POLITECNICO DI TORINO
Repository ISTITUZIONALE

iPlace: An Interference-aware Clustering Algorithm for Microservice Placement

Original
iPlace: An Interference-aware Clustering Algorithm for Microservice Placement / Adeppady, M., Chiasserini, C.F., Karl, H., Giaccone, P.. - STAMPA. - (2022). (IEEE International Conference on Communications (IEEE ICC) 2022 Seoul, Korea, Republic of 16-20 May 2022) [10.1109/ICC45855.2022.9839222].

Availability:
This version is available at: 11583/2951275 since: 2022-01-19T11:19:12Z

Publisher:
IEEE

Published
DOI:10.1109/ICC45855.2022.9839222

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

13 September 2026

iPlace: An Interference-aware Clustering Algorithm for Microservice Placement

Madhura Adeppady[†], Carla Fabiana Chiasserini[†], Holger Karl[‡], Paolo Giaccone[†]
[†] Politecnico di Torino, Italy [‡] Hasso Plattner Institute, University of Potsdam, Germany

***Abstract*—Efficiently deploying microservices (MSs) is critical, especially in data centers at the edge of the network infrastructure where computing resources are precious. Unlike most of the existing approaches, we tackle this issue by accounting for the interference that arises when MSs compete for the same resources and degrades their performance. In particular, we first present some experiments highlighting the impact of interference on the throughput of co-located MSs. Then, we formulate an optimization problem that minimizes the number of used servers while meeting the MSs' performance requirements. In light of the problem complexity, we design a low-complexity heuristic, called iPlace, that clusters together MSs competing for resources as diverse as possible and, hence, interfering as little as possible. Importantly, the choice of clustering MSs allows us to exploit the benefit of parallel MSs deployment, which, as shown by experimental evidence, greatly reduces the deployment time as compared to the sequential approach applied in prior art. Our numerical results show that iPlace closely matches the optimum and uses 10-63% fewer servers compared to alternative schemes, while proving to be highly scalable.**

I. Introduction

Edge computing is a prominent and promising technology for 5G-and-beyond networks; it enables offloading of service tasks from either mobile devices or the core network to the edge, thus reducing end-to-end latency and resource utilization. While edge computing scenarios vary widely, a prevalent characteristic are services that are composed of simpler components. To witness, service function chains (SFCs) [1] comprise individual virtual network functions (VNFs) or even other, simpler chains; microservice chains are similarly composed of individual microservices (MSs) or other chains. Differences exist: VNFs might run close-to hardware; MSs might run inside general-purpose containers. But while details and terminology certainly differ, many core ideas and issues are very similar across these domains. In the following, for the sake of concreteness, we tackle an MS architecture running inside containers, but we emphasise that our ideas and results apply to SFCs/VNFs as well.

Services as such are deployed by an orchestrator that places, deploys, connects, and configures the needed components in one or several edge data centers, so as to meet the associated Service Level Agreement (SLA). Locally, components run inside a container, facilitated by a hypervisor. Current hypervisors isolate containers running on the same server by, e.g., placing them on dedicated cores [2], [3], [4], [5], thus allowing to *consolidate* multiple components on the same hardware. Nonetheless, containers still compete for other hardware resources, predominantly memory subsystem resources [6], [7], [8], [9]. Thus, unregulated competition for a server's shared resources by the MSs degrades throughput compared to them running alone on the same server. Such performance degradation experienced by an MS is referred to as *interference* or *noisy neighbor problem* [10].

Interference is complicated by the multitude of different MSs, each with its own code: they contend for resources differently (e.g., emphasizing memory over I/O) and, hence, experience interference differently [11]. Thus, resources that might have sufficed to meet a particular MS's SLA goal in some combination of components might no longer suffice when combined with other components. This makes guaranteeing SLAs challenging when dynamically consolidating MSs on a limited set of servers, which is the typical operational condition in edge computing.

Recently, several research efforts have been made to address this issue [6], [9], [12], [13]. The proposed solutions use either resource partitioning schemes [9] or supply-demand models [12], [13] to quantify interference. However, none of these methods fully addresses interference completely, as they fail to consider all resources responsible for interference [6]. A few notable approaches [6], [7], [8] have built models to predict the throughput of a target MS when co-located with other MSs and, using such prediction models, they have proposed placement solutions (which component executes where). But even though the prediction models are accurate [6], these placement approaches are somewhat straightforward and suffer from scalability issues.

Unlike prior art, in this paper we propose iPlace, an *Interference-aware Microservice Placement* (IMSP) approach based on clustering and on a prediction model. We start from the observation that, usually, requests for new services do not comprise just a single MS but a set, and also that multiple services might be requested simultaneously. Hence, instead of placing individual MSs sequentially as done in previous work, our solution uses a clustering phase and a placement phase for the sake of scalability. The key idea of clustering the MSs prior to placement stems from the experimental evidence showing that *parallel MSs deployment is much faster than sequential deployment*. Also, while MS clustering has been widely used for placement, e.g., in [14], existing approaches are not suitable for the problem under study as they do not consider interference. Instead, we cluster newly requested MSs so that MSs contending for *different* resources are grouped *together*, thereby *minimizing intra-cluster interference*. Then, we place each cluster in the server whose MSs interfere least

with that cluster's MSs, *minimizing inter-cluster interference* and also ensuring that no SLAs are violated. To account for interference in the clustering and placement phases, we use the *contentiousness* metric [6], [7], which captures the pressure a MS places on shared resources, and build a prediction model inspired by [6], which takes into account the interference among co-located MSs.

To summarize, our main contributions are as follows:

1) We introduce a system model capturing the major characteristics of the network system and the virtualized services; we do so by leveraging both previous work [6] and our own experiments;
2) We formulate an optimization problem for IMSP at the network edge to minimize the number of servers used to place the MSs while minimizing the adverse effects of performance interference;
3) Owing to the problem's NP-hardness, we develop iPlace, a heuristic, interference-aware algorithm for cluster-based MS placement;
4) Through extensive simulations, we demonstrate that iPlace efficiently solves the IMSP problem and outperforms state-of-the-art solutions.

To our knowledge, our work is the first to explore clustering to mitigate the effects of interference during MS placement.

The paper is organized as follows. Sec. II introduces our experimental method. Sec. III presents the system model and formulates IMSP as an optimization problem. Sec. IV describes our heuristic, iPlace, which is then evaluated in Sec. V. Finally, Sec. VI concludes the paper.

II. Measuring and Predicting Interference

We start by giving experimental evidence on how throughput degrades due to interference among competing MSs, despite a resource isolation setup in the server. We then show how to build a prediction model for estimating the throughput of competing MSs, which is required to develop an interference-aware MS placement solution. We stress, however, that our solution, introduced in Sec. III, can work with any other appropriate interference prediction model.

A. Experimental interference assessment

To observe the throughput degradation due to interference, we carried out several experiments using snort, a well-known open-source intrusion detection system, and pktstat, which displays real-time packet activities. The testbed we used is depicted in Fig. 1. The experiments were conducted on an Intel Core(TM) i7-7700K server with 4 CPU cores, 16 GB memory, and 8 MB LLC cache shared across all the CPU cores, while individual cores have 1 MB L2 cache and 128 KB L1 cache (resp.). Each MS runs on a *Docker container* pinned to a dedicated core using Docker runtime option cpuset-cpus, while iperf3 is used to generate traffic.

We first fed each MS with 100 flows and ran it individually, on a server dedicated to a single Docker container. Measurements conducted with increasing per-flow traffic load showed that, when running separately, pktstat and snort reach a

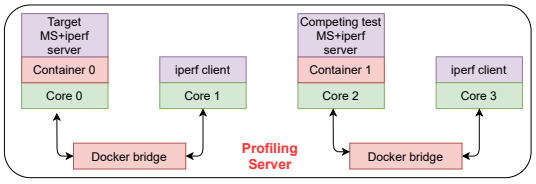

Fig. 1. Experiment setting: target MS and competing MS.

maximum throughput of 17 Gbps and 30 Gbps (resp.). We then evaluated the throughput of each MS with the other running as competing MS, for a varying number of flows and per-flow offered load. The results, shown in Fig. 2, highlight a throughput drop of 28.5 % for pktstat and of 22.5 % for snort, relative to their solo run, both for 100 concurrent flows. Also, as the workload and the number of concurrent flows of the competitor MS increase, the throughput degradation of the target MS becomes more severe. Thus, *despite the isolation of the CPU resources, interference is practically relevant.*

Throughput degradation of the target MS depends upon traffic load and packet processing logic of the competing MSs. As the competitor's number of concurrent flows or its packet rate grow, competition for memory subsystem resources increases, thus degrading performance. It is thus evident that interference plays an important role in MS placement and that inattentive co-location of MSs would seriously degrade throughput.

B. Building a prediction model

It is now clear that, to optimally place MSs, it is necessary to predict an MS's throughput taking into account interference. For the sake of concreteness and later evaluation, here we describe the model based on [6], which leverages two main concepts: *contentiousness* and *sensitivity*. Contentiousness measures the pressure (i.e., load) applied on shared server resources by an MS in the presence of competing MSs; sensitivity models the target MS's throughput as a function of its competitors' aggregate contentiousness.

This prediction model includes an *offline profiling phase* and an *online prediction phase*. In the former phase, contentiousness and sensitivity are computed a-priori, considering a target MS running on a server in the presence of a synthetic load. By letting this load increase, the increasing pressure of competing MS(s) on the shared resources is measured. Thus, contentiousness profiling consists of determining a set of vectors, one for each pressure level of the synthetic competitor(s). Profiling sensitivity then builds on a regression model leveraging the throughput of the target MS in the presence of varying synthetic contentiousness vectors. In the *online phase*, the sensitivity model predicts the target MS throughput, given the contentiousness vector of any real competitor(s) as input.

1) Offline profiling phase: To evaluate the contentiousness vector, we have considered various system-level metrics (e.g., instructions/cycle, L2/L3 cache misses/hits/occupancy, memory read/write operations) exposed by Intel's PCM framework, which is a performance monitoring API to collect real-time,

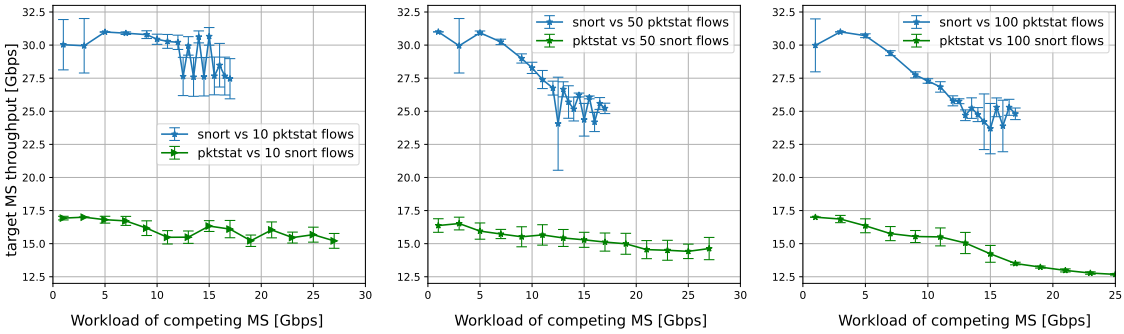

Fig. 2. Throughput of the target MS running with a competing MS, as the workload of the latter varies: 10 (left), 50 (center), 100 (right) concurrent flows, and traffic load equally distributed across the flows.

TABLE I
MOST MEANINGFUL SYSTEM LEVEL METRICS BASED ON THE CORRELATION COEFFICIENT (CC)

Snort		pktstat	
Metric	CC	Metric	CC
Core-1 EXEC	0.96	System L2MPI	0.98
System READ	0.90	Core-0 IPC	0.98
Core-1 IPC	0.89	Core-1 EXEC	0.96
System WRITE	0.88	Core-2 L2MISS	0.95
Core-2 L3MISS	0.85	System L2MISS	0.95
Core-1 L2MISS	0.83	Core-0 softirqs	0.94

architecture-specific resource usage metrics. Intel PCM outputs a wide range of metrics but not all of them are relevant. Out of those, we selected components for the contentiousness vector that are highly correlated with the target MS throughput, i.e., the Pearson correlation coefficient is larger than 0.7. Tab. I lists such system-level metrics for snort and pktstat.

The contentiousness vector $V_r^{(k)}(\mathbf{x})$ of MS r depends upon the competing workload, $\mathbf{x}$, generated by k MSs, each with a specific configuration and operational setting (e.g., traffic rate). As in [6], the sensitivity model of MS r, denoted by M_r, is then obtained by training a regression model mapping the contentiousness vector $V_r^{(k)}(\mathbf{x})$ into the observed throughput $P_r^{(k)}(\mathbf{x})$. More specifically, we use a Gradient Boosting Regressor model, as sensitivity is a non-linear, non-continuous function of the contentiousness vectors.

Further, the experimental results are used to compute the *representative* contentiousness vector $V_r^{(k)}$ of MS r, obtained by *averaging over all* the observed contentiousness vectors with k competing MSs, with respect to the workload values $\mathbf{x}$. $V_r^{(k)}$ is then fed as input to the sensitivity model in the online prediction phase.

2) Online prediction phase: It leverages the specific contentiousness vectors of the competitors and the sensitivity model of the target MS to predict the throughput of the latter. As an example, let us consider three MSs, r_a, r_b and r_c, running on the same server; similar arguments hold for an arbitrary number of MSs. To predict the throughput of r_a in the presence of r_b and r_c, we compute the *aggregate contentiousness* $V_{r_b,r_c}^{(2)}$ jointly imposed by the two competing MSs by combining their representative contentiousness vectors: $V_{r_b,r_c}^{(2)} = V_{r_b}^{(2)} + V_{r_c}^{(2)}$ where, with an abuse of notation, $+$ denotes an appropriate linear operator (e.g., sum for cache occupancy or the cache read/write operations, or average for cache hit or miss probability) applied to each component of the contentiousness vector, to reflect the combined effect of the two competing MSs. We stress that this approach showed to be very accurate [6]. Next, the throughput of r_a can be predicted via the sensitivity model as: $P_{r_a}(\{r_a, r_b, r_c\}) = M_{r_a}(V_{r_b,r_c}^{(2)})$.

Generalizing the above case, the throughput of r_a when running on server s with set $\mathcal{Y}_s \setminus \{r_a\}$ of competing MSs is predicted as:

$$P_{r_a}(\mathcal{Y}_s) = M_{r_a}\Big(\sum_{r \in \mathcal{Y}_s \setminus \{r_a\}} V_r^{(|\mathcal{Y}_s|-1)}\Big). \quad (1)$$

III. SYSTEM MODEL AND PROBLEM FORMULATION

We now describe the system model under study and formalize the IMSP problem to optimally place MSs.

A. System model

Let us focus on a single data center and let $\mathcal{S}$ be the set of servers available therein. We consider an online MS placement scenario in which a subset of servers in $\mathcal{S}$ run some pre-existing MSs, each of them currently satisfying its SLA. Let $\mathcal{F}_s$ be the set of pre-existing MSs running on server s; $\mathcal{F}_s = \emptyset$ if s is idle. Then, consider a set $\mathcal{R}$ of requests for MS instances, (possibly) related to different services, arriving at the orchestrator. Let t_r be the minimum required throughput, as per SLA, for an MS $r \in \mathcal{R}$.

We assume that the data center has ample bijection bandwidth and thus the throughput of an MS depends only upon its server's processing capacity, potentially influenced by interference. Thus, each MS $r \in \mathcal{R}$ can be placed independently from other $r' \in \mathcal{R}$. Also, each server $s \in \mathcal{S}$ has CPU and memory resources, denoted by $\hat{\tau}_s$, $\hat{\mu}_s$, respectively, while other resource types are sufficiently available. In addition, each MS request r entails CPU and memory demand as denoted by τ_r and μ_r, respectively.

While placing new MSs, pre-existing ones are not moved and their SLAs must be met even after the new MSs have been placed. If an MS request cannot be placed in any of the existing servers without violating the SLAs, then an additional server is provisioned.

Let $y_{r,s} \in \{0,1\}$, with $r \in \mathcal{R}$ and $s \in \mathcal{S}$, be a binary decision variable expressing whether a new MS r should be placed on server s or not, and let $\mathcal{Y}_s = \{r \in \mathcal{R} | y_{r,s} = 1\}$ be the set of MSs placed on server s. A server s is active (indicated by $n_s \in \{0,1\}$) if and only if it serves at least one MS r. Using the prediction model introduced in Sec. II, we can predict the throughput of any MS in a server with co-located MSs. We denote by $P_r(\mathcal{Y}_s)$ the predicted throughput of MS $r \in \mathcal{R}$ when running in server s and competing with MSs in $\mathcal{F}_s \cup \mathcal{Y}_s \setminus \{r\}$, i.e., pre-existing and newly placed MSs.

B. The IMSP problem formulation

Given the set of requested MSs, $\mathcal{R}$, the objective is to minimize the number of servers used to place the MSs, i.e.,

$$\min \quad \sum_{s \in \mathcal{S}} n_s \tag{2}$$

subject to system and SLA constraints:

$$\sum_{s \in \mathcal{S}} y_{r,s} = 1 \quad \forall r \in \mathcal{R} \tag{3}$$

$$n_s \leq \sum_{r \in \mathcal{R}} y_{r,s} + |\mathcal{F}_s| \quad \forall s \in \mathcal{S} \tag{4}$$

$$\sum_{r \in \mathcal{Y}_s} y_{r,s} \cdot \mu_r + \sum_{r \in \mathcal{F}_s} \mu_r \leq n_s \cdot \hat{\mu}_s \quad \forall s \in \mathcal{S} \tag{5}$$

$$\sum_{r \in \mathcal{Y}_s} y_{r,s} \cdot \tau_r + \sum_{r \in \mathcal{F}_s} \tau_r \leq n_s \cdot \hat{\tau}_s \quad \forall s \in \mathcal{S} \tag{6}$$

$$P_r(\mathcal{Y}_s \cup \mathcal{F}_s) \geq t_r \quad \forall s \in \mathcal{S}, r \in \mathcal{R} \cup \mathcal{F}_s \tag{7}$$

$$n_s \in \{0,1\} \quad \forall s \in \mathcal{S}, \tag{8}$$

$$y_{r,s} \in \{0,1\} \quad \forall s \in \mathcal{S}, r \in \mathcal{R}\,. \tag{9}$$

Eq. (3) specifies that a new MS must be placed on exactly one server. Eq. (4) ensures that a server is turned off if no MS is assigned to it. Eqs. (5)–(6) mandate that the memory and computing resource requirement of all (new and pre-existing) MSs allocated in server s cannot exceed the available server memory or computing capability; they also ensure that server s is active if any MS is assigned to it. Eq. (7) leverages (1) and imposes that the predicted throughput for any new and pre-existing MS must satisfy the throughput required specified in the SLA, thus the mutual interference across all MSs is acceptable.

Theorem: The IMSP problem in (2), subject to constraints (3)–(9), is NP hard.

The proof, omitted for brevity, shows that any instance of the bin packing problem, which is NP-hard, can be reduced to a simplified, off-line version of the IMSP in polynomial time. This makes a heuristic approach plausible: iPlace, described in the next section.

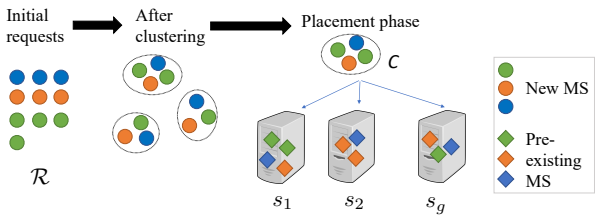

Fig. 3. Example of the clustering phase for a batch of requests (color indicates type of resources an MS competes for; shape distinguishes old vs. new MSs).

IV. iPlace: The Interference-Aware MS Placement

The key idea is to partition the set of new MSs into clusters in which the *MSs contend for different types* of resources. Clustering is motivated by experimental evidence showing that, e.g., deploying a batch of 50 MSs using Docker Swarm orchestrator reduces the deployment time by 46% when compared to sequential deployment. Further, while clustering MSs, we reduce the mutual interference of MSs in the same cluster, which allows them to coexist on the same server.

Then, as depicted in Fig. 3, the algorithm works in two phases: the *clustering phase*, which clusters the new MS placement requests based on their contentiousness, and the *placement phase*, which places each created cluster in the server accounting for the pre-existing MSs in the server.

In the clustering phase, we define the *distance* between any $r_i, r_j \in \mathcal{R}$ $(r_i \neq r_j)$ of MS placement requests with the following criterion: *larger distance* between MSs means that their corresponding contentiousness vectors are *more similar* and *compete more for similar resources*. Formally, we write:

$$d(r_i, r_j) = \|V^{(1)}_{r_i} - V^{(1)}_{r_j}\|_2^{-1}. \tag{10}$$

The MSs in $\mathcal{R}$ are initially clustered using the mean-shift clustering [15] technique that automatically discovers the number of clusters and the MSs to be included therein based on the chosen distance metric.

After clustering the MSs in $\mathcal{R}$ into clusters, the placement phase starts and the clusters are put in a queue in a random order, processed until each cluster is assigned to a server. More specifically, we define the distance between the server s running pre-existing MSs in $\mathcal{F}_s$ and the cluster $\mathcal{C}$ as:

$$\left\| V^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w \in \mathcal{F}_s} - V^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w \in \mathcal{C}} \right\|_2^{-1} \tag{11}$$

where $V^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w \in \mathcal{F}_s}$ is the aggregate contentiousness vector of all MSs placed in s and $V^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w \in \mathcal{C}}$ is the one for $\mathcal{C}$. All the eligible servers having sufficient computing and memory resources to host the MSs of cluster $\mathcal{C}$ are sorted in the increasing order of their distance with cluster $\mathcal{C}$ according to (11). To assess the impact of interference of the new MSs in $\mathcal{C}$ on the nearest server s consisting of $\mathcal{F}_s$ pre-existing MSs, we use the prediction model for the most critical MS placement request $\hat{r} \in \mathcal{C} \cup \mathcal{F}_s$:

$$P_{\hat{r}}(\mathcal{C} \cup \mathcal{F}_s) = M_{\hat{r}}(V^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w \in \mathcal{F}_s \cup \mathcal{C} \setminus \{\hat{r}\}}) \tag{12}$$

Algorithm 1 iPlace: Interference-aware MS placement

procedure MsPlacement($\mathcal{R}, \mathcal{V}, \widehat{\mathcal{S}}, \mathcal{M}$) ▷ $\widehat{\mathcal{S}}$: set of currently active servers
  $\mathcal{Z} \leftarrow$ MeanShiftClustering $(\mathcal{R}, \mathcal{V})$ ▷ Initially apply mean-shift clustering on the placement requests based on $\mathcal{V}$
  while $\mathcal{Z} \neq \emptyset$ **do**
    $\mathcal{C} \leftarrow \mathcal{Z}$.pop() ▷ Remove the first cluster from $\mathcal{Z}$ to $\mathcal{C}$
    $\mathcal{A} \leftarrow \{s \in S | \mu_{\mathcal{C}} \leq \hat{\mu}_s \wedge \tau_{\mathcal{C}} \leq \hat{\tau}_s\}$ ▷ Servers with enough resources
    Sort $\mathcal{A}$ in increasing Distance $(\mathcal{C}, \mathcal{F}_a, \mathcal{V})$
    for every s in $\mathcal{A}$ **do** ▷ For each server
      $\hat{r} \leftarrow$ most critical MS in $\mathcal{C} \cup \mathcal{F}_s$
      if PredictSLAViolations$(\hat{r}, \mathcal{F}_s, \mathcal{V}, \mathcal{M}) =$ no violations **then**
        $\mathcal{F}_s \leftarrow \mathcal{F}_s \cup \mathcal{C}$ ▷ Place the cluster $\mathcal{C}$ on the server s
        break ▷ Consider a new cluster
    if Cluster $\mathcal{C}$ is not placed **then** ▷ Placing $\mathcal{C}$ in any $s \in \mathcal{A}$ violates SLA
      if $|\mathcal{C}| = 1$ **then** ▷ If size of $\mathcal{C}$ is 1, then a new server is created
        $\widehat{\mathcal{S}} \leftarrow \widehat{\mathcal{S}} \cup \{n\}$ ▷ Start a new server n and place $\mathcal{C}$ there
        $\mathcal{F}_n \leftarrow \mathcal{C}$ ▷ Update pre-existing MSs in n to include cluster $\mathcal{C}$
      else ▷ The cluster is too large and must be split
        $\mathcal{Z} \leftarrow \mathcal{Z}$.append(K-meansClustering$(\mathcal{C}, \mathcal{V})$) ▷ Apply K-means clustering on $\mathcal{C}$ based on $\mathcal{V}$ with $K = 2$
end procedure

where the most critical MS placement request $\hat{r} \in \mathcal{C} \cup \mathcal{F}_s$ is the one that has minimum throughput drop relative to its solo run, and we considered the aggregate contentiousness vectors of the $(|\mathcal{F}_s| + |\mathcal{C}| - 1)$ competitors of $\hat{r}$ as:

$$V_{w \in \mathcal{F}_s \cup \mathcal{C} \setminus \{\hat{r}\}}^{(|\mathcal{F}_s|+|\mathcal{C}|-1)} = V_{w \in \mathcal{F}_s \setminus \{\hat{r}\}}^{(|\mathcal{F}_s|+|\mathcal{C}|-1)} + V_{w \in \mathcal{C}}^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}. \quad (13)$$

If the predicted throughput of $\hat{r}$ following (13) satisfies its SLA requirements, then we can safely place cluster $\mathcal{C}$ on server s. Otherwise, $\mathcal{C}$ will be provisionally placed on the next nearest server and the procedure is repeated until we find a server where we can place it without violating the SLA or we ran out of all the active servers. In the latter case, if $\mathcal{C} > 1$, we partition the cluster into two smaller ones using K-means clustering with K=2 and add these two new clusters to the end of the cluster queue. If $\mathcal{C} = 1$, we create a new server to place $\mathcal{C}$.

It is worth to note that the algorithm will tend to consolidate the MSs in the minimum number of servers, in line with the considered cost function in (2). The pseudocode of the proposed approach is provided in Alg. 1.

Notably, the overall complexity of the clustering phase is linear in the number of requested MSs; the placement phase is linear in the product of the number of active servers and clusters. Thus, the approach will scale well in the number of allocated MSs; it will *not* grow quadratically in the number of already running MSs. Rather, its worst-case complexity will be $|\widehat{\mathcal{S}}||\mathcal{R}|^2$, with $|\mathcal{R}|$ being the number of newly requested MS instances and $|\widehat{\mathcal{S}}|$ the number of currently, still partially empty, active servers.

V. Performance Evaluation

We evaluate iPlace against the optimum in a small-scale scenario, as well as against state-of-the-art alternatives in a larger-scale scenario. To do so, we consider snort and pktstat instance requests, arriving in batches of size $|\mathcal{R}|$, each with its associated SLA, i.e., the required throughput. Specifically, for each MS in $\mathcal{R}$, the latter is chosen from a uniform distribution between 70% and 100% of its solo performance.

Small-scale scenario. We first compare the results yielded by iPlace to the optimal solution. The latter is obtained through

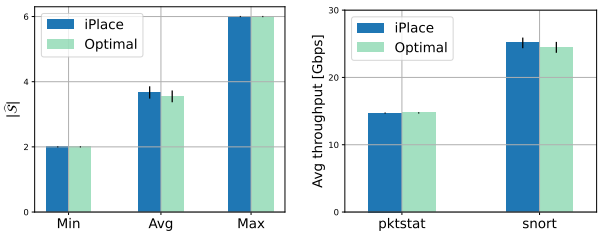

Fig. 4. iPlace vs. optimal: number of used servers (left); throughput (right).

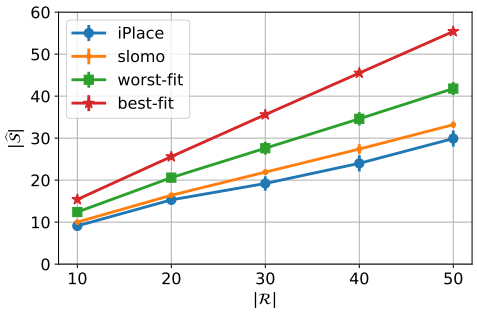

Fig. 5. iPlace vs. benchmarks: number of used servers as $|\mathcal{R}|$ varies.

brute-force search by generating all possible placement combinations and selecting the one that uses the minimum number of servers while satisfying the SLAs of all MSs. We consider that each request arriving at the orchestrator includes six MSs, and that two servers are already active: one running a pre-existing snort MS and the other a pre-existing pktstat MS. We repeat the experiment 100 times, each time varying the MSs' required throughput, and compute the confidence interval with a confidence level of 95%.

Fig. 4 shows that iPlace requires the same minimum and maximum number of used servers as the optimal, while the average is just slightly higher. Interestingly, Fig. 4 also shows that iPlace can provide better performance, as it reduces the interference among co-located MSs, by placing in the same server MSs competing for different types of resources.

Large-scale scenario. We now vary $|\mathcal{R}|$ from 10 to 50 in steps of 10, such that $\mathcal{R}$ contains an equal number of snort and pktstat instances. The initial number of active servers is fixed to six, each of them running one instance of either pktstat or snort, such that they satisfy their SLAs. As benchmarks, *worst-fit*, *best-fit*, and *slomo* [6] are considered. In worst-fit and best-fit approaches, the new requests are allocated to the server with, respectively, lowest and highest cumulative throughput of the pre-existing MSs running on it. If the request cannot be placed on the server with the lowest (highest) cumulative throughput, a new server is provisioned. In slomo, new requests are placed in a greedy incremental way that evaluates for every server whether the placement of a new request will lead to SLA violations using the same prediction model as in iPlace. If the request cannot be placed in any of the active servers without violating the SLAs, then a new server is launched.

Fig. 5 shows the number of servers used to place the MSs

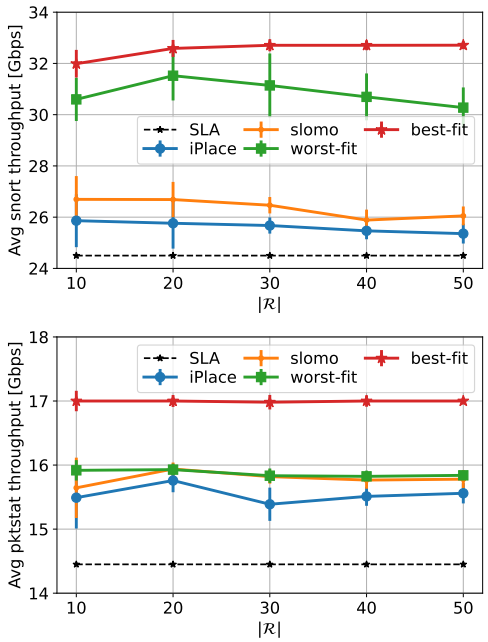

Fig. 6. iPlace vs. its benchmarks: average pktstat throughput (top) and average snort throughput (bottom).

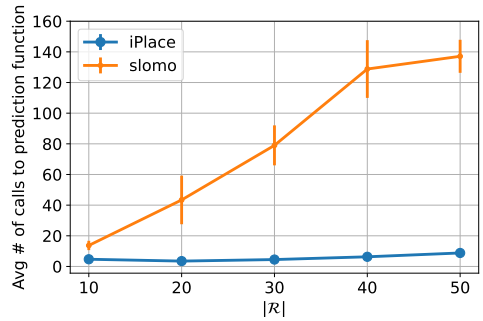

Fig. 7. iPlace vs. slomo: number of calls made to the prediction function as $|\mathcal{R}|$ varies.

as $|\mathcal{R}|$ varies. Compared to slomo, worst-fit and best-fit, iPlace utilizes 9.9%, 38.25% and 63.13% fewer number of servers, respectively, for $|\mathcal{R}| = 50$. Importantly, as depicted in Fig. 6, it can do so while satisfying the SLAs of pre-existing as well as newly placed MSs. Clearly, the throughput of pktstat and snort is higher under the alternatives we considered, but they have used a remarkably higher number of servers to place the MSs. Further, we recall that, as shown by experimental evidence, deployment of MS clusters is much faster than sequential deployment. Thus, iPlace can also greatly reduce the MSs deployment time with respect to all its alternatives.

Finally, to demonstrate the scalability of iPlace, we compare the number of calls made to the prediction function, which is the most CPU consuming function, for both iPlace and slomo. To better highlight the significant improvement, we have scaled down the MSs throughput requirements by a factor of 10 so that a higher number of clusters can be placed in the same server. Fig. 7 shows the results obtained using cProfile: for $|\mathcal{R}| = 50$, iPlace makes 93.58% fewer number of calls to the prediction function than slomo.

VI. Conclusions

We have addressed the placement of microservices (MSs) in data centers with the aim to minimize the number of used servers, while meeting the MSs performance requirements. In doing so, we experimentally characterized the gain of parallel versus sequential MSs deployment and the interference among MSs competing for the same resources, and formulated an optimization problem that aims at minimizing the number of used servers. Given the problem NP-hardness, we developed a low-complexity heuristic that aims at placing on the same server batches of MSs that compete for different resources. Our numerical results show that the proposed approach closely matches the optimum and, when compared to existing solutions, reduces the number of used servers by 10-63%, while proving to be highly scalable.

Future work will extend the experimental evaluation to diverse MSs as well as to MSs memory requirements, and it will further enhance the performance prediction model.

References

[1] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *J. of Network and Computer Applications*, 2016.

[2] 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 *ACM HotMiddlebox*, 2016.

[3] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *USENIX NSDI*, 2014.

[4] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *ACM OSDI*, 2016.

[5] S. Palkar and et al., "E2: a framework for nfv applications," in *ACM SOSP*, 2015.

[6] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *ACM SIGCOMM*, 2020.

[7] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *IEEE/ACM MICRO*, 2011.

[8] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packetprocessing platforms," in *ACM NSDI*, 2012.

[9] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling SLOs in network function virtualization," in *USENIX NSDI*, 2018.

[10] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense)," in *ACM CCS*, 2021.

[11] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the performance interference of co-located virtual network functions," in *IEEE INFOCOM*, 2018.

[12] Q. Zhang, F. Liu, and C. Zeng, "Adaptive interference-aware VNF placement for service-customized 5G network slices," in *IEEE INFOCOM*, 2019.

[13] Q. Zhang, F. Liu, and C. Zeng, "Online adaptive interference-aware VNF deployment and migration for 5G network slice," *IEEE/ACM Transactions on Networking*, 2021.

[14] S. Song, C. Lee, H. Cho, G. Lim, and J.-M. Chung, "Clustered virtualized network functions resource allocation based on context-aware grouping in 5G edge networks," *IEEE Transactions on Mobile Computing*, 2020.

[15] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on Information Theory*, 1975.