VNF Chain Allocation and Management at Data Center Scale

Nodir Kodirov
University of British Columbia
knodir@cs.ubc.ca

Sam Bayless
University of British Columbia
sbayless@cs.ubc.ca

Fabian Ruffy
University of British Columbia
fruffy@cs.ubc.ca

Ivan Beschastnikh
University of British Columbia
bestchai@cs.ubc.ca

Holger H. Hoos
Universiteit Leiden,
University of British Columbia
hh@liacs.nl

Alan J. Hu
University of British Columbia
ajh@cs.ubc.ca

ABSTRACT

Recent advances in network function virtualization have prompted the research community to consider data-center-scale deployments. However, existing tools, such as E2 and SOL, limit VNF chain allocation to rack-scale and provide limited support for management of allocated chains.

We define a narrow API to let data center tenants and operators allocate and manage arbitrary VNF chain topologies, and we introduce NetPack, a new stochastic placement algorithm, to implement this API at data-center-scale. We prototyped the resulting system, dubbed Daisy, using the Sonata platform.

In data-center-scale simulations on realistic scenarios and topologies that are orders of magnitude larger than prior work, we achieve in all cases an allocation density within 96% of a recently introduced, theoretically complete, constraint-solver-based placement engine, while being 82× faster on average. In detailed emulation with real packet traces, we find that Daisy performs each of our six API calls with at most one second of throughput drop.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; **Cloud computing**; *In-network processing*; *Network management*;

KEYWORDS

Network Function as a Service, VNF chain allocation algorithms, Management API

ANCS '18, July 23–24, 2018, Ithaca, NY, USA

ACM ISBN 978-1-4503-5902-3/18/07.
https://doi.org/10.1145/3230718.3230724

ACM Reference Format:
Nodir Kodirov, Sam Bayless, Fabian Ruffy, Ivan Beschastnikh, Holger H. Hoos, and Alan J. Hu. 2018. VNF Chain Allocation and Management at Data Center Scale. In *ANCS '18: Symposium on Architectures for Networking and Communications Systems, July 23–24, 2018, Ithaca, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3230718.3230724

1 INTRODUCTION

Network processing is increasingly being outsourced to third-party hardware (e.g., [50]). Outsourcing reduces complexity and operational cost [21, 41] in much the same way that public clouds do so for compute and storage.

To outsource network processing, a tenant requests a network function (NF) topology encapsulated in a chain[1]. Fig. 1 shows an example placement of the 4-node NF chain in a data center (DC). In this example, the NAT is placed on a top-of-rack switch (consuming TCAM), the Firewall is placed on a server, and the IDS and VPN are placed on another server. Traffic enters and exits through the gateway switch, and traverses each of the NFs in a chain.

The DC operator, therefore, takes on the difficult task of allocating and managing large numbers of such chains. This can be broken down into three challenges. First, the mapping of chains onto physical DC resources (CPU, memory, TCAM, link bandwidth, etc.) must satisfy the tenants' SLAs. Ensuring sufficient throughput may require replicating some NF elements in a chain across dozens of servers and/or switches. Furthermore, NF placement must guarantee sufficient bandwidth between chain elements, across the entire network, and may require NF elements to communicate using multiple paths. Second, the operator wants to maximize DC utilization to serve as many tenants as possible using the given, limited resources. Third, requirements change over time, so

[1]We use the term *chain* to be consistent with the literature, although we support arbitrarily connected directed graphs of NFs. The ETSI standardization community refers to these as NF *forwarding graphs* [10].

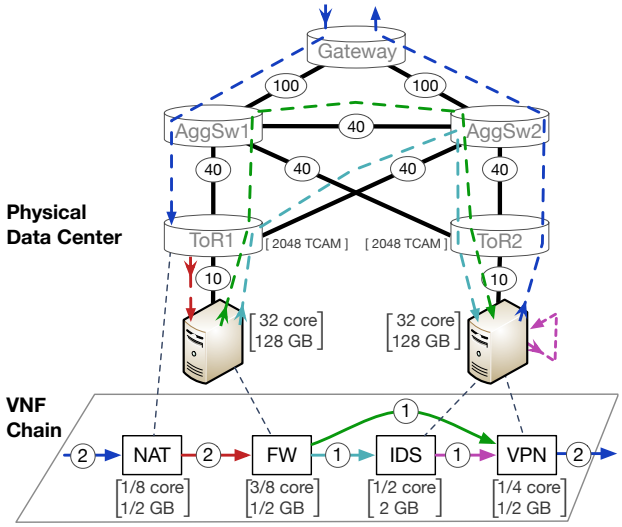

Figure 1: Example of a 4-node VNF chain allocation on a physical DC. Placement of each element must satisfy physical resource constraints and bandwidth constraints between chain elements.

API	Description
$cid \leftarrow$ allocate-chain(C, bw)	Allocate the VNF chain topology C with aggregate throughput bw; return chain identifier cid.
add-node(f, cid)	Add NF f to allocated chain cid.
add-link-bandwidth(a, b, bw, cid)	Add bw bandwidth between NFs a, b in chain cid.
remove-e2e-bandwidth(cid, bw)	Decrease end-to-end throughput in chain cid by bw bandwidth.
remove-node(f, cid)	Remove NF f from allocated chain cid.
remove-link-bandwidth(a, b, bw, cid)	Decrease the bandwidth between NFs a, b by bw in chain cid.

Table 1: Proposed (abstract) chain management API.

the operator needs chain update mechanisms, e.g., to scale up bandwidth, or take down a server for maintenance.

Prior work addresses the challenges of small-scale allocation [2, 5, 14, 25, 30, 31, 39, 47]. In this work, we tackle the problem of NF placement at DC scale, with the goal of allocating chains consisting of 5–10 NFs to physical DCs with 1000+ servers quickly and in a way that permits optimal or near-optimal utilization of the given resources.

We define an API of six operations that jointly permit not only chain allocation, but also efficient in-place chain modifications, such as NF element *upgrades*, chain capacity *scale-out*, and chain *expansion* with new NF nodes. We demonstrate how those operations can be realized with chain allocation algorithms that support end-to-end, multi-path bandwidth guarantees across the entire network infrastructure, from servers to top-of-rack and gateway switches.

Initially, we consider a simple stochastic placement algorithm, Random, introduced as a baseline. Next, we introduce NetPack, a new stochastic algorithm, which performs well in practice at DC scales and greatly improves network throughput. Finally, we compare to VNFSolver, which is based on an algorithm in the constraint-solving literature [4]. VNFSolver is *complete* (guaranteed to find an allocation if one exists), but is orders of magnitude slower than NetPack.

We prototyped a system, called Daisy, using the Sonata platform [32] to empirically evaluate the performance of these algorithms. Our prototype uses each of the above placement algorithms to allocate and manage VNF chains. Using Daisy we tested the proposed six APIs on dozens of emulated virtual hosts and realistic VNF chains with real enterprise traffic. Furthermore, we simulated the algorithms at DC scale (with as many as 1200 nodes, across three families of realistic topologies) and evaluated (1) their DC utilization, and (2) the performance of the chain allocation and chain operations.

To summarize, we make three contributions: (1) we define an API the data center tenants use to allocate and manage VNF chains, (2) we develop a scheduling algorithm, NetPack, to allocate and manage VNF chains at data center scale, (3) we implement a prototype, Daisy, that integrates NetPack and supports the proposed six API in the Sonata platform [33]. Our simulation results show that at DC scales, NetPack is able to achieve at least 96% of the throughput of VNFSolver, while requiring only a small fraction of the compute time as VNFSolver (in some cases, seconds rather than hours). In detailed emulation with real network packet traces, we find that Daisy is able to perform each of our six API calls while experiencing at most one second of throughput drop.

2 CHAIN MANAGEMENT OPERATIONS

Table 1 presents our narrow API of chain management operations. We demonstrate their utility via three use cases:

Use case 1: Chain scale-out/in. Chains must be dynamic to respond to changes in traffic. For example, when the ratio of unsafe traffic grows, an operator (or a monitoring tool) may need to update the allocated chain to handle the increase in load. Fig. 2a illustrates this scale-out when an extra bandwidth unit of suspicious traffic must be handled by an existing chain. In this case, Daisy[2] uses the *add-link-bandwidth* operation five times to increase the bandwidth along the IDS path of the chain. Alternatively, the *remove-link-bandwidth* API can be used to scale-in VNF chains.

[2]Daisy refers to our prototype that implements the API, and also more generally to any system that aims to support the API.

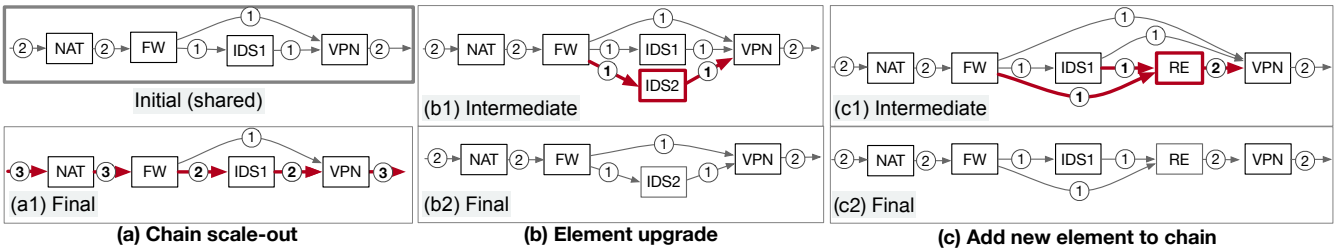

Figure 2: Illustrations of initial/intermediate/final VNF chains in 3 use cases. Changes are marked in bolded red.

Use case 2: Chain upgrade. When a new software version for an NF is released, an operator needs to upgrade deployed chains without disrupting existing flows. We model this workflow as an *in-place upgrade*; Fig. 2b illustrates how an in-place upgrade of an IDS element is expressed using the API. To go from the chain in Fig. 2.Initial to the one in Fig. 2.b1, Daisy first uses *add-node* to create a new IDS instance (IDS2), and then uses *add-link-bandwidth* to connect IDS2 to the destination and source elements of IDS1 with equal amount of bandwidth. IDS1 will keep running until all active flows terminate or migrate to IDS2. Once no traffic is passing through IDS1, Daisy will transition from Fig. 2.b1 to Fig. 2.b2 (effectively disconnecting IDS1 from the chain) using *remove-link-bandwidth* followed by *remove-node*.

Use case 3: Traffic engineering. Chains should be extensible and permit traffic optimization and monitoring. Fig. 2c shows a case when an tenant observes redundant traffic and wants to add a Redundancy Eliminator (RE) to prevent such traffic from passing to the corporate network via the VPN. This change should be transparent to the existing flows. The operator would use the *add-node* and *add-link-bandwidth* API to add the RE (Fig. 2.c1). Then, new flows are directed to pass through RE, and once no further traffic is flowing through the initial links to the VPN, two calls to *remove-link-bandwidth* would remove these links (Fig. 2.c2).

The three APIs at the top of Table 1 require new physical resources and may fail. By default, Daisy handles these failures transparently, by deallocating the existing chain and allocating a new, updated, chain. Since extra resources exist elsewhere in the DC, in this mode, Daisy hides the chain movement. Operators may disable this behavior to manually handle failures, e.g., by gracefully terminating existing chains or provisioning extra hardware to avoid chain relocation.

The three use cases hint at the generality of the API in Table 1. More broadly, any VNF topology can be transformed into any other VNF topology using a finite sequence of these API operations. This follows because the API calls can independently change a VNF chain's links/nodes/bandwidths.

Abstract and concrete chains. Our discussion so far assumed that an operator defines, allocates, and then manages a single chain using the API in Table 1. In practice, a tenant may request a chain that requires more physical server/link capacity than is available on any single server or switch in the DC. To enable scaling VNF chains past the physical resource constraints, we introduce the notions of *abstract* and *concrete* chains. The chain that the operator defines and operates on, and a tenant behaviorally observes, is an **abstract chain**. This abstract chain captures the SLA constraints, the NF elements, and their sequence. However, an abstract chain does not necessarily map as a whole onto the physical resources. In particular, Daisy may realize, or implement, an abstract chain on physical resources as several **concrete chains**. In Section 6 we demonstrate how Daisy implements an abstract-to-concrete chain mapping mechanism.

For example, in use case 1 above, the original abstract chain in Fig. 2.Initial may be instantiated as a single concrete chain. The additional unit of bandwidth added to the abstract chain by the operator may require instantiating a second concrete chain. This may happen because, for example, the existing physical resources hosting the concrete chain cannot cope with the new demand, or because one or more of the NF elements cannot handle the new load. Daisy automatically determines the set of concrete chains that are necessary to support an abstract chain and performs the allocation of concrete chains, rather than abstract chains, onto the physical resources.

3 CHAIN ALLOCATION ALGORITHMS

The core algorithmic problem in VNF chain allocation is to place the NFs of a concrete chain onto servers and switches, and then to allocate sufficient bandwidth between them. Here, we formalize the problem and present three algorithms for solving it. Then, we show how we implemented our chain management API using the allocation algorithms.

In both algorithms described in the next two subsections, allocate-concrete() takes a physical network PN and a concrete chain CN as input. The physical network PN consists of a set of servers and switches S, and a graph (S, L), with capacities $c(u, v)$ for each link in L. The VNF chain CN consists of a set of NFs F and a set of pairwise bandwidth requirements $R \subseteq F \times F \times \mathbb{Z}^+$. For each server/switch $s \in S$, we

are also given a vector of integers $P[s]$ representing the physical resources available for consumption by NFs placed on s; and similarly, for each network function $f \in F$, we are given vector $P[f]$ representing the required server resources for that function. For example, $P[f][0]$ might represent the number of cores required; $P[f][1]$, the amount of RAM; $P[f][2]$, the number of TCAM entries, etc. In order to place NF f on server/switch s, the following condition should be met: $(P[f] \leq P[s])$, i.e., s should have sufficient resources to host f. The objective is to find an assignment $A : F \mapsto S$ of NFs $f \in F$ to servers/switches $s \in S$, and, for each bandwidth requirement $(u, v, bw) \in R$, an assignment of non-negative bandwidth $B_{u,v}(l)$ to links $l \in L$, such that the following sets of constraints are satisfied:

Local Resource Allocation Constraints: (1) Ensure that each NF is assigned to exactly one server/switch (of course, multiple NFs may be assigned to a server/switch), and (2) ensure that each server/switch has sufficient resources $P[s]$ available to serve the sum total of requirements of the NFs allocated to it:

$$\forall s \in S, \forall i \in 1..|P[s]| : \left(\sum_{\{f \in F | A(f) = s\}} P[f][i] \right) \leq P[s][i]$$

Global Bandwidth Allocation Constraints: Ensure that sufficient bandwidth is available in the physical network to satisfy all bandwidth requirements simultaneously. Formally, we require that $\forall (u, v, bw) \in R$, the assignments $B_{u,v}(l)$ form a valid $A(u)$–$A(v)$ network flow greater or equal to bw, and that we respect the capacities of each link l in the physical network: $\forall l \in L : \sum_{(u,v,b) \in R} B_{u,v}(l) \leq c(l)$. We model bandwidths using integer values and assume that communication bandwidth between NFs allocated to the same server/switch is unlimited.

3.1 Random Allocation

Our first allocation algorithm, Random (Algorithm 1), is a simple, stochastic placement algorithm, which serves as a baseline for our empirical experiments. Random performs concrete chain allocation in two stages. First, for each NF $f \in F$, it assigns f to a random server $s \in S$ with sufficient resources. Then, it tries to find sufficient bandwidth to satisfy the global bandwidth constraints of the VNF. Both the server placement step and the bandwidth allocation step are greedy processes, which can fail even in cases where a placement is feasible. For this reason, if either allocation step fails, Random restarts and tries again, up to *max_attempts* times (set to 100 in practice). In the Algorithm 1, assume the variable *InitialAllocation* is the empty set. We will use it in Section 3.4 to extend the behaviour of allocate-concrete().

For each function in $f \in F$, Random visits each server at most *max_attempts* times. Each time a server is visited,

Algorithm 1 Random allocation algorithm.

```
procedure ALLOCATE-CONCRETE(PN : (S, L), CN : (F, R), P)
    Physical network PN has servers/switches S and links L. Chain CN
    has NFs F with bandwidth requirements R. P contains resource vectors
    for each NF, server, or switch.
    repeat up to max_attempts times:
        failed ← False, P' ← P, A ← InitialAllocation
        for all f ∈ F (in random order) do
            S' ← {s ∈ S : P[s] ≥ P[f]}
            if S' = ∅ then failed ← True, break
            else s ← RandomChoice(S')
                P[s] ← P[s] − P[f], A[f] ← s
        if failed or not AllocatePaths(PN, CN, A) then
            P ← P'                ▷ Undo resources used by failed allocation.
        else return True
    return False
procedure AllocatePaths(PN, CN : (F, R), A : F ↦ S)
    A is an allocation of network functions to servers.
    PN' ← PN
    for all u, v, bw ∈ R do
        while bw > 0 do
            path ← ShortestPath(PN, A[u], A[v])
            if path = ∅ then
                PN ← PN'          ▷ Restore PN to original value
                return False
            else
                bw' ← min(bw, {PN[a, b]|(a, b) ∈ path})
                bw ← bw − bw'
                for (a, b) ∈ path do
                    PN[a, b] ← PN[a, b] − bw'
                    if PN[a, b] = 0 then RemoveEdge(PN, a, b)
    return True
```

AllocatePaths() may be called at most once. AllocatePaths() repeatedly computes shortest paths in the unweighted network (using depth first search), quiting when either no more bandwidth can be allocated, or bw units of bandwidth have been allocated. Assuming integer bandwidth values, each iteration either decrements bw or exits. As a result, AllocatePaths() takes $O(bw \cdot |S|)$ steps. Random as a whole then requires $O(BW \cdot |S|^2 \cdot |F|)$ time in the worst case, with BW sum of the bandwidths requirements of F. However, experimentally we have found the runtime to be approximately linear on realistic instances (see Fig. 4 and Fig. 6).

3.2 Stochastic Bin-Packing (NetPack)

Random makes no attempt to place adjacent NFs together on the same server, resulting in excessive bandwidth usage. NetPack (Algorithm 2) improves on this in three ways. The first improvement is to allocate the NFs of the chain in *topological*, rather than random, order. Finding this ordering requires linear time (using Kahn's algorithm [20]), and needs to be done once for each chain, as a preprocessing step. For example, one topological ordering of the 10-node chain in Fig.3e is (*VPN, NAT, LB, FW3, FW1, FW2, WC, DPI,*

Algorithm 2 NetPack allocation algorithm.

```
procedure ALLOCATE-CONCRETE(PN : (S, L), CN : (F, R), P)
    Arguments are as in Alg. 1. Additionally, Racks and Clusters each
    contain a list of subsets of S.
    for all ServerSets ∈ {{S}, Racks, Clusters} do
        if ALLOCATE-LOCAL(PN, CN, P, ServerSets) then
            return TRUE
    return FALSE
procedure ALLOCATE-LOCAL(PN, CN, P, ServerSets)
    ServerSets contains one or more subsets of S.
    repeat up to max_attempts times:
        for all Servers ∈ ServerSets (in random order) do
            failed ← FALSE, P' ← P, A ← InitialAllocation
            s_l ← nil
            for all f ∈ F (in topological order) do
                if s_l = nil or P[s_l] < P[f] then
                    S' ← {s ∈ Servers : P[s] ≥ P[f]}
                    if S' = ∅ then failed ← TRUE, break
                    else s_l ← RANDOMCHOICE(S')
                P[s_l] ← P[s_l] − P[f], A[f] ← s
            if failed or !ALLOCATEPATHS(PN, CN, A) then
                P ← P'                          ▷ Restore P to original value
            else return TRUE
    return FALSE
```

IPS, GW). Allocating NFs in a topological sorted order avoids unnecessary bandwidth consumption. For example, in Fig. 1, a random allocation order might swap the placement of the Firewall and VPN. This would result in the chain consuming 5 Gbps of extra bandwidth from the aggregation layer switches, and 4 Gbps extra from ToR switches as compared to the (current) topologically ordered placement.

The second optimization in NetPack is *network-locality*. NetPack gradually increases the network scope for allocation. First it tries to place all NFs of the chain on the same server. If that is impossible, it explores servers on the same rack, then servers within the same cluster, etc. Network-locality further reduces network consumption of the chain.

The last optimization in NetPack is *server-locality*, which preferentially re-uses the previously selected server when placing consecutive NFs (when possible). This differs from the network-locality optimization described above: if the chain does not fit onto a single server, the network-locality optimization will try to place the chain in the same rack, but will make no effort within that rack to place consecutive NFs on the same server. By applying both optimizations, we attempt to achieve high density packing within each server, and low-latency packet processing for the chain as a whole.

Notice that in Algorithm 2, each server set is processed at most once, and each server appears in each server set at most once (once at the cluster level, once at the rack level, and once at the individual server level). When placing a chain F, for each NF $f \in F$, each server is visited at most $max\_attempts \cdot |ServerSets|$ times (both of which are constant factors). As with Random, each time a server is visited, AllocatePaths() may be called once, requiring $O(bw \cdot |S|)$ time. As the topological sort step requires $O(|F|)$ and is performed only once, the algorithm as a whole requires worst-case $O(BW \cdot |S|^2 \cdot |F|)$ runtime, where BW is the sum of the integer bandwidth requirements of F. However, as with Random, we have found the runtime to be approximately linear in practice (and faster than Random in most cases).

While the optimizations proposed are straightforward improvements to naive random bin-packing, as we will see in Section 6, these optimizations greatly improve the allocation density achieved by NetPack (as compared to Random), in many cases achieving 300% as many allocations as Random. In fact, we will show experimentally that across a wide variety of realistic scenarios, these optimizations are always able to achieve within 96% of the allocations achieved by a theoretically complete (but much more expensive) constraint-based allocation algorithm we describe next.

3.3 Constraint-Based Allocation (VNFSolver)

A natural question, of course, is how much improvement is possible over NetPack? To attempt to answer this question, we use VNFSolver, which allocates concrete chains by directly solving the formal constraints. Although such an approach is not guaranteed to find the globally optimum utilization of a DC (because earlier allocations are done without knowledge of later requests),[3] it is *complete* in the sense that it will always find an allocation for a chain if one exists.

Prior work has used constraint solving for VNF allocation, but scaled only to under 100 servers [14]. We have leveraged recent advances in constraint solving in order to reach DC scale. In particular, VNFSolver builds closely on our recently published placement algorithm called NetSolver [4]. NetSolver is a SAT-based network allocation algorithm, originally designed to perform virtual data center (VDC) allocation, as opposed to VNF allocation. Briefly, NetSolver frames VDC allocation in terms of a constrained multi-commodity flow problem, in which the multi-commodity flow enforces global bandwidth connectivity, while the added constraints enforce that the sources and sinks of the flow problem correspond to legal mapping of VMs to servers.

Although NetSolver was not designed for performing VNF allocation, the local and global allocation constraints defined above are the same. As such, NetSolver can directly perform ALLOCATE-CONCRETE() without major modifications. However, to support the API in Table 1, we needed to modify NetSolver in two significant ways: First, we added support

[3] Indeed, in Section 6, we will describe one case that we are aware of where NetPack got lucky and was able to achieve slightly greater throughput than VNFSolver.

for affinity constraints in order to force some NFs to pre-selected servers, instead of allowing the solver to select them. This change requires adding an extra constraint into the underlying SAT solver, for each NF in question, forcing its assignment to the selected server. Secondly, we modified NetSolver to record the servers that each assigned NF are placed on, as well as the associated bandwidth they utilize (if any), to facilitate deallocation.

In general, the main algorithmic contribution of this work is NetPack. As we demonstrate in our evaluation in Section 6 NetPack scales well to data center settings while being 82× faster than VNFSolver on average.

3.4 Chain Management Primitives

Returning to the six operations in Table 1, the last three operations remove allocated bandwidth or resources from the network. They are implemented with simple book-keeping operations to remove the allocation and release the resources used. The first three operations, however, allocate new bandwidth or resources, and hence require using the placement algorithm as described next. In order to support *allocate-chain*, an implementation must decompose a requested abstract chain into reasonably-sized concrete chains. In our implementation, we find the greatest common divisor, D, among the requested edge bandwidths in the full NF chain, and split the request up into D separate calls to allocate-concrete(). As we show in Section 6, this allocation process proceeds quickly in practice, with NetPack requiring less than 0.05 seconds per individual concrete allocation on even the most challenging DC network that we consider.

The *add-node* primitive inserts a new NF into an already allocated abstract chain, simultaneously placing the new NF on a physical server or switch with sufficient local resources, while also allocating the required bandwidth between that node and its neighbours in the VNF chain. As the abstract chain to be upgraded may be composed of multiple concrete chains, our algorithms proceed iteratively through those concrete chains, adding nodes to each of them individually.

For each concrete chain, the algorithms initially attempt to incrementally upgrade that concrete chain, adding the new node while leaving the rest of the concrete chain in place. Let (F, L) be the concrete NF chain that has already been allocated, while (F', L') is a new concrete NF chain, consisting of a single node to be placed, $F' = \{f'\}$, and one or more links between f' and the nodes of F. Notice that while L' contains links between f' and the nodes of F, F' (the set of nodes to be allocated) contains only the new node. If the allocation algorithm is Random or NetPack, we set the variable *InitialAllocation* to hold the allocation of the original concrete NF (so that the AllocatePaths() knows which servers the nodes of F are located at). If the allocation algorithm is VNFSolver, we use the affinity constraints described in the previous section to force the previously allocated nodes of F to their existing hosts during allocation.

As we will show in Section 6, this process is fast. However, this incremental approach to *add-node* might not always be feasible; even in cases where there is lots of bandwidth available in the DC, it may be that some of the NFs are allocated to a part of the DC that is locally congested, in such a way that no additional bandwidth can be added between those NFs and f'. In the case where the above approach to *add-node* is infeasible, the algorithm deallocates the congested NFs completely (with calls to *remove-link-bandwidth* and *remove-node*), and then makes a separate call to *allocate-chain* to re-allocate F' all in one go, elsewhere in the DC.

The remaining API is *add-link-bandwidth*, which allocates additional bandwidth between two existing NFs in an already allocated VNF chain. The algorithms implement *add-link-bandwidth* exactly as *add-node*, except that there are only new links in (F', L'), and no new node. As with *add-node*, *add-link-bandwidth* can potentially fail, in which case we would re-allocate the chain.

Both *add-node* and *add-link-bandwidth* APIs use full concrete chain re-allocation for locality. Partial chain re-allocation can scatter NFs across servers and racks, increasing chain packet processing latency. We plan to study the trade-off between partial and full re-allocation in our future work.

4 DAISY PROTOTYPE EVALUATION

To verify the feasibility of our proposed API (Table 1), and to evaluate the network impact of selected operations, we implemented the Daisy prototype. Daisy is built on Sonata, an ETSI affiliated NFV management and orchestration stack [32, 33] that uses Mininet [22] to deploy and link NFs encapsulated in Docker containers. Sonata allows us to quickly prototype VNF chains and perform management operations with arbitrary topologies and resource constraints, while steering real traffic to test chain functionality. Each emulated DC contains a number of containers connected to a central Open vSwitch [28]. The DC topology and NF chains are enforced by a Ryu controller [37] that configures VLANs and flow routing rules. Resource constraints are enforced by Linux *cgroups* and by Sonata. However, Sonata is appropriate for modeling only rack-scale DCs, in which the allocation of NF chains is relatively trivial. In Section 6, we will explore the performance of our allocation algorithms beyond the scale that Sonata and our physical hosts can support, including multi-rack, DC scale settings.

We deployed Daisy on an Azure E64s v3 instance with 64 cores and 432 GB of memory [3]. As a sample scenario, we implemented the 4-node chain in Fig. 3c (the same VNF chain we used to motivate our examples in Fig. 1 and Fig. 2).

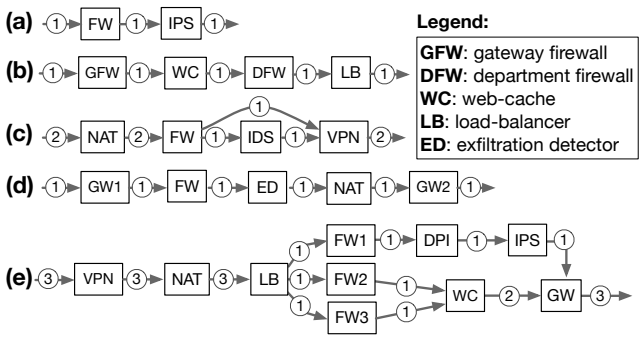

Figure 3: VNF chains used in experiments in Section 6.

Network function type	CPU (core)	Memory (GB)	Switch support
DPI (IDS, Exfiltr. detection)	1/2	2	No
Firewall	3/8	1/2	Yes
Load-balancer	3/8	1	No
VPN, Gateway	1/4	1/2	No[5]
Web-cache, Redund. Elim.	1/4	3/2	No
NAT	1/8	1/2	Yes

Table 2: Resource requirements of different NFs to process 1 Gbps traffic. CPU and memory has linear increase as traffic volume passing through NF grows. Switch support means NF can be placed on the switch by consuming one TCAM space.

All NFs were run as Docker containers with a Ubuntu Trusty image using the 4.11.0 kernel. The NAT and the Firewall use an *iptables* configuration with the firewall being configured to redirect FTP traffic directly to the tenant VPN. All the remaining packets are processed by the IDS, a Snort container that inspects packet payloads and generates alerts for SSH connections across all ports. The last element in the chain is an OpenVPN client that connects to a VPN server, acting as the sink NF. A source container connected to the NAT generates traffic by replaying packet traces from an enterprise network [7].

4.1 Chain allocate

In this scenario, we emulate a cloud provider with a rack of 40 servers to host tenant VNFs. Traffic comes into the network from 10 off-cloud servers. That is, we split the resources of our host machine to emulate a rack-scale topology with 50 servers: 40 are *chain-servers* and 10 serve as *source-sink* servers to generate/receive traffic. The 50U rack is connected to a single ToR switch. In order to compare our chain allocation algorithms' performance in different DC settings, we used a rack with *homogeneous* servers and a rack with *heterogeneous* servers. The homogeneous rack contains identical servers, while the heterogeneous rack contains two generations of servers: 20 *chain-servers* are identical to those in the homogeneous rack, and 20 have 2x more resources. The *source-sink* servers are likewise of two types. The heterogeneous rack has 1/3 more VNF hosting capacity than the homogeneous rack.

To model the limited resources of each server, we use Sonata's modeling techniques. An homogeneous server is represented by 20 compute units (CU), a total of 1000 CU for the DC[4]. Each server has approximately 8 GB of RAM

[4]20 CU corresponds to 1.2 virtual cores per server. Refer to the original paper about Sonata framework [32] for a more detailed description of CUs.

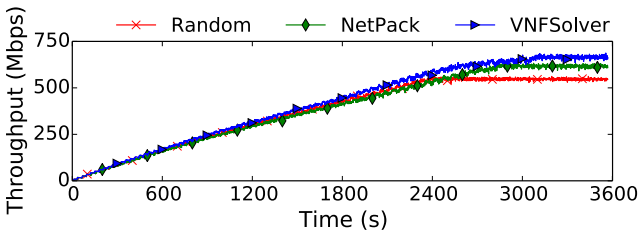

Figure 4: Aggregate throughput of chains allocated by different algorithms in the Daisy prototype.

and can provide space for at most seven VNFs before oversubscription. We deploy NF containers with resources based on Table 2. The NAT consumes 1 CU for a unit of bandwidth, the firewall 3 CUs, the IDS 4 CUs, and the VPN 2 CUs. A complete chain consumes 16 CUs for 2 units of bandwidth passing through the chain (1 unit of bandwidth passing through the IDS, but 2 units from all other NFs). Thus, with a total of 800 CUs available across 40 *chain-servers*, up to 50 chains can be allocated. Since the heterogeneous setting has 1/3 more server capacity, it can host up to 75 chains before running out of compute resources.

Fig. 4 shows the aggregate throughput achieved by the chains allocated in a heterogeneous setting. Here, Random allocated 61 chains, and NetPack made 67 chain allocations, while VNFSolver allocated an optimal number (75) of chains. In this experiment, we iteratively allocate each chain and steer traffic through it, i.e., allocate the chain once the previous one is deployed and network traffic is flowing. As our network traffic we replay real enterprise traffic [7] at 10 Mbps and measure throughput through a chain at the sink interface. Fig. 4 shows increasing aggregate throughput

[5]The switch model we consider does not have VPN or load-balancer support, but other models do [17]. For the experiments in this paper, we do not support the placement of VPN and load-balancer NFs on switches, but our algorithms can place any NF on switches that support them.

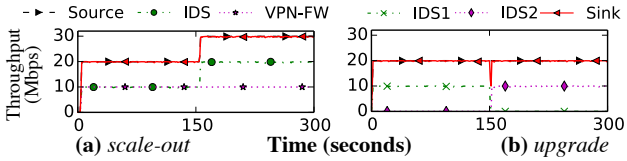

Figure 5: Throughput of inter-NF element links in a single concrete chain in the scale-out and upgrade scenarios in Daisy.

as more chains are allocated and the throughput plateaus once the chains have been allocated. The 75 chains allocated by VNFSolver achieve 687 Mbps of throughput, 67 chains by NetPack achieve 633 Mbps, and 61 chains by Random achieve 561 Mbps. The chain bandwidths are not precisely 10 Mbps due to the irregular nature of the enterprise traffic and the resource limitations of the physical host, which is running over 500 Docker containers to support 75 chains. Due to the *tcpreplay* overhead, our host machine consumed 93% CPU after allocating all chains. We repeated this experiment with TCP traffic generated by *iperf3* and achieved the expected throughput (750 for VNFSolver, 670 for NetPack, 610 for Random) with only 3% host CPU utilization (figure omitted due to lack of space). This experiment illustrates the practical benefit of NetPack and VNFSolver: high throughput/DC utilization by packing more chains onto the same rack. It also illustrates the elasticity offered by the abstract-concrete chain decoupling.

We repeated this experiment with a homogeneous rack. Due to the simple topology, the three algorithms performed similarly: Random allocated 47 chains with 495 Mbps aggregate throughput, NetPack allocated 48 chains achieving 507 Mbps, and VNFSolver got the optimal (50) chains with 526 Mbps aggregate throughput.

4.2 Chain scale-out and chain upgrade

We also evaluated the first two use-cases in Section 2. The chain scale-out use-case exercises the *allocate-chain* and *add-link-bandwidth* API calls, and the chain upgrade use-case utilizes *allocate-chain*, *add-node*, *add-link-bandwidth*, *remove-link-bandwidth* and *remove-node*. Combined, these two experiments make full use of our API. For these experiments, we emulated a rack with one *chain-server*, one *source-sink* server, and one ToR switch to perform chain scale-out and upgrade. In both cases, we reuse the 4-node chain from Fig. 3c, and pass 10 Mbps of real traffic plus 10 Mbps of additional FTP traffic to stress the firewall-to-VPN link (VPN-FW). This FTP traffic rate is expected to remain constant throughout the test. We run the experiments for 300 seconds and call the respective API function after 150s. For scale-out (as in Fig. 2a), we increase the link-bandwidth of all VNF links except VPN-FW; for upgrade (as in Fig. 2b), we switch an IDS running Snort 2.9.6 on Ubuntu Trusty to a new IDS element that uses Snort 2.9.7 on Xenial.

Fig. 5a shows the throughput impact of the scale-out use case on a single concrete chain (the figure omits some of the chain links for readability). In Fig. 5a the VPN-FW link maintains a 10 Mbps throughput rate on both links, while the VPN and sink VNFs receive 20 Mbps until *add-link-bandwidth* is triggered at the 150th second. Except for the VPN-FW link, which is held constant in this use case, the API call increases the bandwidth of all links by 10 Mbps. Fig. 5b highlights the link throughput during the upgrade experiment. We only show the source VNF egress, the to-be-upgraded IDS ingress (IDS1), the upgraded IDS ingress (IDS2), and the sink VNF ingress. The source and sink VNF throughputs remain nearly constant during the upgrade, experiencing a short throughput drop, under 1s, when we trigger *remove-link-bandwidth* and *remove-node* to replace the IDS. The throughput drop occurs because the network path is switched from IDS1 to IDS2 without state-awareness. When the switch is performed, packets of all ongoing flows are dropped and the throughput is restored because of newly established flows through IDS2. Thus, the throughput drop window could be longer than 1s if our realistic packet traces had a large number of long-running (elephant) flows. State-aware path switching, also known as *flow migration*, has been extensively studied in the literature [12, 36, 45] and various mechanisms exist to perform zero-loss flow migration. We leave it to future work to optimize flow migration during upgrades and will build on existing research efforts [12, 36].

5 SIMULATION METHODOLOGY

Our emulation with Sonata does not scale beyond rack-scale, so we use simulation to evaluate the allocation algorithms at DC scale. Here, we present our methodology, and Section 6 presents our results.

Physical topologies. In our simulation experiments, we consider three classes of physical topologies. The first is based on a rack-scale topology used by E2 [30], where a ToR switch has N ports of which K are external and $N-K$ are internal. External ports are northbound interfaces connecting the ToR switch to a higher-level network component, such as an access or gateway switch. Internal ports are southbound interfaces connecting the ToR switch to servers. E2 uses Intel Seacliff Trail switches [9], which have 48 internal ports of 10 Gbps bandwidth each and 4 external 40 Gbps ports. Thus, each ToR has 160 Gbps uplink and 480 Gbps downlink (1:3 oversubsription). We extrapolate from this setting to generate topologies with as many as 32 racks and 1536 servers.

The extrapolated multi-rack setting has a leaf-spine topology, where the leaf consists of 32 ToR switches, each with 160 Gbps aggregate uplink to the single spine switch. The spine switch has no oversubsription and acts as the gateway switch with full-bisection bandwidth. Each ToR switch in the E2 topology also has support for 2048 TCAM rules [9]. We use TCAM to offload some NFs from servers to ToRs.

The second physical topology we consider is a real-world commercial DC topology, used for hosting a private cloud.[6] This private cloud is deployed across four DCs in two geographic availability zones (AZs): us-west and us-middle. These DCs contain between 280 and 1200 servers, arranged into 1, 2, or 4 clusters, with 14 to 60 racks in total. Each server has 32 cores, 128 GB RAM, and 20 Gbps network bandwidth (over two 10 Gbps links). The network in each DC has a leaf-spine topology, where all ToR switches connect to two distinct aggregation switches over 40 Gbps links each (a total of 2 links with 80 Gbps; one on each aggregation switch), and aggregation switches are interconnected with four 40 Gbps links each. For each cluster, there is a gateway switch with a 240 Gbps link connected to each aggregation switch.

The third physical topology is from Facebook's Altoona DC [1]. This topology has a modular design with no oversubscription across its networking fabric. Facebook uses a *pod* as a building block where each pod has 48 ToR switches, and each rack connects together 16 servers with 10 Gbps per server. Thus, a rack has 160 Gbps throughput to anywhere across the DC. A pod has 768 servers and its network fabric consists of ToR, *fabric*, *edge*, and *spine* switches, each with 48 ingress and egress ports with 40 Gbps bandwidth. This network guarantees full-bisection bandwidth across the entire DC, and is well suited for VNF service providers in general, and our algorithms in particular. Because no part of the network is oversubscribed, this design prevents any network segment from being a bottleneck during chain allocation. Further, its modularity allows scalable deployment of our algorithms: each algorithm instance can manage each pod.

Network functions. We consider VNF chains composed of as many as 10 NFs, listed in Table 2. For example, some of the NFs we consider include DPI (Deep Packet Inspection), NAT, Firewall, and VPN. Given that a DC-grade server CPU core typically operates at around 3 GHz, we estimate that each core can sustain a DPI with 2 Gbps traffic (or two 2 DPIs with 1 Gbps, if each DPI belongs to different tenant and each DPI is provisioned for 1 Gbps traffic). We account 2 GB memory for each 1 Gbps of DPI traffic, i.e., 4 GB of RAM per core. We also empirically confirmed such CPU and RAM consumption per Gbps traffic with our Daisy prototype (Section 4). This 4:1 RAM/CPU core ratio roughly matches that of commodity DC servers [8, 15], including the commercial DC servers we consider (128 GB RAM per 32 core server).

The DPI element is one of the most compute- and memory-intensive NFs in common use. Therefore, we model other NF requirements relative to DPI (Table 2). Since IDS and exfiltration detection services can be implemented with the same software as DPI [42], we model these two NFs and DPI as having the same resource requirements. Other NFs, such as NAT or Web Cache, have relatively low compute and memory footprints. Furthermore, some NFs can be placed directly on switches, using TCAM rules. In our experiments, we allow NAT and Firewall to be allocated to TCAM (as both of these functions are supported by this switch model [17]); the remaining NFs we consider in Table 2 cannot be implemented using TCAM. If placed on a switch, NAT and Firewall NF consume one TCAM space, and if placed on the server, they consume core and memory as shown in Table 2. See Section 7 for further discussion on the variability of NF resource footprints and the metrics a VNF scheduler may take into account to perform VNF chain placement.

VNF Chains. We consider five different VNF chains from the literature, depicted in Fig. 3. These chains cover a variety of sizes, functionalities, and use cases, ranging from 2 to 10 NFs. Chains (a) and (b) are from OpenBox [5], (c) and (e) are from E2 [30], and (d) is from Embark [21].

6 SIMULATION EVALUATION RESULTS

We ran our simulation experiments on a server with two 2.66GHz (12MB L3 cache) Intel Xeon x5650 CPUs, with 12 cores per CPU and 96GB of RAM, running Ubuntu 12.04. All processes were limited to 16GB of RAM and 30,000s of CPU time, however neither of these limits were ever reached in practice (all processes ran to completion successfully).

6.1 Allocations from single tenants

In Fig. 6, we evaluate the scalability and DC utilization of our algorithms on DCs of increasing sizes. We did this by using each algorithm to allocate as many concrete chains as possible. These two graphs show the total allocated bandwidth achieved by Random, NetPack, and VNFSolver, as well as the time required to find these allocations, for different physical topologies. These experiments were run for two VNF chains: (1) 4-node chain from Fig. 3c, and (2) 10-node chain from Fig. 3e. We show results for the 10-node chain (4-node results provide similar insights). The two chain types are representative: the 4-node chain represents linear VNF chains while the 10-node chain represents VNF chains with complex topologies. Later, in Section 6.2, we perform chain allocation using all five VNF chains in Fig. 3.

[6] The company that manages this DC provides network security for enterprises and has requested to remain anonymous. Although the company's portfolio includes hosting third party NFs on its DC, we do not know if this particular topology actually hosts these NFs.

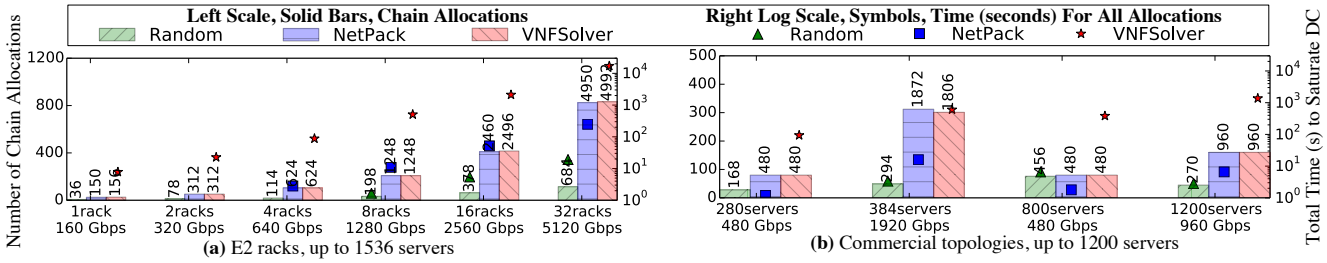

Figure 6: Runtime and total throughput achieved by RANDOM, NETPACK, and VNFSOLVER algorithms allocating 10-node VNF chain from Fig. 3e. Left. Allocating within DCs of the E2 [30] topology with increasing size (up to 1536 total servers). Right. Allocating the same chain in the commercial DCs of Section 5. Here, the largest DC has 1200 servers arranged in 60 racks. In both settings, NETPACK always achieves at least 96% of VNFSOLVER's allocations, while completing in less than 2.5% of VNFSOLVER's runtime.

In Fig. 6a (left), we plot each algorithm's performance on DCs in the E2 topology with 1–32 racks, when allocating the 10-node VNF chain. In these topologies, the bandwidth out of the gateway switch becomes a bottleneck. As described in Section 5, each ToR switch has 4 external ports with a total bandwidth of 160 Gbps to the gateway switch. As the chain must start and end at the gateway, the maximum throughput through this one rack instance is 156 Gbps since each concrete 10-node chain requires 6 Gbps of combined bandwidth for incoming and outgoing traffic. In the one rack experiment NETPACK allocates 150 Gbps of throughput, requiring a total of 0.19 CPU seconds to allocate the chains. For the same instance, RANDOM allocates only 36 Gbps also in 0.04s, and VNFSOLVER allocates 156 Gbps but in 7.77s.

In the remainder of Fig. 6a, we can see that the number of concrete chain allocations grows linearly as we increase the total number of racks, reaching 5120 Gbps of total throughput for 32 racks. NETPACK required a total of 247s to allocate 4950 Gbps in this 1536-server DC with median time of 0.2s per concrete chain. For the same instance, VNFSOLVER allocates 4992 Gbps (+0.84% from NETPACK) in 17105 seconds, requiring a median time of 19.31s per concrete chain.

Fig. 6b (right) shows results from an analogous experiment conducted on the commercial physical topologies described in Section 5. Here, we can see that in all but one case, VNFSOLVER and NETPACK achieve the maximum possible throughput, saturating the gateway switch. For example, in the 280-server region, the total possible bandwidth out of the gateway is 480 Gbps, allowing at most 240 Gbps of throughput into the chain (as 240 Gbps must also exit the chain back through the gateway switch). VNFSOLVER fully utilizes this bandwidth, allocating all 480 Gbps through the chain, requiring a total of 93.84s to make the allocations. For the same instance, RANDOM allocates only 168 Gbps aggregate bandwidth in 0.71s, and NETPACK gets the optimum 480 Gbps in 1.19s. For the largest DC with 1200 servers, VNFSOLVER allocates the maximum possible bandwidth of 960 Gbps in 1370s, while NETPACK gets the same bandwidth in 6.6s.

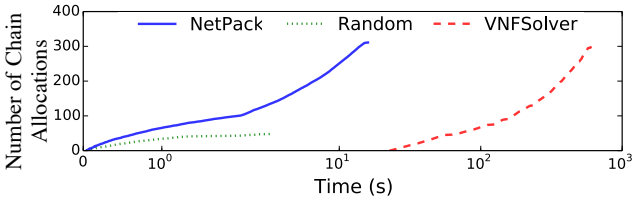

Figure 7: Number of VNF chain allocations over time by each algorithm, for the 384 server commercial DC from Fig. 6b. In this case, NETPACK achieved slightly more allocations in total than VNFSOLVER, while also being nearly 2 orders of magnitude faster.

In all but one of our experiments, VNFSOLVER achieves either the same throughput as NETPACK, or a higher throughput. However, in one instance (384 servers, in Fig. 6b), NETPACK was able to achieve a greater allocation density than VNFSOLVER. In fact, as can be seen in Fig. 7, NETPACK was in this case able to make all of its allocations before VNFSOLVER was able to achieve even a single allocation. As described in Section 3, even though VNFSOLVER is based on a complete allocation process and NETPACK is not, it is possible for VNFSOLVER to make suboptimal allocations because it makes repeated, greedily allocated calls to the underlying constraint solver. As NETPACK is stochastic, one potential explanation for this is that variability due to the random seed in NETPACK allowed it to make unusually good choices in this particular example. To address this question, we re-ran each algorithm 10 times on the 384 server instance. Across these runs, the total number of allocations found by VNFSOLVER varied by less than 3.7%; while NETPACK's allocations varied by less than 0.7%, both much less than RANDOM (which varied by as much as 10.4%). In fact, across these 10 runs, even the least

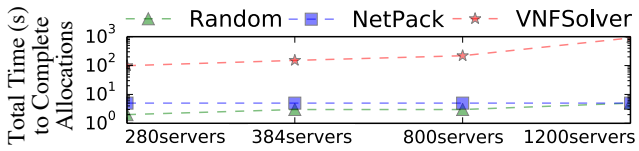

Figure 8: Chains with five different topologies from the literature [5, 21, 30]. In these experiments, all three algorithms allocated a total of 160 Gbps throughput, combined, to these five chains, in each of the four commercial topologies described in Section 5.

number of allocations made by NetPack was larger than the best solution found by VNFSolver.

In this particular case, a total of 1920 Gbps can theoretically be allocated through the chain. However, VNFSolver achieves only 1806 Gbps in 603s while NetPack achieves 1872 Gbps (+3.6%) in 15.95s, corresponding to 94% and 99% of the maximum possible throughput, respectively. Even though NetPack and VNFSolver are unable to achieve full utilization in this one case (due to the complex structure of this particular topology), both algorithms' total utilization remains reasonable.

In addition to the E2 rack and commercial topologies, we performed chain allocation on the Facebook DC topology described in Section 5, with between 1 and 48 pods. As in the previous settings, we found that NetPack consistently achieved within 99% of VNFSolver's allocations (while requiring < 1% of the runtime). In one case (discussed in the next section), Random nearly matched NetPack's allocations; in the remaining cases Random achieved < 40% of the allocations as NetPack. There, Random only managed to allocate chains to saturate 606 Gbps (32%) bandwidth under 18s, while both NetPack and VNFSolver were able to saturate the full 1920 Gbps pod bandwidth requiring 31s and 9328s, respectively. This experiment confirms that NetPack is fast, and is a good fit for modular DCs with no oversubscription.

In addition to data center size, the algorithm's time to allocate a chain also depends on the chain length. As we show in Section 3.2, the chain length is a multiplying factor in NetPack's algorithmic complexity. However, in practice, NetPack performs well for chain lengths that are likely to be encountered during VNF allocation. Our experiments with allocating 4-node chain and 10-node chain across three classes of DC topologies show that in the worst case NetPack consumes 94% more time to allocate 10-node chain, and only 54% on average. These results demonstrate that NetPack can handle chains with various lengths, 10 being the longest considered in the literature.

In all of the above experimental settings, NetPack and VNFSolver achieve a high degree of locality, placing all nodes on either a single server, or on a single server and that

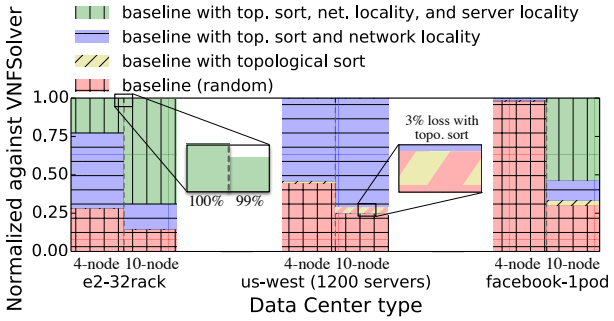

Figure 9: Contribution of each NetPack optimization on the number of allocations, as compared to Random. Allocations are shown as a percentage of those achieved by VNFSolver. In most cases, topological sorting provided a small benefit, though in some cases it decreased allocations slightly (dashed region above).

server's ToR switch. This ensures that the allocated chains also achieve low end-to-end latency.

6.2 Allocations from multiple tenants

So far, we discussed cases in which an operator allocates bandwidth for a single chain topology. In practice, we expect allocations to the same infrastructure for multiple tenants with different topologies. To demonstrate our support for this case, we simultaneously allocated a combined 160 Gbps of bandwidth through five VNF chain topologies from Section 5, in each of the four commercial DCs. That is, we choose a chain at random from Fig. 3 with its corresponding bandwidth, and allocation continues until 160 Gbps is reached. Results are shown in Fig. 8, ranging from under 5s required in the smallest DC with NetPack, to 907s in the largest DC (with 1200 servers) with VNFSolver.

6.3 Evaluating each NetPack optimization

Fig. 9 shows the contribution to NetPack by each optimization in Section 3: topological sort, network-locality, and server-locality. We allocate chains with two different lengths (the 4-node chain from Fig. 3c and 10-node chain from Fig. 3e) on each of the largest topologies from our three DC topology classes. To compare NetPack with Random and VNFSolver, we start with Random as a baseline and normalize the total chain allocations against VNFSolver.

As Fig. 9 shows, Random does poorly on all instances except one: 4-node chain allocation on a Facebook pod. Random does well here because locality matters less for short chains, and this DC topology has full-bisection bandwidth. On the same topology Random struggles with a 10-node

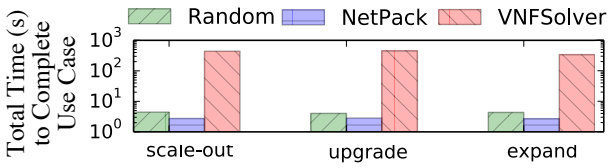

Figure 10: Use case completion time of *scale-out*, *chain-upgrade*, and *expand* on the commercial top. with 1200 servers. Operations are applied to an allocated abstract 4-node chain with 100 Gbps throughput.

chain, while topological sort contributes an extra 3%, network-locality an additional 12%, and server-locality an additional 53%, which completely closes the gap with VNFSolver. The other point of significance is one case when topological sort (on its own) results in 3% fewer allocations than Random. This happens with the 10-node chain allocation on the 1200 server commercial DC. The 3% drop is indicated in Fig. 9 with a *dashed* box on top of the baseline (zoomed in, right). Our experiments show that when used by itself, topological sorting results in at most a small increase, and occasionally a small decrease, in the total number of allocations.

Overall, enabling all three optimizations (including topological sort) yields best results for NetPack (typically within 99% of VNFSolver's allocations), while also being as fast or faster than Random, as seen in Fig. 8.

6.4 Chain API operations

Finally, we evaluate management use cases from Section 2: *scale-out*, *chain-upgrade*, and *expand*. For each use case we first place a single abstract 4-node chain to handle 100 Gbps. Then, we perform the API operations to achieve the three use cases on the allocated concrete chains. Each of these use case experiments were run independently.

We run three use cases on all commercial topologies. Fig. 10 shows results for the largest topology (due to lack of space). For the smallest topology with 280 servers (not shown) NetPack completes *scale-out* in 0.32s and VNFSolver completes it in 57.07s. On the largest topology (Fig. 10), NetPack completes scale-out in 1.74s and VNFSolver in 438s, which is reasonable given the 100 Gbps throughput along the allocated chain. The *chain-upgrade* and *expand* cases are similar, and require 1.84s/1.68s for NetPack, and 447s/337s for VNFSolver, respectively, for the smallest/largest topologies.

All three use cases remain practical with NetPack on E2 and Facebook topologies, as well (not shown for brevity). For the E2 topology with 32 racks, NetPack required 1.75s for *scale-out*, 2s for *chain-upgrade*, and 1.64s for *expand*. For the same use case VNFSolver required 711s, 675s, and 708s, respectively. A Facebook pod with 48 racks required 1.79s for *scale-out*, 1.91s for *chain-upgrade*, and 1.84s for *expand* from NetPack, while VNFSolver required 1199s, 1196s, and 632s for each.

7 DISCUSSION

7.1 Steady-state operation

Cloud operators would like to maintain high DC utilization during steady-state operation, where VNF chains arrive, mutate, and depart. In this work, we mainly focus on a VNF chain scheduler's ability to achieve high DC utilization during initial allocation and demonstrate the scheduler's support for lifecycle operations. However, its performance during chain deallocation and reallocation remains under-explored. Although we expect that NetPack will be able to maintain high DC utilization during steady-state operation, it might introduce several challenges in practice. The main practical concern is *reconfiguration minimization*, where existing VNF chains should not be relocated to optimize the overall placement. Relocation disturbs the existing flows and potentially violates tenant SLAs by causing latency variations. Thus, a good VNF scheduler should maximize DC utilization during steady-state while minimizing reconfiguration for already allocated chains. Similar optimizations have been previously explored in SOL [14]; we see this as a promising direction for future research.

As discussed in Section 2, we split each tenant's abstract chain into multiple concrete chains and request the scheduler to allocate those concrete chains. We believe this chain decoupling mechanism will help maintain high utilization during steady-state operation. This decoupling allows the scheduler to operate on chains with fine-grained resource footprints. Thus, the scheduler should be able to spread thin-footprint concrete chains into different parts of a potentially fragmented DC. Although this seems promising, such allocation must be accomplished without violating tenant SLAs, in particular latency requirements. We leave empirical validation of this intuition to future work.

7.2 NF profiles and scheduler input

Our VNF chain scheduling algorithms consume three types of input described in Table 2. These include NF's compute and memory footprint per unit of bandwidth, and a flag indicating whether this NF can be placed on a switch. Note that all three inputs are internal to the cloud provider (tenants only specify the chain bandwidth) and the scheduler can be extended with additional input types.

Cloud providers can leverage existing tools, such as NFVPerf [26], NFV-Vital [6], or Probius [27], to create compute and memory profile of each NF. This prior work demonstrates that VNF resource requirements depend on many factors, such as NF configuration (rulesets in the Firewall), traffic

pattern (packet size, burst rate), and NF position in the chain. For example, Probius [27] reported that the throughput of a chain with four NFs can vary by up to 5× when the NF sequence is shuffled.

We argue that such variations should be resolved outside the scheduler. The only requirement for the scheduler is to produce a valid placement given an accurate input. For example, to achieve high DC utilization while guaranteeing chain SLA (e.g., throughput and latency), a cloud provider can use NF's worst-case profile and adjust DC's overcommit ratio [29] appropriately. For example, if a cloud provider finds the actual DC utilization to be only 60% when the scheduler reports 100% allocation, the cloud provider can increase the overcommit ratio from 1.0 to 1.4 to make the DC capacity appear 40% greater to the scheduler. Such indirect solutions allow the scheduler to handle a wide range of NFs in diverse DC settings.

Our approach to handling the compute, memory, and switch resource requirements of an NF is mostly consistent with the literature [14, 25, 30, 35]. However, there are other work which model additional constraints during chain placement. These include reduced packet processing latency due to CPU core-affinity of chain NFs [48], and packet (or flow, or request) arrival rate and size [24, 38, 46]. We consider such fine-grained metrics to fall out of scope of the data center scale VNF chain scheduler. For example, Zhang et al. [47] found that such fine-grained metrics will overwhelm a VNF scheduler even in the single host setting, given a high packet arrival rate.

Cloud providers can also combine a DC-level *global* chain scheduler with a VNF-aware *local* OS scheduler. In such a setting, the host-local scheduler accepts the VNFs assigned to a single host (by the global scheduler) and further optimizes the host-level placement by adjusting core-affinity, Rx/Tx queue sizes, flow priorities, etc. Splitting the scheduling duties in this way allows for high DC utilization while guaranteeing chain SLAs [25].

We believe that this approach is particularly appealing in the context of lightweight VNF chains at scale [23, 47]. For example, the authors of Flurries [47] consider an OS-level VNF scheduler in which over 80,000 VNFs run on a single server, each second, where each VNF handles a separate flow. We believe cooperative VNF chain scheduling between NetPack and a VNF-aware OS scheduler would be able to handle such high-churn VNF chain allocation.

7.3 Failures

Hardware failures. Failures in large-scale deployments are inevitable. A study across tens of geographically distributed Microsoft DCs found that over 20% of devices have availability of three nines [13]. In other words, over 20% of devices experience 8.76 hours of annual downtime. The same study found that network redundancy reduces the median impact of failures by up to 40%.

The abstract-concrete chain decoupling that we propose in this paper allows for failure masking: concrete chains implementing the same abstract chain can be assigned to different hardware resources, improving fault tolerance. This is analogous to approaches based on replication and hardware redundancy. Note that *server-locality* described in Section 3.2 refers to the individual NFs of the concrete chain, not to concrete chain instances. This is important as our allocation algorithms do not co-locate concrete chain instances at the same server as this would void the fault tolerance benefit.

Individual VNF failures. VNFs may also fail due to software bugs, mis-configuration, and upgrades. A recent study of 2000+ physical NFs across 10+ DCs found that 5% of Firewall, 4% IDS, and 7% of load-balancer failures are due to software issues [34]. Although orthogonal to the VNF chain allocation and management, which is the focus of this paper, reliability of an individual VNF is a critical operational aspect. Recent work, such as FTMB [40], can improve individual NF robustness. Our allocation algorithms and management API can be extended with support for such techniques.

API failures. Our prototype assumes no API failures. We believe that API failures should be handled transparently to the tenants, for example by using recent work on providing ACID semantics within a SDN [43]. For example, this can be achieved with a shim layer between the controller and the network infrastructure [49]. Such built-in design not only prevents inconsistent packet processing due to partial API failures but also simplifies the tenant API.

7.4 Emulator limitations

Because we leverage Sonata's VNF chaining API to build individual concrete chains, our Daisy implementation does not handle several practical concerns: (1) efficiency loss due to duplicated NF elements, and (2) complexity of coordinating state between several, logically identical, NF elements. We believe that both of these must be solved at lower layers of the stack. Recent work on NF consolidation [18] may help with the first concern, and stateless NFs [19] or S6 [44] may help with the second concern.

8 RELATED WORK

We review the VNF literature in relation to our primary contribution: scalable VNF chain allocation and management.

Chain allocators. Slick [2] and SOL [14] address chain allocation and placement. Slick provides a heuristic-based algorithm while SOL uses a constraint solver (CPLEX [16]) to perform the allocation. Neither of them consider scales beyond 100 servers nor develop an API for scalable chain

management. A body of VNF chain allocation work assumes that a single VNF instance can process flows from different tenants [24, 38, 46]. This assumption is counter to existing cloud isolation guarantees and is unrealistic due to concerns around security, performance variation, and the need to support custom NF images. The algorithms by Yu et al. [38] and Zhang et al. [46] are therefore incompatible with ours, since in our setting tenants explicitly request *isolated chains*. Additionally, the algorithm by Zhang et al. [46] simplifies the problem by excluding *network infrastructure* as a bottleneck, considering only edge server resources for optimization. This is an over-simplification as most of the chain allocation complexity stems from the end-to-end bandwidth guarantees.

The ILP constraint-based algorithm proposed in VNF-P [24] can be applied in our setting. However, that algorithm is not scalable. It takes around 2s to allocate 100 VNF chains with length two on DC with 19 elements (10 edge nodes, 5 switches, 4 core routers). NetPack takes only 0.69s (3x shorter) to allocate 100 chains of length 10 (5x longer) on the largest commercial topology with 1265 elements (66x larger).

Another ILP based VNF chain allocation algorithm is developed by Qazi et al. in SIMPLE [35]. In that work, the authors optimize middlebox policy enforcement in an enterprise setting, where network topology and middlebox policies (which can be modeled as a VNF chain) are expected to change infrequently. Such infrequent change assumption allows authors to apply a *pruning stage* to reduce the number of physical nodes considered for a chain placement. Their evaluation shows that the pruning stage takes around 1800 seconds for a DC with 250 nodes, and needs to be redone when middlebox policies change. However, middlebox policy changes frequently in our setting: each time a new chain allocation (or update, or deallocation) request is made, which we expect to happen every second (or fraction of a second) at DC scale. Such a high churn rate leaves no room for an expensive pruning stage, and renders SIMPLE's ILP based algorithm prohibitively expensive for chain placement at data center scale.

VNF frameworks. APLOMB [41] surveys NF outsourcing and builds a system for low-latency packet processing in the cloud. Embark [21] goes a step further with a mechanism for handling encrypted traffic. Dysco [45] proposes a protocol to dynamically chain network functions. E2 [30] and NetBricks [31] are frameworks for developing and deploying high-performance NF chains. E2 develops a chain placement heuristic and demonstrates its ability to handle chain allocation on one rack. None of these solve chain allocation and management beyond a single rack. In our work, we consider a DC scale with dozens of racks and 1000+ servers.

VNF consolidation. CoMb [39], OpenBox [5], and mOS [18] are examples of systems that use VNF consolidation to achieve high-speed packet processing. These works address the important problem of hardware resource utilization by an individual NF or a VNF chain, but none directly address chain allocation and management.

VNF state management. Split/Merge [36], Stratos [11], OpenNF [12], Stateless [19], and S6 [44] are examples of systems which deal with NF state consistency during chain management scenarios, like the scale-out and other scenarios we consider in Section 2. These works are complementary to our chain allocation and management API, and can be used as a more robust mechanism to implement these APIs.

9 CONCLUSION

We introduced techniques for VNF chain allocation and management at data-center-scale: a set of operations that serve as an API for chain management, and NetPack, a heuristic bin-packing algorithm for scalable VNF chain allocation.

In addition to handling physical topologies that are orders of magnitude larger than prior work, our work supports (1) realistic topologies composed of racks, switches, and servers; (2) locality-aware allocation of arbitrary VNF chain topologies; and (3) management of allocated chains, such as scaling out and in-place upgrades (adding/removing NFs).

Despite the simplicity of NetPack, in our experiments it always achieved *at least* 96% of the throughput allocated by VNFSolver, a *complete* algorithm based on constraint solving, while running 82× faster. We prototyped our approach by building Daisy and evaluated it on real traffic. We also compared the utilization and runtime of the algorithms on 3 types of realistic DC topologies and 5 different VNF chains.

ACKNOWLEDGMENTS

We thank our shepherd, Timothy Wood, and the anonymous reviewers for their thoughtful feedback. Mihir Nanavati, Tony Mason, and Amanda Carbonari gave feedback on early drafts of this paper. This research was funded by NSERC discovery grants to Ivan Beschastnikh, Holger Hoos, and Alan Hu, and an NSERC discovery accelerator supplement to Alan Hu. Nodir Kodirov is supported by a UBC four-year doctoral fellowship (4YF). This work is supported in part by a gift from Microsoft Azure, and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

REFERENCES

[1] Alexey Andreyev. 2014. Introducing data center fabric, the next-generation Facebook data center network. (2014). [Online at code.facebook.com/posts/360346274145943; accessed 03-31-2018].

[2] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. 2015. Programming Slick Network Functions. In *Proceedings of the Symposium on SDN Research (SOSR '15)*. 14:1–14:13. https://doi.org/10.1145/2774993.2774998

[3] Microsoft Azure. 2018. Linux Virtual Machines. (2018). [Online at azure.microsoft.com/en-ca/pricing/details/virtual-machines/linux; accessed 03-31-2018].

[4] Sam Bayless, Nodir Kodirov, Ivan Beschastnikh, Holger H. Hoos, and Alan J. Hu. 2017. Scalable Constraint-Based Virtual Data Center Allocation. In *International Joint Conference on Artificial Intelligence, IJCAI*. 546–554. https://doi.org/10.24963/ijcai.2017/77

[5] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 511–524. http://doi.acm.org/10.1145/2934872.2934875

[6] Lianjie Cao, Puneet Sharma, Sonia Fahmy, and Vinay Saxena. 2015. NFV-VITAL: A framework for characterizing the performance of virtual network functions. In *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 93–99. https://doi.org/10.1109/NFV-SDN.2015.7387412

[7] Digital Corpora. 2009. Real packet traces. (2009). [Online at downloads.digitalcorpora.org /corpora/scenarios/2009-m57-patents/net/; accessed 03-31-2018].

[8] Dell. 2018. PowerEdge R730xd Rack Server. (2018). [Online at dell.com/ca /business/p/poweredge-r730xd/pd; accessed 03-31-2018].

[9] ADI Engineering. 2018. Seacliff Trail: Intel FM6000. (2018). [Online at adiengineering.com/products/seacliff-trail-switch; acc.d 05-31-2018].

[10] ETSI. 2013. NFV Whitepaper. (2013). [Online at portal.etsi.org/NFV /NFV_White_Paper2.pdf; accessed 03-31-2018].

[11] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. 2013. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR* (2013). http://arxiv.org/abs/1305.0209

[12] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM SIGCOMM Conference*. 163–174. http://doi.acm.org/10.1145/2619239.2626313

[13] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the 2011 ACM SIGCOMM Conference*. 350–361. http://doi.acm.org/10.1145/2043164.2018477

[14] Victor Heorhiadi, Michael K. Reiter, and Vyas Sekar. 2016. Simplifying Software-defined Network Optimization Using SOL. In *Conference on Networked Systems Design and Implementation (NSDI'16)*. 223–237. http://dl.acm.org/citation.cfm?id=2930611.2930627

[15] HPE. 2018. HPE Integrity BL870c i4 Server Blade. (2018). [Online at hpe.com/us/en/product-catalog/servers/integrity-servers/pip.hpe-integrity-bl870c-i4-server-blade.5330439.html; accessed 03-31-2018].

[16] IBM. 1997. Optimization, CPLEX. (1997). [Online at www-03.ibm.com/software/products/en/ibmilogcpleoptistud; accessed 03-31-2018].

[17] Intel. 2018. Ethernet Switch FM6000 Series Datasheet. (2018). [Online at intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf; accessed 03-31-2018].

[18] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Conference on Networked Systems Design and Implementation (NSDI'17)*. 113–129. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/jamshed

[19] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *Conference on Networked Systems Design and Implementation (NSDI'17)*. 97–112. http://dl.acm.org/citation.cfm?id=3154630.3154639

[20] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562. https://doi.org/10.1145/368996.369025

[21] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. 2016. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Conference on Networked Systems Design and Implementation (NSDI'16)*. 255–273. http://dl.acm.org/citation.cfm?id=2930611.2930629

[22] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*. 19:1–19:6. https://doi.org/10.1145/1868447.1868466

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

[24] Hendrik Moens and Filip De Turck. 2014. VNF-P: A model for efficient placement of virtualized network functions. In *10th International Conference on Network and Service Management (CNSM) and Workshop*. 418–423. https://doi.org/10.1109/CNSM.2014.7014205

[25] Ali Mohammadkhan, Sheida Ghapani, Guyue Liu, Wei Zhang, K. K. Ramakrishnan, and Timothy Wood. 2015. Virtual function placement and traffic steering in flexible and dynamic software defined networks. In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*. 1–6. https://doi.org/10.1109/LANMAN.2015.7114738

[26] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. 2016. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 154–160. https://doi.org/10.1109/NFV-SDN.2016.7919491

[27] Jaehyun Nam, Junsik Seo, and Seungwon Shin. 2018. Probius: Automated Approach for VNF and Service Chain Analysis in Software-Defined NFV. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. 14:1–14:13. https://doi.org/10.1145/3185467.3185495

[28] Open vSwitch. 2018. Open Virtual Switch. (2018). [Online at openvswitch.org; accessed 03-31-2018].

[29] OpenStack. 2018. Overcommitting CPU and RAM. (2018). [Online at docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html; accessed 05-31-2018].

[30] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '15)*. 97–112. http://dl.acm.org/citation.cfm?id=3154630.3154639

[31] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Conference on Operating Systems Design and Implementation (OSDI'16)*. 203–216. http://dl.acm.org/citation.cfm?id=3026877.3026894

[32] Manuel Peuster, Holger Karl, and Steven van Rossem. 2016. MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments. In *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 148–153. https://doi.org/10.1109/NFV-SDN.2016.7919490

[33] Manuel Peuster, Holger Karl, and Steven van Rossem. 2017. Sonata NFV SDK. (2017). [Online at github.com/sonata-nfv/son-emu; accessed 03-31-2018].

[34] Rahul Potharaju and Navendu Jain. 2013. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. In *Internet Measurement Conference (IMC '13)*. 9–22. https://doi.org/10.1145/2504730.2504737

[35] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the 2013 ACM SIGCOMM Conference*. 27–38. http://doi.acm.org/10.1145/2486001.2486022

[36] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Conference on Networked Systems Design and Implementation (NSDI'16)*. 227–240. http://dl.acm.org/citation.cfm?id=2482626.2482649

[37] Ryu Community. 2018. Ryu SDN Framework. (2018). [Online at osrg.github.io/ryu; accessed 02-24-2018].

[38] Yu Sang, Bo Ji, Gagan R Gupta, Xiaojiang Du, and Li Yehnology. 2017. Provably Efficient Algorithms for Joint Placement and Allocation of Virtual Network Functions. In *IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057036

[39] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *Conference on Networked Systems Design and Implementation (NSDI'12)*. 24–24. http://dl.acm.org/citation.cfm?id=2228298.2228331

[40] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM SIGCOMM Conference*. 227–240. http://doi.acm.org/10.1145/2785956.2787501

[41] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the 2012 ACM SIGCOMM Conference*. 13–24. https://doi.org/10.1145/2342356.2342359

[42] The Snort Team. 2018. Network Intrusion Detection & Prevention System. (2018). [Online at snort.org; accessed 03-31-2018].

[43] Anduo Wang, Wenchao Zhou, Brighten Godfrey, and Matthew Caesar. 2014. Software-Defined Networks as Databases. In *Open Networking Summit*. https://www.usenix.org/conference/ons2014/technical-sessions/presentation/wang

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

[45] Pamela Zave, Ronaldo A. Ferreira, Xuan Kelvin Zou, Masaharu Morimoto, and Jennifer Rexford. 2017. Dynamic Service Chaining with Dysco. In *Proceedings of the 2017 ACM SIGCOMM Conference*. 57–70. https://doi.org/10.1145/3098822.3098827

[46] Qixia Zhang, Yikai Xiao, Fangming Liu, John C.S. Lui, Jian Guo, and Tao Wang. 2017. Joint Optimization of Chain Placement and Request Scheduling for Network Function Virtualization. In *International Conference on Distributed Computing Systems (ICDCS)*. 731–741. https://doi.org/10.1109/ICDCS.2017.232

[47] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. 3–17. http://doi.acm.org/10.1145/2999572.2999602

[48] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. 2016. Performance management challenges for virtual network functions. In *IEEE NetSoft Conference and Workshops (NetSoft)*. 20–23. https://doi.org/10.1109/NETSOFT.2016.7502435

[49] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. 2015. Enforcing Customizable Consistency Properties in Software-Defined Networks. In *Conference on Networked Systems Design and Implementation (NSDI'15)*. 73–85. http://dl.acm.org/citation.cfm?id=2789770.2789776

[50] Zscaler. 2018. Cloud Security. (2018). [Online at zscaler.com; accessed 03-31-2018].