μNF: A Disaggregated Packet Processing Architecture

Shihabur Rahman Chowdhury, Anthony, Haibo Bian, Tim Bai, and Raouf Boutaba
David R. Cheriton School of Computer Science, University of Waterloo
{sr2chowdhury | a3anthon | haibo.bian | tim.bai | rboutaba}@uwaterloo.ca

***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 deploying 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 and independently deployable components. Experimental results show that compared to monolithic VNF based SFCs, μNF-based ones achieve the same throughput by using less CPU cycles per packet on average.**

I. Introduction

Network operators ubiquitously deploy hardware *middleboxes* (*e.g.*, NATs, Firewalls, WAN Optimizers *etc*.) to realize different network services [1]. Despite being an integral part of modern enterprise and telecommunication networks, middleboxes are proprietary, have little or 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 [2]. 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 over the years [3], including for: resource provisioning, middlebox outsourcing, management platforms, fault-tolerance, state management, traffic steering through VNFs, and programming models and runtime systems. A common trait observed in these works is the *one-to-one substitution of monolithic hardware middleboxes by monolithic VNF counterparts*. Indeed, this is a logical first step for NFV. However, as demonstrated in [4], monolithic VNFs can lead to wasted infrastructure resources.

A fundamental problem demonstrated with monolithic VNF implementation is that many packet processing tasks, *e.g.*, packet I/O, parsing, classification, TCP session reconstruction *etc*., are repeated across wide range of enterprise NFs [4]. 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 [5]. 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 implementing 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?*

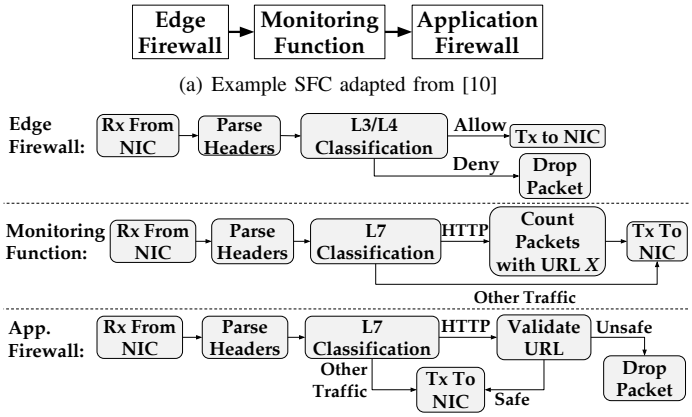
There is a substantial body of research on modular packet processing software [6]–[9]. However, in most cases the end-product is still a monolithic software, which typically executes in a run-to-completion mode, *i.e.*, applies all the functionality of an NF or even an SFC on a batch of packets read 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 packet processors, that we call *Micro Network Functions* (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 [4] that consolidating common packet processing tasks from multiple NFs can lead to better resource utilization. However, CoMb's focus was not to address the implementation challenges for realizing such a system (*e.g.*, software architecture, performance optimizations *etc*.), which is the key contribution of this paper. Specifically, we have the following contributions:

- An architecture for composing VNFs and SFCs from reusable, lightweight, independently deployable and

978-1-5386-9376-6/19/$31.00

loosely-coupled components that we call μNFs (§IV).
- Implementation of architecture components including the μNFs and the communication primitives between μNF (§VI).
- Optimizations for improving the performance of μNFs on multi-socket NUMA machines, and packet processing latency (§V).
- Evaluation of our system through testbed experiments (§VII). A key finding is that an SFC composed from μNFs can achieve the same throughput using less CPU cycles per packet on average compared to that composed from monolithic VNFs.

II. Motivation



(a) Example SFC adapted from [10]

(b) Functional decomposition of NFs from Fig. 1(a)

Fig. 1. Common packet processing tasks across NFs

Our motivation for developing a disaggregated packet processing architecture stems from the observation that many packet processing tasks, *e.g.*, I/O, packet classification, payload inspection *etc*. are repeated when VNFs are chained in an SFC. We demonstrate this using an SFC in Fig. 1(a), similar to the ones typically found in enterprise Data Centers (DCs) [10]. This SFC consists of the following VNFs:

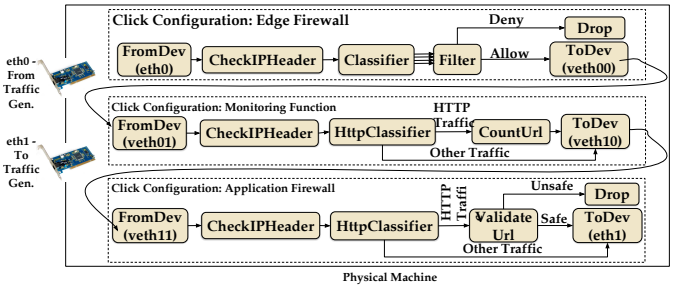
- ***Edge Firewall*:** Allows or denies packets based on layer 2-4 header signature.
- ***Monitoring Function*:** Consists of different counters such as a packet size distribution counter, a counter for packets containing certain URLs *etc*.
- ***Application Firewall*:** Filters packets based on Layer 7 information, *e.g.*, block HTTP requests with embedded SQL injection attacks (similar to [11]).

We can decompose these VNFs into smaller packet processing tasks as shown in Fig. 1(b). Clearly, tasks such as packet I/O, parsing, classifying HTTP packets *etc*. 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.

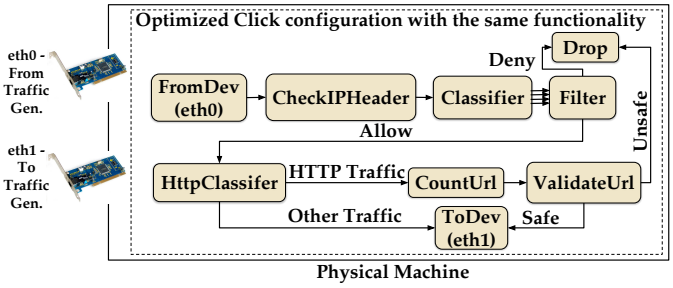
We also perform an experimental study to demonstrate possible performance implications of repeating common packet processing tasks in the SFC from Fig. 1(a) by comparing between the following two deployment configurations: (i) Click [7] based monolithic VNFs chained using virtual Ethernet (veth) pairs (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 web-service (≈15K hits/month)) and measure the average CPU cycles/packet required by each type of Click element. We instrumented all the elements to measure the CPU cycles spent in processing each packet. We also implemented our own Click elements (`HttpClassifier`, `CountUrl`, and `ValidateUrl`) as needed. 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 [4] by demonstrating the impact on an SFC rather than considering single middlebox applications.

TABLE I
Results from Motivational Experiment

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



(a) Monolithic VNFs chained with veth pairs



(b) One single optimized click configuration

Fig. 2. Motivational Experiment Scenarios

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.

This result further motivates re-architecting VNFs by exploiting the commonality in packet processing in a way to achieve better resource usage. To this end, we argue in favor of adopting a microservice-like architecture [12] 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 §VII-C.

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 [13]:

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/processes, each performing a specific packet processing task.

The state-of-the-art modular VNF designs (*e.g.*, ClickOS [6] and NetBricks [8]) 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/process. When more processing capacity is required, the whole VNF (or SFC) instance is scaled out and traffic is split between the instances (*e.g.*, using NIC features such as Receiver Side Scaling). 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 an element processes a packet from the underlying communication pattern between them (transparency).

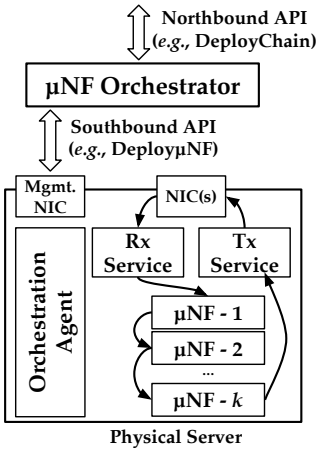


Fig. 3. System Components

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 operator central offices converted into edge data centers [14]. When SFCs are deployed inside these edge data centers their constituent 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 [8]. 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/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 YANG [15].

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. It comprises the following components: a *μNF orchestrator*, per physical server *orchestration agent*, *μNFs*, and Rx/Tx services for reading packets from/to the NICs. 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

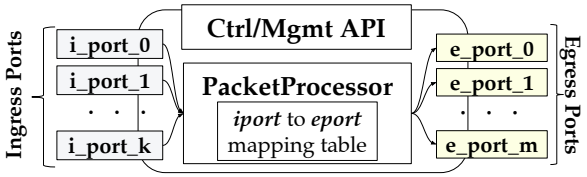


Fig. 4. μNF Architecture

μNF placement across physical servers to realize SFCs, make μNF migration decisions, *etc*.

The orchestration agent acts as the local orchestration endpoint for a given machine. A southbound API between the global orchestrator and orchestration agents facilitate 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 in §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. The architecture of a μNF is shown in Fig. 4. It contains a number of *IngressPort*s, a number of *EgressPort* and a *PacketProcessor*. 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, an 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. One of our design goals is to keep the packet processing logic of μNFs oblivious to μNF to μNF communication pattern. The port abstraction simplifies μNFs' design and implementation, and keep them loosely coupled with each other. For instance, a specific implementation of EgressPort can perform load balancing by distributing packets to multiple next-stage μNFs in a round-robin fashion. From a μNF's point-of-view, such load balanced distribution of packets to the next stage μNFs remains transparent. In §VI we describe the implementation of different ports in detail.

4) Rx Service: Rx service is the interface between host NIC(s) and the μNFs. Rx service keeps the hardware specific configurations (*e.g.*, number of NICs, number of Rx queues *etc*.) 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 [16] 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:

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

***VNF templates*:** A VNF in our system is represented as a packet processing graph composed of the μNFs. 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).

***SFC*:** An SFC request is a directed graph, where the nodes are the constituent 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 μNFs.

2) Sequence of Operations for SFC Deployment: The 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.

After the 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. The orchestrator provides the agents with the configuration parameters of each μNF in the subgraph. 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.

V. Optimizations

A. Pipelined Cache Pre-fetching

One issue that might arise from our design of μNF is when using multiple processors in 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 §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. 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 executions 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 §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

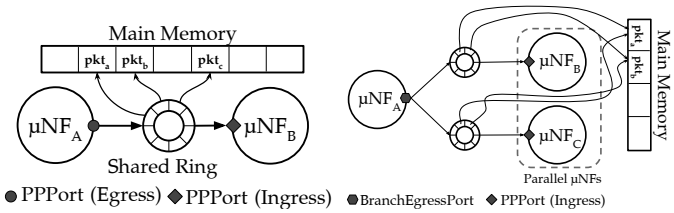
We have implemented a prototype of the proposed system using C++ (agent and μNFs) and Python (orchestrator). Our current focus is more on developing the μNFs and their communication primitives. Therefore, our current orchestrator is limited in functionality and acts more as a convenience mechanism for testing. We use Intel DPDK (http://dpdk.org/) for packet I/O and `hugetlbfs` [17] for sharing memory between μNFs. In the remainder of this section we describe the implementation details of each 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, 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 §VI-D).

B. μNF

μNFs are implemented to run as stand-alone 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.



(a) Point-to-Point Port (b) Branched Egress Port

Fig. 5. Port Implementations

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: *(srcIP, dstIP, 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.

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.

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. 5(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 ring library.

3) BranchEgressPort: This port connects a μNF to multiple μNFs that are processing packets in parallel. For instance, in Fig. 5(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 meta-data 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 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.

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 2.1Ghz CPU (hyper-threading disabled), 32GB memory (distributed evenly between two sockets), and a DPDK compatible Intel 10G Ethernet adapter.

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 kernel feature to ensure a consistent hugepage mapping across the μNFs. We also allocated a total of 4GB hugepages (evenly divided

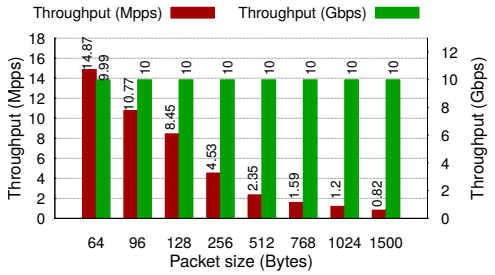


Fig. 6. Baseline Performance

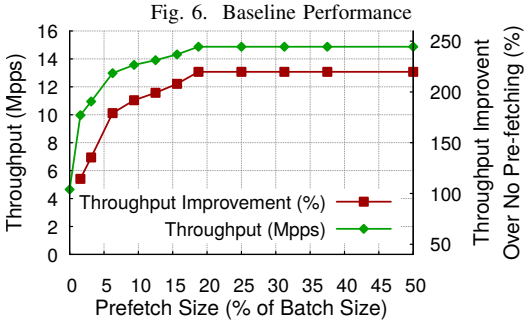


Fig. 7. Impact of Pipelined Cache Pre-fetching

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*.

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

MacSwapper: swaps the source and destination MAC address of each packet.

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 malformed URLs.

CountUrl: Counts the number of packets in a batch that contains a certain URL in its payload.

4) Traffic Generation: We used *pktgen-dpdk* (http://git.dpdk.org/apps/pktgen-dpdk/), and *Moongen* [18] 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 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

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

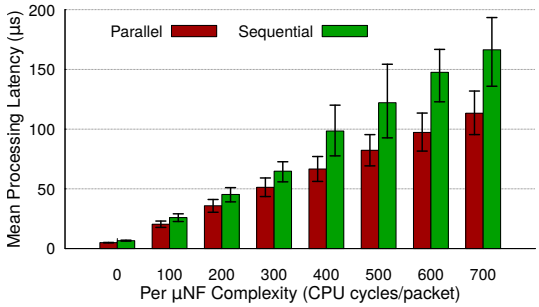


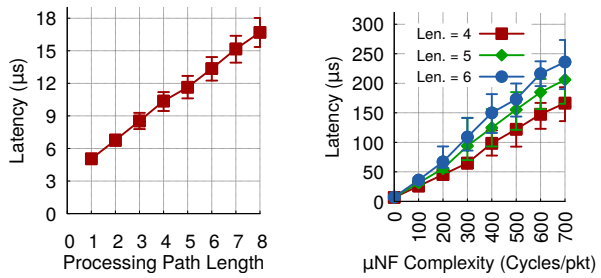
Fig. 8. Impact of Parallelism in Processing Graph

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 10G 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 [19]. In this experiment, we evaluate the impact of cache-prefetching optimization from §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 improves by more than ≈3× and goes back to the line rate for smallest packet size.

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 §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



(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

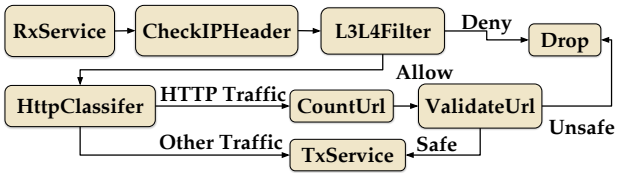


Fig. 10. μNF Realization of the SFC from Fig. 2(a)

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 cycles/packet processing cost).

4) Impact of μNF Processing Graph Diameter: We create μNF chains of different lengths 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 §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 §VII-B3) and measure latency for different lengths of μNF packet processing path (varied from 4 to 6). 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.

C. Performance of μNF-based SFC

We have developed a set of μNFs (described in §VII-A3) for realizing realistic VNFs and SFCs. We use these μNFs to deploy the SFC used for the motivational experiment in §II, *i.e.*, Edge Firewall → Monitor → Application Firewall. The resulting μNF processing graph is shown in Fig. 10. We implemented each individual μNF as close as possible to their Click counterpart. We played the same traffic trace used in §II. Results in Table II show the relative savings in average CPU

TABLE II
CPU CYCLES SAVED PER-PACKET ON AVERAGE

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

cycles per packet when using μNF processing graph over monolithic VNFs (*i.e.*, configuration-(i) from §II). To be fair, we did not compare packet I/O from NIC since it is fundamentally different in μNF and in 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. 10 by generating packets with the packet size set to 200 bytes (average packet size reported in data center networks [20]). The deployment was able to reach a throughput of 2.08Mpps or 3.67Gbps.

VIII. RELATED WORKS

Modular Packet Processing The development of modular packet processing software has a long history that dates back to the late 90s. Click [7], 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, FPGA acceleration *etc.*) to improve packet processing performance and add flexibility to VNF composition [6], [8], [9], [21]–[23]. However, these proposals are centered around the assumption that a middlebox is a monolithic software. More recently, Slick [24] and OpenBox [5] proposed different approaches to achieve the 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 focuses more on the protocol design between VNF control and data planes and not on the design of a modular data plane. A chaining mechanism for lightweight VNFs has been proposed in [25], 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 [26].

Industry Efforts in Microservice-based VNFs There has been some movement in the industry for re-designing large VNFs using microservice architecture. As part of the CORD project [14], 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 [27] re-architected using microservices design principle. However, the independently deployable components themselves are rather complex and can be further decomposed into more manageable sizes.

Middlebox Functionality Consolidation CoMb [4] 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 advantages of consolidating multiple NFs on commodity hardware as opposed to using purpose-built hardware middleboxes. In contrast, E2 [28] proposed to consolidate management tasks such as resource allocation, fault-tolerance, monitoring, and auto-scaling into a single framework. More recently, Microboxes proposed to consolidate TCP protocol processing functions such as bytestream reconstruction and endpoint termination of multiple middleboxes [29]. Consolidation has the advantage of reducing redundant development efforts in implementing and optimizing common tasks. In this paper, we focus on the software architecture and performance optimizations for realizing a disaggregated packet processing platform to consolidate packet processing tasks and thus reduce the development efforts.

IX. Conclusion

We proposed μNF, a system for building VNFs and SFCs from independently deployable, 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 a monolithic implementations.

Acknowledgment

This work was supported in part by the NSERC CREATE for Network Softwarization Program.

References

[1] 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," in *Proc. of ACM SIGCOMM'12*, pp. 13–24.

[2] "Network Functions Virtualisation – Introductory White Paper," White paper, Oct 2012, accessed: Dec 02, 2018. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[3] 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.

[4] 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. 24–24.

[5] 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.

[6] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda *et al.*, "Clickos and the art of network function virtualization," in *Proc. of USENIX NSDI*, 2014, pp. 459–473.

[7] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *Proc. of ACM SOSP'99*, pp. 217–231.

[8] 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.

[9] M. Gallo and R. Laufer, "Clicknf: a modular stack for custom network functions," in *Proc. of USENIX ATC*, 2018.

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

[11] "Barracuda web application firewall," accessed: Dec 02, 2018. [Online]. Available: https://www.barracuda.com/products/webapplicationfirewall

[12] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi *et al.*, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

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

[14] L. Peterson, A. Al-Shabibi, T. Anshutz, S. Baker, A. Bavier *et al.*, "Central office re-architected as a data center," *IEEE Communications Magazine*, vol. 54, no. 10, pp. 96–101, 2016.

[15] R. Penno, P. Quinn, D. Zhou, and J. Li, "Yang data model for service function chaining," Working Draft, IETF Secretariat, Internet-Draft draft-penno-sfc-yang-15, June 2016.

[16] 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.

[17] "hugetlbfs documentation," accessed: Dec 02, 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[18] 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.

[19] 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 ACM CAN*, 2017, pp. 7–12.

[20] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. of ACM SIGCOMM*, 2015, pp. 123–137.

[21] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proc. of ACM/IEEE ANCS*, 2015, pp. 5–16.

[22] 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, p. 22.

[23] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo *et al.*, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proc. of ACM SIGCOMM*, 2016, pp. 1–14.

[24] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proc. of ACM SOSR*, 2015.

[25] R. Kawashima and H. Matsuo, "A generic and efficient local service function chaining framework for user vm-dedicated micro-vnfs," *IEICE Transactions on Communications*, vol. E100.B, pp. 2017–2026, 2017.

[26] 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 (To appear)*, 2019.

[27] "Clearwater ims," accessed: Dec 02, 2018. [Online]. Available: http://www.projectclearwater.org/technical/clearwater-architecture/

[28] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda *et al.*, "E2: a framework for nfv applications," in *Proc. of ACM SOSP*, 2015, pp. 121–136.

[29] 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.