POLITECNICO DI TORINO
Repository ISTITUZIONALE

Reducing Microservices Interference and Deployment Time in Resource-constrained Cloud Systems

Original
Reducing Microservices Interference and Deployment Time in Resource-constrained Cloud Systems / Adeppady, M., Giaccone, P., Karl, H., Chiasserini, C.F.. - In: IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT. - ISSN 1932-4537. - STAMPA. - 20:3(2023), pp. 3135-3147. [10.1109/TNSM.2023.3235710]

Availability:
This version is available at: 11583/2974353 since: 2023-10-11T07:45:18Z

Publisher:
IEEE

Published
DOI:10.1109/TNSM.2023.3235710

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

©2023 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)

27 November 2026

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

Reducing Microservices Interference and Deployment Time in Resource-constrained Cloud Systems

Madhura Adeppady, *Student Member, IEEE,* Paolo Giaccone, *Senior Member, IEEE*, Holger Karl, *Member, IEEE*, Carla Fabiana Chiasserini, *Fellow, IEEE*

***Abstract*—In resource-constrained cloud systems, e.g., at the network edge or in private clouds, it is essential to deploy microservices (MSs) efficiently. Unlike most of the existing approaches, we tackle this issue by accounting for two important facts: (i) the interference that arises when MSs compete for the same resources and degrades their performance, and (ii) the MSs' deployment time. In particular, we first present some experiments highlighting the impact of interference on the throughput of MSs co-located in the same server, as well as the benefits of MSs' parallel deployment. 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, clustering MSs also allows us to exploit the benefit of parallel deployment, which greatly reduces the deployment time as compared to the sequential approach applied in prior art and by default in state-of-the-art orchestrators. Our numerical results show that iPlace closely matches the optimum and uses 21-92% fewer servers compared to alternative schemes while proving to be highly scalable. Further, by deploying MSs in parallel using Kubernetes, iPlace reduces the deployment time by 69% compared to state-of-the-art solutions.**

***Index Terms*—Microservices, Performance Interference, Edge Computing, Microservice Placement, Resource Contention**

I. Introduction

Cloud computing provides isolated environments to run heterogeneous applications of multiple tenants on the same underlying hardware using virtualization techniques. Customers can often request computing and memory resources from public cloud systems such as Amazon EC2 and Microsoft Azure to execute their applications. While such public systems have abundant resources for their users, edge computing and private cloud systems are often resource-constrained.

Applications deployed on either public or edge cloud systems are increasingly composed of multiple, simple components. For example, service function chains (SFCs) [1] comprise individual virtual network functions (VNFs) [2] or even other, simpler chains. Likewise, microservice chains are composed of individual microservices (MSs) [3] or other chains. Differences exist: VNFs might run close-to hardware, whereas 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 focus on an MS architecture running inside containers, however 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 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 [4]–[7], thus allowing to *consolidate* multiple components on the same hardware. Nonetheless, containers still compete for other hardware resources, predominantly memory subsystem resources [8]–[11]. 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 competing MSs is referred to as *interference* or *noisy neighbor problem* [12]–[14].

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 [15]. 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 MSs have to be consolidated dynamically on a limited set of resource-constrained servers, which is the typical operational condition in edge computing[1].

Recently, several research efforts have been made to address the above issue [11], [14], [16]–[18]. The proposed solutions leverage either resource partitioning schemes [11], [14], [19] or supply-demand models [16], [17] to quantify interference. However, none of these methods fully addresses interference completely, as they fail to consider all resources responsible for interference, according to [8]. A few notable approaches [8]–[10], [20] have built models to predict the

M. Adeppady, P. Giaccone, and C. F. Chiasserini are with the Electronics and Telecommunications Dept., Politecnico di Torino, Italy, and with CNIT, Parma, Italy. Email: {firstname.lastname@polito.it}. C. F. Chiasserini is also with IEIIT-CNR, Torino, Italy. H. Karl is with the Hasso Plattner Institute, University of Potsdam, Germany. Email: holger.karl@hpi.de

This work was supported partially through H2020 MSCA-ITN SEMANTIC (Grant No. 861165), and partially by the European Union's NextGenerationEU instrument, under the Italian National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.3, enlarged partnership "Telecommunications of the Future" (PE0000001), program "RESTART".

[1]Although we mainly focus on edge computing scenarios, our approach applies to other environments, e.g., data centers, as well.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

throughput of a target MS when co-located with other MSs and, using such prediction models, they have proposed *placement solutions* to decide where to execute each component. But even though the prediction models are accurate [8], these placement approaches may suffer from scalability issues.

Often, multiple requests for new services arrive at the orchestrator at the same time. Moreover, each request will typically be an SFC that comprises multiple individual microservices (or network functions). Also, these requests are usually not known in advance, prohibiting proactive deployment and requiring on-the-spot activation of the related services. In summary, the orchestrator usually needs to place *batches* of MSs on the available servers *as fast as* possible to ensure low deployment time.

Current orchestrators [21], [22] and MS placement algorithms [23], [24] focus on improving data locality or reducing contention for shared resources, however they pay little attention to reducing deployment time. Although there exist some few prior works that address deployment time by lowering the container startup latency [25]–[27], they have not paid any attention to the adverse effect of interference on the performance of MSs placed in the same server.

To provide this missing feature combination, we here propose iPlace, an Interference-aware Microservice Placement (IMSP) approach based on clustering, with clusters being deployed on different servers. iPlace specifically seeks to group services such that cluster members have minimal interference with each other. A cluster is then placed on that server with whose MSs it least interferes. In more detail, this is done via a recursive process where a large cluster of MSs is split into smaller ones until each cluster can be placed in a server without violating the SLA requirements of any of the involved MSs. Importantly, although clustering has been widely used in the literature [18], [28]–[30], existing approaches are not suitable to effectively cope with the problem we address. Indeed, the use of interference among and between cluster members has not yet been investigated, nor has the setup of existing MSs on servers been taken into account.

Additionally, a naive clustering approach would need many comparisons between cluster members and existing MSs, which is costly when using interference as a clustering metric and prevents a placement scheme to scale well with the number of MSs. Hence, we replace some of these comparisons by only looking at cluster representatives, resulting in an approximation of actual interference. Interference is then predicted using an ML approach. Specifically, we use the *contentiousness* metric [8], [9], which captures the pressure of an MS on shared resources, and build a prediction model inspired by [8], which takes into account the interference among MSs co-located in the same server. As shown in our performance evaluation, this prediction-based approximation yields excellent results at a fraction of the computational overhead.

Once clusters have been identified, it would now be straightforward to deploy the MSs inside a cluster onto their respective servers, one after the other, as typically performed by state-of-the-art, real-world orchestrators. However, this may result in very high deployment times. Instead, we show experimentally the advantage of deploying the MSs included in a cluster in parallel onto the selected server, compared to the default, sequential deployment strategy.

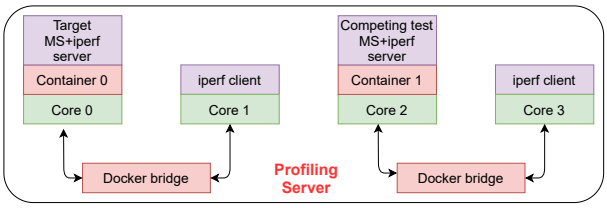

Fig. 1. Experimental testbed with target MS and competing MS.

To summarize, our main contributions are as follows:

1) Using our testbed and such real-world orchestrators as Kubernetes and Docker Swarm, we provide experimental evidence for the necessity to carefully cluster, place, and start up MSs in resource-limited environments. Indeed, our results show that interference among co-located MSs can degrade performance by up to 50%.
2) We formulate the Interference-aware Microservice Placement (IMSP) as an optimization problem that minimizes the number of servers needed to place the MSs at the network edge, while still meeting the MSs target performance.
3) Owing to the problem's NP-hardness, we develop iPlace, an interference-aware heuristic for cluster-based MS placement, which has cubic worst-case complexity. We remark that our work is the first to explore clustering to mitigate the effects of interference and to reduce the deployment time during MS placement.
4) Through extensive simulations and experiments with real systems, we show that the proposed interference-based clustering using prediction-information approximations effectively solves the IMSP and outperforms state-of-the-art alternatives. We also demonstrate that starting up MSs in parallel, according to the iPlace's clustering strategy, yields much shorter deployment time than sequential start up.

The rest of the paper is organized as follows. Sec. II demonstrates experimentally both how interference can affect the MSs performance and the reduction in deployment time obtained through a parallel start up of the MSs. In Sec. III, we describe the background necessary to understand our proposed work and interference prediction model. Then Sec. IV introduces the methodology we use for interference prediction, while Sec. V presents the system model and formulates IMSP as an optimization problem. Sec. VI describes our heuristic, iPlace, which is then evaluated in Sec. VII. Sec. VIII summarizes related work and highlights our novel contributions. Finally, Sec. IX concludes the paper.

II. Experimental Evidence and Work Motivation

We start by giving experimental evidence of how throughput degrades due to interference among competing MSs, despite a resource isolation setup in the server. We then show that MS deployment can be sped up significantly by working with *batches of MSs* instead of placing MSs sequentially.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

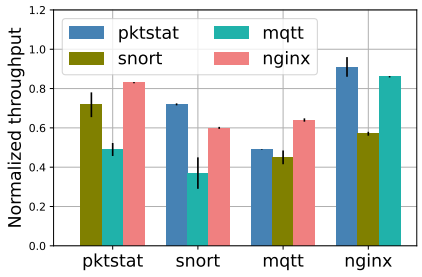

Fig. 2. Throughput of target MSs running with different competitors, normalized to the throughput obtained when the target MS runs solo.

A. Experimental interference assessment

To assess the throughput degradation experienced by MSs, we develop a testbed, as 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 Last Level Cache (LLC) cache shared across all the CPU cores, while individual cores have 1 MB L2 cache and 128 KB L1 cache, respectively. Each MS runs on a *Docker container*, pinned to a dedicated core using Docker runtime option `cpuset-cpus`.

Concerning the MSs, we consider both network services and application-level services:

- Pktstat[2]: a flow-monitoring tool that displays the real-time packet activity of a specific network interface. It reads the packet header to identify the flows and shows the data rate associated with each flow. The iperf3 tool is used for generating the traffic incoming to Pktstat.
- Snort[3]: a well-known intrusion prevention system that uses predefined rules to identify anomalous network activities. snort reads the packet header and payload of every incoming packet and compares them against the set of rules, in order to perform real-time intrusion detection. As before, iperf3 acts as a traffic generator for snort..
- Nginx[4]: We use it as a load balancer that forwards incoming requests to one of two web servers by modifying the requests' IP addresses. The web server MSs used in the experiments are developed using Python Flask. The httperf tool is used to generate input traffic to nginx.
- MQTT[5]: It is a lightweight message transmission protocol that relies on the publish-subscriber model. We have used Mosquittoas an MQTT broker to which publishers and subscribers are connected. MQTTLoader [31] is used on both publisher and subscriber sides to apply load on the MQTT broker MS and measure the throughput.

Our experimental settings pin each MS to a dedicated CPU core, however competing MSs still share memory resources, namely, last-level cache and memory bandwidth. In the first set of experiments, we investigate how the performance of MSs degrades due to interference by running them in pairs, despite the fact that, as mentioned, each of them is pinned to a dedicated CPU core. Normalized throughput is used as the metric for identifying the performance interference. It is defined as the ratio of throughput when running the target MS with the other MS to the target MS solo performance, i.e., the throughput of target MS when it is running alone on the server. Hence, lower values of normalized throughput indicate higher performance degradation. Note that computing and memory resources allocated to the MSs remain the same when running alone or in pairs.

As the per-MS load is small enough to be handled by a single server, in the absence of interference, one would expect to see a normalized throughput of 1. Instead, as shown in Fig. 2, all considered MSs suffer from performance degradation when run in pairs. The horizontal axis of Fig. 2 indicates the target MSs, while the MS with which they run in pair is the competitor MS. Interestingly, each pair of MSs experiences different normalized throughput. MQTT and snort as target MSs suffer from considerable performance degradation, regardless of their competitor MS, while nginx suffers from the competition with another MS least.

In the next set of experiments, we evaluate the throughput of a target MS in the presence of a competitor MS when varying the number of flows and per-flow offered load of the competitor MS. The results, shown in Fig. 3, highlight a throughput drop of 28.5 % for pktstat and of 22.5 % for snort, relative to their solo performance, both for 100 concurrent competing flows. Also, as the workload and the number of concurrent flows of the competitor MS increase, the throughput of the target MS degrades more.

In summary, our experiments reveal that performance interference is a relevant problem, *despite the isolation of CPU cycles*. Furthermore, throughput degradation of the target MS depends upon the traffic load and processing logic of the competing MSs. As the competitor's workload characteristics vary, competition for various hardware resources changes, degrading throughput differently. It is thus evident that interference plays a vital role in MS placement and that interference-oblivious co-location of MSs would seriously degrade throughput.

B. Experimenting sequential versus parallel MSs deployment

MS placement requests often arrive at the scheduler simultaneously, and these requests need to be deployed as fast as possible to provide efficient services. However, state-of-the-art MS scheduling algorithms deploy MSs only sequentially, and such is also the default deployment strategy in real-world orchestrators, with MS placement requests being kept in a queue following their order of arrival. According to prior MS algorithms, the scheduler acts upon these placement requests one after another and selects the best node to place the request depending on various scheduling policies. Such sequential deployment of the MSs has a high deployment latency, making it unacceptable for strictly latency-constrained edge services. This section briefly demonstrates the significant improvement in the deployment latency that can instead result from *parallel deployment* of MSs compared to that of *sequential deployment*.

The experiments have been conducted on a local Linux machine with 4 CPU cores and 16 GB of memory. To deploy the containerized MS placement requests, we used both

[2]https://linux.die.net/man/1/pktstat
[3]https://www.snort.org/
[4]https://nginx.com
[5]https://mqtt.org

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

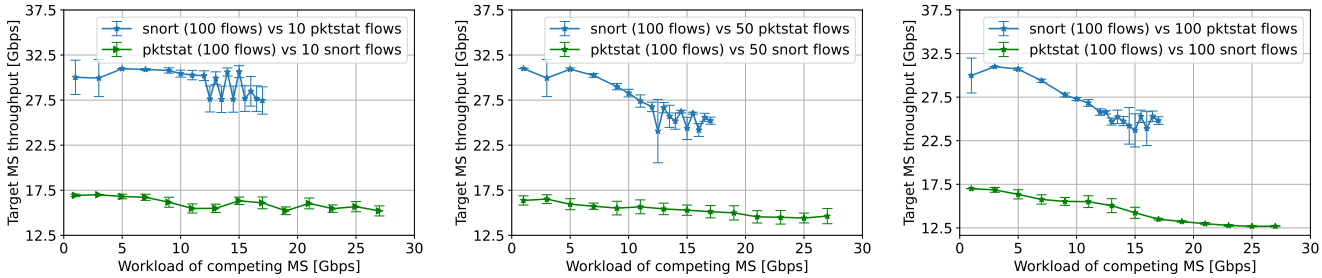

Fig. 3. Throughput of the target MS running with a competing MS, as the workload of the latter varies: 10 (left), 50 (center), 100 (right) concurrent flows, and traffic load equally distributed across the flows.

Kubernetes [21] and Docker Swarm [22]. Kubernetes is a well-known open-source cloud infrastructure tool that automatically deploys, scales, and manages containerized applications. Docker Swarm is a container orchestration tool native to the Docker platform, and it has been used to cross-check Kubernetes characteristics. To run Kubernetes locally, we used `minikube`, a single-node Kubernetes cluster. As native schedulers of Kubernetes and Docker Swarm orchestrators do not explicitly support parallel deployment of MSs, we have proceeded as set forth below.

For the experiments with Kubernetes, we leveraged the Volcano scheduler. Indeed, Volcano defines a new job, called Volcano job, that deploys a group of pods simultaneously, making it an excellent choice for parallel deployment. In our experiments, we created a Volcano job containing pods varying from 10 to 100 in steps of 10, each time measuring the per-pod deployment time.

For Docker swarm, we realized parallel deployment using the `--replicas` option. Docker swarm replicas aim at a stable set of running containers at any time. We created two batches for Docker swarm parallel deployment experiments and scaled the number of containers present in those batches using the `--replicas` option of Docker swarm from 5 to 50 in steps of 5, in order to mimic the same workload as in the Kubernetes scenario.

For the sequential deployment scenario using Kubernetes, we deployed sequentially Volcano jobs with a single container. In a sequential deployment scenario using Docker swarm, instead, we just used the default scheduler. We evaluated the average per-container start-up time for the sequential deployment and parallel deployment scenarios, using either Kubernetes or Docker swarm, as a function of the number of created containers. The experimental evidence in Fig. 4 demonstrates that parallel deployment (i.e., deploying MSs in batches) reduces the per-container start-up time by 81.25% and 80% compared to sequential deployment for Kubernetes and Docker swarm, respectively. This observation further motivated us to design an algorithm that schedules MS clusters instead of individual MSs, thereby reducing the incurred deployment time.

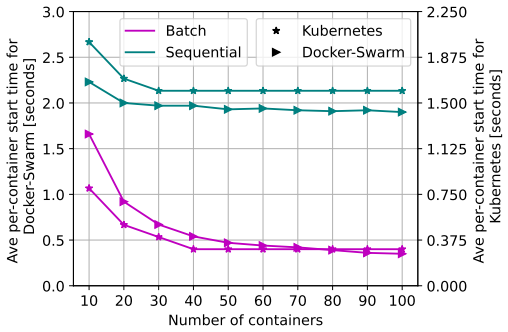

Fig. 4. Average per-container start-up time for batch and sequential deployment, using Kubernetes and Docker swarm.

III. Background

For the sake of concreteness and later evaluation, we describe here a throughput prediction model based on [8], 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 two phases: *offline profiling* and *online prediction*. 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 increasing this load, the increasing pressure of competing MS(s) on the shared resources can be measured. Thus, contentiousness profiling consists of determining a set of vectors, with each vector corresponding to a different pressure level of the synthetic competitor(s). Such vectors also comprise various system-level metrics, as detailed below. Then, sensitivity profiling builds on a regression model leveraging the target MS's throughput in the presence of varying synthetic contentiousness vectors.

In the *online prediction phase*, the sensitivity model predicts the target MS's throughput, given the contentiousness vector of any real competitor(s) as input.

A. Offline profiling phase

For each considered MS, the offline profiling phase characterizes its contentiousness and sensitivity. Here, we discuss further the methodology we adopted gtom [8].

The contentiousness vector of MS r is denoted by $\widehat{\mathbf{V}}_r^{(k)}(\mathbf{x})$, where k is the number of the competing MSs and $\mathbf{x}$ represents the competing workload and its specific configuration (e.g., traffic rate, number of flows, etc.). More in detail, $\mathbf{x}$ is a multi-dimensional vector $[x_1, \ldots, x_k]$ where the generic entry x_i,

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

$i = 1, \ldots, k$, is a tuple of the form $(\text{name}, \text{configuration})$: the name of the competing MS and its configuration. For instance, let us assume the contentiousness vector of snort consists of two metrics, LLC-misses and system-read. Also, suppose LLC-misses $= 100\,\text{s}^{-1}$ and system-read $= 20\,\text{MB/s}$, when snort runs with an instance of pktstat as a competitor processing 5 Gbps incoming traffic consisting of 10 concurrent flows. Then, in this case the contentiousness vector of snort is given by $\widehat{V}^{(1)}_{\text{snort}}(\mathbf{x}) = [100, 20]$ where $\mathbf{x} = [(\text{pktstat}, 5\,\text{Gbps}, 10\,\text{flows})]$. During the offline profiling phase for MS r, we vary $\mathbf{x}$ and k to measure $\widehat{\mathbf{V}}^{(k)}_r(\mathbf{x})$ as well as the corresponding observed throughput $P^{(k)}_r(\mathbf{x})$.

Concerning the sensitivity model of an MS r, this is defined as a function $M_r : \widehat{\mathbf{V}}^{(k)}_r(\mathbf{x}) \rightarrow P^{(k)}_r(\mathbf{x})$ [8]. It is obtained by training a regression model mapping the contentiousness vector $\widehat{\mathbf{V}}^{(k)}_r(\mathbf{x})$ into the observed throughput $P^{(k)}_r(\mathbf{x})$, for different $\mathbf{x}$ and k.

Finally, we leverage experimental results to compute the *representative* contentiousness vector $\mathbf{V}^{(k)}_r$ of MS r, obtained by *averaging over all* the observed contentiousness vectors with k competing MSs, with respect to the workload values $\mathbf{x}$. $\mathbf{V}^{(k)}_r$ is then fed as input to the sensitivity model in the online prediction phase. For brevity, we will refer to such a vector as the contentiousness vector hearafter.

B. Online prediction phase

It leverages the specific contentiousness vectors of the competitors and the sensitivity model of the target MS, in order 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* $\mathbf{V}^{(2)}_{r_b,r_c}$, characterizing the pressure on the resources jointly caused by MSs r_b and r_c, by combining their representative contentiousness vectors as follows: $\mathbf{V}^{(2)}_{r_b,r_c} = \mathbf{V}^{(2)}_{r_b} + \mathbf{V}^{(2)}_{r_c}$. In this expression, 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, so as to reflect the combined effect of the two competing MSs. We stress that this approach proved to be very accurate, as reported in [8]. Next, the throughput of r_a can be predicted as:

$$P_{r_a}(\{r_a, r_b, r_c\}) = M_{r_a}(\mathbf{V}^{(2)}_{r_b,r_c})\,. \tag{1}$$

Generalizing (1), the throughput of r_a when running on server s with set $\mathcal{Y}_s \setminus \{r_a\}$ of competing MSs is predicted as:

$$P_{r_a}(\mathcal{Y}_s) = M_{r_a}\Big(\sum_{r \in \mathcal{Y}_s \setminus \{r_a\}} \mathbf{V}^{(|\mathcal{Y}_s|-1)}_r\Big)\,. \tag{2}$$

IV. Measuring and Predicting Interference

We now show how to build a prediction model for estimating the throughput of competing MSs, which is required for an interference-aware MS placement solution. We stress, however, that our solution, introduced in Sec. V, can work with any other appropriate interference prediction model.

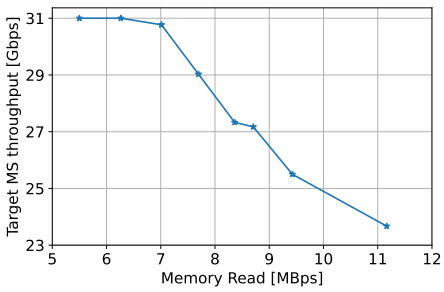

Fig. 5. Degradation of snort performance when competing with pkstat.

A. Selecting metrics for contentiousness vector

To determine 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, architecture-specific resource usage metrics.

We recall that each core of a modern system comprises its own L1 and L2 cache while all cores share LLC and memory bandwidth. Even running MSs on dedicated cores does not perfectly isolate their performance from each other as they will still compete for LLC and memory bandwidth. Furthermore, the Linux Kernel does not pin the Interrupt Service Routines (ISR) to the same core as the code causing these interrupts: code pinned to core 1 and causing an interrupt might still have its ISR executed on core 2. These ISRs then impact the performance of code pinned to core 2. As a consequence, containers running MSs with heavy network traffic can interfere with MSs co-located on the same machine. This motivated us to collect core-wise software interrupts using `mpstat` [32].

Intel PCM and `mpstat` output a wide range of metrics, but not all of them are relevant for the interference. Out of those, we selected components for the contentiousness vector that are highly correlated with the target MS's throughput, i.e., for which the Pearson correlation coefficient [33] is larger than 0.7. Table I lists such system-level metrics for the considered MSs. The READ and WRITE operations appear to be very relevant since the interference is due to the LLC and memory bandwidth contentions. To better characterize this effect, Fig. 5 shows, for snort competing with pktstat, the high performance degradation due to such contention. Furthermore, softirq are also relevant because of the sharing of interrupt handling among the cores, as discussed before.

B. Building the Interference Prediction Model

Recall that sensitivity models the performance of a target MS as a function of its contentiousness. The sensitivity is measured by building a model that takes the contentiousness vector as input and outputs the MS performance. Such a prediction model is a regressor, as input and output are continuous variables.

We train the sensitivity prediction model per MS by using its contentiousness vectors generated during the offline profiling

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

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

Snort		Pktstat		MQTT		Nginx	
Metric	CC	Metric	CC	Metric	CC	Metric	CC
Core-1 EXEC	0.96	System L2MPI	0.98	System READ	0.97	Core-0 IPC	0.91
System READ	0.90	Core-0 IPC	0.98	Core-0 IPC	0.98	System L3MISS	0.85
Core-1 IPC	0.89	Core-1 EXEC	0.96	Core-0 L3MPI	0.79	Core-4 L3MISS	0.83
System WRITE	0.88	Core-2 L2MISS	0.95	Core-1 IPC	0.75	System READ	0.81
Core-2 L3MISS	0.85	System L2MISS	0.95	Core-4 softirqs	0.74	Core-0 L2MPI	0.80
Core-1 L2MISS	0.83	Core-0 softirqs	0.94	Core-1 L3MISS	0.73	System L3MPI	0.74

phase. Specifically, we use a Gradient Boosting Regressor model as sensitivity is a non-linear, non-continuous function of the contentiousness vectors. For each trained sensitivity model, we measured the prediction error using the absolute mean prediction error, which is 2.5%, 3.6%, 4.5%, and 5.2% for snort, pktstat, MQTT, and nginx, respectively. Thus, despite the differences between our experimental setup and the one used in [8], we obtain an accurate interference prediction that is coherent with that presented in [8].

V. SYSTEM MODEL AND PROBLEM FORMULATION

We now describe our system model and formalize the IMSP problem to optimally place MSs at the network edge.

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, with $\mathcal{F}_s = \emptyset$ 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 specified by the SLA, for an MS $r \in \mathcal{R}$.

We assume that the data center has ample symmetric bandwidth and, hence, 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 $r' \in \mathcal{R}$. Furthermore, each server $s \in \mathcal{S}$ has limited CPU and memory budget denoted by $\hat{\tau}_s$ and $\hat{\mu}_s$, respectively, while there is ample availability of other resource types. 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 still met. 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 model introduced in Sec. IV, 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., with both pre-existing and newly placed MSs. Key parameters of the system model, along with their notation, are listed in Table II.

TABLE II
LIST OF SYMBOLS USED IN THE PROBLEM FORMULATION

Symbol	Description
	Parameters for Servers
$\mathcal{S}$	Set of available servers
$\hat{\mu}_s$	Memory capacity of server s
$\hat{\tau}_s$	Computation capacity of server s
$\mathcal{F}_s$	Set of MSs already placed on server s
	Parameters for MS
μ_r	Required memory of MS r
τ_r	Required computation of MS r
t_r	minimum throughput for MS r based on its SLA
$\mathcal{R}$	Set of MSs to deploy in a new batch of requests, $\mathcal{R} \cap \mathcal{F}_s = \emptyset$, $\forall s$
$P_r(\mathcal{Y}_s)$	Predicted throughput of MS r on server s when competing with MSs in $\mathcal{F}_s \cup \mathcal{Y}_s \setminus \{r\}$, with $\mathcal{Y}_s$ being the set of requests placed on server s
	Variables
$y_{r,s}$	Binary decision variable, indicating whether MS r is running on server s
n_s	Binary decision variable, indicating whether server s is actively running at least one MS

B. The IMSP problem formulation

Given the set of requested MSs, the goal is to minimize the number of edge servers used to place the MSs, i.e.,

$$\min \quad \sum_{s \in \mathcal{S}} n_s \tag{3}$$

subject to system and SLA constraints:

$$\sum_{s \in \mathcal{S}} y_{r,s} = 1 \quad \forall r \in \mathcal{R} \tag{4}$$

$$n_s \leq \sum_{r \in \mathcal{R}} y_{r,s} + |\mathcal{F}_s| \quad \forall s \in \mathcal{S} \tag{5}$$

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

$$\sum_{r \in \mathcal{Y}_s} y_{r,s} \cdot \tau_r + \sum_{r \in \mathcal{F}_s} \tau_r \leq n_s \cdot \hat{\tau}_s \quad \forall s \in \mathcal{S} \tag{7}$$

$$P_r(\mathcal{Y}_s \cup \mathcal{F}_s) \geq t_r \quad \forall s \in \mathcal{S}, r \in \mathcal{R} \cup \mathcal{F}_s \tag{8}$$

$$n_s \in \{0,1\} \quad \forall s \in \mathcal{S}, \tag{9}$$

$$y_{r,s} \in \{0,1\} \quad \forall s \in \mathcal{S}, r \in \mathcal{R}\,. \tag{10}$$

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

Eq. (4) specifies that a new MS must be placed on exactly one server. Eq. (5) ensures that a server is turned off if no MS is assigned to it. Equations. (6)–(7) mandate that the memory and computing resource requirements 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. (8) leverages (2) and imposes that the predicted throughput for any new and pre-existing MS must satisfy the minimum required throughput as specified in the corresponding SLA, thus the mutual interference across all MSs is acceptable.

About the problem complexity, the following result holds.

Theorem 1. *The IMSP problem in* (3)*, subject to constraints* (4)–(10)*, is NP-hard.*

Proof. We consider a simplified offline version of our IMSP problem where (i) the interference among MSs is neglected, (ii) each server has infinity memory, (iii) no pre-existing MSs are running, i.e., $\mathcal{F}_s = \emptyset$, (iv) no throughput constraints exist, i.e., $t_r = 0$, $\forall r \in \mathcal{R}$. In this case, each of the requested MSs in $\mathcal{R}$ has only a computing requirement and overall the server capacity cannot be exceeded.

Next, consider the bin packing problem, which is NP-hard [34], where we are given a set of items and bins. Observe that each item can be mapped onto an MS request with the required computation equal to the item size, and each bin can be mapped onto a server with a computation capacity equal to the bin size. It follows that any instance of the bin packing problem can be reduced in polynomial time to the above simplified version of the IMSP, thus proving the thesis. □

In light of the complexity of the IMSP problem, we introduce below a heuristic, which has cubic wort-case complexity and, as shown in Sec. VII closely matches the optimum.

VI. iPlace: The Interference-Aware MS Placement

This section first presents in Sec. VI-A our heuristics, along with an example of how the iPlace algorithmic framework works. Then it discusses the complexity of iPlace in Sec. VI-B.

A. Algorithmic framework

The key idea of our MS placement algorithm, iPlace, is to partition the set of new MSs into clusters in which the *MSs contend for different types* of resources. Importantly, clustering is motivated by the much smaller time for parallel deployment compared to sequential deployment, as showed experimentally in Sec. II-B. Furthermore, while clustering MSs, we reduce the mutual interference of MSs in the same cluster, which allows them to better coexist on the same server.

As depicted in Fig. 6 and presented in Alg. 1, the iPlace algorithm works in two phases:

- the *clustering phase*, which clusters the new MS placement requests based on their contentiousness, and
- the *placement phase*, which selects the server where to place each created cluster, accounting for the MSs that are already hosted in the active servers.

In the clustering phase, iPlace leverages 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*. More formally, we define the distance between two MSs $r_i, r_j \in \mathcal{R}$ as:

$$d(r_i, r_j) = \|\mathbf{V}^{(1)}_{r_i} - \mathbf{V}^{(1)}_{r_j}\|_2^{-1}. \tag{11}$$

The MSs in $\mathcal{R}$ are initially clustered using the mean-shift clustering [35] technique, which automatically discovers the number of clusters and the MSs to be included therein based on the chosen distance metric.

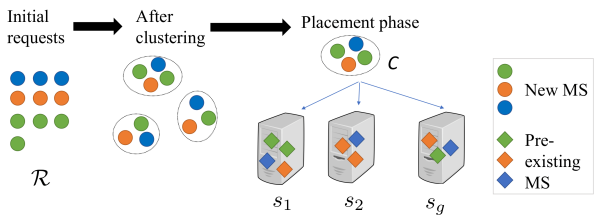

Fig. 6. Example of the clustering and placement phases for a batch of requests: colors indicate the different types of resources an MS competes for; shapes distinguish old from new MSs).

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. Servers with enough computing and memory resources are considered as eligible servers for placing a given cluster $\mathcal{C}$. We denote the set of eligible servers as $\mathcal{A}$. For any eligible server $a \in \mathcal{A}$, we compute the distance between cluster $\mathcal{C}$ and server a as presented in Alg. 2. More specifically, we define the distance between server a running pre-existing MSs in $\mathcal{F}_a$ and cluster $\mathcal{C}$ as:

$$\left\|\mathbf{V}^{(|\mathcal{F}_a|+|\mathcal{C}|-1)}_{w\in\mathcal{F}_a} - \mathbf{V}^{(|\mathcal{F}_a|+|\mathcal{C}|-1)}_{w\in\mathcal{C}}\right\|_2^{-1} \tag{12}$$

where $\mathbf{V}^{(|\mathcal{F}_a|+|\mathcal{C}|-1)}_{w\in\mathcal{F}_a}$ is the aggregate contentiousness vector of all MSs placed in a and $\mathbf{V}^{(|\mathcal{F}_a|+|\mathcal{C}|-1)}_{w\in\mathcal{C}}$ is the one for $\mathcal{C}$. All the eligible servers in $\mathcal{A}$ are sorted in the increasing order of their distance with cluster $\mathcal{C}$ according to (12).

To assess the impact of interference of the new MSs in $\mathcal{C}$ on the nearest server $s \in \mathcal{A}$ consisting of $\mathcal{F}_s$ pre-existing MSs, we use the `PredictSLAViolations` function described in Alg 3. `PredictSLAViolations` returns `True` if placing cluster $\mathcal{C}$ on the nearest server s violates the minimum required throughput specified in the SLA of the most critical MS placement request $\hat{r}$. The most critical request $\hat{r}$ is the one exhibiting the minimum difference between its solo-run throughput and the minimum required throughput. In particular, the function uses the prediction model for $\hat{r} \in \mathcal{C} \cup \mathcal{F}_s$ as follows:

$$P_{\hat{r}}(\mathcal{C} \cup \mathcal{F}_s) = M_{\hat{r}}(\mathbf{V}^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w\in\mathcal{F}_s\cup\mathcal{C}\setminus\{\hat{r}\}}) \tag{13}$$

where the aggregate contentiousness vectors of the $(|\mathcal{F}_s| + |\mathcal{C}| - 1)$ competitors of $\hat{r}$ is given by:

$$\mathbf{V}^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w\in\mathcal{F}_s\cup\mathcal{C}\setminus\{\hat{r}\}} = \mathbf{V}^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w\in\mathcal{F}_s\setminus\{\hat{r}\}} + \mathbf{V}^{(|\mathcal{F}_s|+|\mathcal{C}|-1)}_{w\in\mathcal{C}}. \tag{14}$$

If the predicted throughput of $\hat{r}$ following (14) 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

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

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, $\mathcal{V}$: set of contentiousness vectors
  $\mathcal{Z} \leftarrow$ `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() ▷ Move 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`$(\hat{r}, \mathcal{C}, \mathcal{F}_s, \mathcal{V}, \mathcal{M})$ = 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
        $\widehat{\mathcal{S}} \leftarrow \widehat{\mathcal{S}} \cup \{n\}$ ▷ 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

Algorithm 2 Compute cluster-server distance

procedure `Distance`$(\mathcal{C}, \mathcal{F}_a, \mathcal{V})$
  $c \leftarrow |\mathcal{C}| + |\mathcal{F}_a| - 1$ ▷ Every MS will compete with c number of competitors
  $\mathbf{V}_c \leftarrow \sum_{r \in \mathcal{C}} \mathbf{V}_r^{(c)}$ ▷ Find the aggregate contentiousness vector of the cluster $\mathcal{C}$
  $\mathbf{V}_a \leftarrow \sum_{r \in \mathcal{F}_a} \mathbf{V}_r^{(c)}$ ▷ Find the aggregate contentiousness vector of the server a
  return $\|\mathbf{V}_c - \mathbf{V}_a\|_2^{-1}$ ▷ Distance evaluated as in (12)
end procedure

Algorithm 3 Predict SLA violations

procedure `PredictSLAViolations`$(\hat{r}, \mathcal{C}, \mathcal{F}_s, \mathcal{V}, \mathcal{M})$
  c $\leftarrow |\mathcal{C}| + |\mathcal{F}_s| - 1$ ▷ Compute number of competing MSs
  $\mathbf{V}_{\text{aggr}} \leftarrow \sum_{\forall i \in \mathcal{F}_s \cup \mathcal{C}, i \neq \hat{r}} \mathbf{V}_i^{(c)}$ ▷ Aggregate cont. vector of $\hat{r}$'s competitors
  $p_{\hat{r}} \leftarrow \mathrm{M}_{\hat{r}}(\mathbf{V}_{\text{aggr}})$ ▷ Predict the performance
  if $p_{\hat{r}} < t_{\hat{r}}$ **then** ▷ Compare with throughput according to SLA
    return True ▷ At least one SLA violation is experienced
  return False ▷ No SLA violation is experienced
end procedure

server and the procedure is repeated until a server is found that can host $\mathcal{C}$ without violating the SLA of any involved MS, or all the active servers have been examined. In the latter case, if $\mathcal{C} > 1$, we partition the cluster into two smaller ones using K-means clustering with K=2, and we add such two new clusters to the end of the cluster queue. If $\mathcal{C} = 1$, we open a new server to place $\mathcal{C}$.

It is worth noting that the algorithm will tend to consolidate the MSs in the minimum number of servers, in line with the considered objective function in (3). In order to further clarify how iPlace work, we provide the example below.

Example 1 (MS placement using iPlace)**.** *Consider a batch request $\mathcal{R}$ consisting of two MS placement requests arriving at the orchestrator, $\mathcal{R} = \{snort, pktstat\}$. Further, consider that there is a single active server running one instance of nginx, represented as $\widehat{\mathcal{S}} = \{\mathcal{S}_0\}$ where $\mathcal{S}_0 = \{nginx\}$. Each MS in the considered system is associated with an SLA that specifies minimum required throughput, and let us say that $t_{snort} = 1\,\text{Gbps}$, $t_{pktstat} = 2\,\text{Gbps}$, and $t_{nginx} = 4\,\text{Gbps}$. Without loss of generality, let us assume that the contentiousness vector of the MSs comprises of two metrics; LLC-miss and system-read. For simplicity, we have considered the contentiousness vectors of the MSs as shown in Table III consisting of arbitrary values for the considered metrics.*

TABLE III
A SIMPLE EXAMPLE OF CONTENTIOUSNESS VECTORS OF MSS

	LLC-misses [s^{-1}]	system-read [MB/s]
snort	120	40
pktstat	10	200
nginx	50	40

Initially, iPlace creates clusters of the MSs based on their contentiousness. We can represent the created clusters as $\mathcal{Z} = \{\mathcal{C}_0\}$, with $\mathcal{C}_0 = \{snort, pktstat\}$. It is obvious from Tab. III that snort and pktstat belong to the same cluster because they compete for different kinds of resources. During the placement phase, we check whether the placement of cluster $\mathcal{C}_0$ on server $\mathcal{S}_0$, already running nginx, violates the minimum required throughput in the SLA of the most critical MS. Assume that the most critical MS here is pktstat, then we calculate the aggregate contentiousness vector of snort and nginx and give it as input to prediction model $M_{pktstat}$. The aggregate contentiousness vector is calculated by combining the contentiousness vectors of snort and pktstat using appropriate linear operators; sum for system-read and average for LLC-miss. Let the output of $M_{pktstat}$ be 2.5 Gbps, which is greater than $t_{pktstat}$; then $\mathcal{C}_0$ can be placed on $\mathcal{S}_0$ without violating the SLAs of the MSs present in $\mathcal{C}_0$ and $\mathcal{S}_0$.

B. iPlace complexity

The overall worst-case complexity of iPlace can be determined by inspecting Alg. 1–Alg. 3.

Alg. 1 performs `MeanShiftClustering`, which has complexity $O(|\mathcal{R}|^2)$, once, and `meansClustering`, which has complexity $O(|\mathcal{R}|)$, $\log_2 |\mathcal{R}|$ times, where we recall that $|\mathcal{R}|$ is the number of newly requested MS instances. Furthermore, iPlace executes the `distance` and `predictSLAViolations` functions for every created cluster and for every, still partially empty, active server, with the number of such servers being denoted with $|\widehat{\mathcal{S}}|$. The worst-case complexity of the `distance` function is equal to $O(|\mathcal{R}|)$, as it might need to loop over the contentiousness vectors of the number of MSs present in the cluster and the server. Similarly, the `predictSLAViolations` function has a worst-case complexity of $O(|\mathcal{R}|)$, as in the worst-case, we might need to calculate the aggregate contentiousness vector using the contentiousness vectors of $|\mathcal{R}|-1$ competitors of the target MS. Thus, iPlace's worst-case is dominated by that of

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

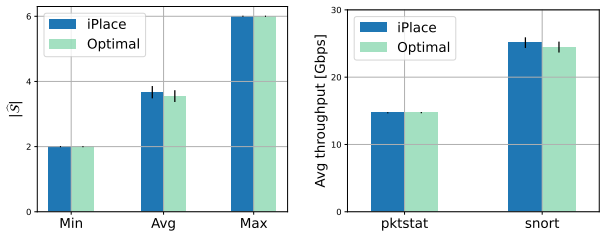

Fig. 7. iPlace vs. optimal: number of used servers (left); throughput (right). The thin bars represent 95%-level confidence intervals.

the `distance` and `predictSLAViolations` functions, which is $O(|\widehat{\mathcal{S}}||\mathcal{R}|^2)$.

We therefore remark that iPlace will scale well in the number of allocated MSs, and, importantly, it will *not* grow quadratically in the number of already running MSs.

VII. Performance Evaluation

In order to evaluate iPlace against the optimum, we first investigate a small-scale, stationary scenario, where the IMSP problem can be solved by brute force. Then we move on to a larger-scale, dynamic scenario that we use to compare iPlace against state-of-the-art alternatives. In both scenarios, we consider MS placement requests, arriving in batches of size $|\mathcal{R}|$, each with its associated minimum required throughput. Specifically, for each MS in $\mathcal{R}$, the associated minimum required throughput is chosen from a uniform distribution between 50% and 70% of the MS solo performance.

Small-scale, stationary scenario. We first compare the results yielded by iPlace to the optimal solution. The latter is obtained through brute-force search by generating all possible placement combinations and selecting the one that uses the minimum number of servers while satisfying the SLA 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 pkstat MS. We repeat the experiment 100 times, each time varying the MSs' required throughput, and we compute the confidence interval with a confidence level of 95%.

Fig. 7 shows that iPlace requires the same minimum and maximum number of servers as the optimum, while the average is just slightly higher. Interestingly, iPlace can provide better[6] performance, thanks to the higher number of activated servers, mitigating the interference among co-located MSs.

Large-scale, dynamic scenario. We used "short-lived" MSs to capture a dynamic scenario, in which the container is activated when a request arrives and is deactivated after serving the request. We assume that the MSs do not maintain state in their containers, and the startup time is negligible compared to the service execution time; this is quite typical for the "serverless" computing paradigm. Note that, if MSs are instead "long-lived", iPlace is still perfectly applicable, but since start-ups happen much less frequently, this scenario is much less of a challenge, thus we have not considered it in our evaluation.

[6]Since the throughput performance is not the goal of the optimization, it is indeed possible that iPlace outperforms the optimum in this respect.

We now model MS arrivals according to a Poisson process and the service duration according to a negative exponential distribution. As it is common that the requested services are chains of multiple functions, we consider batch arrivals and departures where the batch size $|\mathcal{R}|$ follows a geometric distribution with mean varying between 5 to 30 MSs. Each MS in the batch is picked randomly from a pool of profiled MSs and their associated minimum required throughput is chosen from a uniform distribution between 50%-70% of their solo performance. In the considered model, the arrival rate of each MS batch and the expected service duration of all the MSs within the same batch are denoted with λ and $1/\mu$, respectively. Then let $\rho = \lambda/\mu$ be the utilization factor. Whenever a batch arrival occurs, iPlace processes the newly arrived MS requests and places the corresponding MSs in the available servers, avoiding violating the SLAs of new and pre-existing MSs. In case of any SLA violation, a new server is provisioned. When an MS batch's service time expires, all of the MSs in that batch are removed and a server becomes inactive if it is not running any MS.

We fix $\lambda = 3$ arrivals per time unit and the system is evaluated in and with $\rho = 0.2$ (low-load) and $\rho = 0.9$ (high-load). Each simulation begins by setting the initial number of servers to 0, and it lasts 1000 time units. At every event, i.e., batch arrival or departure, we calculate the number of active servers in the system and the number of active MSs.

We evaluate the average number of servers, average MSs per server, and average active MSs, as $\mathcal{R}$ varies, for both high-load and low-load scenarios. While doing so, we compare the performance of iPlace to the following alternative solutions, which consider sequentially all the MSs within each batch and deploy each of them individually:

slomo [8]: A new MS is placed in the first available server, according to a fixed order, that can host the MS without any SLA violation. To detect SLA violations, slomo uses the same prediction model as iPlace.

bestfit: A new MS is placed in the server with the highest cumulative throughput of the pre-existing MSs.

worstfit: A new MS is placed in the server with the lowest cumulative throughput of the pre-existing MSs.

In all the three benchmark algorithms, if a new MS request cannot be placed in any of the active servers, then a new server is provisioned. Thus, the MSs of the same batch can be eventually placed on different servers. Also, notice that, due to the sequential approach, the overall deployment time of these three variants will be larger than iPlace, as shown by our experimental results in Sec. II-B.

The two plots of Fig. 8 present the average number of servers required by iPlace and its benchmarks to deploy a batch of MSs in low-load and high-load scenarios, respectively. For an average batch size of 30 MSs, in a high-load scenario, iPlace utilizes 21%, 92% and 72% fewer servers than slomo, bestfit, and worstfit approaches, respectively. In a low-load scenario, iPlace requires 25%, 89% and 68% fewer servers for placing the MS requests compared to slomo, bestfit, and worstfit approaches, respectively. In a nutshell, in all the cases iPlace can reduce substantially the number of used servers, with respect to its alternatives.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

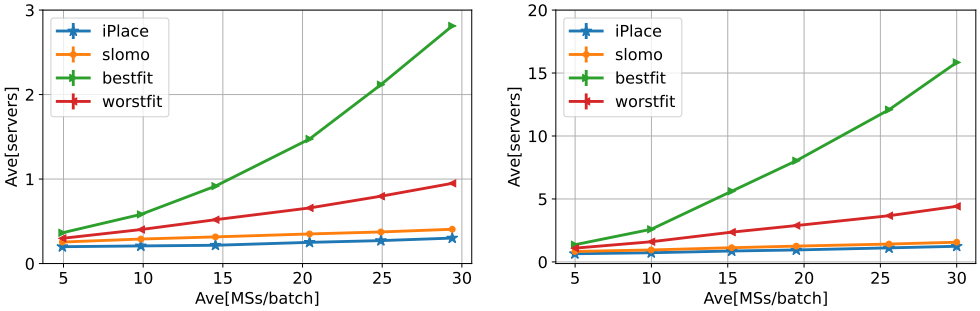

Fig. 8. iPlace vs. its benchmarks: average number of servers under low load (left) and high load (right).

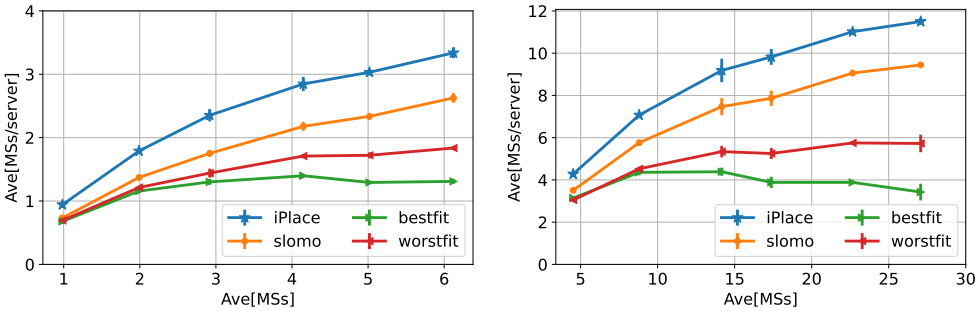

Fig. 9. iPlace vs. its benchmarks: average number of MSs per server under low load (left) and high load (right).

Furthermore, Fig. 9 depicts the average number of MSs per server as a function of the average number of MSs present in the system. The plots demonstrate that iPlace and slomo approaches consolidate a larger number of MSs on the same server than bestfit and worstfit approaches, without violating any SLA. Moreover, iPlace outperforms slomo by accommodating between 31% and 41% more MSs on the same server, depending on the scenario settings.

Deployment time. We now evaluate experimentally the deployment time in a realistic scenario according to the following methodology. We simulate the MS batch arrivals and compute the placement according to a given policy. For each batch, iPlace computes a sequence of MS clusters and the corresponding servers. Then we run the Volcano scheduler of Kubernetes to place each cluster individually by deploying in parallel all the MSs within each cluster on the same server. On the contrary, slomo placement algorithm computes a sequence of MSs and a server associated to each individual MS. Thus, in this cases we run the Volcano scheduler of Kubernetes to deploy sequentially each MS on the desired server.

In each experiment we consider the arrival of 10 batches, with an average number of MSs per batch varying between 5 and 30. Each experiment is repeated 5 times to estimate the average deployment time (the results show sample mean and confidence intervals for a 95 % confidence level).

Fig. 10 presents the average per-container deployment time for iPlace and slomo, as a function of the number of MSs per batch. Remarkably, one can observe that iPlace reduces the deployment time by 69% as compared to slomo. Thus, in addition to reducing the number of used servers by packing more MSs on the same server, iPlace also benefits greatly from batch deployment when compared with state-of-the-art solutions and achieves a lower deployment time.

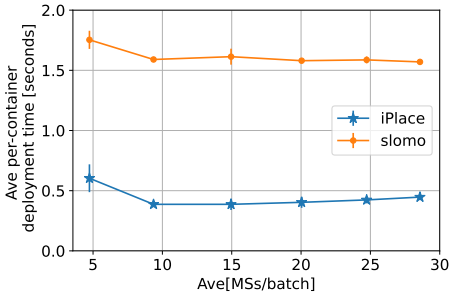

Fig. 10. Average per-container deployment time for iPlace and slomo.

VIII. Related Work

The problem of MS, or, alternatively, VNF, placement has been extensively studied in the literature with multiple scopes and objectives. Examples include works that have aimed at properly accounting for user mobility, or at minimizing the delay or the number of used servers. However, most of the existing studies have not considered performance interference, which is instead vital when deciding on which edge server an MS/VNF has to be placed.

Among the few prior approaches that propose interference-aware MS/VNF placement, [16], [17], [36] leverage a supply-demand model to quantify interference. In particular, [16], [17] propose an Adaptive Interference Aware (AIA) VNF placement to automatically place VNFs maximizing the total throughput of the accepted placement requests. AIA quantifies interference experienced by the consolidated VNFs through a demand-supply model by profiling each VNF with the aim to measure CPU and memory utilization and meet diverse service requirements. Unlike iPlace, the proposed supply-

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

demand model in AIA does not consider other sources of contention such as LLC, memory bandwidths and software interrupts. Moreover, AIA proposes a sequential deployment of the placement requests to maximize the total throughput of the accepted requests, which incurs a large deployment time as discussed in Sec. II-B and shown in Fig. 10.

Many efforts have addressed performance interference using partitioning techniques. ResQ [11] proposes a contention-aware VNF scheduler, based on the observation that LLC and DDIO[7] packet buffers are the critical factors for performance degradation of consolidated VNFs. This complements our approach, although we did not experience any effect of DDIO since our experiments are always based on virtual interfaces. ResQ combines two techniques: a profiling scheme to determine the LLC partition size required to achieve a target SLA, and a Cache Allocation Technology, which is a hardware-based method method to partition the LLC among the competing VNFs. The ResQ scheduler uses both an online greedy heuristic and offline mixed-integer linear programming to find an optimal schedule for the arriving requests based on profiling information. ResQ partitions the LLC among competing VNFs, but it does not isolate other resources degrading throughput, making it an incomplete solution to IMSP. Also, unlike iPlace, ResQ works sequentially on each MS individually and is not able to exploit parallel deployment.

Other works [9], [10] model the contention-induced performance drop suffered by a packet processing unit as a function of competitors' aggregate Cache Access Rate (CAR). Their main observation is that some VNFs are sensitive to co-located VNFs' behavior, and that VNFs can also be "aggressive" in causing such sensitive VNFs to suffer. This is coherent with our experimental evidence presented in Sec. II-A. Their proposed solution is to co-locate such sensitive VNFs with non-aggressive VNFs. They have considered a single metric to quantify the interference and classify the VNFs. The modern memory architecture is complex; thus, measuring interference based on a single metric may be inefficient. In contrast, the iPlace clustering scheme tends to co-locate the MSs by considering several system-level metrics to quantify the interference.

DeepDive [18] proposes a solution to transparently identify and manage performance interference between co-located virtual machines using a warning system based on low-level metrics and an interference analyzer for determining the culprit resource. Unlike iPlace, which takes interference-aware decisions while deploying the MSs in the first place, DeepDive identifies the interference after the initial placement of the MS and, when necessary, migrates an MS to a different physical machine, which incurs additional migration cost.

The most relevant study to ours is however [8], which presents an interference-aware VNF placement solution. We adopt the same machine learning approach to estimate the interference of slomo, as discussed in Sec. IV-A. Even if the interference model is the same, the way it is adopted to solve the IMSP is different. Notably, the cluster-based approach in iPlace is compatible with any method to estimate the interference. To solve the placement problem, slomo proposes a greedy incremental approach to minimize the number of active servers, according to which it verifies whether adding a new VNF request to a server leads to SLA violations for each scheduling request. If there is no feasible solution, slomo provisions a new server to place the request. The greedy incremental approach in slomo is inefficient as it involves tentatively placing new requests in each active server and checking each VNF individually for SLA violations. Instead, iPlace introduces a two-stage method consisting of clustering the MSs that do *not* contend for the same resources and placing these clusters on separate servers to minimize interference. Moreover, as evident in the experimental results, the batch approach exploited by iPlace dramatically reduces the deployment time compared to slomo.

Finally, our preliminary work in [37] explores interference-aware MS placement using clustering, which has shown improvement of 10-60% in the number of used servers against various state-of-the-art solutions for a batch size of 50 consisting of only two MS instances. In our preliminary work, we considered only network services, i.e., pktstat and snort, while building the prediction model and evaluating the performance of iPlace. In this work, we have extended the set of MSs to include both network and application-level services, i.e., MQTT and nginx. This extension allowed us to better represent the real-world scenario where network and application-level services coexist. Furthermore, we highlight the benefit of parallel MS deployment over a sequential approach in reducing the deployment time using real-world orchestrators (Kubernetes and Docker swarm). This important observation motivated us to design iPlace so that it deploys clusters of MSs rather than a single MS each time. We further extended the simulation of our prior work to consider a large-scale dynamic scenario in which MS batch arrivals and their service time are modeled using a Poisson process and negative exponential distribution, respectively. The simulated extension demonstrates that iPlace reduces the number of used servers both in high and low-load scenarios. Furthermore, the number of consolidated MSs per server is higher in iPlace than in considered benchmarks.

IX. Conclusions

We addressed resource-constrained cloud systems, a typical operational condition in edge computing and private clouds. In this context, we designed a microservice placement algorithm that minimizes the use of computing resources while still meeting the performance requirements of MSs. In doing so, we showed experimentally the gain of parallel versus sequential MSs deployment, and the substantial impact that interference among MSs co-located in the same server can have on the MSs performance. We formulated the interference-aware MSs placement as an optimization problem that aims at minimizing the number of used servers. Given the problem's NP-hardness, we developed a low-complexity heuristic that places batches of MSs that compete for different resources on the same server, thus also allowing for parallel MSs deployment.

Our numerical results show that the proposed approach closely matches the optimum and, when compared to state-

[7]Intel Data Direct I/O Technology is a proprietary solution to optimize data plane forwarding performance in physical network cards.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3235710

of-the-art solutions, it reduces the number of used servers by 21-92%, while proving to be highly scalable. Furthermore, by exploiting parallel deployment, iPlace can reduce the deployment time by 69%.

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] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-art and research challenges," *IEEE Comm. Surveys & Tutorials*, vol. 18, no. 1, 2017.

[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today and Tomorrow*. Springer International Publishing, 2017.

[4] 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.

[5] 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.

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

[7] S. Palkar and et al., "E2: A Framework for NFV Applications," in *ACM SOSP*, 2015.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] S. Akundi, S. Prabhu, N. U. B.K., and S. C. Mondal, "Suppressing noisy neighbours in 5g networks: An end-to-end nfv-based framework to detect and suppress noisy neighbours," in *ACM ICDCN*, 2020.

[14] P. Veitch, E. Curley, and T. Kantecki, "Performance evaluation of cache allocation technology for NFV noisy neighbor mitigation," in *IEEE NetSoft*, 2017.

[15] 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.

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

[17] 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.

[18] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: Transparently identifying and managing performance interference in virtualized environments," in *USENIX ATC*, 2013.

[19] K. Nikas, N. Papadopoulou, D. Giantsidi, V. Karakostas, G. Goumas, and N. Koziris, "Dicer: Diligent cache partitioning for efficient workload consolidation," in *ACM ICPP*, 2019.

[20] A. Baluta, J. Mukherjee, and M. Litoiu, "Machine learning based interference modelling in cloud-native applications," in *ACM ICPE*, 2022.

[21] "Kubernetes." https://kubernetes.io.

[22] "Docker-swarm." https://docs.docker.com.

[23] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *ACM SOSP*, 2009.

[24] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *ACM EuroSys*, 2010.

[25] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *USENIX HotEdge*, 2020.

[26] L. Gu, D. Zeng, J. Hu, B. Li, and H. Jin, "Layer aware microservice placement and request scheduling at the edge," in *IEEE INFOCOM*, 2021.

[27] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *IEEE INFOCOM*, 2021.

[28] 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.

[29] H. Bouattour, Y. B. Slimen, M. Mechteri, and H. Biallach, "Root cause analysis of noisy neighbors in a virtualized infrastructure," in *IEEE WCNC*, 2020.

[30] O. A. Wahab, N. Kara, C. Edstrom, and Y. Lemieux, "Maple: A machine learning approach for efficient placement and adjustment of virtual network functions," *J. of Network and Computer Applications*, vol. 142, pp. 37–50, 2019.

[31] "Mqttloader." https://github.com/dist-sys/mqttloader.

[32] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based CPU in container environments," in *ACM NSDI*, 2018.

[33] J. Benesty, J. Chen, Y. Huang, and I. Cohen, *Pearson Correlation Coefficient*, pp. 1–4. Springer, 2009.

[34] E. G. C. Jr, M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: A survey," 1996.

[35] 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.

[36] Y. Mu, L. Wang, and J. Zhao, "Energy-efficient and interference-aware vnf placement with deep reinforcement learning," in *2021 IFIP*, 2021.

[37] M. Adeppady, C. F. Chiasserini, H. Karl, and P. Giaccone, "iPlace: An interference-aware clustering algorithm for microservice placement," in *IEEE ICC*, 2022.

Madhura Adeppady has received her Master of Technology in Computer Science and Engineering from IIT Hyderabad, in 2020. She is currently pursuing her Ph.D. at Politecnico di Torino.

Paolo Giaccone (SM'16) received his Ph.D. degree from the Politecnico di Torino, Italy, where he is currently Associate Professor. In 2000-2001 and 2002 he was with Stanford University. His main area of interest is the design of network control and optimization algorithms.

Carla Fabiana Chiasserini (F'18) worked as a visiting researcher at UCSD and as a Visiting Professor at Monash University in 2012 and 2016. She is currently a Professor at Politecnico di Torino and EiC of Computer Communications.

Holger Karl leads the Internet Technology and Softwarization working group at the Hasso Plattner Institute, University Potsdam. His research interests are in network softwarization, machine learning in and for networks, and mobile systems.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Politecnico di Torino. Downloaded on January 12,2023 at 09:38:22 UTC from IEEE Xplore. Restrictions apply.