`sfc2cpu`: Operating a Service Function Chain Platform with Neural Combinatorial Optimization

Patrick Krämer[1], Philip Diederich[1], Corinna Krämer[2], Rastin Pries[3], Wolfgang Kellerer[1], Andreas Blenk[1]
[1]Technical University of Munich [2] Papierfabrik Louisenthal GmbH [3]Nokia Bell-Labs

***Abstract*—Service Function Chaining realized with a micro-service based architecture results in an increased number of computationally cheap Virtual Network Functions (VNFs). Pinning cheap VNFs to dedicated CPU cores can waste resources since not every VNF fully utilizes its core. Thus, cheap VNFs should share CPU cores to improve resource utilization. `sfc2cpu` learns efficient VNF to core assignments that increase throughput and reduce latency compared to three baseline algorithms. To optimize VNF assignments, `sfc2cpu` uses game theory combined with Neural Combinatorial Optimization in a novel way. Measurements in a real hardware testbed show that `sfc2cpu` increases throughput by up to $36\,\%$ and reduces latency by up to $59\,\%$ compared to Round Robin. We show that `sfc2cpu` can be incrementally deployed and easily integrated into existing infrastructures.**

***Index Terms*—Reinforcement Learning, Network Virtualization, Game Theory, Neural Combinatorial Optimization, Digital Twin**

I. Introduction

The Context: Micro-service based network virtualization. Edge computing and Network Function Virtualization (NFV) are two essential aspects of future networks. Infrastructure for edge computing is located at varying distances to customers [1] and can cover a campus, a city or a metropolitan area [1], [2]. NFV is the key technology to facilitate packet processing on top of this infrastructure. NFV is associated with easier deployment, maintenance and better scalability. This is achieved with Virtualized Network Functions (VNFs) running on commodity hardware. Typical VNFs include firewalls, intrusion detection systems and video optimizers [3], [4]. VNFs can be composed to arbitrary Service Function Chains (SFCs) [5]. Each SFC might process only a fraction of the overall traffic, specifically in multi-tenant environments.

While easier to deploy than dedicated hardware, today's VNFs are often monolithic software packages that are difficult to scale, change and maintain [6]. A natural step from a software-engineering point of view is the decomposition of monolithic VNFs into smaller micro-VNFs (µVNFs), each implementing one dedicated functionality [5], [7], [8].

The Problem: Assigning µVNFs to CPU cores. µVNFs are usually computationally cheaper than monolithic VNFs since they implement less functionality [5]. Implementing SFCs based on micro-services increases the number of µVNFs compared to monolithic VNFs. The larger number of µVNFs collides with the current best practice of core-pinning, i.e., assigning each VNF to a dedicated core [7]. Core-pinning can waste resources: A µVNF might not fully utilize a core, specifically when processing only a fraction of traffic in multi-tenant environments.

Contrary to VNFs, µVNFs can be co-located on the same core. To maintain throughput and keep latency low, the demand of co-located µVNFs should not lead to over-utilization of the shared core. This can be formulated as a bin-packing problem — an NP-complete combinatorial optimization problem [9]. Here, N objects with individual weights have to be placed on M bins, such that no bin is overloaded. In our case, objects correspond to µVNFs, weights to computational cost and bins to cores, and our objective is maximizing throughput.

The Challenge: Interference between µVNFs. The problem is further complicated through interference between SFCs and µVNFs as well as system effects. For instance, a reduction in throughput early in the chain reduces the demand, i.e., weight, of later µVNFs [8]. A bottleneck at the end of a SFC can impact earlier µVNFs through backpressure or congestion control [8]. Besides, µVNFs can influence each other's computational cost, e.g. by competing over the cache [10]. As a result, in our bin packing problem, the weights (computational cost) of the objects (µVNFs) are not static but vary based on the actual assignment.

Formally modelling these impacts is difficult, time-consuming and possibly has to be repeated for different hardware and software stacks [11]. Besides, interferences can have a detrimental effect on algorithms that rely on densely packing cores. Those solutions are vulnerable to demand changes and already small spikes can lead to poor performance. An effect that we show in our evaluation.

The Solution: `sfc2cpu`. We propose `sfc2cpu`, a system that uses Neural Combinatorial Optimization (NCO) paired with Reinforcement Learning (RL) to learn how to solve the bin-packing problem by exploiting scenario-specific properties [12]–[14]. This approach has three advantages: (1) `sfc2cpu` can learn interference between µVNFs; (2) `sfc2cpu` can adapt to system effects; and (3) `sfc2cpu` can tailor solutions to a concrete deployment scenario.

Contribution. We design, implement and evaluate `sfc2cpu` in a real testbed. `sfc2cpu` improves throughput up to $36\,\%$ and reduces latency up to $59\,\%$ compared to Round Robin in our measurements. Further, `sfc2cpu` can be incrementally deployed, combined with fall-backs, allows changes to individual µVNFs without touching others, and is easy to integrate into existing infrastructures.

The paper is organized as follows: Sec. II mathemati-

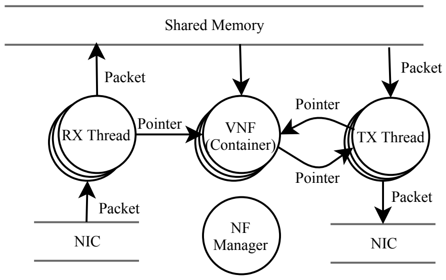


Fig. 1: ONVM architecture.

cally derives how co-locating VNFs on one core influences throughput and latency and gives background information and guidelines for the operation of SFC platforms with core sharing. Sec. III describes the design of sfc2cpu. Sec IV describes how sfc2cpu uses game theory and NCO to learn algorithms. Sec. V compares measurement results of sfc2cpu and baseline algorithms.

II. Background and problem

This section introduces background information on the underlying SFC platform OpenNetVM (ONVM) in Sec. II-A, Linux schedulers in Sec. II-B, and formulates the resulting assignment problem as well as presents guidelines in Sec. II-C.

A. OpenNetVM

Fig. 1 shows the architecture of ONVM [15], which is extended by sfc2cpu. ONVM is designed for Service Function Chaining and focuses on the overhead reduction of copying packets between VNFs of the same SFC, setting ONVM apart from other systems in the line of ZygOS [16], which focus on a single application.

The most important entities of ONVM are the RX- and TX-Threads, the NF Manager and containerized NFs. Upon startup, the NF Manager creates a shared memory region for all NFs. The NF Manager spawns TX- and RX-threads. The RX-thread fetches packets from the NIC, places the packets in the shared memory region, and inserts pointers into the input buffers of the first NFs in the configured SFCs. The TX-threads copy pointers between the output and input buffers of chained NFs. The TX-thread also transfers outbound packets to the corresponding NICs. ONVM uses core-pinning, i.e., assigns each VNF to a dedicated core. Core-pinning can result in significant resource overhead if individual VNFs do not utilize their CPU core fully. Scenarios like this can occur if micro-service architectures are adopted for VNFs [8], or in multi-tenant systems where individual SFCs serve only a small fraction of the overall traffic [7].

B. Task Scheduling on NF Platform

The Completely Fair Scheduler (CFS). NF platforms usually run on top of a Linux system [7], [8], [15], [17], [18]. The default scheduler of Linux systems is CFS. Each CPU core has its own CFS instance [19], [20]. CFS approximates an ideal processor that can execute multiple tasks simultaneously. For example, the ideal processor would grant 50 % of its time to two simultaneously running tasks [19], [20].

CFS has a *target latency*, during which every task gets a turn on the processor. If the target latency is infinitely small, then the processor would be equivalent to the ideal processor. If the processor has N tasks, then each task gets at most a $\frac{1}{N}$ slice of the target latency. For two tasks and a target latency of 24 ms (the default latency on our testbed server), each task gets a slice of 12 ms [19], [20].

A task gets preempted once the task reaches the end of its time slice. Tasks can voluntarily yield the CPU. The remaining time of the time slice is neither granted to subsequent tasks nor added to the yielding task's time slice in later scheduling periods. To give a task more CPU time, CFS allows to weight time slices. By increasing or decreasing the weight of a task, the corresponding time slice can be increased or decreased [19], [20]

Rate-cost proportional fairness. NFVNice shows that CFS can cause unfair processing of traffic, and thus proposes rate-cost proportional fairness (RC) [8]. RC weights the time slices of the NFs proportional to the product of packet arrival rate and processing cost, using the built-in capabilities of CFS, resulting in fair processing of network flows and improved system throughput. Besides, NFVNice uses backpressure (BP) to discard packets that would get dropped on overloaded VNFs and thus frees computational resources along an SFC [8]. sfc2cpu uses RC and BP (see Sec. III-B).

C. Placement and Scheduling Problem

Optimization opportunity. sfc2cpu exploits the observation that once the requested processing demand (load) of µVNFs sharing one CPU core exceeds the capacity of that core, *throughput decreases* and *latency strongly increases*.

Decreasing throughput in overload situations is obvious: If the service rate of µVNFs cannot keep up with the arrival rate of packets because they do not get enough CPU resources, throughput decreases. Note that one limiting µVNF is enough to reduce the throughput of a complete SFC. Also, packets get buffered, resulting in queuing delays. The time-slicing of CFS increases the latency further. In the worst case, packets *have to wait multiple scheduling periods*, leading to latencies in the order of tens of milliseconds *for a single µVNF*. As we will show, effects like this are avoidable with an intelligent assignment of µVNFs to CPU cores. Previous work advocating the scheduling of multiple µVNFs on one CPU core misses this optimization opportunity [7], [8].

Guidelines. The guidelines that follow are: (1) Co-locate µVNFs that do not overload a core; (2) be conservative when estimating processing cost and packet arrival rate, i.e., it is better to overestimate; (3) use core-pinning for computational expensive µVNFs.

The first guideline characterizes the underlying optimization

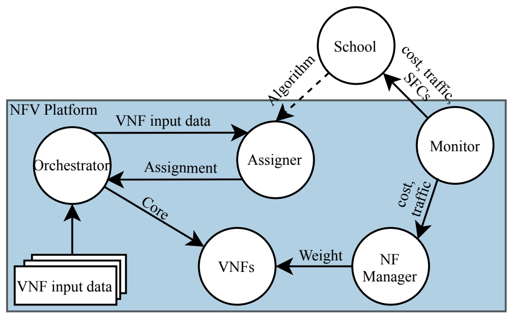


Fig. 2: Overview of pipeline.

problem as a constrained satisfaction problem:

$$\sum_{j \in \mathcal{J}} l(j,m)\, x_{m,j} \leq c(m), \qquad \forall m \in \mathcal{M}, \tag{1}$$

$$\sum_{m \in \mathcal{M}} x_{m,j} = 1, \qquad \forall j \in \mathcal{J}, \tag{2}$$

$$x_{m,j} \in \{0,1\}, \qquad \forall m,j \in \mathcal{M} \times \mathcal{J}, \tag{3}$$

where $\mathcal{J}$ is the set of deployed µVNFsand $\mathcal{M}$ the set of CPU cores. The binary variable $x_{m,j}$ indicates if µVNF j is placed on CPU m. Function $l(j,m)$ returns the load µVNF j induces on CPU m, and $c(m)$ returns the capacity of CPU m.

The above problem is an instance of the well known NP-complete bin-packing problem [9]. Solutions can thus be time-consuming to obtain. Also, finding an assignment that satisfies the constraints is not guaranteed to achieve the highest possible throughput on a real system. As we will show in our analysis, interference that cannot be captured in a model leads to performance degradation for assignments that satisfy the above constraints.

We believe that this particular problem is a good fit for a learned heuristic that can exploit the problem-specific structure and include such system effects [12]–[14].

III. Data-Driven Approach

This section describes the design goals of `sfc2cpu` in Sec. III-A, explains the components of `sfc2cpu` in Sec. III-B and the workload generation in Sec. III-C.

A. Design Goals

`sfc2cpu` is a SFC platform that allows CPU sharing between µVNFs. `sfc2cpu` uses an RL-based algorithm to assign µVNFs to CPU cores. The assignments can be continuously adapted during operation. For example, if new SFCs and µVNFs should be deployed, or a change in the processing cost or arrival rate is detected. In a cluster, a separate orchestration tool is responsible to place µVNFs on individual `sfc2cpu` instances.

`sfc2cpu` uses builtin tools for process management in Linux, i.e., `sfc2cpu` requires no changes to the underlying operating system. To achieve an efficient CPU assignment, `sfc2cpu` has four key properties:

Distributed operation: `sfc2cpu` uses the data of the node it is running on to make assignment decisions. Thus, `sfc2cpu` does not require a complex distributed control plane during operation.

Data efficient: Allocating µVNFs to CPUs is a combinatorial optimization problem with an exploding solution space. `sfc2cpu` should facilitate the learning of general strategies that are independent of perturbations to the problem. For example, `sfc2cpu` should handle a varying number of different µVNFs that could be deployed together in a specific scenario.

Non-preemptive operation: During operation, new SFCs or µVNFs are expected to arrive over time. A SFC platform should find good assignments for newly arrived instances without necessarily changing the assignment of previously deployed µVNFs

Incremental deployment and fallback: Incremental deployment of `sfc2cpu` in production systems must be possible to ensure correct operation. In the case of performance degradation, `sfc2cpu` should revert to a previous working version or a basic algorithm like Round Robin.

B. `sfc2cpu` Overview

Fig. 2 shows the components of `sfc2cpu`: NFV platform, Orchestrator, Monitor, Assigner, NFManager and School for Assigner. The Orchestrator coordinates the start of `sfc2cpu`. The Orchestrator receives a list of NFs chained to SFCs with estimated packet processing costs of the µVNFs and packet arrival rates of each SFC. If this information is not available before deployment, then `sfc2cpu` can start with a default placement, estimate those numbers and then re-assign the µVNFs. The Orchestrator passes this information to the Assigner. The Assigner responds with an assignment of µVNFs to CPU cores. The Orchestrator then starts the NFManager of `ONVM` and the containerized NFs. The NFManager uses `RC` and `BP` to dynamically adjust the CPU usage of NFs, i.e., The NFManager implements `NFVNice`. The Orchestrator can query the Assigner once at startup, or during operation if changes are detected.

The Monitor is integrated into the NFManager of `ONVM` and observes the deployed µVNFs, the cost of and traffic to µVNFs, and how µVNFs are chained. Besides, the NFManager uses this information to implement `RC` and `BP` [8]. The School uses information from the Monitor to generate synthetic workloads to teach Assigners. The simplified system model in Sec. IV provides the necessary feedback during learning. Once an Assigner graduates, it can be deployed to NFV platforms and replace the previous Assigner, e.g., a default algorithm such as Round Robin or a previously learned Assigner. The NFV platforms can benefit from improved assignments and revert to a basic algorithm should the current Assigner prove itself incompetent.

As we will show in our analysis, teaching Assigners with a simple model of the underlying system is enough to learn assignments superior to existing strategies for a specific scenario. For future research, we envision a system that is trained on data from a diverse set of µVNFs, SFCs, traffic patterns and past assignments. Together with recent proposals in offline

Property	Values
#SFCs	$\{1, 2, \ldots, 8\}$
#µVNFs per SFC	$\{1, 2, \ldots, 8\}$
Total #µVNFs	$\{4, \ldots, 16\}$
#CPUs	$\{4, 8, 12, 16\}$
#Dummy Loops	$\{0, 1, \ldots, 130\}$
Arrival rate of SFCs	$[0.02\,\text{Mpps}, 2.5\,\text{Mpps}]$

TABLE I: Properties of generated problem instances.

RL, this approach could produce ASSIGNERs that generalize to unseen scenarios, i.e., be used for transfer learning [21].

C. Problem Workload Generation

NCO can learn algorithms that operate NFV platforms at unmatched efficiency by exploiting patterns in the problem instances. We do not claim that NCO can learn algorithms that produce better assignments for *every* possible problem. What we believe and show in our analysis is that NCO can improve performance in *specific* scenarios, i.e., the typical workload of *one* operator. This idea is in line with the current trend of data-driven algorithm design, where algorithms are designed for specific subsets of general problem instances that outperform existing heuristics on that subset [12]. We design synthetic workloads with a varying number of µVNFs, SFCs, available CPU cores, processing costs and arrival rates. Table I shows the dimensions of the generated data.

Since an extensive library of micro-service based VNFs does not yet exist and implementing those is not the purpose of this work, we follow the same approach as previous work [8], [22] and use dummy µVNFs. The µVNFs for evaluating `sfc2cpu` are based on the `SimpleForward` example from `ONVM`. The `SimpleForward` looks up the next hop of the current packet and updates corresponding metadata in `ONVM`. We vary the packet processing cost of the `SimpleForward` by executing a `for`-loop with a configurable number of iterations, resulting in fine-grained control over the computational cost of µVNFs, allowing us to generate a wide range of different workloads. In future work, we plan to additionally vary the memory access patterns of µVNFs to learn interference in the memory subsystem.

To obtain the processing costs of µVNFs, we emulate the monitoring step from the previous section: We execute the modified `SimpleForward`, vary the number of loops in $\{10i \mid i = 0, 1, \ldots, 100\}$, and report the average processing cost. The cost ranges from 80 to 8 027 cycles. The µVNFs can process between $25.5\,\text{Mpps}$ and $0.275\,\text{Mpps}$.

The arrival rate generation for individual SFCs is based on sampling a Multinomial distribution from a Dirichlet distribution with concentration parameter $\alpha = 5$, and multiply the Multinomial with the system arrival rate. We use values up to $2.5\,\text{Mpps}$ for the system arrival rate. We chose this value as the limit because the TX-thread often limited throughput in our experiments, due to effects reported in [10], [23]–[25]. We found that up to $2.5\,\text{Mpps}$ can be reliably sustained and is enough to handle traffic on links in wide area networks [26].

We generate a problem instance by randomly sampling the number of SFCs and their arrival rates. We construct SFCs by selecting a SFC that has no µVNF yet. If all SFCs have at least one µVNF, we select a SFC at random. We generate a µVNF by sampling the number of iterations of the `for`-loop. The maximum number of iterations is constrained such that the µVNF could process the arrival rate of the SFC via core pinning. We use the previously estimated processing costs of our dummy µVNF for this. We next assign the µVNF to a core that can support the demand of the µVNF. If no such core exists, the number of iterations is reduced until the µVNF fits on the least loaded core. If no such iteration count exists, we stop and return the generated problem. This procedure ensures that the underlying problems are generally feasible but still challenging, which allows a fair comparison between assignment algorithms, i.e., heuristics should find a solution to the problem.

IV. LEARNING PLATFORM DESIGN

This section introduces the heart of `sfc2cpu`: the learning platform. Sec. IV-A and Sec. IV-B explain the role of game theory and RL in `sfc2cpu`. Sec. IV-C introduces the Neural Network (NN) architecture, Sec. IV-D the used RL algorithm, and Sec. IV-E the simple model of the system.

A. Game Formulation

We cast the assignment problem as a job scheduling game and treat it as a sequential game of perfect information. The game consists of a finite set of selfish players $\mathcal{J}$, which correspond to µVNFs. The actions or strategy space of each player corresponds to the available cores $\mathcal{M}$.

The players choose their actions one after the other, i.e., in sequence. All chosen actions, and the players themselves (how many, properties such as packet arrival rate or computational cost), are common knowledge. After every player has chosen her action, each player j receives a payoff π_j corresponding to the fraction of achieved rate divided by the requested rate of network packets.

Since the game is finite and sequential, it has at least one Subgame Perfect Nash Equilibrium (SPNE) in pure strategies [27]. An SPNE corresponds to an action profile from which no player can unilaterally deviate and achieve a better payoff. Further, the Price of Stability (PS) is one [28], i.e., the ratio of highest social welfare[1] across all SPNEs of the game, and the maximum social welfare is one. Thus, the assignment of µVNFs to cores with which maximum throughput is achieved is an SPNE of the game. If the optimal assignment would not be an SPNE of the game, then there would exist one player that could change her strategy and receive a higher payoff. This is possible only if the throughput of the player is currently restricted, i.e., the core the player is on is overloaded. Thus, the target core the player changes to must have a smaller load than the current core the player is

[1]Social welfare is the sum of individual player's payoff. In our case, the sum of the throughput of all µVNFs [27].

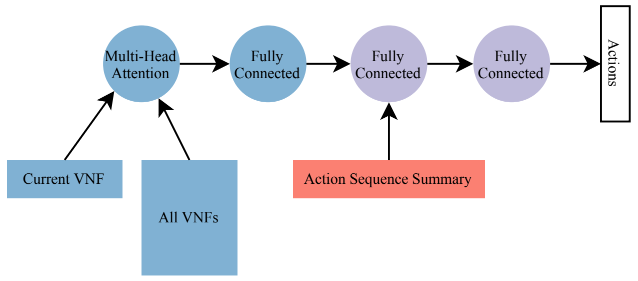


Fig. 3: Architecture of the NN for payoff prediction.

located on. This in turn would contradict the assumption that the system is in the optimum state □.

Formulating the assignment problem as a sequential game has two advantages: It makes `sfc2cpu` flexible and interpretable. Players can be represented with different learned models and can have varying objective functions. The sequential nature of the game allows `sfc2cpu` to predict placements for arbitrary subsets of µVNFs. The game itself can be analyzed, allowing operators to conjecture on the potential behavior of the system.

B. The Role of RL

Since the game is sequential, the best action for each player can be determined through Backwards Induction (BI) [27]. BI requires the evaluation of every possible strategy profile, i.e., every possible assignment of µVNFs to CPU cores [27]. Since this number grows exponentially, BI is not feasible in practice. Instead, we use RL to learn a function parameterized through a NN that predicts the expected payoff for player j of choosing action a given the previously played actions and the players that have yet to make their decision. That is, we use RL to learn the best responses, resulting in an SPNE of the game [29]. RL thus learns the local optima of the underlying optimization problem guided by the game dynamics. This approach has been proven to result in best-responses with high probability, even for games with imperfect information [29]. That is, this approach can learn actions resulting in an SPNE of the underlying game. SPNEs correspond to local optima of the actual optimization problem, i.e., assignments that maximize throughput.

C. Neural Network Architecture

Fig. 3 shows the NN architecture used to predict the expected payoff for all actions of one µVNF. The input to the NN is divided into data summarizing the already played actions shown as red rectangle in Fig. 3, and a representation of the current µVNF and all other µVNFs depicted as two blue rectangles. The size of the output layer corresponds to the number of cores. This data is exactly the data that is required for the game to have perfect information, i.e., the players know exactly which contingency they are in and can act correspondingly [27].

We use parameter sharing to reduce training time and sample complexity, i.e., we use the same weights for all µVNFs. Individual µVNFs are discriminated through the `Current VNF` input. Parameter sharing is common practice in deep reinforcement learning [30].

Action sequence summary. We represent the played actions with their effect on the CPU load. Each CPU core is represented by its relative load and the number of players that chose the core. The relative load of a core m is calculated as: $\frac{1}{c(m)} \sum_{j \in \mathcal{J}} x_{j,m} l\left(j, m\right)$, i.e., the sum of loads of µVNFs that chose m divided by the cycles of m. The summary is thus represented as a vector in $\mathbb{R}^{|\mathcal{M}|}$.

VNF representation. Each µVNF is represented with four numbers: The packet processing cost, the packet arrival rate in Mpps, the resulting load and a binary value that is one if the µVNF has already chosen an action and zero otherwise. The processing cost is normalized to a value between zero and one. The load is divided by the clock speed, resulting in a value between zero and one. The resulting vectors are combined in the `All VNFs` input, resulting in a $\mathbb{R}^{|\mathcal{J}| \times 4}$ matrix. The `Current VNF` vector is one row from `All VNFs` and corresponds to the µVNF that currently makes its move.

Relation between VNFs. The NN has a `Multi-Head Attention (MHA)` module. This module helps the current µVNF to set itself into relation to all other µVNFs [30], [31]. Thus, the NN can learn which of the other players constitute important information for its action. The module consists of multiple self-attention mechanisms using scaled-dot-product attention and a learnable non-linearity [31]. The output vectors of each module are concatenated to a single vector. This allows the NN to focus on different parts of the input [31].

The attention mechanism takes as input three tensors: `Queries`, `Keys` and `Values`. The `Keys` and `Values` correspond to the `All VNFs` tensor, i.e., are identical, and the `Queries` to the `Current VNF` vector. The inputs are linearly transformed through learnable weight matrices. The transformed `Queries` and `Keys` are used to calculate attention scores. Those are used as weights in a convex combination of the rows of the transformed `Value` tensor. Note that the attention mechanism is independent from the number of µVNFs, i.e., rows in the `All VNFs` tensor. Further, the attention mechanism is invariant to permutations of rows in the `All VNFs` tensor [31]. The NN architecture can thus be used for a variable number of µVNFs.

Action calculation. The `Action Sequence Summary` input and the `MHA` output is concatenated and passed through a sequence of fully connected layers, which finally produce the output, i.e., the expected payoff for the current µVNF to choose any of the available CPU cores.

The game formulation and NN architecture allow the resulting network to assign a single µVNF, a subset of µVNFs, or all µVNFs to cores given the placement of the remaining µVNFs. In an online scenario, new µVNFs or SFCs can be integrated without touching the assignment of already running µVNFs. Similarly, if changes in load or cost are detected, affected µVNFs or SFCs can be re-assigned individually.

Parameter	Value
train batch size	2048
buffer size	500 000
Attention heads	3
Attention dimension	4
Attention Dense	16
Fully Connected	$\{48, 48\}$
type	EpsilonGreedy
initial epsilon	1.0
final epsilon	0.02
epsilon timesteps	500 000
learning rate	Annealed with population-based training.

TABLE II: Hyper Parameter for model and training.

D. RL Algorithm

We use double Q-learning [32] with prioritized experience replay and a dueling architecture [33] implemented in [34] to train the NN. We use `ASHA` [35] and population-based training [36] implemented in `Tune` [37] for hyperparameter tuning. Q-learning is a temporal difference method based on dynamic programming. In Q-learning, an agent learns action values of states: The expected cumulative reward the agent can obtain by taking action a in state s. In our case, the action values correspond to the payoff of the game, i.e., the ratio of achieved to requested packet rate.

E. Ideal System Model

Obtaining training samples on the real system is expensive. Evaluating a single assignment takes tens of seconds. A digital twin of the environment that allows the generation of necessary data is thus an important contribution.

In our twin, we calculate the expected service rate of a µVNF given its estimated computational cost and arrival rate. The time slice $t_{j,m}$ granted to µVNF j on CPU m is $t_{j,m} = T \cdot \frac{l(j,m)}{\sum_{i \in \mathcal{J}} x_{i,m} l(i,m)}$, where T is the scheduling latency. The number of packets processed by j during this time is: $s_j = \gamma \frac{t_{j,m} c(m)}{c(j)}$, where $c(j)$ returns the cost to process one packet in cycles, $c(m)$ returns the clock speed of CPU m in cycles per second, and $\gamma \in (0, 1)$ is a scalar factor to account for lost time due to context switches.

The model is simple and can be evaluated very fast. As our evaluation shows, this model suffices to guide the learning. The model can be improved with performance models of µVNFs that capture interference, and data gathered during operation. This could help to fine-tune and generalize the learned algorithms.

V. Evaluation

This section presents the results. Sec. V-A discusses hyper-parameters for RL and the NN. Sec. V-B discusses settings for the SFC platform itself. Sec. V-C evaluates VNF scheduling techniques and Sec. V-D compares the performance obtained with `sfc2cpu` against heuristic assignment algorithms.

All results are obtained on a server with two AMD EPYC 7301 16-Core CPUs and two Intel 10G X550T NICs. Traffic is generated on a separate server using MoonGen [38],

Parameter	Value
`CFS` Scheduling Latency	1 ms
Number of TX threads	5
Number of RX threads	1
CPU location	Cores 1-8 on Socket 1, Cores 8-16 on Socket 2
Packet Size	64 Byte
Arrival Rate	2 Mpps

TABLE III: Parameter Settings for `ONVM`.

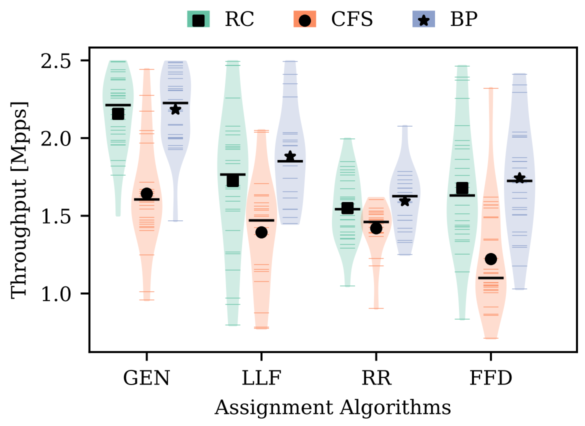


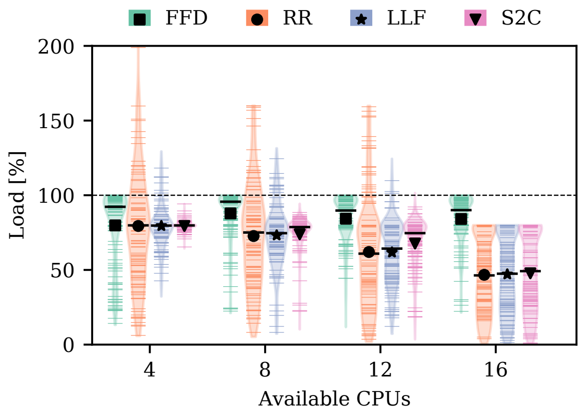
Fig. 4: Black lines represent the median, black markers the mean, colored lines the actual samples, the shaded area represents a density estimate. `CFS` achieves worst throughput.

which is also used to measure latency and throughput. The servers are connected back-to-back. The heuristic algorithms are Least Loaded First (`LLF`), Round Robin (`RR`), and First Fit Decreasing (`FFD`).

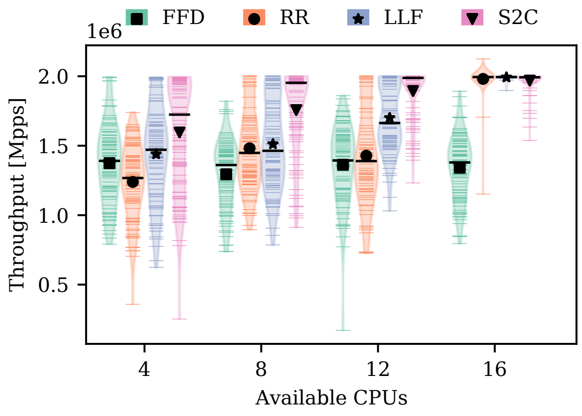
A. RL Hyperparameters

Tbl. II lists the hyperparameters used during learning. We use a batch size of 2 048 and a replay buffer size of 500 000 samples. For training, we use the epsilon greedy exploration strategy. We start with $\epsilon = 1$, i.e., fully random actions, and anneal the value to $\epsilon = 0.02$ over 500 000 time steps. Once ϵ is annealed, two percent of the actions are random. We do not use a fixed learning rate, but use population-based training [36] to adjust the learning rate over the course of training. We use a population size of ten. During evaluation, ϵ is set to zero.

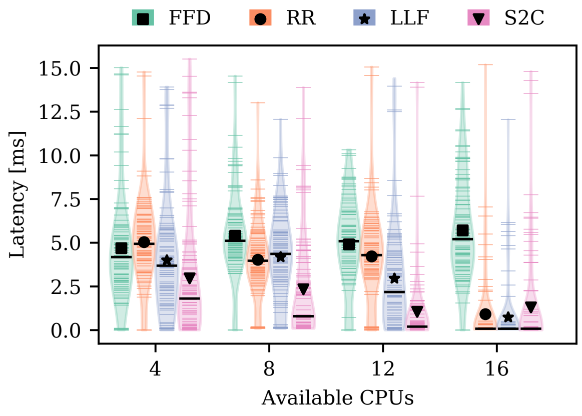
The `MHA` module of the NN has three attention heads. Input tensors are transformed into a four dimensional latent space. The fully connected layer following the `MHA` module has 16 neurons. The last two fully connected layers have 48 neurons each. We train separate models for different number of available CPUs, but always use the same architecture. The resulting networks have sizes from 50 KBytes in case of 4 CPUs to 160 KBytes for 16 CPUs. The most expensive operation for all networks is the calculation of the input to the second dense layers, which requires $48^2 + 48$ operations. The models are small, fit into a CPU cache and are computationally cheap.



(a) CPU Load.



(b) System Throughput.



(c) Average latency.

Fig. 5: Black lines represent the median, black markers the mean, colored lines the actual samples, the shaded area represents a density estimate. `sfc2cpu` achieves higher throughput, lower latency and better load balancing.

All fixed hyper-parameters are obtained through a parameter search using `ASHA` [35]. We used `OpenAI`'s `Tune` and `RLlib` library to scale training [34], [37]. We chose the architecture with best performance.

B. CPU Core, Thread, and `CFS` Settings

Tbl. III lists the hardware settings. We use one RX-Thread and five TX-threads. The NF Manager runs on core `C0`, the RX-Thread on `C1` and the TX-threads on `C2-C6`. We use one dedicated µVNF to split traffic between SFCs. This µVNF is pinned to `C7`. For the placements of µVNFs, we use `C8-C24`. Cores `C8-C15` are on one CPU socket, `C16-C24` on a the other socket. The `CFS` scheduling latency is set to 1 ms.

C. What is the impact of assignments on throughput?

We compare the throughput of the schedulers `BP`, `CFS`, and `RC` for the heuristic algorithms *RR*, *LLF*, *FFD* and *GEN* on 20 problem instances *GEN* represents the placements obtained from the generative process for the problem instances in Sec. III-C. Note that `BP` corresponds to an implementation of `NFVNice`. We generate 64 Byte packets with a total rate of 2.5 Mpps using MoonGen.

Fig. 4 shows that the placements from the generator (*GEN*) perform best with an average throughput of 2.21 Mpps. Fig. 4 shows that `CFS` results in the worst throughput. Reduction in throughput for *GEN* and *FFD* is 27.97 % and 47.11 %. Both algorithms allocate more µVNFs to a single CPU core than *LLF* and *RR*, and thus benefit from rate-cost proportional fairness. `BP` does not result in significant performance gains. But `BP` reduces the variance in the measured throughput. `BP` increases performance in overload scenarios where upstream µVNFs become a computational bottleneck [8]. Dropping packets frees resources along the chain, which can then be used by other µVNFs. In our evaluation, only *LLF* and *RR* overload cores and `BP` can improve over `RC`. *GEN* and *FFD* do not overload cores in theory, so `BP` can have only a limited impact.

In summary, the evaluation in Fig. 4 shows that a careful placement can improve the performance of systems like `NFVNice`. Further, we restrict our evaluations to the `RC` scheduling going forward.

D. `sfc2cpu` Performance

The RL algorithm is trained on the synthetic workload described in Sec. III-C, and evaluated with problem instances not seen during training. We compare `sfc2cpu` (abbreviated with *S2C*) to *RR*, *LLF* and *FFD*. All algorithms are compared across four CPU settings (4, 8, 12, and 16 cores), with 100 problem instances per setting. For each setting, we use MoonGen to generate 64 Byte packets with a rate of 2 Mpps, which is enough to handle traffic on a back-bone link [26].

`sfc2cpu` provides better load balancing. Fig. 5a shows the load of all cores over all experiments and confirms that `sfc2cpu` avoids over-utilization of cores. *FFD* densely packs the cores in all scenarios with a median load between 89.76 % and 95.65 %. The mean and median of *RR*, *LLF* and *S2C* are similar across all settings and decreases from 80.0 % for 4 cores to 47.5 % for 16 cores. The load induced on cores in settings with 4 - 12 cores varies strongly for assignments produced with *LLF* and *RR*, overloading (requesting more cycles than available) some cores up to 150 % and even more than $200\,\%$ in case of 4 cores. In contrast, `sfc2cpu` keeps the load consistently below $100\,\%$ for all settings.

`sfc2cpu` achieves higher throughput. Fig. 5b shows the achieved throughput for all scenarios. `sfc2cpu` achieves the highest throughput in scenarios with 4 - 12 cores, with a median throughput of $1.72\,$Mpps, $1.95\,$Mpps and $1.99\,$Mpps, respectively. The result for 16 cores is different, but expected. Here, *RR* and *LLF* result in core pinning, i.e., each µVNF is placed on one dedicated core and achieves a median throughput of $1.99\,$Mpps. *FFD* performs worst with a median throughput of $1.38\,$Mpps. As Fig. 5a shows, *FFD* densely packs µVNFs on cores. The densely packed µVNFs are more vulnerable to variations in processing cost and thus achieve lower throughput in every scenario. `sfc2cpu` learns this type of interference and is thus able to mitigate it.

Our results highlight the potential of data-driven algorithms: From Fig. 5a, *FFD* and `sfc2cpu` should always achieve full throughput. System level effects reduce the effective throughput. Thus, improving the system model to include system effects is an important next step.

`sfc2cpu` achieves lower latency. `sfc2cpu` provides the best latency values for 4 - 12 cores with $1.81\,$ms, $0.79\,$ms and $0.20\,$ms, respectively. `sfc2cpu` better distributes the workload, i.e., µVNFs among the CPU cores. Thus, the risk that µVNF j on CPU m cannot meet its arrival rate is reduced and packets wait at most $T - t_{j,m}$ seconds, i.e., less than $1\,$ms per µVNF. In contrast, *RR* and *LLF* overload CPUs in some cases, which results in increased latency. *FFD* densely packs CPUs resulting in interference between µVNFs, which causes buffering of packets and a median and average latency close to $5\,$ms for all settings. Again, when core pinning is possible, *RR* and *LLF* benefit from their simple assignment strategy. Fig. 5c shows that for all settings, scenarios with large tail latencies of up to $15\,$ms exist, i.e., packets get buffered.

VI. Related Work

Placement papers. Most prior work focuses on placing VNFs *on servers* [39]. Instead, our work focuses on assigning VNFs *to CPU cores*. Closest to our work is [40] and [10]. Both minimize data transfer between VNFs in the same SFC between NUMA nodes; both perform core pinning. In contrast, we target systems that allow core sharing. Also, our procedure can be applied to arbitrary NF platforms, whereas results in [40] and [10] are limited to specific technologies.

Performance modeling. The memory subsystem, data transfer between NUMA nodes, and interference on the NIC can impact the throughput, latency and CPU usage of NF systems [11], [23], [41]–[43]. Interference and computational cost can be modeled with ML [11], [44]. The focus of this work is not on the performance or interference modeling between VNFs. Actually, performance models can be used with `sfc2cpu` to learn assignment algorithms.

Thread scheduling. Our work is closely related to thread scheduling [24], [25], [45], [46]. Similar to OS schedulers, those algorithms are not designed with the specific workload of network processing in mind. Above solutions rely on frequent migration of threads between cores to improve performance metrics. However, frequent migration of VNFs is harmful for the specific networking workload [10].

NCO. RL and NCO can be used to learn the admission of jobs [47], and the assignment of jobs to compute resources [48]–[51]. Our work differs from previous work wrt. the underlying optimization problem that is solved with RL, and the problem domain. Previous results are thus not applicable. Also, we are the first to combine NCO with game theory.

NFV platforms. A range of NFV platforms and systems exist [7], [8], [15], [17], [18], [52]–[55]. Our work extends `ONVM` [15] and allows sharing of CPU cores between VNFs as in [8]. Our work is orthogonal to the previous work in that we focus on how to assign VNFs to CPUs to improve overall system performance. While implemented on top of `ONVM`, our modifications can be combined with any NFV platform to improve the underlying core assignment.

Dataplane operating systems. The objective of Dataplane operating systems [16], [22], [56]–[58] is to provide microsecond tail latencies for data-center applications. They are not designed for the SFC use-case, e.g., they do not support zero-copying of packets between VNFs as `ONVM` does. Techniques used in those systems could be combined with existing NFV platforms, though. Further, the goal of `sfc2cpu` is the co-location of VNFs such that overall system performance is improved which none of those systems consider.

VII. Conclusion

This paper investigates SFC platforms that allow CPU sharing between VNFs. We explain the impact of VNF to CPU assignments on throughput and latency and give guidelines on the operation of SFC platforms that ensure high throughput and low latency. We further propose `sfc2cpu` that learns to solve the assignment problem with Reinforcement Learning (RL). We compare `sfc2cpu` with three assignment algorithms and show with testbed measurements that `sfc2cpu` improves the throughput of SFC systems up to $36\,\%$ and reduces latency up to $59\,\%$ compared to Round Robin. `sfc2cpu` is incrementally deployable and easy to integrate into existing infrastructures, achieved through a novel combination of game theory and neural combinatorial optimization.

We believe that our work opens interesting future avenues. In particular, less predictable system behaviors or more complex VNFs and SFCs might increase the challenge to achieve efficient workload processing, which is a perfect application for NCO. Investigating how `sfc2cpu` can generalize from collected data to new scenarios is an essential next step towards more efficient network virtualization.

Acknowledgment

This work has been supported in part by the European Union's Horizon 2020 research and innovation program (grant agreement No 647158-FlexNets) and was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 438892507.

References

[1] P. Kalmbach, A. Blenk, W. Kellerer, R. Pries, M. Jarschel, and M. Hoffmann, "GPU Accelerated Planning and Placement of Edge Clouds," in *NetSys*, Garching b. München, Germany, 2019, pp. 1–3.

[2] C. Bachhuber, A. S. Martinez, R. Pries, S. Eger, and E. Steinbach, "Edge Cloud-based Augmented Reality," in *MMSP*, Kuala Lumpur, Malaysia, 2019, pp. 1–6.

[3] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges," *Proc. IEEE*, vol. 107, no. 4, pp. 711–731, 2019.

[4] M. Hoffmann, M. Jarschel, R. Pries, P. Schneider, A. Jukan, W. Bziuk, S. Gebert, T. Zinner, and P. Tran-Gia, "SDN and NFV as Enabler for the Distributed Network Cloud," *Mob. Netw. Appl.*, vol. 23, no. 3, pp. 521–528, Jun. 2018.

[5] S. Sahhaf, W. Tavernier, M. Rost, S. Schmid, D. Colle, M. Pickavet, and P. Demeester, "Network service chaining with optimized network function embedding supporting service decompositions," *Computer Networks*, vol. 93, pp. 492 – 505, 2015.

[6] R. C. Martin, *Clean Architecture - A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2018.

[7] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization," in *CoNEXT'16*. Irvine, California, USA: ACM, 2016, pp. 3–17.

[8] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains," in *SIGCOMM '17*. Los Angeles, CA, USA: ACM, 2017, pp. 71–84.

[9] G. Dsa and J. Sgall, "First Fit bin packing: A tight analysis," in *STACS 2013*, ser. Leibniz International Proceedings in Informatics (LIPIcs), N. Portier and T. Wilke, Eds., vol. 20. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2013, pp. 538–549.

[10] Z. Zheng, J. Bi, H. Yu, H. Wang, C. Sun, H. Hu, and J. Wu, "Octans: Optimal Placement of Service Function Chains in Many-Core Systems," in *IEEE INFOCOM 2019*. Paris, France: IEEE, 2019, pp. 307–315.

[11] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-Aware Performance Prediction For Virtualized Network Functions," in *SIGCOMM '20*. Virtual Event, USA: ACM, 2020, pp. 270–282.

[12] R. Gupta and T. Roughgarden, "Data-Driven Algorithm Design," *Commun. ACM*, vol. 63, no. 6, pp. 87–94, May 2020.

[13] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural Combinatorial Optimization with Reinforcement Learning," *CoRR*, vol. abs/1611.09940, 2016. [Online]. Available: http://arxiv.org/abs/1611.09940

[14] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *CoRR*, vol. abs/1811.06128, 2018. [Online]. Available: http://arxiv.org/abs/1811.06128

[15] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A Platform for High Performance Network Service Chains," in *HotMIddlebox '16*. Florianopolis, Brazil: ACM, 2016, pp. 26–31.

[16] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *SOSP '17*. Shanghai, China: ACM, 2017, p. 325341.

[17] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A Framework for NFV Applications," in *SOSP '15*. Monterey, California, USA: ACM, 2015, pp. 121–136.

[18] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *NSDI 14*. Seattle, WA, USA: USENIX Association, Apr. 2014, pp. 459–473.

[19] T. kernel development community. (2020) Cfs scheduler. [Online]. Available: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html

[20] R. Love, *Linux Kernel Development*, 3rd ed. Crawfordsville, Indiana, US: Pearson Education, Inc, 2010.

[21] A. Kumar, A. Zhou, G. Tucker, and S. Levine, "Conservative Q-Learning for Offline Reinforcement Learning," *arXiv e-prints*, p. arXiv:2006.04779, Jun. 2020.

[22] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazires, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency," in *NSDI 19*. Boston, MA, USA: USENIX Association, Feb. 2019, pp. 345–360.

[23] C. Sieber, R. Durner, M. Ehm, W. Kellerer, and P. Sharma, "Towards Optimal Adaptation of NFV Packet Processing to Modern CPU Memory Architectures," in *CAN 17*. Incheon, Republic of Korea: ACM, 2017, pp. 7–12.

[24] J. Rao, K. Wang, X. Zhou, and C. Xu, "Optimizing virtual machine scheduling in NUMA multicore systems," in *HPCA*. Shenzhen, China: IEEE, 2013, pp. 306–317.

[25] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," in *ISCA*, Minneapolis, MN, USA, 2014, pp. 325–336.

[26] C. for Applied Internet Data Analysis (CAIDA). (2020) Trace statistics for caida passive oc48 and oc192 traces. [Online]. Available: https://www.caida.org/data/passive/trace_stats/

[27] R. Gibbons, *A primer in game theory.* Pearson Academic, 1992.

[28] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic Game Theory*. New York, NY, USA: Cambridge University Press, 2007.

[29] A. Greenwald, J. Li, and E. Sodomka, *Solving for Best Responses and Equilibria in Extensive-Form Games with Reinforcement Learning Methods*. Cham: Springer International Publishing, 2017, pp. 185–226.

[30] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent Tool Use From Multi-Agent Autocurricula," *arXiv e-prints*, p. arXiv:1909.07528, Sep. 2019.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need," in *NIPS 2017*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Long Beach, CA, USA: Curran Associates, Inc., 2017, pp. 6000–6010.

[32] H. v. Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[33] Z. Wang, N. d. Freitas, and M. Lanctot, "Dueling Network Architectures for Deep Reinforcement Learning," *CoRR*, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[34] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica, "Ray rllib: A composable and scalable reinforcement learning library," *CoRR*, vol. abs/1712.09381, 2017. [Online]. Available: http://arxiv.org/abs/1712.09381

[35] L. Li, K. G. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar, "Massively parallel hyperparameter tuning," *CoRR*, vol. abs/1810.05934, 2018. [Online]. Available: http://arxiv.org/abs/1810.05934

[36] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu, "Population based training of neural networks," *CoRR*, vol. abs/1711.09846, 2017. [Online]. Available: http://arxiv.org/abs/1711.09846

[37] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[38] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *IMC '15*. Tokyo, Japan: ACM, 2015, p. 275287.

[39] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *J. Netw. Comput. Appl.*, vol. 75, pp. 138 – 155, 2016.

[40] Y. Wang, "NUMA-aware design and mapping for pipeline network functions," in *2017 4th International Conference on Systems and Informatics (ICSAI)*. Hangzhou, China: IEEE, 2017, pp. 1049–1054.

[41] R. Durner, C. Sieber, and W. Kellerer, "Towards Reducing Last-Level-Cache Interference of Co-Located Virtual Network Functions," in *ICCCN*. Valencia, Spain: IEEE, 2019, pp. 1–9.

[42] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the Performance Interference of Co-Located Virtual Network Functions," in *INFOCOM*. Honolulu, HI, USA: IEEE, Apr. 2018, pp. 765–773.

[43] Z. Peng, W. Feng, A. Narayanan, and Z.-L. Zhang, "NFV Performance Profiling on Multi-core Servers." Paris, France: IEEE, Jun. 2020, pp. 91–99.

[44] A. Mestres, E. Alarcon, and A. Cabellos, "A machine learning-based approach for virtual network function modeling," in *WCNCW*, Barcelona, Spain, Apr. 2018, pp. 237–242.

[45] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in *ASPLOS XV*, ser. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 129–142.

[46] B. Lepers, V. Quma, and A. Fedorova, “Thread and Memory Placement on NUMA Systems: Asymmetry Matters,” in *USENIX ATC 15*, ser. USENIX ATC 15. Santa Clara, CA, USA: USENIX Association, 2015, pp. 277–289.

[47] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, “Resource Management with Deep Reinforcement Learning,” in *HotNets ’16*. Atlanta, GA, USA: ACM, 2016, pp. 50–56.

[48] “Learning Scheduling Algorithms for Data Processing Clusters,” in *SIGCOMM’19*, Beijing, China.

[49] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, “Device Placement Optimization with Reinforcement Learning,” *CoRR*, vol. abs/1706.04972, 2017. [Online]. Available: http://arxiv.org/abs/1706.04972

[50] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, “A Hierarchical Model for Device Placement,” in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Hkc-TeZ0W

[51] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. M. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, A. Babu, Q. V. Le, J. Laudon, R. C. Ho, R. Carpenter, and J. Dean, “Chip Placement with Deep Reinforcement Learning,” *CoRR*, vol. abs/2004.10746, 2020. [Online]. Available: https://arxiv.org/abs/2004.10746

[52] G. P. Katsikas, T. Barbette, D. Kostiundefined, R. Steinert, and G. Q. Maguire, “Metron: NFV Service Chains at the True Speed of the Underlying Hardware,” in *NSDI’18*. Renton, WA, USA: USENIX Association, 2018, pp. 171–186.

[53] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, “Iron: Isolating Network-based CPU in Container Environments,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA, USA: USENIX Association, Apr. 2018, pp. 313–328.

[54] V. Sciancalepore, F. Z. Yousaf, and X. Costa-Perez, “z-TORCH: An Automated NFV Orchestration and Monitoring Solution,” *IEEE TNSM*, vol. 15, no. 4, pp. 1292–1306, 2018.

[55] L. Rizzo, P. Valente, G. Lettieri, and V. Maffione, “PSPAT: Software packet scheduling at hardware speed,” *Comput. Commun.*, vol. 120, pp. 32 – 45, 2018.

[56] T. Barbette, G. P. Katsikas, G. Q. Maguire, and D. Kosti, “RSS++: Load and State-Aware Receive Side Scaling,” in *CoNEXT ’19*. Orlando, Florida, USA: ACM, 2019, pp. 318–333.

[57] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, “Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads,” in *NSDI 19*. Boston, MA, USA: USENIX Association, Feb. 2019, pp. 361–378.

[58] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun, “Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs,” in *APNet ’19*. Beijing, China: ACM, 2019, pp. 71–77.