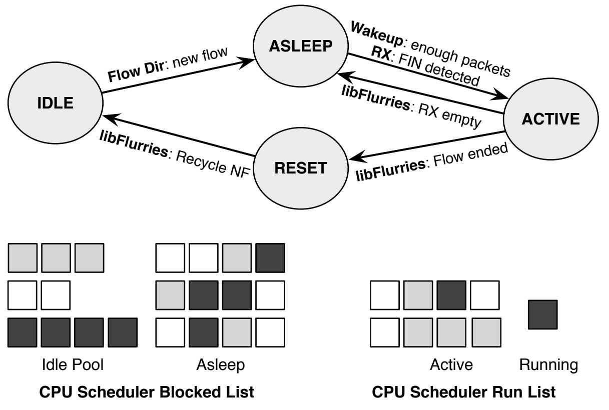


Figure 3: Flurry NFs transition between states depending on whether they are currently in use and the number of packets in their queues. The mapping at bottom shows how NFs for different service types (colors) are mapped to different states, and states are mapped to the CPU scheduler queues.

statistics for the flow which are used by Flurries to differentiate between long versus short flows. Since we use RSS, flows are consistently distributed across RX threads. This allows us to partition the flow table across RX threads to avoid contention since two RX threads will never need to perform lookups for packets from the same flow.

An optimization we add is for the RX thread to check for FIN or RST flags that indicate the end of a TCP flow. If either of these flags are observed, the RX thread updates the meta data for the flow so that it will be woken up immediately. We add this optimization in order to avoid the case where packets for the end of a flow arrive at an NF, but it is not immediately scheduled to handle them, delaying when sockets can close and when the NF can be recycled. Having RX threads perform full TCP state tracking would incur too high a cost since they are on the critical path, but checking for these flags is a low overhead way to schedule the NF so that libFlurries can determine if the flow is really ending.

TX threads read a batch of packets from an NF's transmit ring and do flow table lookups on each packet to determine whether they should be sent to another NF or out a NIC port. To minimize cache contention and avoid locks, TX threads only read from the flow table, and never make updates to the table or packet meta data. Like RX threads, TX threads use polling and are dedicated a core. This is necessary since a small number of TX threads are used to move millions of packets per second between the different NFs that make up the processing pipelines. NFs are partitioned across TX threads using *NF_ID % number of TX threads*; this keeps a consistent mapping since the number of TX threads is static but the number of NFs may fluctuate. Since Flurry NFs are so transient and lightweight, we have found that this leads to a reasonably even workload distribution.

Wakeup Thread and Monitor: Although the Flurries Manager relies on polling and dedicated CPUs to achieve high overall performance, individual NFs need to be carefully woken up and put to sleep so that they can efficiently share CPU cores. The Manager's Wakeup thread is responsible for checking the receive queues of each NF to determine if there are sufficient packets to warrant waking up the NF. We use an adaptive threshold on the number of packets to decide whether to wake up, and a timeout to avoid starvation, as described in Section 5.1.

While one can think of Flurry NFs as being "interrupt driven," that is not completely accurate since interrupts are not sent for each packet. Instead, Flurry NFs go to sleep by waiting on a linux semaphore used specifically for that NF. When the Wakeup thread finds that the number of packets in the receive queue for an NF exceeds the threshold or the NF's timeout has expired, it marks a flag in the flow entry to indicate the NF is being woken up and signals the semaphore. This transitions the NF to the ACTIVE state, as shown in Figure 3, which means it will be moved from the CPU scheduler's blocked list to the run queue. Once the NF is selected as the running process, libFlurries will update a flag in the NF meta data to indicate it has woken up. This prevents any further signals from the Wakeup thread until the NF empties its queue and transitions back to the ASLEEP state. This protocol ensures that only one semaphore signal will be performed between the time when the queue size first exceeds its threshold and when that batch is fully processed.

The Monitor periodically gathers aggregate statistics about NFs for display on the console and to provide to an NF resource management system. This thread does not need to be dedicated a core since it only runs periodically.

Flow Director: The Flow Director is a special NF that maps incoming flows to new NFs. When an RX thread has a flow table miss for a packet, it sends the packet to the Flow Director to determine how it should be processed. The Flow Director can use an internal rule set, or communicate with an SDN controller using OpenFlow to decide what service chain of NFs a new flow should be sent through. It then adds a new entry to the flow table and initializes the statistics for the flow. The flow table is shared with the RX and TX threads. To avoid locks, only the Flow Director adds or removes entries, while only the RX and Wakeup Threads update the statistics within a flow entry.

If Flurries is used with an NF for each flow, then the TCP handshake ensures that only the very first packet in a flow will go to the Flow Director. By the time the second packet in the flow (the server's SYN-ACK) arrives, the flow table rule will have been installed and the RX thread will deliver it and all subsequent packets directly to the NF. This minimizes the work done by the Flow Director, although we still run it in polling mode on a dedicated core to ensure efficient flow startup.

The Flow Director is responsible for tracking the pool of idle NFs. As shown in Figure 3, it maintains a list of idle NFs for each service type (e.g., IDS, proxy, VPN gateway, router, etc.); NFs are added to the appropriate list when their flow terminates, and NFs are recycled in FIFO order. Idle NFs are "blocked" processes that will not be scheduled on the CPU. They consume relatively little memory since the majority of the memory needed by an NFV platform is for the packet buffer pools, which are shared across all NFs by the NF manager. In order to run thousands of processes concurrently, some Linux and Docker configuration parameters need to be adjusted, e.g., to increase the number of open file descriptors and process IDs. While concurrently running thousands of processes is generally ill-advised when they are contending for resources such as the Linux networking stack, our approach alleviates these issues since NFs typically use entirely user-space processing, and only interact with the network via libFlurries which provides efficient zero-copy I/O to the management layer.

libFlurries and Container-based NFs: Each NF is a process that can be encapsulated within a container. NFs are implemented with the help of libFlurries, by registering call backs that NF developers define to handle a packet and clear their state at the end of a flow. The libFlurries API provides functions for NFs to perform common packet operations such as manipulating the header and forwarding packets out NIC ports or to other NFs. After initialization, NFs enter the libFlurries main loop, which checks the receive queue for a batch of packets, issues call backs to process them if available, and then sleeps on the NF's semaphore if the queue is empty.

In order to access the receive/transmit queues and packet buffer pools, the NF uses the DPDK library to map the shared huge page region at start up. This is achieved even in container environments by granting the container access to the Linux huge page file system (hugetlbfs), which exposes a file-based API to map the memory. The NF will map the huge pages with the exact same address mappings as the Flurries Manager by reading the configuration of virtual and physical addresses from the /var/run/.rte_config file. This ensures that the manager and all NFs can use identical addresses when referencing packets and ring buffers, eliminating the need to translate addresses when moving between the host and containers.

After each packet is processed by the NF's packet handler code, libFlurries checks the header bits in the packet to maintain a lightweight TCP state machine for the flow. Flurries only needs to detect the end of a flow by watching for FIN and RST flags, although we also have it track SYN flags at the start of each flow for debugging purposes. Accurately finding the end of a flow within the NF is critical, since otherwise NFs will not be able to free themselves. Once the end of a connection is found, libFlurries calls the NF's reset callback to ensure that relevant memory is cleared before it is recycled for a new flow.

Correctly tracking the end of a TCP connection can be a challenge, especially since occasionally extraneous RST, FIN, or ACK packets may be retransmitted. If such packets arrive after an NF has been reset, Flurries currently drops the packets since it views the connection as closed. This could be avoided by adding a delay before NFs are reset. A timeout could also be used to detect the end of flows that do not cleanly shut down, similar to how SDN switches use timeouts to expire flow rules.

API and Extensibility The libFlurries API is designed to abstract most platform management functions away from the NF developer. The primary functions used in libFlurries are listed below:

Function Name	Description
register_pkt_handler	register packet handler defined by NF developer
register_state_cleaner	register state cleaner defined by NF developer
sleep_nf	wait for interrupt message from wakeup thread
flow_close	detect flow close based on FIN and RST flags
adjust_prio	adjust NF priority based on flow characteristic

The two key functions which a developer must implement are the `pkt_handler` and `state_cleaner` functions, which are then registered with the functions listed above. The first function is called for each new packet that arrives, while the latter is used to clean up any NF-specific state when the flow ends and the NF must be decommissioned. While the other functions listed here are not currently exposed to NF developers, they could be extended to add new features to the platform, for example to allow NFs to request changes to their own priority or to define alternative triggers to close a flow for non-TCP traffic.

The Flurries Management layer includes the following:

The `wakeup_nf` and matching `sleep_nf` function in libFlurries are currently implemented using semaphores, however, other communication mechanisms could be considered, as discussed in [24]. Flurries currently assumes NFs of the appropriate type for any new flows are already running (although possibly busy). The `start` and `destroy_nf` functions could be extended to automatically instantiate the appropriate type of NFs based on requests from an SDN controller, or to build a more dynamic NF management system that adjusts the number of idle NFs at runtime, as discussed in the following section.

Function Name	Description
wakeup_nf	determine if an NF must be woken up
adjust_thresh	adjust the wakeup threshold based on the current flow type and the algorithm in §5.1
sem_post	send the interrupt message to wake up the NF
start_nf	start a new NF of specified type
destroy_nf	Destroy an NF of specified type

5. OPTIMIZATIONS

This section describes some of the performance optimizations that enable Flurries to handle thousands of flows/NFs per second. Specifically, we describe how the Flurries platform manages wake up messages for large numbers of NFs to improve both latency and throughput, and how the per-flow NF abstraction allows for fine grained prioritization.

5.1 Adaptive Wakeup Batching

Waking an NF whenever at least one packet is in its receive queue can become very inefficient if it prevents batching of packets. In the worst case, an NF might be scheduled to handle a packet and then descheduled just when another packet arrives for it to be processed. On the other hand, holding a packet until more arrive can incur high latency, especially during TCP startup and for interactive flows where a packet must be processed and sent out before a new one will arrive.

To balance latency and context switch overhead, the Flurries Wakeup system classifies flows as latency or bandwidth sensitive. It then adjusts a wakeup threshold for each NF to determine how many packets must arrive before it is woken up. The Wakeup system also maintains a timer for each flow that is started when its queue becomes non-empty; if the timer expires before the NF is woken up, then the NF will be preemptively scheduled to avoid starvation.

Since all TCP flows are latency sensitive during the handshake phase, and short flows tend to be latency sensitive, Flurries considers any flow that has received less than T_{short} packets to be a latency sensitive flow. In our evaluation we configure $T_{short} = 36$ packets. These latency sensitive flows are always woken up when at least one packet arrives, allowing them to quickly finish the TCP handshake and handle short, interactive flows such as HTTP requests.

Once a flow has received more than T_{short} packets over its lifetime, we use an additive increase, multiplicative decrease controller to automatically adapt the wakeup threshold, T_{adapt}. Whenever the flow's queue contains at least T_{adapt} packets, the NF is woken up and T_{adapt} is updated to $T_{adapt} + 1$. If further packets do not arrive and the NF is scheduled because of the flow timeout, then we adjust the threshold to $T_{adapt} = T_{adapt}/2$.

System administrators can tune these parameters and set maximum or minimum wakeup thresholds on a per-flow basis. The wakeup parameters and statistics for each flow are maintained inside the flow table.

5.2 NF Priority Management

Tuning the parameters for the adaptive wakeup algorithm described above provides an indirect way to adjust performance, but Flurries also provides direct prioritization of NFs. Since each Flurry

NF handles traffic for a single flow or traffic class, this provides fine grained performance management.

In the early stages of our research we explored sharing NF performance statistics (e.g., queue lengths) with a custom CPU scheduler in order to have NF-aware process scheduling. However, we found that even when using shared memory between user space and kernel space, it was costly and difficult to keep fine grained statistics up to date in the kernel due to the high speed of packet arrivals. Further, since Flurries NFs run for only a short time before switching to a new NF, having a more complex scheduling algorithm actually hurt performance and limited the scalability.

Instead, Flurries uses coarser grained scheduling priorities to increase or decrease the performance of flows. To minimize the context switches, we use the Linux Real Time scheduler with the Round Robin policy. This is because RT scheduler limits context switches, especially when a woken up process has the same priority with the current process. The standard Completely Fair Scheduler (CFS) allows a newly awoken process to preempt the current one, which can lead to poor batching and more context switches since Flurries frequently wakes up NFs. When an idle NF is assigned a new flow, Flurries sets the scheduler priority for the NF. This priority is retained until the flow finishes. The scheduler maintains a queue for each priority and selects processes from a queue in round robin order. This allows Flurries to provide a total performance ordering for different flows, for example ensuring that latency sensitive voice chat flows will be scheduled before bulk data transfer flows. However, adaptive wakeups and flow prioritization are not enough to provide strict performance guarantees, especially for service chains. Our future work will explore how monitoring data from multiple NFs in a chain can be used to automatically adapt priorities to meet target SLAs.

5.3 Minimizing Startup Times

Creating a new NF process for each arriving flow would result in high latency due to the process setup delay. Consequently, Flurries maintains a sufficiently large idle pool of NFs from which NFs are assigned to new incoming flows. The wait time experienced by an incoming flow depends on the availability of idle NFs in the pool. When the workload consists of short connections, as is usually the case for web-based workloads, new flows are less likely to experience any wait time. Even in complex web applications that initiate multiple flows, individual flows are typically short-lived. When the majority of connections are long-lived, there is a higher chance of new flows experiencing longer delays.

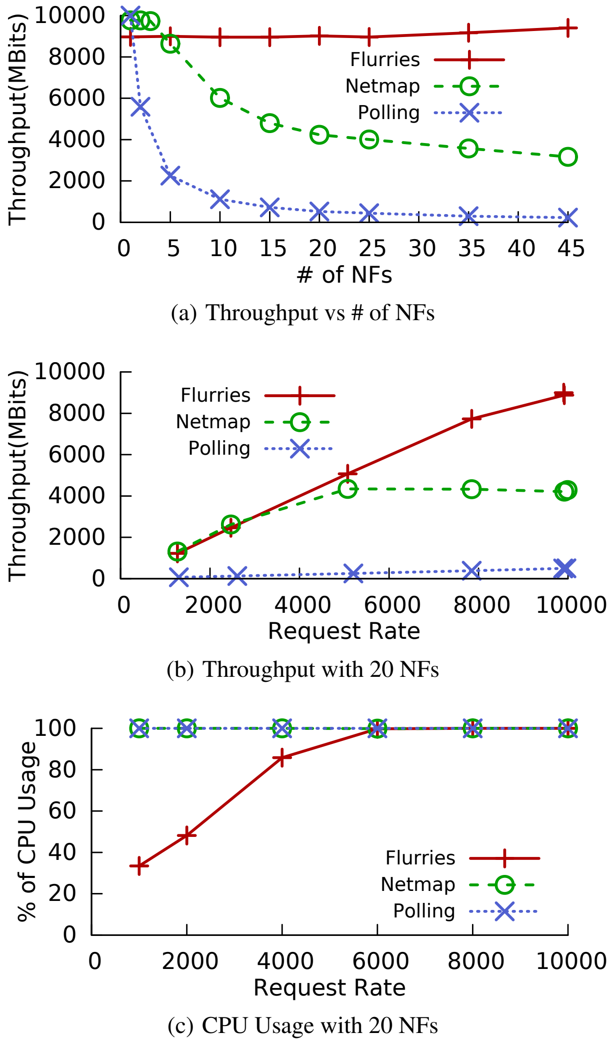
To handle highly dynamic workloads whose characteristics vary dramatically within short periods of time, the idle pool size can be dynamically resized based factors like the flow arrival rate, average lifetime of a flow, etc. Our current implementation does not support the dynamic idle pool resizing feature.

6. EVALUATION

In this section, we evaluate Flurries to:

- Show the hybrid approach of interrupt and polling provides better throughput and resource utilization than purely interrupt or purely polling solutions (§6.1)
- Demonstrate the benefits of Flurries (§6.2)
- Analyze the impact of the adaptive wakeup threshold (§6.3)
- Show the scalability of Flurries when using up to six 10Gbps NICs (§6.4)

In our experimental setup, we use seven severs. Six workload generator servers have dual Xeon X5650 @ 2.67GHz CPUs (2x6 cores), 384KB of L1 cache, 1536KB of L2 cache per core, and a shared 12MB L3 cache, an Intel 82599EB 10G Dual Port NIC and 48GB memory. One server used to host Flurries has an Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz (2x14 cores), three Dual Port NICs (Intel 10G X520-DA2) and 164GB memory. Every server runs Ubuntu 14.04.4 with kernel 4.2.0-36-generic for Flurries testing and kernel 3.13.0 when using Netmap. We use MoonGen as a high speed packet generator, apache bench (ab) for generating latency-sensitive TCP flows, and iperf for bandwidth-sensitive TCP flows. On the web servers, we run haproxy 1.4.24 and nginx 1.4.6 which are capable of sustaining 10Gbps rates when ab requests 32KB files.



(a) Throughput vs # of NFs

(b) Throughput with 20 NFs

(c) CPU Usage with 20 NFs

Figure 4: Flurries provides high throughput and scalable CPU usage, even when running multiple NFs on each core.

6.1 Hybrid Interrupt & Polling

The hybrid notification mechanism of Flurries is designed to provide high performance at the NIC-layer, while allowing multiple NFs to share one core. This allows NFs to efficiently use CPU resources when the workload is low, and for a larger number of NFs than cores to run on a server. To show that Flurries can provide high performance when multiple NFs share a single core, we measure the achieved throughput when increasing the number of NFs.

We measure the throughput in Mbits/sec for three cases: Flurries, Polling, and Netmap. The first two cases are based on our OpenNetVM platform, so the RX and TX threads use polling to efficiently receive and send packets from the NIC; Flurries then uses NFs that sleep between batches, while the Polling case runs NFs that spin looking for packets in their receive queues. For netmap, we run the load distributor (lb) to receive the packets from the NIC on a dedicated core (similar to Flurries' RX thread), then distribute packets to pkt-gen receiver NFs that share a different core. Unlike the Flurries and Polling cases, the netmap NFs simply drop all packets that are received rather than forwarding them back out the NIC, so we expect netmap to have better performance all else being equal.

The experiment uses MoonGen with a 64 byte packet size. Since MoonGen only sends a single large packet flow, we use round robin to distribute packets evenly across the configured number of NFs and we do not recycle them. This setup mimics the case where you have N different traffic classes that each need to be processed by their own NF.

In this scenario, Flurries consistently achieves 9.4Gbps for up to $N = 45$ NFs, as shown in Figure 4(a). In contrast, polling quickly drops to 5.5Gbps even when just two NFs share one core. This is because a polling NF may hog CPU cycles even when there are no packets available, causing other waiting NFs to drop packets from their overflowing queues. Netmap performs better than the Polling NFs, but its performance drops as the number of NFs rises over five since it is not well optimized to run many NFs per core.

When the workload is low, Flurries avoids wasteful polling to get good performance and CPU utilization. Figure 4(b) and 4(c) respectively show the throughput and CPU usage vs. request rate for the three systems when one core runs 20 NFs. The throughput of Flurries linearly increases as the request rate goes up. With an arrival rate of 1270 Mbps, the 20 Flurries NFs consume a total of 33% of the CPU, whereas the other approaches consistently consume all available CPU resources with that many NFs. Even the interrupt driven netmap system uses 100% of the CPU if there are 20 NFs running, although with only a single NF it can meet the line rate using 20% of the CPU, similar to Flurries.

NF Complexity: The above experiments use light-weight NFs that only perform forwarding, not complex packet analysis or processing. To evalute how Flurries reacts differently when facing complex NFs compared to simple NFs, we create a more heavy-weight NF that uses about three times as many CPU cycles for each packet processed (incurring a 162 ns processing cost compared to 67 ns in our simple NF). We then run either a single NF of the given type, or a group of 45 Flurry NFs, all of the same type. This allows us to see the overhead of Flurries when running different types of NFs.

The throughput of simple NF vs complex NF is shown in figure 5. For a simple forwarding NF, we see about 10.3% performance drop when we run 45 Flurry NFs instead of a single NF that is dedicated the CPU. The primary cause of this overhead is the context switch cost to cycle between the different Flurry NFs servicing the packet stream. With more complex NFs, the maximum throughput that can be achieved drops since the CPU becomes the primary bottleneck; however, the overhead of Flurries falls to only 2.8%. This is because in a more complex NF, the context switch overhead becomes relatively smaller compare against lighter NF. This suggests that as NFs become more complex, it may be necessary to use more resources (e.g., CPU cores) to host them in order to meet a target throughput, but that the amount of extra resources required to run Flurry NFs as opposed to traditional, monolithic NFs will continue to fall.

NF Scalability: Figure 6 shows the throughput and latency when

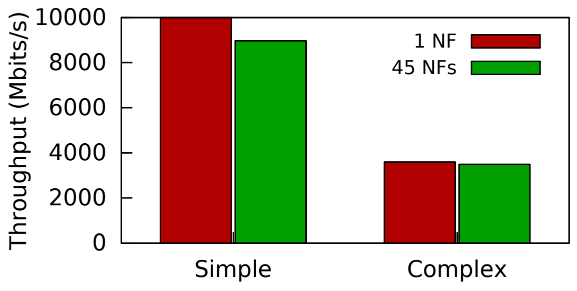


Figure 5: The overhead incurred by Flurries falls as NFs become more complex.

scaling to up to 900 Flurry NFs, well beyond the capacity of Netmap or our Polling system. The "Flurries-Single" case is identical to that described above, so the 900 NFs can be thought of as 900 different traffic classes that each are sent to their own NF. The "Flurries-Chain 4" case means that packets are sent through a chain of four NFs before being sent out. In the latter case, we use two cores to split the NFs across, so each core runs at most 450 NFs. Note that a chain of four NFs puts four times as much pressure on the NFs and requires the management layer to do four times as many packet transfer operations to maintain the same rate as a single NF.

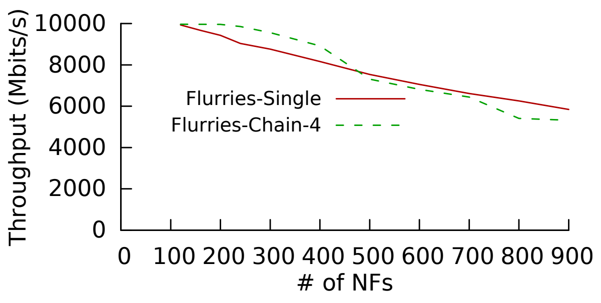
When having a single NF process each traffic class, Flurries is able to maintain a throughput over 9Gbps and latency under 1ms when running 240 NFs (i.e., 240 traffic classes) on just one core. With 900 NFs, Flurries can still meet a 6Gbps rate and a 6ms latency, running almost one hundred times as many NFs per core as netmap while meeting the same throughput. With a chain of 4 NFs per traffic class, Flurries initially gets even better performance since the NFs are spread across two cores instead of one. Even with 900 NFs (i.e., $900/4 = 225$ different traffic class chains), Flurries achieves a throughput of over 5Gbps for 64 byte packets. Latency rises when there are a large number of chains, but we discuss ways to mitigate this issue by scheduling NFs in a service chain-aware manner later in our evaluation.

Port Scalability: We next evaluate the scalability of each approach when using two NIC ports instead of one. Here we focus on performance of the I/O layer in each system, so we use a single core to receive packets from both NIC ports. In Flurries and Polling we run a single RX thread on the core. In netmap we run two of its interrupt driven load balancer programs (one per port) on the same core since each can only read from one NIC port. The RX threads and load balancers distribute packets in a round robin fashion to a pair of NFs running together on a second core.

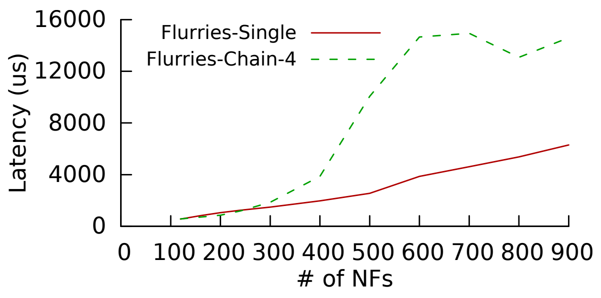
We measure the throughput of each approach when sending 128 byte packets. Flurries achieves 15Gbps, which is the maximum our hardware supports when using two ports on the same NIC, even for the minimalist L2fwd DPDK code. Netmap can only get 10.3Gbps in this setup, illustrating the limitations of an interrupt-driven I/O architecture. Polling achieves 9.3Gbps with two NFs, an improvement over the single port case since it is more likely that a spinning NF will find packets from one of the ports to process.

6.2 Benefits of Flurries

Given the high performance potential of Flurries, we now investigate some of the benefits of our Flurries architecture for flow-based fine-grained scheduling, flow affinity, and performance isolation.

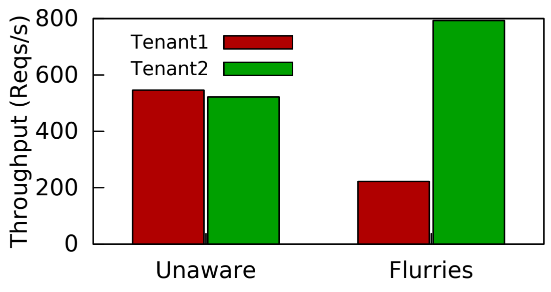


(a)

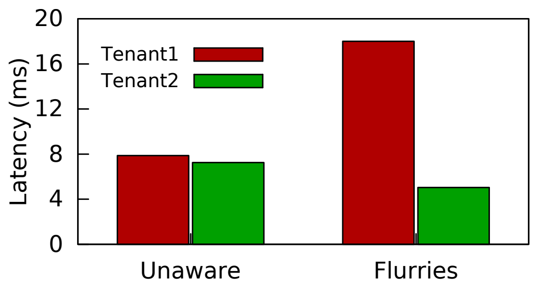


(b)

Figure 6: Even with hundreds of NFs on two cores, Flurries can achieve over 5Gbps when processing 64 byte packets through a chain of 4 FNs



(a) Throughput



(b) Latency

Figure 7: Flurries dynamically sets the priority of NFs based on tenant flow characteristics.

Fine-grained Flow Priority: To illustrate the potential for traffic prioritization in Flurries, we first setup a scenario where two tenants, Tenant 1 and Tenant 2, send traffic through NFs of the same type of service. Tenant 2 is a VIP customer (high priority), thus expects better performance for her flows than the non-VIP user, Tenant 1. If the system is unable to differentiate traffic at the flow level, flows from both tenants will be assigned to NFs with equal shares of resources, and thus will receive similar throughput and latency. However, since Flurries assigns NFs to individual traffic classes (one per tenant in this case), it can easily achieve differentiated levels of service.

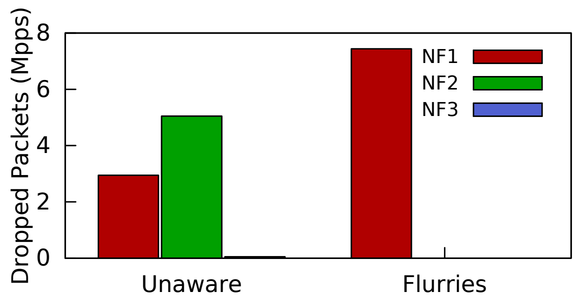
We demonstrate this in Figure 7(a) and 7(b), where HTTP requests from two clients are sent through a middlebox system that is either oblivious of the tenant priorities ("Unaware") or priority-aware ("Flurries"). In the priority-aware version of Flurries, a common idle pool is used for NFs from both tenants, all running identical forwarding functions. When the Flow Director assigns a new flow to an NF, it also sets the scheduler priority for that NF, increasing or decreasing the likelihood it will run for the length of that flow. As a result, Tenant 2 achieves substantially higher throughput and lower latency compared to the priority unaware case. This illustrates how Flurries provides the basic mechanisms that can be used to dynamically tune NF performance on a per-flow basis.

Service Chain-Aware Scheduling: Next we consider how adjusting priorities of NFs in a service chain can improve performance by avoiding wasted work. We configure a three NF service chain, where each NF in the chain has an increasing computation cost with ratio 1:3:7 across the chain. For simplicity, we generate a single large UDP flow with MoonGen and send it through the service chain without recycling NFs.

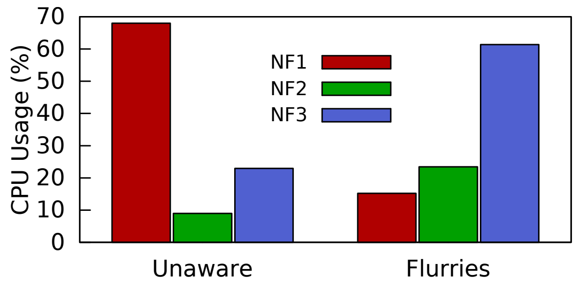
Since, for example, the 3rd NF has seven times higher computation cost than the first, we would expect it to receive approximately seven times as much CPU time. However, this is not the case, and NF 1, despite having the lowest cost per packet, ends up receiving the most CPU as shown in Figure 8(b). The reason for this is that since packets are arriving into the NFV platform at a high fixed rate, the RX threads are constantly receiving packets for NF 1 and marking it as runnable. This makes it more likely for NF 1 to run and process a full batch of packets. However, when these packets are sent on through the chain, it becomes increasingly likely that they will arrive at an NF that already has a full receive queue and is waiting for NF 1 to relinquish the CPU. In Figure 8(a), this causes a large number of drops at the input queue for NF 2, essentially causing it to become a rate limiter before packets can reach NF 3. However, having packets dropped at NF 2 is wasteful since it means we have already paid the cost of partially processing them through the chain; it would be far preferable to drop packets when they first arrive at the service chain if there is insufficient capacity. Such a problem is exacerbated with even longer chains.

To prevent this form of wasted work in service chains, Flurries can assign an increasing scheduler priority to each NF in the chain. This ensures that if the system becomes overloaded, the extra workload will be dropped as early as possible. Figure 8(c) shows that when Flurries is service chain-aware and assigns priorities to NFs in increasing order, it achieves a more appropriate CPU usage assignment and improves throughput by over 2.5 times.

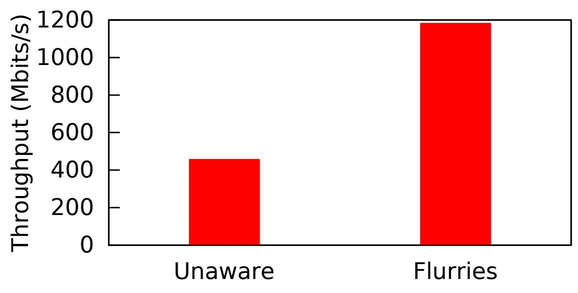
Flow Affinity: Flurries can provide NF affinity at the flow or traffic class level, meaning that all of the requests and responses of the same type go through the same set of NFs. In this experiment, we use apache bench to send realistic traffic and vary the number of NFs, then measure performance when Flurries uses three different NF-assignment algorithms: Round Robin, RSS, and Per-Flow. The first simply assigns batches of incoming packets to NFs in a round robin fashion, potentially causing out of order problems since pack-



(a) Drop Rate

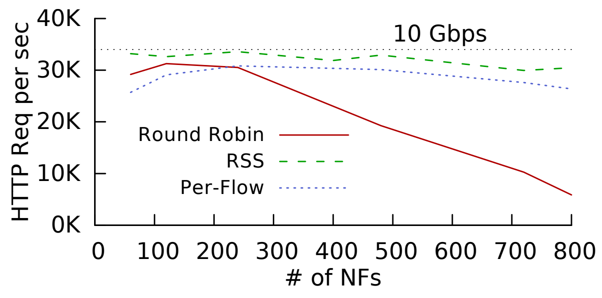


(b) CPU Usage

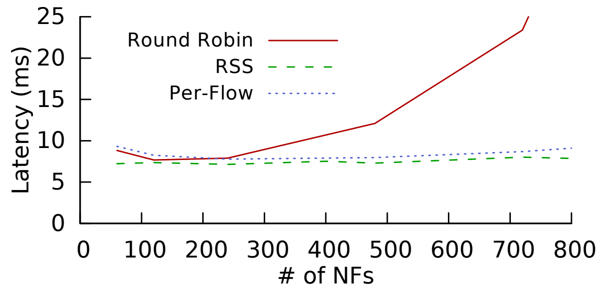


(c) Throughput

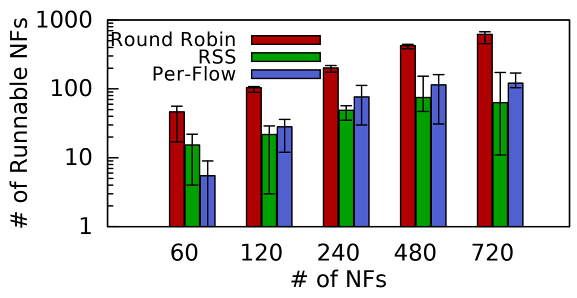
Figure 8: Using service chain position to set NF priority helps throughput by dropping packets early to avoid wasted work.



(a) Throughput



(b) Latency



(c) Runnable NFs

Figure 9: Using RSS or Per-Flow assignment of NFs allows Flurries to maintain close to 10Gbps even with hundreds of NFs.

ets from the same flow may be delivered to different NFs. The RSS approach provides flow affinity by assigning flows to NFs based on the packet header's *RSS value mod number of NFs*. While this prevents out of order packets, it does not guarantee a unique NF is assigned to each flow, so this multiplexing can cause interference as shown in the next section. Finally, the Per-Flow case represents the standard Flurries deployment where each incoming flow is matched to its own private NF which is recycled for a different flow once it completes.

We fix the HTTP workload concurrency to 240 flows and adjust the number of available NFs. When the number of NFs increases, the throughput of Round Robin decreases significantly as shown in Figure 9(a), and the request latency quickly rises up from 9*ms* to 25*ms* as shown in Figure 9(b). This is due to two factors: first, the rate of packets arriving out of order reaches up to 4.5% since round robin is not flow-aware, and second, round robin spreads incoming packets across all NFs, causing more NFs to be in the Runnable state contending for the CPU, but having fewer packets in their queues, reducing the benefits of batching. Figure 9(c) shows the average number of runnable NFs for each approach; the error bars indicate the 25th and 75th percentiles. When the number of available NFs is lower than the workload concurrency, the Per-Flow case performs worse since it does not have enough NFs to assign one for each flow. Otherwise, it is able to perform similarly to the RSS case, maintaining a throughput close to 33K requests per second which utilizes the full 10Gbps bandwidth. RSS performs the best since it does not incur NF recycling overheads and it tends to multiplex a smaller number of NFs (leading to better batching) than the Per-Flow case.

Performance Isolation: While RSS-based flow direction provides the highest performance, it does not offer any way to carefully control which flows are delivered to which NFs. As a result, if different flows have different traffic characteristics, they may be assigned to the same NF causing interference. In Table 1, we compare the performance when latency-sensitive web traffic and bandwidth-sensitive iperf traffic need to go through the same type of NF. Using RSS, iperf uses up almost all the bandwidth and incurs high interference to web traffic. While Flurries can schedule to steer the web flows and iperf flows to different NFs and control the resource al-

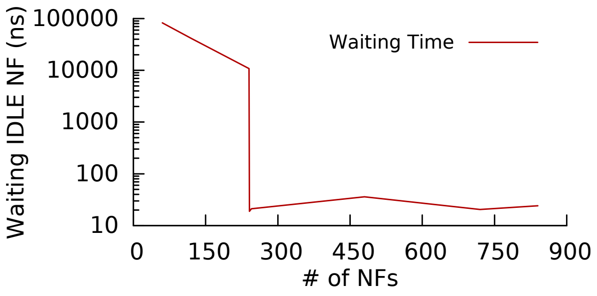


Figure 10: Waiting times are small if there are enough NFs to meet the concurrency level (240 flows)

location in a fine-grained fashion, the web throughput doubles, the web latency drops by 50%. Therefore, Flurries can achieve the comparable iperf throughput compared to RSS.

	Web Reqs/s	Web Latency	Iperf Tput
RSS	3378	11.8ms	9.3BGbps
Flurries	6650	6.0ms	8.8Gbps

Table 1: Impact of flow direction on mixed workloads

Per-Flow Overheads: Running Flurries with Per-Flow NFs puts more stress on the Flow Director since it must assign NFs to each incoming flow recycle them when they are finished. To evaluate the minimum cost of flow setup we maintain a large pool of idle NFs and measure the latency between when a packet arrives to Flurries and when the new NF is allocated for the flow. We find that this takes an average of 708*ns*, including the time for the Flow Director to find an idle NF of the appropriate service type, establish a new flow table rule, initialize the flow entry statistics, and enqueue the packet for the new NF.

To show how the size of the idle pool affects performance, we vary the number of NFs and measure the cost to find an idle NF. We fix the flow concurrency to 240 and use apache bench to send requests through Flurries. Figure 10 shows that the time to find a free idle NF is linearly decreased as the number of NFs increases up to 240. After that, the waiting time becomes stable since there is always an NF available.

6.3 Adaptive Wakeup Threshold

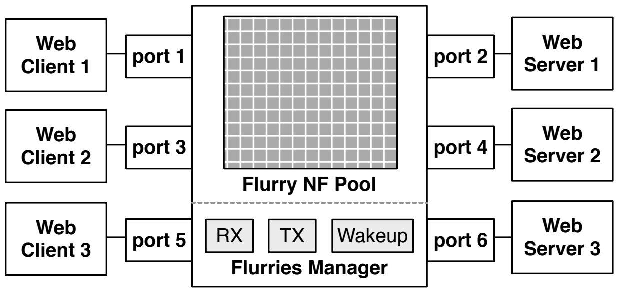
The Flurries adaptive wakeup threshold algorithm is designed to balance the needs of latency and throughput focused flows. We evaluate its effectiveness by comparing against several simpler wakeup strategies. *Min* tries to minimize the latency of flows by always transitioning an NF to the runnable state once it has at least one packet in its queue. *Naive-100us* represents a simple approach to balance latency and throughput: it requires at least 32 packets to be in the queue before waking an NF and uses a timeout of 100 microseconds to avoid starvation. The *Naive-1ms* is similar, but uses a 1 millisecond timeout in order to further reduce the number of context switches. Finally, we run *Adaptive*, which uses a dynamic approach: during flow startup it mimics the Min technique, but once flows are established it uses an adaptive threshold and a 1ms timeout. We run a mix of web flows, which are particularly sensitive to TCP handshake latency, and iperf flows, which are throughput focused.

The table shows that the Min approach incurs the highest number of context switches, while keeping the minimum web latency. The

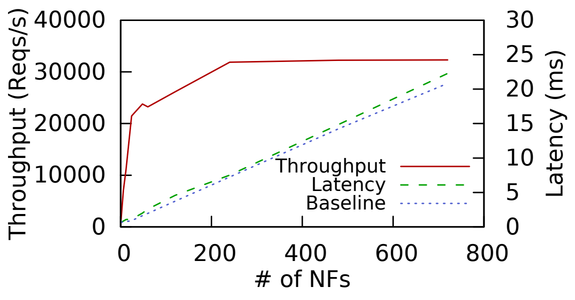
	Web Req/s	Web Latency	Iperf Tput	CPU	Context Switches
Min	4002	3.7ms	6.0 Gbps	80%	258K/s
Naive-100us	3899	3.8ms	5.8 Gbps	99%	237K/s
Naive-1ms	2694	5.6ms	4.7 Gbps	67%	136K/s
Adaptive	5022	4.0ms	6.9 Gbps	99%	217K/s

Table 2: Impact of wakeup algorithms on mixed workloads

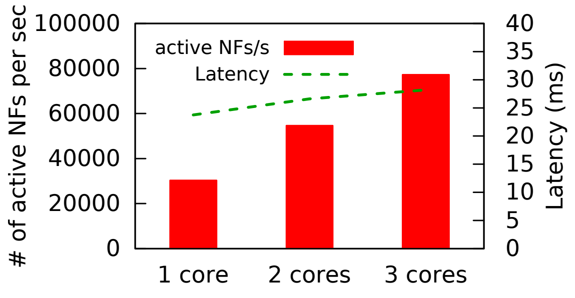
Naive approaches both perform relatively poorly, with high latency and low throughput when using a timeout of 1ms. This approach leaves the CPU idle about 33% of the time. Flurries' Adaptive algorithm provides the best overall performance, getting the highest web and iperf throughput with only a small latency degradation.



(a) Multi NIC Testbed



(b) Single NIC



(c) Multi NIC

Figure 11: Testing Flurries with 6 NIC ports shows its capacity to run up to 80,000 NFs in a one second interval, while meeting a 30Gbps traffic rate and incurring minimal added latency to web traffic.

6.4 Multi-NIC Scalability

Finally, we investigate the scalability of Flurries when using web traffic across multiple NIC ports and a large number of NFs. We setup our Flurries server with three dual port NICs as a "bump in the wire" between three client generator servers and three web servers as shown in Figure 11(a). Flurries is used to bridge ports between the two servers, and we use Per-NF flow allocation to dedicate a distinct NF to each flow. We use one thread each for RX, TX, and Wakeup.

First, we show how Flurries' performance changes as we adjust the number of flows on one core and with one NIC (i.e., only client 1 and web server 1). Figure 11(b) shows the throughput and latency vs. # of flows on one core (the number of flows is equal to the number of NFs). We use apache bench to request a file (32KB), and adjust the concurrency of the clients so the total concurrency is equal to the number of NFs in each test. As the number of flows goes up, the throughput increases until the full NIC bandwidth is used. Using 32KB files for this test results in Flurries seeing a large number of relatively short flows–as of September 2016, the average size of web page content is 2.4MB [25], although this may be split over several HTTP connections. For flows larger than 32KB, we expect Flurries' performance to improve, since relatively less time will be spent on assigning idle NFs and generating the flow entry.

Note that since Flurries is bridging two ports that each support 10Gbps in both directions, the total data rate being sent and received through the NFV platform is well over 20Gbps—approximately 10Gbps of server responses are received and sent out each port plus client requests are also being transmitted. Increasing the number of flows causes a minimal increase in latency because there are more flows contending for the CPU, however, the majority of the latency rise is because there is simply more load on all the client and server machines–running the two machines directly connected gives a baseline latency that is consistently within 2ms of Flurries, as shown in the figure.

Next we measure the scalability when using multiple NIC ports and multiple cores. In each stage of the experiment we add one more core hosting 720 NFs and one more dual port 10Gbps NIC, until all three NICs (6 ports) are in use and there is a total of 2,160 NFs in the idle pool. We keep the management configuration the same in all cases, with one thread each for RX, TX, and Wakeup. We measure the number of "active NFs" within each one second interval, i.e., the number of NFs that are selected when a new flow arrives and then recycled for a subsequent flow. Figure 11(c) shows that over the course of the experiment, three cores can support an average of nearly 80,000 active NFs each second. This is about 2.6 times greater than when only a single core and a single port are used. At this scale, each of the 80,000 NFs could be assigned different priorities based on the needs of each flow, and since the active NFs are drawn from a pool of over 2,000 unique NFs, it would be possible to host a vast variety of different types of functions each being flexibly assigned on a per-flow basis.

In the three core case, Flurries saturates all three of the dual port NICs, so adding additional cores of NFs does not affect the scalability results. We believe that adding more NICs and cores should continue this trend, while maintaining a relatively flat latency. Latency rises in part due to the increasing load on the RX thread, but our current implementation only supports running one RX thread, so we are not able to evaluate scalability further.

7. RELATED WORK

Several NFV platforms have demonstrated high-speed networking virtualization on commodity off-the-shelf servers [9, 18, 23, 26]. In 2009, Greenhalgh et. al. [27] first proposed Flowstream, an architecture designed to enable flexible flow processing and forwarding, and leading to a growing interest in managing network functions in a more fine-grained fashion. More recently, the Superfluid cloud [28] project sought to increase and manage the number of concurrently running network functions up to 10,000 using an optimized platform plus lightweight network functions; their focus was on fast startup of NFs, and they do not provide performance results. Cerrato [29] also considers running large numbers of tiny NFs, and faces many performance problems that our platform overcomes with our hybrid polling-interrupt I/O scheme. The ClickOS platform [30] builds on top of netmap to provide a lightweight Click router based operating system with high speed network processing. Our evaluation compares against netmap directly since our testing suggests it generally has higher performance. Their work, and others such as Jitsu [31] and Miniproxy [32], illustrate the potential of unikernels to reduce NF startup time to the 20 *ms* range. Flurries relies on an idle pool of NFs and we have not yet explored ways to reduce the startup delay of our NFs (approximately 500ms).

Managing a large number of network functions at fine-grained granularity requires maximizing the parallelism and efficiency of hardware resources. E2 [33] is an NFV management platform, which includes NF placement across the cluster, dynamic scaling and migration avoidance for flow affinity. Our work focuses on a different scale: running large number of NFs on a small number of cores. Especially, for a service chain [34, 35] spanning multiple NFs requires good resource schedulability in order to not increase latency or drop packets. The RouteBricks project [36] increases the speed of software routers by exploiting parallelism both across multiple servers and across multiple cores within a single server. Li et. al. [37] sought to provide timing guarantees in network function virtualization environment, and enable a set of service chains that each consist of some network functions. Cao et. al. [38] characterized the performance of virtual network functions.

Flurries makes use of a mix of polling and interrupts, a concept which has been explored in prior work. Dovrolis et. al. [39] proposed the hybrid approach for the network interface to benefit merits of both. Deri et. al. [40] proposes a new approach to passive packet capture that combines with device polling to allow packets to be captured and analyzed using the NetFlow protocol at (almost) wire speed on Gbit networks. Liu et. al. [41] significantly accelerates the I/O virtualization by using the dedicated cores to poll the network interfaces. Also, Yang et. al. [42] investigated cases where polling is better than interrupt. To our knowledge, we are the first NFV platform to try to balance these two approaches to maximize performance at the I/O layer while efficiently using resources at the application layer.

8. CONCLUSION

NFV offers unprecedented power to manipulate and analyze traffic as it flows through the network. Existing NFV platforms have been designed to maximize performance of a small number of NFs, but this prevents truly exploiting the flexible per-flow routing provided by SDNs. Flurries draws from recent trends such as containerization and transient microservices to build an NFV platform optimized for running thousands of distinct NFs per server, each of which can be flexibly assigned to a single flow or traffic class. We believe that this can dramatically expand the diversity of functionality and performance management in the network. Our prototype has shown that Flurries can run 100 times as many NFs as netmap while meeting the same throughput, and that over 2000 distinct NFs can share 3 CPU cores while fully utilizing six 10Gbps NIC ports. This provides a substantial increase in performance, flexibility, and isolation compared to the state of the art.

Acknowledgements: This work was supported in part by NSF grants CNS-1422362 and CNS-1522546. We would like to thank the anonymous reviewers and our shepherd, Ihsan Ayyub Qazi, for their comments and feedback.

References

[1] Wei Zhang, Guyue Liu, Ali Mohammadkhan, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Sdnfv: Flexible and dynamic software defined control of an application- and flow-aware data plane. In *Middleware*, 2016.

[2] T. Wood, K. K. Ramakrishnan, J. Hwang, G. Liu, and W. Zhang. Toward a software-based network: integrating software defined networking and network function virtualization. *IEEE Network*, 29(3):36–41, May 2015.

[3] Ciena Toolkit Applies DevOps To Control SDN/NFV Networks. https://virtualizationreview.com/articles/2016/05/27/ciena-devops.aspx, May 2016. [ONLINE].

[4] Achieving DevOps for NFV Continuous Delivery on Openstack - Verizon Case Study. https://www.openstack.org/videos/video/achieving-devops-for-nfv-continuous-delivery-on-/openstack-verizon-case-study. [ONLINE].

[5] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 44–57, New York, NY, USA, 2016. ACM.

[6] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 501–521, Santa Clara, CA, March 2016. USENIX Association.

[7] Wei Zhang, Timothy Wood, Jinho Hwang, Shriram Rajagopalan, and K. K. Ramakrishnan. Performance management challenges for virtual network functions. In *Proc. NetSoft*. IEEE, 2016.

[8] Intel Corporation. Intel data plane development kit: Getting started guide. 2013.

[9] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, 2016.

[10] Sam Neuman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, February 2015.

[11] Martin Fowler and James Lewis. Microservices. http://martinfowler.com/articles/microservices.html, 2014. [ONLINE].

[12] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1 edition, July 2010.

[13] ORBITZ. Enabling Microservices at Orbitz. *DockerCon (2015)*.

[14] HUBSPOT. How We Deploy 300 Times a Day. http://product.hubspot.com/blog/how-we-deploy-300-times-a-day, November 2013. [ONLINE].

[15] Wei Zhang, Timothy Wood, and Jinho Hwang. Netkv: Scalable, self-managing, load balancing as a network function. In *IEEE International Conference on Autonomic Computing*. IEEE, 2016.

[16] Wei Zhang, Timothy Wood, K.K. Ramakrishnan, and Jinho Hwang. Smartswitch: Blurring the line between network infrastructure & cloud applications. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, Philadelphia, PA, June 2014. USENIX Association.

[17] Intel Corporation. Intel data plane development kit: Intel 64 and ia-32 architectures software developer's manual. 2013.

[18] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[19] Radu Stoenescu, Vladimir Olteanu, Matei Popovici, Mohamed Ahmed, Joao Martins, Roberto Bifulco, Filipe Manco, Felipe Huici, Georgios Smaragdakis, Mark Handley, et al. In-net: in-network processing for the masses. In *Proceedings of the Tenth European Conference on Computer Systems*, page 23. ACM, 2015.

[20] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 227–240, Berkeley, CA, USA, 2013. USENIX Association.

[21] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[22] VPP. https://fd.io/, 2016. [ONLINE].

[23] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[24] I. Cerrato, M. Annarumma, and F. Risso. Supporting Fine-Grained Network Functions through Intel DPDK. In *2014 Third European Workshop on Software Defined Networks*, pages 1–6, September 2014.

[25] HTTP Archive - Interesting Stats. http://httparchive.org/interesting.php. [ONLINE].

[26] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *Proc. NSDI*, NSDI'14, pages 445–458, Berkeley, CA, USA, 2014. USENIX Association.

[27] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev.*, 39(2):20–26, March 2009.

[28] Filipe Manco, Joao Martins, Kenichi Yasukata, Jose Mendes, Simon Kuenzer, and Felipe Huici. The case for the superfluid cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[29] Ivano Cerrato, Mauro Annarumma, and Fulvio Risso. Supporting fine-grained network functions through intel dpdk. In *EWSDN*, pages 1–6. IEEE, 2014.

[30] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, April 2014. USENIX Association.

[31] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *Proc. NSDI*, pages 559–573, Oakland, CA, May 2015. USENIX Association.

[32] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. On-the-fly tcp acceleration with miniproxy. In *Proceedings of the*

2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, 2016.

[33] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[34] Seungik Lee, Myung-Ki Shin, EunKyoung Paik, and Sangheon Pack. Resource Management in Service Chaining. Internet-Draft draft-lee-nfvrg-resource-management-service-chain-01, Internet Engineering Task Force, October 2015. Work in Progress.

[35] ETSI. Network functions virtualization (nfv); architectural framework. *ETSI GS NFV V002*, 2013.

[36] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28. ACM, 2009.

[37] Y. Li, L. T. X. Phan, and B. T. Loo. Network functions virtualization with soft real-time guarantees. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

[38] L. Cao, P. Sharma, S. Fahmy, and V. Saxena. Nfv-vital: A framework for characterizing the performance of virtual network functions. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 93–99, Nov 2015.

[39] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. Hip: Hybrid interrupt-polling for the network interface. *SIGOPS Oper. Syst. Rev.*, 35(4):50–60, October 2001.

[40] Luca Deri, Netikos S. P. A, Via Del Brennero Km, and Loc La Figuretta. Improving passive packet capture: Beyond device polling. In *In Proceedings of SANE 2004*, 2004.

[41] Jiuxing Liu and Bulent Abali. Virtualization polling engine (vpe): Using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 225–234, New York, NY, USA, 2009. ACM.

[42] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 3–3. USENIX Association, 2012.