A Disaggregated Packet Processing Architecture for Network Function Virtualization

Shihabur Rahman Chowdhury, *Student Member, IEEE*, Anthony, Haibo Bian,
Tim Bai, and Raouf Boutaba, *Fellow, IEEE*

***Abstract*—Network Function Virtualization (NFV) promises to reduce the capital and operational expenditure for network operators by moving packet processing from purpose-built hardware to software running on commodity servers. However, the state-of-the-art in NFV is merely replacing monolithic hardware with monolithic Virtual Network Functions (VNFs), *i.e.*, software that realizes different network functions. This is a good first step towards transitioning to NFV, however, common functionality is repeatedly implemented in monolithic VNFs. Repeated execution of such redundant functionality is particularly common when VNFs are chained to realize Service Function Chains (SFCs) and results in wasted infrastructure resources. This stresses the need for re-architecting the NFV ecosystem, through modular VNF design and flexible service composition. From this perspective, we propose MicroNF (μNF in short), a disaggregated packet processing architecture facilitating the deployment of VNFs and SFCs using reusable, loosely-coupled, and independently deployable components. We have implemented the proposed system, including the different architecture components and optimizations for improving packet processing throughput and latency. Extensive experiments on a testbed demonstrate that: (i) compared to monolithic VNF based SFCs, those composed of μNFs achieve the same packet processing throughput while using less CPU cycles per packet on average; and (ii) μNF-based SFCs can sustain the same packet processing throughput as those based on state-of-the-art run-to-completion VNF architecture while using lesser number of CPU cores.**

***Index Terms*—Network function virtualization, microservices, middleboxes, virtual network function decomposition.**

I. Introduction

Network operators ubiquitously deploy hardware *middleboxes* [2] (*e.g.*, Network Address Translators (NATs), Firewalls, WAN Optimizers, Intrusion Detection Systems (IDSs), *etc.*) to realize different network services [3]. Despite being an integral part of modern enterprise and telecommunication networks, middleboxes are proprietary, have little to no programmability and vertically integrate packet processing software with the hardware. Such closed and inflexible ecosystem explains the high capital and operational expenditures incurred by network operators. This led to the *Network Function Virtualization (NFV)* movement initiated in 2012 [4]. NFV proposes to disaggregate the tightly coupled *Network Functions (NFs)* and hardware middleboxes, and deploy the NFs as *Virtual Network Functions (VNFs)* on commodity servers. Through this disaggregation, NFV promises to reduce CAPEX by consolidating multiple NFs on the same hardware, and reduce OPEX by enabling on-demand flexible service provisioning.

Significant effort has been dedicated to NFV research since its inception [5], including for: resource allocation and scheduling [6], middlebox outsourcing [3], [7], management platforms [8]–[10], fault-tolerance [11]–[14], state management [15]–[18], traffic steering through VNFs [19], and programming models and runtime systems to support VNFs and SFCs [20]–[26]. However, a common trait observed in these works is the *one-to-one substitution of monolithic hardware middleboxes by their monolithic VNF counterparts*. Indeed, this is a logical first step for transitioning to NFV. However, monolithic VNFs can be a barrier to achieving fine-grained resource allocation and scaling, and can lead to wasted infrastructure resources.

A fundamental problem with monolithic VNF implementation is that many packet processing tasks such as packet I/O, parsing and classification, and payload inspection are repeated across a wide range of enterprise NFs [27]. This has several negative consequences. First, redundant development and optimization effort on these common tasks across different VNFs. Second, monolithic VNFs restrict how many packet processing tasks can be consolidated on the same hardware. For instance, a Firewall and an IDS, both perform packet classification [24]. Since the VNFs are monolithic, we cannot consolidate packet classification as a single function, allocate just enough resources for processing the cumulative traffic of the Firewall and the IDS, and deploy the classifier as a single entity. Third, monolithic VNFs impose coarse-grained resource allocation and scaling. This non-exhaustive list of issues poses a barrier in achieving the agility promised by NFV. In this regard we set out to answer the following question: *What is an appropriate software architecture for VNFs that will enable better function consolidation on the same hardware and finer-grained resource allocation while maintaining the same level of performance as state-of-the-art approaches?*

There is a substantial body of research on modular packet processing software [20], [22], [23], [30], [31]. However, in most cases the end-product is still a monolithic software, which typically executes in a *run-to-completion* mode, *i.e.*,

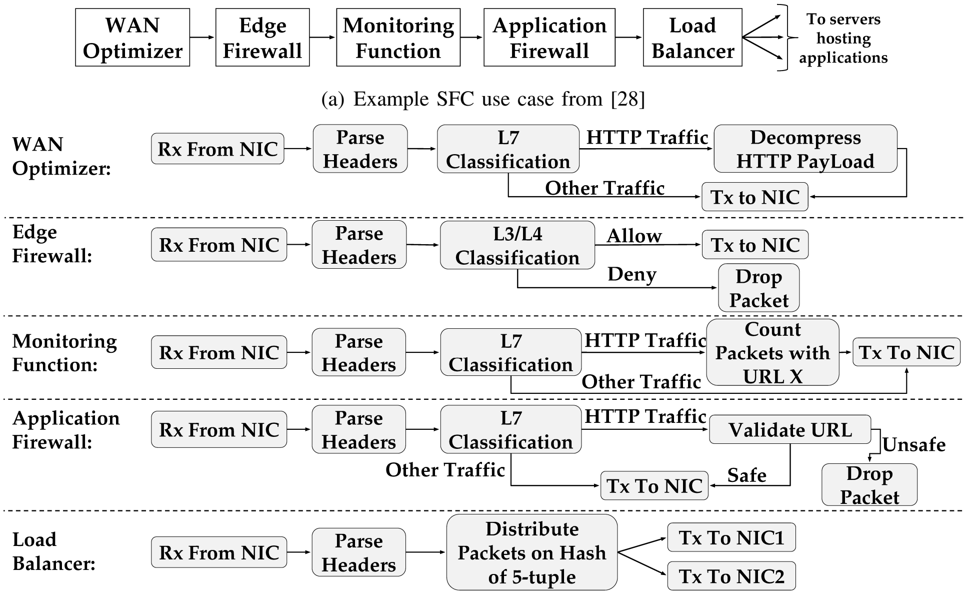
Manuscript received April 15, 2019; revised November 26, 2019; accepted January 28, 2020. An earlier version of this work appeared in [1]. This work was supported in part by the NSERC Create program on Network Softwarization and in part by an NSERC Discovery Grant. This work also benefited from the use of Tembo compute cluster at the University of Waterloo.

Shihabur Rahman Chowdhury, and Raouf Boutaba are with the David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (email: sr2chowdhury@uwaterloo.ca, rboutaba@uwaterloo.ca)

Anthony is with Huawei Technologies Canada, Markham, ON L3R 5A4, Canada (email: anthony.anthony@uwaterloo.ca)

Haibo Bian is with Bioinformatics Solutions Inc., Waterloo, ON N2L 6J2, Canada (email: haibo.bian@uwaterloo.ca)

Tim Bai is with Desire2Learn Canada, Kitchener, ON N2G 1H6, Canada (email: tim.bai@uwaterloo.ca)



(a) Example SFC use case from [28]

(b) Functional decomposition of NFs (adapted from [29])

Fig. 1. Common packet processing tasks across NFs

applies all the functionality of an NF or even an SFC on a batch of packets read from the Network Interface Card (NIC) before they exit the system. This model is usually easier to scale, however, it still suffers from the coarse-grained resource allocation imposed by monolithic software.

In this paper, we aim at building VNFs from simple building blocks by taking advantage of the commonality of packet processing tasks. To this end, we propose μNF, a disaggregated packet processing architecture. μNF takes the disaggregation of middleboxes one step further and decompose VNFs into independently deployable, loosely-coupled, lightweight, and reusable packet processors, that we call *MicroNFs* (μNFs for short). VNFs or SFCs are then realized by composing a packet processing pipeline from these independently deployable μNFs. Such decomposition will allow finer-grained resources allocation, independent scaling of μNFs thus increased flexibility, and independent development and maintenance of packet processing components. μNF is built on the thesis of CoMb [27] that consolidating common packet processing tasks from multiple NFs may lead to better resource utilization. However, CoMb's focus was not to address the engineering challenges for realizing such a system (*e.g.*, software architecture, performance optimizations), which is the key contribution of this paper. Specifically, we have the following contributions:

- A quantitative study to demonstrate how repeated application of common packet processing tasks in an SFC can affect CPU resource utilization (Section II).
- An architecture for composing VNFs and SFCs from independently deployable, loosely-coupled, lightweight, and reusable components that we call μNFs (Section IV).
- Implementation of architecture components including the μNFs, communication primitives between μNF, and CPU sharing between μNFs to improve CPU utilization without sacrificing packet processing throughput (Section VI).
- Optimizations for improving packet processing throughput of μNFs on multi-socket NUMA machines, and packet processing latency in μNF-based network services (Section V).
- Evaluation of the system through testbed experiments (Section VII). Our key findings are: (i) compared to an SFC composed from monolithic VNFs, μNFs can achieve the same throughput using less CPU cycles per packet on average; (ii) μNFs can sustain the same packet processing throughput as the state-of-the-art run-to-completion VNF architecture [23] using lesser number of CPU cores.

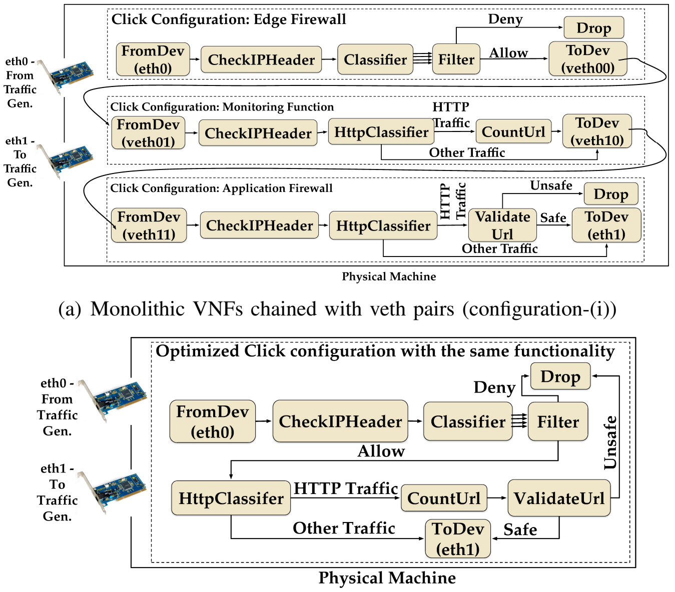
II. Motivation

Our motivation for developing a disaggregated packet processing architecture stems from the observation that many packet processing tasks, such as packet I/O, parsing and classification, and payload inspection are repeated when VNFs are chained in an SFC. We demonstrate this using the SFC in Fig. 1(a), typically found in enterprise Data Centers (DCs) [28]. This SFC consists of the following VNFs:

- **WAN Optimizer:** Placed at a DC and WAN boundary for optimizing WAN link usage, *e.g.*, compresses/decompresses HTTP payload to reduce WAN traffic [32].
- **Edge Firewall:** Allows or denies packets based on layer 2-4 header signature.
- **Monitoring Function:** Consists of different counters such as a packet size distribution counter, a counter for packets containing certain URLs, *etc.*
- **Application Firewall:** Filters packets based on application layer information, *e.g.*, block HTTP requests with embedded SQL injection attacks (similar to [33]).
- **Load Balancer:** Distributes packets to back-end servers based on flow signature.

We can decompose these VNFs into smaller packet processing tasks as shown in Fig. 1(b). Clearly, tasks such as packet I/O, parsing and classifying HTTP packets are repeated in these VNFs. In a monolithic implementation, developers will separately implement and optimize these tasks in the respective VNFs. Among other consequences, the benefits of optimization in one implementation cannot be leveraged into others because of the tight coupling between the tasks.

An elaborate qualitative discussion on the drawbacks of repeating common tasks across VNFs in an SFC can be found in [29]. In this paper, we perform an experimental study to demonstrate possible performance implications of repeating common packet processing tasks in an SFC by comparing between the following two deployment configurations: (i) Click [30] based monolithic VNFs chained using virtual Ethernet (veth) pairs (Fig. 2(a)); and (ii) a single Click configuration implementing the functionality of the same SFC from configuration-i, while removing the repeated common elements (Fig. 2(b)). For both cases we play the same traffic (HTTP packet trace generated from access log for a moderate size public web-service (≈15K hits/month)) and measure the average CPU cycles/packet required by each type of Click element. Our objective is to measure the wasted CPU cycles for repeating common tasks across an SFC. Note that this study complements that of the one presented in [27] by demonstrating the impact on an SFC rather than considering single middlebox applications.



(a) Monolithic VNFs chained with veth pairs (configuration-(i))

(b) One single optimized click configuration (configuration-(ii))

Fig. 2. Motivational Experiment Scenarios

We deployed the following simplified form of the SFC from Fig. 1(a): *Edge Firewall* → *Monitoring Function* → *Application Firewall*. We implemented our own Click elements (*HttpClassifier*, *CountUrl*, and *ValidateUrl*) when Click's element library did not have any elements with similar functionality. We also instrumented the Click elements to measure the number of CPU cycles spent in processing each packet.

We present the savings in CPU cycles obtained from removing repeated elements in the optimized configuration, *i.e.*, configuration-(ii) in Table I. We observed a per element savings of up to ≈70%. However, as shown in Table I, not all elements contribute equally to packet processing, hence, the overall gain at the end is 29.5%, which is still significant.

TABLE I
RESULTS FROM MOTIVATIONAL EXPERIMENT

Click Element Type	**CPU Cycles Saved in configuration-(ii)**	**Element Weight in configuration-(i)**
FromDevice	71.9%	0.22%
ToDevice	67.1%	0.25%
CheckIPHeader	65.1%	0.44%
HttpClassifier	48.28%	47.8%
Overall	**29.5%**	**–**

This result further motivates re-architecting VNFs by exploiting the commonality in packet processing in a way to achieve better resource utilization. To this end, we argue in favor of adopting a microservice-like architecture [34] for building VNFs and SFCs. We propose to disaggregate VNFs into independently deployable packet processors, that we call μNFs. VNFs or SFCs can then be realized by orchestrating a packet processing pipeline composed from the μNFs. With this, one can think of applying optimizations such as consolidating multiple instances of a common packet processing function into a single instance for better CPU utilization. We will experimentally demonstrate CPU utilization gains from using a μNF-based SFC over that composed from monolithic VNFs (*i.e.*, configuration-(i)) in Section VII-C2.

III. Design Goals and Choices

Our objective is to re-architect the VNFs by exploiting their overlapping functionality enabling finer-grained resource allocation and achieving better resource utilization. To achieve these objectives we start with the following design goals:

Reusability Frequently appearing packet processing functions should be developed once and shared across VNFs.

Loose-coupling: Packet processing functions should not be tightly coupled, so that they can be deployed and scaled independently, allowing fine-grained resource allocation.

Transparency: Implementation of a packet processing function should not be affected by their communication pattern (*e.g.*, one-to-one, one-to-many, *etc.*).

Lightweight communication primitives: Communication between packet processing elements should not incur significant overhead hurting the overall performance.

The first goal can be achieved by dividing large packet processing software into smaller packet processing tasks or functionality. Then to achieve the rest of the goals we have the following two design alternatives [35]:

Run-to-completion: Packet processors are implemented as a set of identical threads or processes, each implementing the entire packet processing logic (*i.e.*, an NF or even an SFC).

Pipelining: Packet processors are implemented by composing a pipeline of heterogeneous threads or processes, each performing a specific packet processing task.

The state-of-the-art modular VNF designs such as ClickOS [20] and NetBricks [23] have adopted a run to completion model, where packets are passed between different functions in the same address space and processed in a single thread or process. When more processing capacity is required, the whole VNF (or SFC) instance is scaled out and traffic is split between the instances using NIC features such as RSS [36]. One limitation of this model is that it is hard to right size resource allocation to individual components because of the tight coupling between them. In contrast, pipelining mode satisfies more of our design goals. Individual components can be allocated their own resource, independently deployed and scaled (loose-coupling), and it is easier to decouple how elements process packets from their underlying communication pattern (transparency).

IV. System Description

A. Assumptions

We assume that the network operator owning the infrastructure has control over the VNFs that are being deployed. These VNFs can be deployed at the telecommunication central offices or Internet Service Provider point-of-presences (PoPs) converted into edge data centers [37], [38]. When SFCs are deployed inside these edge data centers their VNFs are typically in the same layer-2 domain.

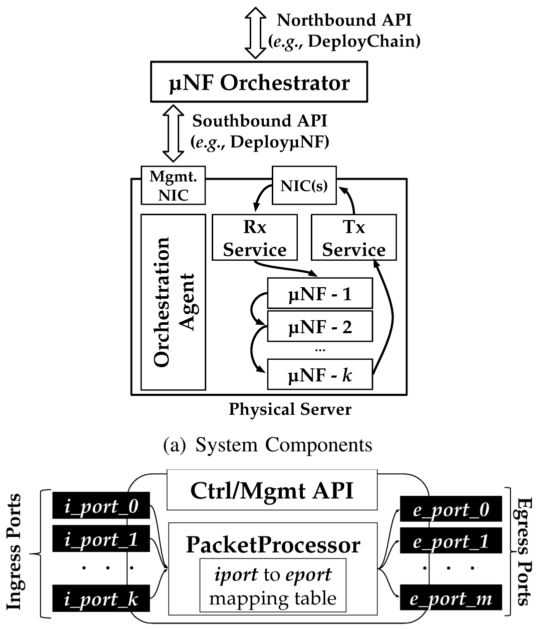
We do not consider Virtual Machines (VMs) as the choice of deployment for individual μNFs since that would add a significant overhead for μNF to μNF communication [23]. Moreover, we also do not require separate OSs and kernel features for deploying the μNFs, which is typically provided by VMs. Rather we choose using either processes or containers for μNF deployment. At this point we leave the choice of using processes or containers to the network operator since our evaluation results demonstrated similar performance.

We assume that the μNF descriptions (*e.g.*, what type of operation the μNF performs on what part of the packet header or payload) and template for composing VNFs from μNFs will be provided by the VNF providers. The SFC request will come from the network operator. Currently, we use JSON format for SFC specification. However, we do not restrict ourselves as to what can be used for specifying SFCs. We plan to support standards such as TOSCA [39] and YANG [40].

Finally, we assume that the μNF developers will provide configuration generator for each μNF. This will generate the necessary configuration for a μNF (*e.g.*, the types of communication primitives to create), when presented with a μNF type and its connectivity with neighboring μNFs.

B. System Architecture: Birds Eye View

A high level view of our system is presented in Fig. 3(a). It comprises the following components: a *μNF orchestrator*, per physical server *orchestration agent*, *μNFs*, and Rx and Tx services for reading packets from and to the NICs, respectively. The northbound API facilitates SFC life-cycle management and monitoring, and allows network operators to interact with the system. The μNF orchestrator is responsible for making global decisions such as μNF placement across physical servers to realize SFCs and make μNF migration decisions, among others.



(a) System Components

(b) μNF Architecture

Fig. 3. System Architecture

The orchestration agent acts as the local orchestration endpoint for a given machine. A southbound API between the global orchestrator and orchestration agents facilitates their communication. For example, the μNF orchestrator can use the southbound API for requesting local orchestration agents to allocate resources for μNFs, deploying μNFs with proper configuration and create the communication primitives for μNF to μNF communication.

The smallest deployable units in the system are the μNFs. μNFs usually perform a specific packet processing task and are independently deployable loosely-coupled entities. As described earlier in Section III, one of our design goals is to keep the μNFs simple and keep the communication pattern between μNFs transparent from how they process the packets.

Finally, we have two special μNFs, namely the Rx and Tx services, responsible for reading packets from and writing packets to the NIC, respectively. These two services collectively form a lightweight software data path for the μNFs. By isolating these two services from the μNFs we have the flexibility to adjust I/O batch sizes according to the consumption/production rate of the μNFs. Moreover, such separation allows us to make the operations on hardware transparent to other packet processing μNFs.

C. System Components

1) μNF Orchestrator: The μNF orchestrator is responsible for realizing an SFC by orchestrating a packet processing pipeline consisting of μNFs across multiple machines. Network operators can interact with the orchestrator through a north-bound API. The orchestrator is also responsible for global management decisions such as handling machine failures, making scaling decision, *etc.*

2) μNF Orchestration Agent: μNF orchestration agent is the local orchestration endpoint on a physical machine. It has a northbound API for the μNF orchestrator to act on it. The agent is responsible for performing local actions such as deploying μNFs, creating communication primitives to enable inter μNF communication on the same machine, *etc*.

3) μNFs: A μNF is the unit of packet processing in the system as well as the unit of deployment and resource allocation. It consists of a number of *IngressPort*s, a number of *EgressPort* and a *PacketProcessor* (Fig. 3(b)). The IngressPorts and EgressPorts provide methods to pull packets from and push packets to the previous and the next μNF in the packet processing pipeline, respectively. When μNFs from different VNFs are consolidated, the IngressPort to EgressPort mapping table helps in routing packets to different branches of the pipeline.

The aforementioned ports are of abstract type and can have different implementations (details in Section VI). One of our design goals is to keep packet processing logic of μNFs oblivious to μNF to μNF communication pattern. The port abstraction simplifies μNFs' design and implementation and keeps them loosely coupled with each other. For instance, we implement a *LoadBalancedEgressPort* that has the same interfaces as EgressPort. However, the implementation distributes packets to multiple next-stage IngressPorts in a round-robin fashion. From a μNF's point-of-view this distribution of packets to multiple next stage μNFs is completely transparent.

4) Rx Service: Rx service is the interface between host NIC(s) and the μNFs. Rx service keeps hardware specific configurations (*e.g.*, number of NICs, number of Rx queues) and operations (*e.g.*, flow classification in either hardware or software based on NIC capabilities) transparent to the μNFs. The Rx service can be thought of as a lightweight data path (similar to [41], except that complex data path functions are implemented as independent μNFs in our system).

5) Tx Service: Tx service sits between the μNFs and the host NIC. Common Tx specific tasks such as tagging packets of the same SFC, rewriting destination MAC address with next hop MAC address, writing packets to different NIC Tx queues, *etc*., are consolidated inside the Tx service.

D. SFC Deployment

As discussed earlier, the μNF orchestrator is the entry point for the network operators to deploy an SFC composed of μNFs. One of our goals is to ensure that from the network operators point-of-view the SFC request does not look different from what they are used to seeing, *i.e.*, they should not be required to specify μNF specific configurations. It is up to the orchestrator to determine the optimal composition of μNFs that offers the semantics of the user requested SFC.

1) Inputs: In what follows, we describe the inputs to the orchestrator in a bottom up fashion:

***μNF Descriptor*:** A μNF descriptor defines different attributes of a μNF. Currently, we support the following attributes: statefulness of the μNF and types of action (*e.g.*, No Operation (NOP), ReadOnly, or ReadWrite) a μNF performs on the packet headers at different protocol layers. For instance, the following is a descriptor for a layer 3-4 classifier that performs only ReadOnly operation on the packet headers:

```
PacketProcessorClass: "TCPIPClassifier"
Stateful: "Yes"
L2Header: "NOP"
L3Header: "ReadOnly"
```

Meta-data about the μNFs assist in performing optimizations (detail discussion in Section V) when composing SFCs.

***VNF templates*:** A VNF template is a blueprint of realizing a VNF from μNFs and we represented it by a packet processing graph composed of the constituent μNFs. VNF templates can be considered analogous to VNF descriptors defined in ETSI NFV MANO specification [42]. A VNF template consists of the nodes of the processing graph (*i.e.*, the μNFs) and the links representing the order of packet processing between μNFs. The links can be labeled with the output of the source μNF for that link. Labels act as a filter, *i.e.*, only packets producing results equal to the label are forwarded along that link. Examples of VNFs and VNF templates are presented in Fig. 1(b). If we take the Application Firewall VNF from Fig. 1(b) as an example, it is composed from six independently deployable μNFs. Annotations on the edges represent classification results at different stages, *e.g.*, whether a packet contains HTTP payload or not.

***SFC*:** An SFC request is a directed graph, where the nodes are the VNFs and a directed link between two nodes represents the order that traffic should follow. Links can have labels in an SFC indicating VNF specific output. μNF descriptors provided by VNF providers may include more or less information than what we have described. The lesser information they contain, the lesser constraints we may have in placing the constituent μNFs.

2) Sequence of Operations for SFC Deployment: The μNF orchestrator combines the constituent VNF templates of an SFC, removes redundant μNFs and builds a *μNF forwarding graph* with the same semantics as the SFC request. The graph construction phase can take μNF specific meta-data into account to perform optimizations such as consolidating multiple μNF instances of the same type into one and performing optimization such as parallelizing the executing of multiple μNFs on the same packet whenever possible.

After the μNF orchestrator builds an optimized μNF processing graph and determines the placement of μNFs, it then requests agents on the selected machines to deploy their parts of the graph. μNF orchestrator also generates configuration of each μNF in the graph by leveraging the developer provided configuration generators and provides the agents with these generated configurations. Upon receiving the μNF processing subgraph and the configurations, the agent first allocates the necessary resources, creates the communication primitives, and deploys and connects the μNFs using the instantiated communication primitives.

E. Auto-scaling

We propose to use a simple packet drop monitoring based mechanism to take auto-scaling decisions. Once μNFs are deployed, the local agents continuously monitor for packet drops on all EgressPort – IngressPort pairs. A consistent drop indicates that the μNF attached to the IngressPort is not able to match the processing rate of the μNF attached to the EgressPort. This triggers an auto-scaling event in the agent. The agent then spawns another instance of the bottleneck μNF and modifies the corresponding EgressPort in the pair to a LoadBalancedEgressPort (described in Section IV-C), which load balances traffic across the scaled-out instances.

However, there is a delay between detecting consistent packet drop and actually deploying another μNF instance to mitigate packet drops. Since the agent is continuously monitoring, it will keep seeing a packet drop during this period and trigger another scale-out event even before the first one completes. To avoid this, we assign a cool down timer to the μNF that is being scaled-out and do not trigger another scale-out event until the cool down timer has expired.

V. Optimizations

A. Pipelined Cache Pre-fetching

One potential issue that might arise from our design of μNF is when using multiple processors in a NUMA configuration. In such configuration, each processor socket has its local memory bank and the access time to local and remote memory banks are not uniform. Processing packets on a NUMA zone (*i.e.*, socket) other than the one where the NIC is attached has performance implications due to remote memory invocation. To circumvent this problem, we perform a pipelined cache pre-fetching inside every μNF. It works as follows. Before processing a batch of packets, a μNF first pre-fetches a cache-line from the first k packets in the batch. Then it proceeds to process the batch. While packet i from the batch is being processed, a cache-line from packet $i + k$ is pre-fetched into the cache. In this way, when a packet is being processed, the first level cache is very likely to be warm with a cache-line worth data from that packet (which contains the header fields). Thus potentially increasing the first level cache hit rate and masking the remote memory access latencies to some extent. We experimentally evaluate the impact of this optimization in Section VII-B2.

B. Parallel execution of μNFs

In a pipelined packet processing model, the packet processing elements typically operate on a batch of packets in a sequential manner. This is often unavoidable since one μNF only processes the set of packets as determined by the previous stage μNF. For instance, in Fig. 1(b), the L7 classifier μNF in the Application Firewall determines the set of packets to be processed by the URL Validator μNF. However, there are scenarios where sequential packet processing can be avoided. For example, in the monitoring function from Fig. 1(b), the counting function performs a read-only operation on the packets. Therefore, if another counting function was part of the Monitoring function, these two could be safely executed in parallel on the same set of packets.

We parallelize the execution of consecutive μNFs from the μNF processing graph that are placed on the same machine by employing techniques similar to the ones discussed in [25], [43], [44]. Parallelization is performed based on the type of operation they perform on the packet header (specified in μNF descriptor). When consecutive μNFs perform read-only operations on the packet header, or operate on disjoint regions of the header, or do not modify the packet stream (*e.g.*, not dropping packets), only then we parallelize their execution and assign them distinct CPU cores on the same NUMA zone. One issue with parallel execution is to ensure synchronization after the parallel processing stage, *i.e.*, a μNF β that is just after the parallel processing state, should be able to start processing a packet only if the packet has been processed by all the μNFs in the parallel processing stage. Such synchrony is achieved through special IngressPort and EgressPort implementations (details in Section VI-D). These ports embed a counter as packet meta-data before parallel execution begins. At the parallel execution stage, each μNF atomically increases the counter after its processing is complete. At μNF β, the IngressPort ensures that only packets with appropriate counter value are passed on to β's PacketProcessor. Moving the synchrony mechanism into ports thus keeps the μNF design simple.

VI. Implementation

One option for implementing the proposed system is to adapt existing modular packet processing frameworks such as Click [30] to a multi-process model. However, Click comes with a lot of legacy code, some of which is not useful for our case (*e.g.*, scheduling multiple elements inside a Click binary). Also, Click was originally designed and optimized for a run-to-completion packet processing model, which is fundamentally different from the pipeline model adopted by μNF. Therefore, re-engineering Click and similar systems require significant refactoring of many of their subsystems such as component scheduling, packet transfer, *etc.*, to make them efficiently work in a pipeline model. Finally, we wanted to build the system in a way such that it can process packets at 10 Gbps line-rate at least (current *de facto* capacity for commodity NICs) while maximizing CPU usage on the servers. It was becoming cumbersome to optimize Click's performance and refactor its subsystems for pipeline model, hence, we decided to build the system from scratch.

We have implemented a prototype of our system using C++ (agent and μNFs) and Python (orchestrator). At this point we focus more on developing the μNFs and their communication primitives. Therefore, our current orchestrator is limited in functionality and acts more as a convenience mechanism for testing. We use Intel DPDK [45] for kernel bypass packet I/O and `hugetlbfs` [46] for sharing memory between μNFs. We plan to open-source our current implementation in the near future. In the remainder of this section we describe the implementation of the system components.

A. Agent

Agents are implemented in C++ and run as primary DPDK processes. During initialization, an agent pre-allocates memory buffers for the NIC to store incoming packets, and exposes an RPC-based control API for the orchestrator. The orchestrator can use this API to deploy part of a μNF processing graph on a machine. When such a request is received by an agent, it deploys the μNFs according to the orchestrator specified configuration and creates the necessary communication primitives (details in Section VI-D). Agents also monitor the μNFs and take scaling out decisions.

B. μNF

μNFs are implemented by leveraging DPDK APIs. Each μNF runs as a stand-alone secondary DPDK process. Since DPDK allows only one process to be the primary, *i.e.*, have the privileges of memory allocation, μNFs run as secondary DPDK processes. When required, μNFs obtain pre-allocated objects from a memory pool shared with the agent. Memory sharing between μNFs and between a μNF and the agent is enabled by hugetlbfs. The hugetlbfs is mounted on a directory accessible to both the μNFs and the agent, and contains virtual to physical memory mapping of the shared memory regions. One caveat in this shared memory model is that each process should have exactly the same virtual address space layout in order to successfully translate the shared virtual memory to their physical locations. To do so we had to disable Address Space Layout Randomization (ASLR), a Linux kernel feature for preventing buffer overflow attacks [47]. This is a security vulnerability and is a limitation in our current implementation. However, this is also a limitation of the technology at hand and solving it can be an interesting future work.

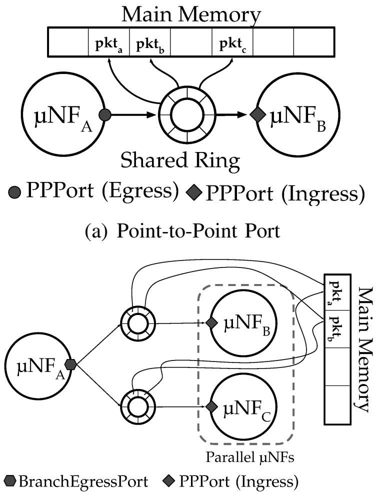
C. Rx and Tx Services

In our design, packet I/O is handled by Rx and Tx services in order to hide hardware specifics from the other μNFs. In our prototype implementation, the Rx service runs as a separate thread inside the agent and is pinned to a physical CPU core on the same socket where the NIC's PCIe bus is attached. It receives packets from a NIC queue in batches and implements a classifier that dispatches the packets to the appropriate μNFs. Currently, the classifier is based on matching the following 5-tuple flow signature: *(source-IP, dest-IP, ip-proto, src-port, dst-port)*.

The Tx service abstracts the NIC Tx queues and implements common functions frequently required by the μNFs. For example, in a multi-node deployment scenario, when a μNF processing graph is deployed across multiple machines, the Tx service encapsulates the packets belonging to a μNF graph destined to another machine in a custom layer 2 tunnel with appropriate tag and destination MAC addresses. The Rx service on the other end of the tunnel distributes packets to the appropriate μNFs based on the tags. These tags are determined and configured by the orchestrator.

D. Port

As discussed earlier, a port provides packet I/O abstraction for μNFs and decouples the implementation of a specific communication pattern from a μNF's packet processing logic. This design choice helps to keep the μNF implementation focused only on the packet processing part. We have two broad classes of ports, *IngressPort* for receiving packets from and *EgressPort* for sending packets to μNF(s). If not stated otherwise, ports provide a zero-copy packet exchange mechanism by exchanging the packet addresses instead of full copies of the packets. IngressPort and EgressPort present the following interfaces to the μNFs while hiding underlying implementation details: (i) pull based *IngressPort::RxBurst*, which populates an array with a burst of packet addresses; (ii) *EgressPort::TxBurst* pushes a burst of packets to the next μNF. Currently, we have the following specific implementations of IngressPort and EgressPort that allow different communication patterns between μNFs.



(a) Point-to-Point Port

(b) Branched Egress Port

Fig. 4. Port Implementations

1) NIC I/O Port: A NIC I/O port abstracts the rx/tx queues in the hardware NIC. It allows μNFs to directly read from or write to the NIC. We have leveraged the NIC specific DPDK poll mode drivers (PMDs) for implementing ingress and egress versions of NIC I/O Port. The DPDK PMDs bypass the OS kernel and allow zero copy packet I/O from the NIC.

2) Point-to-Point Port: A point-to-point port allows a μNF to push packets to or pull packets from exactly one other μNF. We have implemented this port using a circular queue (Fig. 4(a)). The ingress version of the port (*PPIngressPort*) pulls a batch of packet addresses from a circular queue and the egress version (*PPEgressPort*) pushes packet addresses for a batch of packets to the queue. When a μNF's PPIngressPort and another μNF's PPEgressPort share the same circular queue, they can exchange packets with each other. The circular queue in our implementation is an instance of `rte_ring` data structure (a lock-less multi-producer multi-consumer circular queue) from DPDK `librte_ring` library.

3) BranchEgressPort: This port connects a μNF to multiple μNFs that are processing packets in parallel. For instance, in Fig. 4(b), μNF$_B$ and μNF$_C$ are executing in parallel. To realize this execution model, μNF$_A$ can be made aware of this configuration and pushes packet addresses to both of the next state μNFs. μNF$_A$ will also need to embed the necessary metadata in packets to mark the completion of μNF$_B$ and μNF$_C$. This violates our design principle of loose coupling between μNFs, and therefore, we developed BranchEgressPort to transparently handle this type of branching. A BranchEgressPort contains multiple circular queues, each corresponding to one μNF in the next stage. Each of the circular queues can be shared with a PPIngressPort to create a communication channel. For example, one of the circular queues of μNF_A's BranchEgressPort is essentially the underlying circular queue of μNF_B's PPIngressPort. A BranchEgressPort also initializes and embeds a counter inside each packet's meta-data area, which is used to mark the completion of packet processing by all parallel μNFs.

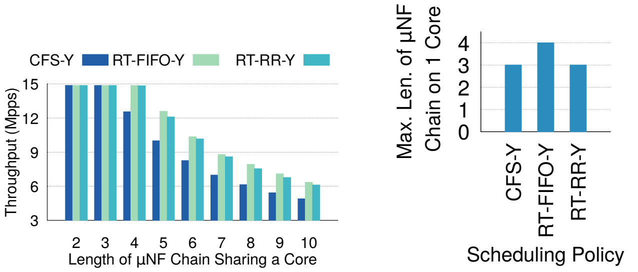
4) MarkerEgressPort: A MarkerEgressPort works in conjunction with a BranchEgressPort. It is the typical EgressPort of a μNF part of a parallel processing group. This port atomically increases the embedded counter in the packet before putting the packet into a shared circular queue.

5) SyncIngressPort: A SyncIngressPort connects a set of parallel μNFs to a single μNF that is potentially modifying packets. This port is also an abstraction over a shared circular queue. The queue is shared with other MarkerEgressPorts in the parallel processing group. SyncIngressPort ensures that any packet that is pulled out has been processed by all the parallel μNFs. This synchronization is done by atomically checking the counter embedded inside every packet by a BranchEgressPort. SyncIngressPort pulls a packet only if the counter value equals the number of μNFs in the parallel processing stage. In this way, the next stage of a parallel processing stage proceeds to process a packet only after all the μNFs from the parallel processing stage have completed their processing. Note that in order to keep the cost of atomically updating and checking the embedded counters, we leverage the atomic instruction set of modern CPUs.

6) LoadBalancedEgressPort: This is an EgressPort that load balances packets pushed by a μNF to a number of next stage μNFs. This port is particularly useful when μNFs are scaled-out. Consider two μNFs a and b, connected with a pair of ingress and egress point-to-point ports. If b is scaled out then packets from a need to be load balanced across b instances. This port transparently performs this load balancing. Our current implementation has a round-robin load balancing policy. However, more complex policies (*e.g.*, ensuring flow affinity) can also be implemented using this abstraction.

E. μNF Scheduling

In order to increase μNF density per physical machine, we share a CPU core between multiple consecutive μNFs from a μNF processing graph. This also enables these consecutive μNFs to better utilize a CPU's warm first level cache. However, like many other DPDK applications, μNFs operate in busy polling mode. Therefore, it can occur that one μNF out of several others sharing the same CPU core, gets scheduled on that core, and there is no packet at that moment to process. This will waste CPU cycles during the time allocated for the μNF. Therefore, a major challenge here is to carefully schedule μNFs to minimize the wasted CPU cycles. This is a problem of its own and merits separate investigation as seen in the literature [48]. For our prototype implementation, we aim to have a simple yet effective solution and first explore which out of the box OS scheduler is the most suitable one.



(a) Impact of scheduler and scheduling policy on μNF chains (sharing same CPU core)

(b) Max. length of a μNF chain able to sustain line rate (64B) while sharing a core

Fig. 5. Impact of Different Scheduling Schemes

Completely Fair Scheduler (CFS) is the default scheduler in most Linux distributions [49]. CFS ensures fair sharing of a CPU between competing processes by periodically preempting them. However, there are other schedulers available in the kernel, *e.g.*, the *Real Time (RT)* scheduler [50]. RT scheduler supports the following two scheduling policies: *First-in-First-out (FIFO)* and *Round Robin* (RR). Unlike CFS, RT scheduler does not ensure fairness, rather it ensures that a process only releases a CPU after it has finished (FIFO) or its allocated time quantum has expired (RR). To better understand which scheduler and scheduling policy is a best fit, we performed the following experimental study.

We deployed μNF chains of varying lengths on a single CPU core, where each μNF performs very minimal packet processing (swaps source and destination MAC addresses). We measured the throughput of these chains for smallest size (64 byte) packets using different scheduler and policy combinations, namely CFS, RT with FIFO, and RT with RR. We observed that CFS was preempting the μNFs too frequently. As a consequence, there was a significant context switching overhead and μNFs from the chain were being scheduled when there was no packet available in their IngressPorts. RT scheduling was not performing well either since μNFs were getting uneven CPU time and were starving. We observed a throughput of only a few thousand packets per second.

Therefore, we added the following optimization in the μNFs. A μNF voluntarily yields CPU in the following events: (i) when there are no packets available in its IngressPort to process, and (ii) after successfully processing k batches of packets. This optimization (voluntary yielding) improved the throughput by three orders of magnitude. We present results for different scheduler and scheduling policies with voluntary yielding optimization in Fig. 5.

Fig. 5(b) shows the maximum length of a μNF chain that can be deployed on a CPU core while maintaining 10Gbps line rate throughput for 64B packets ($\approx$14.88 Mpps). We found that voluntary yielding with RT scheduling and FIFO policy can support the maximum number of chained μNFs while operating at line rate. In Fig. 5(a) we demonstrate the throughput for μNF chains of varying lengths sharing a single CPU core for different combinations of scheduler and policy. From our empirical evaluation, it is clear that the best combination to use is voluntary yielding with RT scheduler and FIFO policy, which is able to sustain higher throughput for any chain length compared to any of the other combinations. The reason being, CFS preempts a process as soon as its allocated time quantum expires. Therefore, CFS can preempt a μNF in the middle of processing a batch, making it less likely for the next scheduled μNF to get packets from its IngressPort, thus wasting CPU cycles. RT with FIFO mitigates the impact of preempting. By combining voluntary yielding, we prevent other μNFs from starving.

VII. Performance Evaluation

A. Experiment Setup

1) Hardware Configuration: Our testbed consists of two machines connected back-to-back without any switch. One of them hosts the traffic generator, while the other hosts the μNFs. Each machine is equipped with 2×6-core Intel Xeon E5-2620 v2 2.1 Ghz CPU (hyper-threading disabled), 32 GB memory (distributed evenly between two sockets), and a DPDK compatible Intel X710-DA 10 Gbps NIC.

2) Software Environment: We used DPDK v17.05 on Ubuntu 16.04LTS (kernel version 4.10.0-42-generic). We disabled Address Space Layout Randomization (ASLR) to ensure a consistent hugepage mapping across the μNFs. We also allocated a total of 4GB hugepages (evenly divided between sockets). Additionally, we configured the machines with the following performance improvement features:

- We isolated all CPU cores except core 0 on socket 0 from the kernel scheduler. μNF processes and agent threads were pinned to these isolated CPUs.
- CPU scaling governor was set to *performance*.
- Flow control in the NIC was disabled.

3) Prototype μNFs: We developed the following μNFs and used them for different scenarios:

- **MacSwapper**: Swaps the source and destination MAC address of each packet.
- **IPTtlDecrementer**: Parses IP header and decrements time-to-live (TTL) field by 1.
- **CheckIPHeader**: Computes and checks the correctness of IP checksum of each packet.
- **L3L4Filter**: Filters packets based on Layer 3-4 signature.
- **HttpClassifier**: Determines if a packet is carrying HTTP traffic by checking the payload.
- **ValidateUrl**: Performs a regular expression matching on URL in HTTP header to detect URLs containing SQL injection attacks.
- **CountUrl:** Counts the number of packets in a batch that contains a certain URL in its payload.

4) Traffic Generation: We used *pktgen-dpdk* [51], and *Moongen* [52] for throughput and latency measurements, respectively. We determine the physical limits of our setup by modifying the agent to receive batches of packets and echo them back (single thread pinned on a CPU core). We observed line rate throughput from this setup (*i.e.*, 10 Gbps for all packet sizes), hence, there are no bottlenecks present in the hardware or configuration. For latency measurements, we set the packet rate to 90% of maximum sustainable rate for that particular deployment scenario.

B. Microbenchmarks

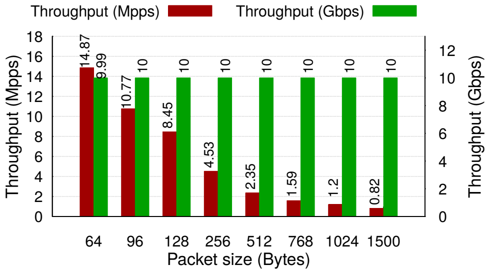


Fig. 6. Baseline Performance

1) Baseline Performance of μNF: We first establish the baseline performance that can be achieved by disaggregating larger VNFs into μNFs. We pinned the agent's Rx thread to a CPU core and run a very simple μNF (MacSwapper) pinned to a different CPU core in the same NUMA zone. We vary packet size from 64 to 1500 Bytes and report the throughput in Fig. 6. Throughput reaches line rate for smallest packet size on 10 Gbps NIC. We also deployed the same μNF inside a Docker container and performed the same experiment to observe any potential impact of containerization. Throughput results for containerized μNF are very similar to those presented in Fig. 6, and are hence not presented.

2) Impact of Pipelined Cache Pre-fetching: We intend to utilize all available CPU cores on a machine for deploying the μNFs. However, in a NUMA system with multiple CPU sockets, processing packets on a NUMA zone other than the one where the packet was received can cause performance degradation due to remote memory access overhead [53]. In this experiment, we evaluate the impact of cache pre-fetching optimization from Section V-B when packets are processed by μNFs on different NUMA nodes.

We receive packets on NUMA zone 0 and process them through a chain of two MacSwapper μNFs deployed on separate cores at NUMA zone 1. We measure throughput of this chain (for smallest size packets) while varying the number of pipelined pre-fetched packets up to 50% of packet batch size (batch size is set to 64). The results are shown in Fig. 7. With pre-fetching disabled throughput drops to ≈30% of line rate. However, with as little as ≈20% packets pre-fetched to cache in a pipeline (8 out of 64 packets in a batch), throughput

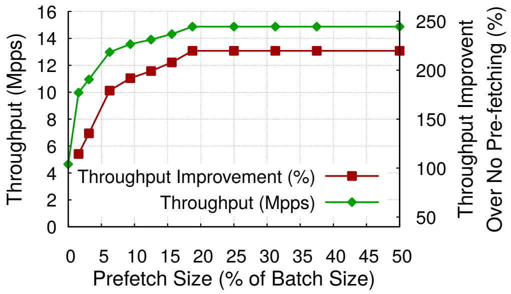


Fig. 7. Impact of Pipelined Cache Pre-fetching

improves by more than ≈3× and goes back to the line rate for smallest packet size.

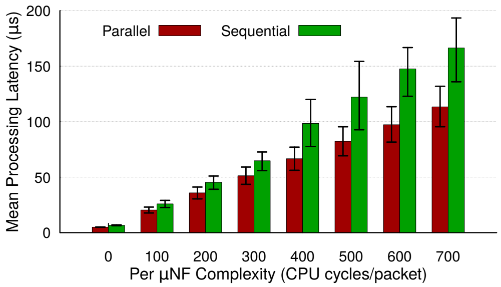
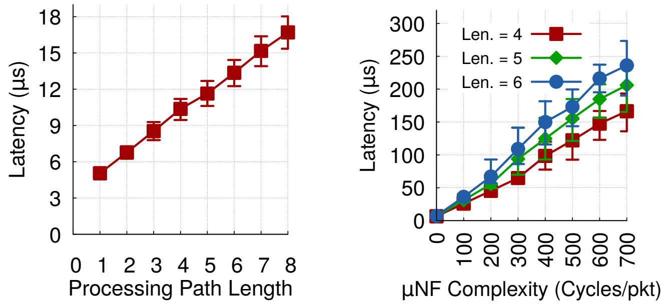


Fig. 8. Impact of Parallelism in Processing Graph

3) Impact of Parallelism in μNF Processing Graph: Intuitively, parallel execution of μNFs in the processing graph is expected to reduce the processing latency for the packets through μNF processing graph. However, overheads are associated with parallel executions because of atomically increasing a counter on each packet during branching and synchronizing as described in Section V-B. Depending on how fast a μNF is processing packets, we may observe different impacts of parallelism. To evaluate the effect of parallelism for different packet processing costs, we add an artificial busy loop after processing each packet in `MacSwapper` μNF. We create a pipeline from four of these μNFs connected linearly for the sequential case. For the parallel case, we create a two-way branching after the first μNF (using BranchEgressPort) and join the branches at the last μNF (using SyncIngressPort). We vary the per packet processing cost from 100 to 700 CPU cycles. We measure packet processing latency of the sequential and parallel configurations using Moongen.

Results of this experiment (mean latency with 5th and 95th percentile error bars) are shown in Fig. 8. When a μNF's processing cost is low (*e.g.*, less than 100 cycles/packet), the gains from parallelism are rather marginal compared to the sequential case (less than 10% improvement in latency). The gains become more evident when μNFs' packet processing cost increases and we see a good potential for improving latency there (more than 20% for μNFs with 700 CPU cycles per packet processing cost).

4) Impact of μNF Processing Graph Diameter: We create μNF chains of different lengths (varied from 4 to 6) and measure packet processing latency along the pipeline using Moongen. The objective is to observe if packets start queuing up in any stage of the processing pipeline or not. We have an experiment setup similar to the scenario in Section VII-B3. We first measure latency with varying chain lengths and without introducing any additional packet processing complexity in our MacSwapper μNF. In this case, we observe a linear increase in mean latency (Fig. 9(a)). Then we introduce additional busy loops to emulate CPU cycles spent for packet processing (similar to Section VII-B3) and measure latency for different lengths of μNF packet processing path. As we observe from Fig. 9(b), latency increases linearly with μNF complexity as well as with μNF processing path length. Therefore, no buffering issues were encountered along the pipeline.



(a) Latency as a Function of Chain Length

(b) Latency as a Function of Packet Processing Cost

Fig. 9. Impact of μNF Processing Path Length

C. Service Level Performance

1) Resource Efficiency over Run-to-Completion Mode: We compare μNF with NetBricks [23], the state-of-the-art in software packet processing platform operating in run-to-completion model. In particular, we perform the same experiment as in [23] to reproduce results from Fig. 7 of the original paper [23]. We developed similar packet processing element using μNF (IPTtlDecrementer) as the one used in [23] and deployed chains of different length in the following configurations: (i) *NB-MC*: NetBricks with multiple threads, each pinned to a dedicated CPU core, (ii) *NB-1C*: NetBricks with single thread, (iii) *μNF-1C*: all μNFs packed on a single CPU core, and (iv) *μNF-MC*: the chain is divided into k clusters of consecutive μNFs such that each cluster packs maximum number of μNFs to sustain line rate while sharing a CPU core. For a fair comparison, for both NetBricks and μNF we read packets from NIC without intervention from a software switching layer. Note that in the original paper [23], the authors spawned ℓ threads for a chain of length ℓ in NB-MC configuration and were able to reach more than 10Gbps throughput. However, we do not have similar hardware in our disposal at this moment, hence, for each chain length, we deploy the minimum number of threads on distinct CPUs until NetBricks reaches line rate for smallest packets. We also performed the suggested performance tuning as in [23], [54].

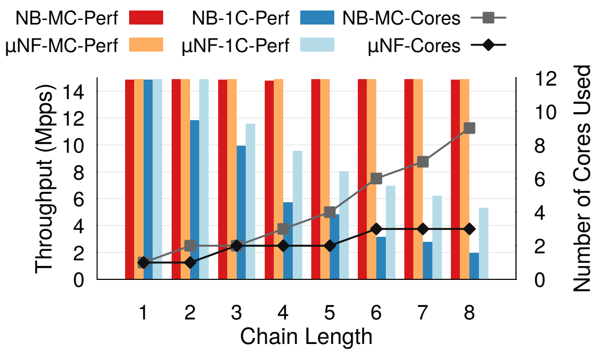


Fig. 10. Comparison with NetBricks [23]

The results from this experiment are shown in Fig. 10. For the single CPU core scenario (*i.e.*, NB-1C and μNF-

1C), μNF achieves better throughput than that of NetBricks with increasing chain length. Because of operating in a run-to-completion mode, NetBricks starts processing a new batch of packets only after the previous batch has finished processing through all the elements in the chain. In contrast, because of its pipeline mode, μNF can schedule a packet processing element to work on a new batch of packets as soon as that element has finished processing the previous batch and hands it over to the next element in the chain. This effectively increases the number of packets in the pipeline, resulting in a better packet processing throughput compared to NetBricks as demonstrated in Fig. 10. Indeed, there is context switching overhead involved in a pipeline mode. However, by carefully yielding the CPU as discussed in Section VI-E, μNF minimizes the impact of such overhead on packet processing throughput.

We also observe from Fig. 10 that for a given chain length, μNF can reach line rate using lesser number of CPU cores compared to NetBricks. This is because, in pipeline mode with appropriate scheduling, it is possible to reduce wastage of CPU cycles and use the CPUs more effectively between packet processing stages, compared to run-to-completion mode. However, to be fair in the comparison, NetBricks provides packet ownership transfer by using underlying compiler features, which is not provided by μNF. Another caveat in the result is that, when we used more than 5 cores for NetBricks, the packets crossed a NUMA zone, which caused some performance penalty as we can see from the non-linear core scaling for longer chains.

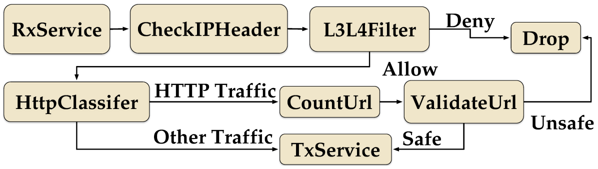


Fig. 11. μNF Realization of the SFC from Fig. 2(a)

2) Performance of μNF-based SFC: We have developed a set of μNFs (described in Section VII-A3) for realizing realistic VNFs and SFCs. We use these μNFs to deploy the SFC used for the motivational experiment in Section II, *i.e.*, *Firewall → Monitor → Application Firewall*. The resulting μNF processing graph is shown in Fig. 11. We implemented each individual μNF as close as possible to their Click counterpart. We played the same traffic trace used in Section II. Results in Table II show the relative savings in mean CPU cycles per packet when using μNF processing graph over monolithic VNFs (*i.e.*, configuration-(i) from Section II). To be fair, we did not compare packet I/O from NIC since it is fundamentally different between μNF and Click. We counted the cycles spent in reading to/from ring-based shared memory since that is an added overhead in this disaggregated architecture. We also benchmarked the deployment from Fig. 11 using pktgen. We set the packet size to 200 bytes, the average packet size reported in a recent study on a production data center [55]. Throughput reached 2.08 Mpps or 3.67 Gbps. We identified the HttpClassifier μNF to be a bottleneck through a separate benchmark. To test the scaling out of individual μNFs and LoadBalancedEgressPort, we deployed the same SFC but with two instances of HttpClassifier. We observed a near linear increase in throughput, which is 4.1 Mpps or 7.2 Gbps.

TABLE II
CPU CYCLES SAVED PER-PACKET ON AVERAGE

Click Element Type/μNF	CPU Cycles Saved in μNF	Element Weight in configuration-(i)
CheckIPHeader	27.8%	0.44%
HttpClassifier	28.9%	47.8%
Overall	**16.8%**	–

VIII. DISCUSSION

Decomposition of Monolithic VNFs into μNFs: In this paper, we proposed a system to compose SFCs and VNFs from independently deployable loosely-coupled μNFs. Orthogonal to the system design is the identification of the set of μNFs in the first place. From our initial survey this appears to be rather challenging primarily because it requires domain specific knowledge. Also determining the granularity of such tasks is also non-trivial. On one hand, most of the academic works propose low level packet processing functions (*e.g.*, TCP processing functions [25]) as VNF building blocks. On the other hand, state-of-the-art commercial VNFs [56] are composed from coarser-grained building blocks. Finer granularity increases re-usability whereas coarser granularity reduces overhead. *The best way to decompose a VNF into μNFs* remains an interesting research question.

Packet Ownership Transfer: When a μNF is finished processing a packet and transfers it to another μNF, the ownership of the packet should be transferred to that other μNF as well, *i.e.*, the previous μNF should not be able to access the packet content using the previously acquired packet handler. Virtual switches provide this abstraction by copying packets between ports, so, the previous copy becomes invalidated. However, this is a difficult problem to solve using a shared memory subsystem. In our implementation, μNFs rely on the `hugetlbfs` to obtain virtual-to-physical memory translation of the packet addresses. This file system should be accessible to the μNFs to ensure that they can always obtain a valid translation. This requirement also raises the issue that μNFs can always read packet content even after the packet has been transferred to other μNFs, and consequently, ownership is not transferred. Ownership transfer between multiple processes has been studied in HPC systems [57]. However, the state-of-the-art in that area still performs at least one message copy, which in our case would add a significant latency in packet processing. *Ownership transfer in shared-memory multi-process system with zero-copy* remains an open question. As a workaround in our implementation we create disjoint segments in the huge table area and assign one area to μNFs of the same processing graph. This does not solve the problem 100%, however, it provides isolation between μNFs from different processing graphs.

Future Work: In this paper, our design and implementation was primarily focused on developing a working solution and addressing the engineering challenges for enabling VNF and

SFC composition from independently deployable μNFs while operating at line rate. We also focused our evaluation on whether our prototype system is able to deliver its promise, *i.e.*, provide better resource utilization by eliminating redundancies and finer grained resource allocation. However, to get the best out of such architecture there are other interesting research questions that we have not addressed in this paper. These include μNF graph optimization to incorporate parallelization and consolidation of μNF instances, optimal placement of μNF graph across multiple machines, scaling out μNF instances across multiple machines, state management between scaled out instances, fault-tolerance, and scheduling of μNF instances for better resource utilization among others. Some of these problems may be of interest to the broader research community. For our part, we plan to improve our current system by addressing some of the aforementioned problems in the future.

IX. Related Works

A. Modular Packet Processing

The development of modular packet processing software has a long history that dates back to the late 90s. Click [30], one of the most influential works in this area proposed to build monolithic packet processing software using reusable packet processing components called *elements*. Click's focus was more on the programmability than performance. Over the years, Click influenced a significant body of subsequent research on building modular yet high performance packet processing platforms that employed different optimization techniques of their own (*e.g.*, NIC offloading, I/O batching, kernel bypass, *etc.*) to improve packet processing performance and add flexibility to VNF composition [20], [23], [58], [59]. However, these are centered around the assumption that a middlebox is a monolithic software. mOS proposed to abstract layer 4-7 packet processing tasks into modular and high-performance libraries for the ease of middlebox development [60]. mOS is complimentary to our work on disaggregating VNFs and can facilitate μNF development.

More recently, Slick [22] and OpenBox [24] proposed different approaches to achieve a similar goal of building packet processing from independently deployable components. Slick focuses more on the programming model for middlebox composition while OpenBox goes one step further and decouples data and control planes of VNFs. In contrast to μNF, OpenBox does not focus on addressing the engineering challenges pertaining to realizing a data plane for modular VNFs and SFCs. Its focus is more on the control aspects such as designing a protocol between VNF control and data planes, optimizing the forwarding graph, *etc*. OpenBox can complement our proposed system by acting as a control and orchestration layer above μNFs.

A chaining mechanism for lightweight VNFs has been proposed in [61], which inserts per-VM SFCs between a VM and a virtual switch for providing QoS, security, and monitoring services. In contrast, our focus is not on per-VM services, rather, on a general software architecture for realizing VNFs and SFCs from lightweight, independently deployable, and loosely-coupled packet processing components. An elaborate discussion on the challenges associated with realizing such microservice-based VNFs and SFCs can be found in [29]. An area of research orthogonal to modular and lightweight packet processing is runtime systems built around unikernels [62]. Unikernels are minimalistic OSs that are custom made to run only a single application, thus losing the benefit of being general purpose OSs. However, they have very low memory footprint (a few megabytes) and high deployment density (order of hundreds per physical machine) compared to traditional VMs or containers [20], [63], hence, can be a potential choice for μNF deployment.

B. Industry Efforts in Microservice-based VNFs

There has been some movement in the industry for redesigning large VNFs using microservice architecture. As part of the CORD project [37], a number of VNFs have been decomposed into having separate control and data planes that are loosely coupled and can be independently scaled. Another example is the Clearwater IP Multimedia System [56] rearchitected using microservices design principle and also made available as an open-source software. However, the independently deployable components themselves are rather complex and can be further decomposed into more manageable sizes. The availability of Clearwater as an open-source software has also fostered academic research, including on enhancing its auto-scaling capabilities [64], [65], and service latency and failure recovery time [66].

C. Middlebox Functionality Consolidation

CoMb [27] is one of the early works to experimentally motivate the consolidation of common functionality into separate services and share them across VNFs. However, CoMb's main focus was not to address the implementation issues related to realizing such a system, rather demonstrate the advantage of consolidating multiple NFs on commodity hardware as opposed to using purpose-built hardware middleboxes. E2 [10] proposed to consolidate management tasks such as resource allocation, fault-tolerance, monitoring, auto-scaling, *etc.*, into a single framework, which is orthogonal to μNF or CoMb's objective. More recently, Microboxes proposed to consolidate TCP protocol processing functions (*e.g.*, TCP bytestream reconstruction, TCP endpoint termination, *etc.*) of multiple middleboxes [25]. Consolidation has the advantage of reducing redundant development efforts in implementing and optimizing common tasks. In this paper, we focus on the engineering efforts related to software architecture, necessary abstractions, and performance optimizations for realizing such a disaggregated packet processing platform, facilitating better consolidate of packet processing tasks with ease.

X. Conclusion

In this paper, we propose μNF, a system for building VNFs and SFCs from reusable, independently deployable, and loosely-coupled components enabling finer-grained resource allocation. Our design goal is to keep the μNFs simple and develop the necessary primitives to transparently enable different communication patterns between them. We demonstrated the effectiveness of our system through a DPDK based prototype implementation and experimental evaluation. The individual techniques used for implementing and optimizing the system are not entirely new (*e.g.*, batched I/O, zero-copy I/O, pre-fetching, *etc*.). However, the bigger picture here is to demonstrate that disaggregating complex VNFs using the proposed software architecture combined with the individual techniques is indeed a viable and competitive solution for composing VNFs and SFCs. This is further supported by our experimental evaluation showing that the combined engineering effort enables finer-grained resource allocation and scaling while attaining comparable performance compared to the state-of-the-art monolithic implementations.

References

[1] S. R. Chowdhury, Anthony, H. Bian, T. Bai, and R. Boutaba, “μNF: A disaggregated packet processing architecture,” in *Proc. of IEEE NetSoft*, 2019, pp. 342–350.

[2] B. Carpenter and S. Brim, “Middleboxes: Taxonomy and issues,” RFC 3234, February 2002.

[3] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, “Making middleboxes someone else’s problem: network processing as a cloud service,” *ACM SIGCOMM Computer Comm. Rev.*, vol. 42, no. 4, pp. 13–24, 2012.

[4] “Network Functions Virtualisation – Introductory White Paper,” White paper, Oct 2012, accessed: Apr 05, 2017. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[5] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, “Network function virtualization: State-of-the-art and research challenges,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[6] J. G. Herrera and J. F. Botero, “Resource allocation in NFV: A comprehensive survey,” *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[7] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, “Embark: securely outsourcing middleboxes to the cloud,” in *Proc. of USENIX NSDI*, 2016, pp. 255–273.

[8] “Open platform for nfv (OPNFV),” accessed: Apr 09, 2017. [Online]. Available: https://www.opnfv.org/

[9] “Open source MANO,” accessed: Apr 09, 2017. [Online]. Available: https://osm.etsi.org/

[10] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, “E2: a framework for nfv applications,” in *Proc. of ACM SOSP*, 2015, pp. 121–136.

[11] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, “Rollback-recovery for middleboxes,” in *Proc. of ACM SIGCOMM*, 2015, pp. 227–240.

[12] Y. Kanizo, O. Rottenstreich, I. Segall, and J. Yallouz, “Optimizing virtual backup allocation for middleboxes,” in *Proc. of IEEE ICNP*, 2016, pp. 1–10.

[13] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, “REINFORCE: achieving efficient failure resiliency for network function virtualization based services,” in *Proc. of ACM CoNeXT*, 2018, pp. 41–53.

[14] J. Duan, X. Yi, S. Zhao, C. Wu, H. Cui, and F. Le, “NFVactor: A resilient nfv system using the distributed actor model,” *IEEE Journal on Sel. Areas in Comm.*, vol. 37, no. 3, pp. 586–599, 2019.

[15] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, “Split/merge: System support for elastic execution in virtual middleboxes.” in *Proc. of USENIX NSDI*, 2013, pp. 227–240.

[16] A. G. Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, “OpenNF: Enabling innovation in network function control,” in *Proc. of ACM SIGCOMM*, 2014, pp. 163–174.

[17] M. Kablan, A. Alsudais, E. Keller, and F. Le, “Stateless network functions: Breaking the tight coupling of state and processing,” in *Proc. of USENIX NSDI*, 2017, pp. 97–112.

[18] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, “Elastic scaling of stateful network functions,” in *Proc. of USENIX NSDI*, 2018, pp. 299–312.

[19] H. Hantouti, N. Benamar, T. Taleb, and A. Laghrissi, “Traffic steering for service function chaining,” *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 487–507, First quarter 2019.

[20] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, “ClickOS and the art of network function virtualization,” in *Proc. of USENIX NSDI*, 2014, pp. 459–473.

[21] J. Hwang, K. K. Ramakrishnan, and T. Wood, “Netvm: High performance and flexible networking using virtualization on commodity platforms,” in *Proc. of USENIX NSDI*, 2014, pp. 445–458.

[22] B. Anwer, T. Benson, N. Feamster, and D. Levin, “Programming slick network functions,” in *Proc. of ACM SOSR*, 2015, pp. 14:1–14:13.

[23] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, “Netbricks: Taking the V out of NFV,” in *Proc. of USENIX OSDI*, 2016, pp. 203–216.

[24] A. Bremler-Barr, Y. Harchol, and D. Hay, “Openbox: A software-defined framework for developing, deploying, and managing network functions,” in *Proc. of ACM SIGCOMM*, 2016, pp. 511–524.

[25] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood, “Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions,” in *Proc. of ACM SIGCOMM*, 2018, pp. 504–517.

[26] J. Duan, X. Yi, J. Wang, C. Wu, and F. Le, “Netstar: A future/promise framework for asynchronous network functions,” *IEEE Journal on Sel. Areas in Comm.*, vol. 37, no. 3, pp. 600–612, 2019.

[27] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, “Design and implementation of a consolidated middlebox architecture,” in *Proc. of USENIX NSDI*, 2012, pp. 323–336.

[28] Surendra, M. Tufail, S. Majee, C. Captari, and S. Homma, “Service function chaining use cases in data centers,” IETF Secretariat, Internet-Draft draft-ietf-sfc-dc-use-cases-06, February 2017.

[29] S. R. Chowdhury, M. A. Salahuddin, N. Limam, and R. Boutaba, “Re-architecting NFV ecosystem with microservices: State of the art and research challenges,” *IEEE Network*, vol. 33, no. 3, pp. 168–176, 2019.

[30] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 217–231, 1999.

[31] M. Gallo and R. Laufer, “ClickNF: a modular stack for custom network functions,” in *Proc. of USENIX ATC*, 2018, pp. 745–757.

[32] “Blue coat® systems proxysg™,” Tech. Rep., accessed: Apr 09, 2017. [Online]. Available: https://bto.bluecoat.com/sites/default/files/tech_pubs/SGOS_4.3.1_Upgrade_Downgrade.pdf

[33] “Barracuda web application firewall,” accessed: Apr 09, 2017. [Online]. Available: https://www.barracuda.com/products/webapplicationfirewall

[34] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: yesterday, today, and tomorrow,” in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[35] C. Dumitrescu, “Design patterns for packet processing applications on multi-core intel architecture processors.” White Paper, December 2008.

[36] “Receiver side scaling,” accessed: Apr 09, 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

[37] L. Peterson, A. Al-Shabibi, T. Anshutz, S. Baker, A. Bavier, S. Das, J. Hart, G. Palukar, and W. Snow, “Central office re-architected as a data center,” *IEEE Communications Magazine*, vol. 54, no. 10, pp. 96–101, 2016.

[38] R. S. Montero, E. Rojas, A. A. Carrillo, and I. M. Llorente, “Extending the cloud to the network edge.” *IEEE Computer*, vol. 50, no. 4, pp. 91–95, 2017.

[39] “TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0,” Committee Specification Draft 03, Mar 2014, accessed: Apr 09, 2017. [Online]. Available: https://docs.oasis-open.org/tosca/tosca-nfv/v1.0/csd03/tosca-nfv-v1.0-csd03.pdf

[40] R. Penno, P. Quinn, D. Zhou, and J. Li, “Yang data model for service function chaining,” IETF Secretariat, Internet-Draft draft-penno-sfc-yang-15, June 2016, accessed: Apr 09, 2017. [Online]. Available: http://www.ietf.org/internet-drafts/draft-penno-sfc-yang-15.txt

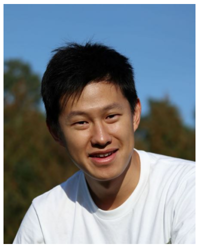
[41] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, “SoftNIC: A software nic to augment hardware,” *Dept. EECS, Univ. California, Berkeley, USA, Tech. Rep. UCB/EECS-2015-155*, 2015.

[42] “Network Functions Virtualisation (NFV); Management and Orchestration ,” White paper, Dec 2014, accessed: Apr 09, 2017. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf

[43] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, “Parabox: Exploiting parallelism for virtual network functions in service chaining,” in *Proc. of ACM SOSR*, 2017, pp. 143–149.

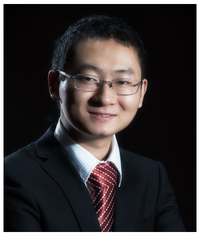
[44] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, “NFP: Enabling network function parallelism in nfv,” in *Proc. of ACM SIGCOMM*, 2017, pp. 43–56.
[45] “Intel dpdk,” accessed: Apr 09, 2017. [Online]. Available: http://dpdk.org/
[46] “hugetlbfs documentation,” accessed: Apr 09, 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
[47] “Address space layout randomization,” accessed: Apr 09, 2018. [Online]. Available: https://www.kernel.org/doc/html/v4.13/security/self-protection.html
[48] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, “NFVnice: Dynamic backpressure and scheduling for nfv service chains,” in *Proc. of ACM SIGCOMM*, 2017, pp. 71–84.
[49] “CFS scheduler,” accessed: Apr 09, 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
[50] “RT scheduler,” accessed: Apr 09, 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt
[51] “pktgen-dpdk,” accessed: Apr 09, 2017. [Online]. Available: http://git.dpdk.org/apps/pktgen-dpdk/
[52] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, “Moongen: A scriptable high-speed packet generator,” in *Proc. of ACM IMC*, 2015, pp. 275–287.
[53] C. Sieber, R. Durner, M. Ehm, W. Kellerer, and P. Sharma, “Towards optimal adaptation of nfv packet processing to modern cpu memory architectures,” in *Proc. of CAN Workshop*, 2017, pp. 7–12.
[54] “Netbricks repository,” accessed: Jan 09, 2018. [Online]. Available: https://github.com/NetSys/NetBricks
[55] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *ACM SIGCOMM Computer Comm. Rev.*, vol. 45, no. 4. ACM, 2015, pp. 123–137.
[56] “Clearwater IMS,” Project documentation, 2018, accessed: Apr 09, 2019. [Online]. Available: https://media.readthedocs.org/pdf/clearwater/latest/clearwater.pdf
[57] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma, “Ownership passing: Efficient distributed memory programming on multi-core systems,” *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 177–186, 2013.
[58] T. Barbette, C. Soldani, and L. Mathy, “Fast userspace packet processing,” in *Proc. of ACM/IEEE ANCS*, 2015, pp. 5–16.
[59] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, “NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors,” in *Proc. of ACM EuroSys*, 2015, pp. 22:1–22:14.
[60] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, “mOS: A reusable networking stack for flow monitoring middleboxes,” in *Proc. of USENIX NSDI*, 2017, pp. 113–129.
[61] R. Kawashima and H. Matsuo, “A generic and efficient local service function chaining framework for user VM-dedicated micro-VNFs,” *IEICE Transactions on Comm.*, vol. E100.B, pp. 2017–2026, 2017.
[62] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: Library operating systems for the cloud,” *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 461–472, 2013.
[63] P. L. Ventre, P. Lungaroni, G. Siracusano, C. Pisa, F. Schmidt, F. Lombardo, and S. Salsano, “On the fly orchestration of unikernels: Tuning and performance evaluation of virtual infrastructure managers,” *IEEE Transactions on Cloud Computing*, Early Access 2018.
[64] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba, “Topology-aware prediction of virtual network function resource requirements,” *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 106–120, 2017.
[65] J. Duan, C. Wu, F. Le, A. X. Liu, and Y. Peng, “Dynamic scaling of virtualized, distributed service chains: A case study of IMS,” *IEEE Journal on Sel. Areas in Comm.*, vol. 35, no. 11, pp. 2501–2511, 2017.
[66] M. T. Raza, S. Lu, M. Gerla, and X. Li, “Refactoring network functions modules to reduce latencies and improve fault tolerance in NFV,” *IEEE Journal on Sel. Areas in Comm.*, vol. 36, no. 10, pp. 2275–2287, 2018.

Shihabur Rahman Chowdhury (S’13) is a PhD candidate at the David R. Cheriton School of Computer Science, University of Waterloo. He received his B.Sc. degree in computer science and engineering from BUET in 2009. His research interests include virtualization and softwarization of computer networks. He is co-recipient of the Best Paper Award at the IEEE/ACM/IFIP CNSM 2019, IEEE NetSoft 2019, and IEEE/ACM/IFIP CNSM 2017 conferences.



Anthony is currently a software engineer with Huawei Technologies Canada. He received MMath degree in Computer Science from the University of Waterloo, Canada in 2018 and Bachelor’s degree in Computer Science from NCTU, Taiwan in 2015. Anthony is a recipient of Mitacs Globalink Research Award in 2018. His research interest includes network softwarization and cloud computing.

Haibo Bian is currently a software engineer at Bioinformatics Solutions Inc. He received the MMath degree from the David R. Cheriton School of Computer Science, University of Waterloo, Canada and BSE degree from Zhejiang University, China in 2019 and 2016, respectively. His research interests include network function virtualization, cybersecurity, and machine learning.



Tim Bai is currently a software engineer with Desire2Learn Canada. He received the MMath and BMath degrees from the David R. Cheriton School of Computer Science, University of Waterloo, Canada in 2019 and 2017, respectively. His current research interests include machine learning, cybersecurity, and network softwarization.

Raouf Boutaba (F’12) received the M.Sc. and Ph.D. degrees in computer science from the University Pierre & Marie Curie, Paris, in 1990 and 1994, respectively. He is currently a professor of computer science and university research chair at the University of Waterloo and holds an INRIA International Chair at INRIA Nancy. He is the founding editor in chief of the IEEE Transactions on Network and Service Management (2007–2010) and is in the editorial board of many other journals. He is a fellow of the Royal Society of Canada, the Institute of Electrical and Electronics Engineers (IEEE), the Engineering Institute of Canada, and the Canadian Academy of Engineering. His research interests include resource and service management in networks and distributed systems.