Containers Resource Allocation in Dynamic Cloud Environments

Oren Katz
Department of Computer Science
Technion
Haifa 32000, Israel
oren.katz@cs.technion.ac.il

Dror Rawitz
Faculty of Engineering
Bar Ilan University
Ramat Gan 52900, Israel
dror.rawitz@biu.ac.il

Danny Raz
Department of Computer Science
Technion
Haifa 32000, Israel
danny@cs.technion.ac.il

***Abstract*—Containers technology has become very popular in recent years, since it allows users to focus on designing their applications in a modular way and abstracting away the environments in which they actually run. Cloud providers such as AWS (Amazon Web Services) and GCP (Google Cloud Platform) offer their users managed containers platforms that orchestrate, schedule and execute multiple containers over a multi-tenant cloud infrastructure. As these services gain popularity, it is becoming more and more challenging to manage them in a way that effectively utilized the existing resources. The latter has a significant economical impact on cloud providers when it comes to their compute infrastructure investment costs and the price they can offer to their customers.**

In this paper, we approach this challenge by developing multi-dimensional container resource allocation algorithms designed to be deployed in dynamic cloud environments with different types of applications under varying loads scenarios. Our algorithms allocate for each container an available engine to execute it, in a way that maximizes the overall revenue. We design our algorithms and provide a constant worst-case approximation bound using the Local Ratio technique. Our evaluation, based on real-world scenarios, indicates that the performance of our algorithms is up to a factor of two better than the performance of existing scheduling algorithms, when the available resources are scarce.

***Index Terms*—cloud, containers, resource allocation, approximation algorithms**

I. Introduction

In recent years, an increasing number of users are building their business applications using public cloud platforms. Public cloud provider such as Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure offer an abundance of basic and complex services to their customers. The on-demand nature of the cloud's resources consumption model as well as its "pay-as-you-grow" pricing models have helped users to architect their applications by using on-demand virtual machines, leveraging design concepts such as "cloud elasticity". These concepts allow applications to use a varying amount of resources in order to handle different load scenarios and thus allows the application owner to provide an adequate level of service with a well defined costs. This is beneficial both for cloud users that can build efficient cost-aware architectures and for the public cloud providers as it allows more cloud users to grow their business by effectively consuming cloud resources.

At the same time, container technology has been adopted off and on the cloud as a preferred technology to build applications. One of the main reasons for that is that the footprint of a container is much smaller than the footprint of a full virtual machine. Moreover, the ability to spawn containers in microseconds, without the need to use a whole virtual machine and move them between development and production environments creates an appealing architecture for running modular and distributed applications on public cloud platforms. Therefore, containers have become the standard unit of software that packages up code and all its dependencies[1].

Containers run on a system that orchestrate their deployment and scaling as part of a containerized management application[2]. A very popular such system is Kubernetes[3]. In Kubernetes, a component named "Scheduler"[4] is responsible for selecting the specific virtual machine on which the container will run on and eventually allocates the required resources to run it.

The problem of effectively allocating containers' resources is of high importance both for public cloud providers and their users. The former try to maximize their profits by utilizing their cloud resources and maintaining a good Quality Of Service. At the same time, users would like to make sure their applications are cost-optimized without worrying about the complexity of the algorithms managing their resource allocations.

In most real-world scenarios, cloud resources are clustered into logical groups. In Kubernetes these groups are called nodes[5], which are virtual machines (VMs) that execute the containers' code. Each such node has a limited amount of resources in terms of CPU, memory and networking. Thus, the number of containers and their aggregated resources are constrained by the node's resources. For example, if the node is installed on a 48 virtual CPUs VM, and it has already run 45 vCPUs allocated to 45 single vCPU containers, it cannot

[1] see https://kubernetes.io/docs/concepts/configuration/overview/
[2] see https://www.datadoghq.com/container- orchestration/
[3] see https://kubernetes.io/
[4] see https://kubernetes.io/docs/concepts/scheduling/ kube-scheduler/
[5] seehttps://kubernetes.io/docs/concepts/architecture/nodes/

ISBN 978-3-903176-39-3 ©2021 IFIP

support a new request to execute a container that requires four vCPUs (in the sequel we use the term CPU instead of vCPU).

In this paper we define the CONTAINER ALLOCATION problem (abbreviated CONALLOC). The input for this problem consists of the number of nodes, k and a sequence of individual container requests including their CPU and memory requirements and the time-interval of the container. Every container request is also associated with a profit that the provider gains if this container is scheduled to run. The goal is to maximize the overall profit by choosing a subset of the containers to be executed. For each container in this subset we need to assign a node (out of the k available nodes) such that the resource constraints are kept. Thus our goal is to maximize the overall profit while maintaining the resource constraints.

From a practical point of view, considering an offline version of the problem is reasonable since, in many scenarios, the workload follows a daily (or weekly) pattern. Thus, cloud operators can predict the essential properties of the series of containers' requests. In addition, our analysis provides insight regarding the containers that can be scheduled and can be used to decide what type of VMs, and thus, what kind of underlying hardware infrastructure they need to purchase to run the execution nodes. These decisions have a long-term economic impact and data-centers operational implications.

We present a 12-approximation algorithm for CONALLOC, which is designed and analyzed using the Local Ratio technique [2], [4]. Roughly speaking, our algorithm partitions the requests sequence into three subsets: requests with low CPU and low memory requirements, requests with high CPU requirements, and high-memory requirements. We provide a generic local ratio algorithm and name it *Cradle*, which we run on the three subsets with different weight functions (see Section IV for more details). The corresponding approximation ratios are 8, 2, and 2. The maximum weight solution among the three candidates is a 12-approximation. We also provide a 9-approximation algorithm for the case where k is constant. In this case, the algorithm partitions the request sequence into two subsets: requests with low requirements, and the rest. We design a dynamic programming algorithm for the latter. Combined with the 8-approximation algorithm for the former, we get a 9-approximation algorithm. We note that Local Ratio algorithms for packing problems are relatively efficient. Moreover, tend to perform much better than their worst-case performance guarantee.

In our experimental study detailed in section V, we use a best-effort algorithm on top of our local ratio algorithm. First, we run our 12-approximation algorithm, and then we try using a best-effort approach to schedule the container requests that the initial algorithm did not serve. Henceforth we refer to this two-phase algorithm as *Cradle*$^+$. We note that the best effort phase does not significantly improve the performance. During our experiments, the best-effort stage increased the number of accepted requests by a few percent, say by 2%-5%. However, considering the scale of a public cloud platform, these percentage gains help cloud providers' efforts to maximize their utility, quality-of-service and may yield substantial economic gains.

We evaluate the performance of our algorithm, *Cradle*$^+$, by testing it using real-world data shared in *Google's Cluster's* data[6] which include detailed traces of 29 consecutive days of Linux container requests. These requests vary in sizes and ratios between their CPU and memory. We normalize the resource requests for each type of resource (CPU or memory) to be a number between 0 and 1. Then, we calculate the *Load Factor* that represents the load on the system under this set of requests. To calculate the load, we sum up all the containers' resource requests in the sequence and divide this sum by the number of available resources. (Section V includes a more precise definition for this calculation.).

Our experimental evaluation results show that *Cradle*$^+$ can schedule up to 2-times more containers than currently used heuristics. However, it happens in situations where the resources are scarce.

We use the term *performance factor* to describe the percentage of successfully serviced container requests out of all the containers in a sequence. In high load situations described above, *Cradle*$^+$ performance factor is just below 50%, while other heuristics achieve only 25%. In more appealing settings, when $60\% - 70\%$ of the requests are served, *Cradle*$^+$ serves 10-20% more requests.

Public cloud providers may use this *performance factor* when they plan their future data-centers capacity. This factor is significant because it helps determine the marginal quality-of-service and the utility gains they expect to deliver to their customers under different load scenarios. We discuss this point further in Section VI.

The main contributions of this paper are as follows. We develop a novel scheduling algorithm for containers in a cloud environment and provide proven guarantees on the expected performance. More specifically, we prove that our algorithm is a constant approximation algorithm for the theoretical problem we describe. We also evaluate the expected performance of the algorithm using real cloud traces. Our results indicate that (a) the algorithm performs much better than the theoretical bounds, and (b) it out preforms the common scheduling policies (Bin Packing and Spread) by 10-50% depending on the actual load on the system.

Paper organization: The rest of the paper is organized as follows. We describe related work in Section II. Then we define the problem and give a short overview of the local ratio technique in Section III. Our theoretical results are described in Section IV. This section contains the general 12-approximation algorithm and the 9-approximation algorithm for the special case where $k = O(1)$. Finally, we present our experimental results in Section V and suggest further steps for our research in Section VI.

II. RELATED WORK

The research around container orchestration and effective resources allocation yielded in recent years a plethora of

[6]https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md

studies. Some studies consider special cases of management and utility functions. For example, recently Zhou et al. [21] developed both off-line and on-line containers' scheduling algorithm while assuming "partial value for partial execution" of containers tasks. This may be a solid assumption for some types of workloads such as in long-lasting training function of machine learning models, some shorter micro-service oriented applications may require a full completion of such jobs to have any value. Mason et al. in [15] and Baig et al. in [19] suggested methods to predict cloud resources consumption - host CPU and data-center resources utilization using neural networks and adaptive prediction models accordingly.

Zheng and Shroff [20] assume partial values and a concave utility function that can deliver partial results, when it comes to online multi-resource allocation and with job preemption.

Menouer and Darmon [17] developed a new scheduling strategy by combining the *Spread* and the *Bin Packing* algorithms. This combination yielded the *TOPSIS* algorithm, which comprises between the *Spread Strategy* and the *Bin Packing Strategy* and provides more flexibility in choosing nodes and improving the global scheduling of several containers. This study assumes un-weighted containers and does not provide an analytical analysis of the expected performance.

Another experimental study by Li et al. [12] analyzed *Swarm* - a Docker container-based cluster management tool. They propose a *Particle Swarm* optimization-based container scheduling of Docker platform to achieve better container clusters load-balancing and resource utilization. This study does not include analytical performance examination. In another recent paper, Chung et al. [9] suggested a specialized scheduler for batch job execution. They demonstrated substantial cost reduction compared to best practices for batch job workloads. However, this work did not address the use-cases where modern applications implement a micro-services architecture, handling both batch and short-lived container demands.

Several experimental studies address challenges in containers performance optimization. In the study of Liu et al. [13], the authors focused on scheduling containers that belong to a specific service. Their algorithm considers the time it takes to transmit a container's image over the network. They define a dissipation function for each factor, such as networking speed, image size, CPU demand, to choose the best node in a container cluster. However, this research does not provide any analytical study of the problem or experimental evaluation on real-world container workloads.

Recent studies focused on workloads like deep-learning and big-data, as in Mao et al. [14] who suggested improved configuration scheme for Dockers and Kubernetes, and Menouer [16] who suggested a multi-criteria scheduling algorithm.

The one dimensional special case of CONALLOC, where $k = 1$ and there are no memory requirements, is called the BANDWIDTH ALLOCATION PROBLEM (BAP)[7]. The unique case of BAP in which a container fully occupies the VM is the MAXIMUM INDEPENDENT SET problem in interval graphs which is solvable in polynomial time (see, e.g., [10]). On the other hand, BAP (and thus CONALLOC) is NP-hard since KNAPSACK it is a special case of BAP, where all requests intervals intersect. Bar-Noy et al. [2] designed a local ratio 3-approximation algorithm for BAP. Calinescu et al. [6] developed a randomized approximation algorithm for BAP with an expected performance ratio of $2 + \epsilon$, for every $\epsilon > 0$. They obtained this result by dividing the given instance into an instance with large tasks and small tasks. They use dynamic programming to compute an optimal solution for the large instance and a randomized LP-based algorithm to obtain a $(1+\epsilon)$-approximate solution for the small instance. They also present a 3-approximation algorithm for BAP that is different from the one given in [2]. Chrobak et al. [8] showed that BAP is strongly NP-hard even for the case of uniform weights. Bonsma et al. [5] developed a $(7+\epsilon)$-approximation algorithm for an extension of BAP in which the available resource may change over time and showed that it is strongly NP-hard even for instances with demands in $\{1, 2, 3\}$. Anagnostopoulos et al. [1] presented a $(2 + \epsilon)$-approximation algorithm for this extension of BAP. This approximation was recently improved by Grandoni et al. [11] who improved the approximation ratio to $(5/3 + \epsilon)$. Finally, Bar-Noy et al. [2] considered a problem related to CONALLOC. In this problem, there are k one-dimensional identical VMs, and each container request fully occupies a VM. On the other hand, each request has several instances. They provide a 2-approximation algorithm for this problem.

[7]Other names: RESOURCE ALLOCATION PROBLEM or UNSPLITTABLE FLOW ON PATHS.

III. MODEL AND PRELIMINARIES

This section contains a formal definition of the container allocation problem and a short introduction to the *Local Ratio Technique* which we use in the next section.

A. The Problem

In the CONTAINER ALLOCATION PROBLEM (CONALLOC) we are given k identical virtual machines available over time, where each machine has a limited processing power and a predefined amount of memory. As explained in the introduction, these are the available Kubernetes nodes. One can assume that both amounts are normalized to 1, by scaling the containers' resource requirements. It follows that the resource requirements of each container request represents the portion of the VM's resources needed. The input consists of the number k and a sequence of n containers requests $C = \{1, 2, \ldots, n\}$, where each container request j, for $1 \leq j \leq n$, is associated with the following parameters:

- a weight $w(j) \in \mathbb{R}_+$,
- a time interval $I_j = (s_j, f_j]$, where $s_j < f_j$, and
- two resource requirements $a_j, b_j \in [0, 1]$.

We assume that a_j represents the amount of CPU and that b_j represents the amount of memory requested by j. Observe that the problem becomes trivial if $n \leq k$, henceforth we assume that $k < n$.

A feasible solution, or an *allocation*, is a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k pairwise disjoint subsets of the requests such that the following conditions are met:

$$\sum_{j \in S_i : t \in I_j} a_j \leq 1 \qquad (1)$$

$$\sum_{j \in S_i : t \in I_j} b_j \leq 1 \qquad (2)$$

for every time t and for every subset S_i. Subset S_i stands for the requests that are allocated to VM i. Conditions (1) and (2) ensure that the VM is not overload. The weight of a solution $\mathcal{S}$ is defined as $w(\mathcal{S}) = \sum_{j \in \cup_i S_i} w_j$. The goal is to find an allocation that maximizes the weight of allocated requests, namely to find $\mathcal{S}$ that maximizes $w(\mathcal{S})$.

Let T be a set that contains all start times and finish times of requests in C, i.e., $T = \cup_{j=1}^{n} \{s_j, f_j\}$. Observe that, given an allocation $\mathcal{S} = \{S_1, \ldots, S_k\}$, the set $\{j \in S_i : t \in I_j\}$ remains the same for all $t \in (t_1, t_2]$, where t_1 and t_2 are two consecutive times in T. Hence, it is enough to require that Conditions (1) and (2) hold for all $t \in T$.

The above observation leads to an integer linear programming (ILP) formulation of CONALLOC. Let x_{ij} be a variable that is set to 1 if request j is placed in the i'th machine. That is $x_{ij} = 1$ if and only if $j \in S_i$. The following ILP formulation captures CONALLOC:

$$\begin{aligned}
\max \sum_{j=1}^{n} w_j \sum_{i=1}^{k} x_{ij} & & \\
\text{s.t.} \sum_{i=1}^{k} x_{ij} \leq 1 & & \forall j \\
\sum_{j : t \in I_j} x_{ij} a_j \leq 1 & & \forall t \in T, i \\
\sum_{j : t \in I_j} x_{ij} b_j \leq 1 & & \forall t \in T, i \\
x_{ij} \in \{0, 1\} & & \forall i, j
\end{aligned}$$

Given two solutions $\mathcal{S}$ and $\mathcal{S}'$ we abuse notation by defining:

- $\mathcal{S} \cup \mathcal{S}' = (S_1 \cup S'_1, \ldots, S_k \cup S'_k)$.
- $\mathcal{S} \setminus \mathcal{S}' = (S_1 \setminus S'_1, \ldots, S_k \setminus S'_k)$.

Let $\ell \in C$ be a container request. A solution $\mathcal{S}$ is called *ℓ-maximal*, if either ℓ is in the solution, or it ℓ cannot be added to the solution. More formally, $\mathcal{S}$ is ℓ-maximal if one of the following conditions is satisfied:

- $\ell \in S_i$, for some i, or
- For every machine i, there exists a time $t \in I_\ell$ such that $\sum_{j \in S_i : t \in I_j} a_j + a_\ell > 1$.

B. Narrow and Wide Container Requests

In this section we describe a partition of the set of container requests into several subsets. We say that a request j is *narrow* if $a_j, b_j \leq \frac{1}{2}$. Otherwise, the request is called *non-narrow*. Let N be the set of narrow requests, i.e., $N = \left\{j : a_j \leq \frac{1}{2} \text{ and } b_j \leq \frac{1}{2}\right\}$.

A non-narrow request if called a-wide, if $a_i > \frac{1}{2}$. Similarly, a non-narrow request is called b-wide, if $b_i > \frac{1}{2}$. Define

$$W^a = \left\{j : a_j > \frac{1}{2}\right\}$$

and

$$W^b = \left\{j : b_j > \frac{1}{2}\right\} .$$

Observe that it may be the case that $W^a \cap W^b \neq \emptyset$.

C. The Local Ratio Technique

The local ratio technique [2], [4] is based on the Local Ratio Lemma, which applies to maximization (or minimization) problems of the following type. The input is a non-negative weight vector $w \in \mathbb{R}^n$ and a set of feasibility constraints $\mathcal{F}$. The problem is to find a solution vector $x \in \mathbb{R}^n$ that maximizes (or minimizes) the inner product $w \cdot x$ subject to the constraints $\mathcal{F}$. The proof of the lemma for the maximization case is given for completeness.

Lemma III.1 (Local Ratio [2])**.** *Let $\mathcal{F}$ be a set of constraints and let w, w_1, and w_2 be weight functions such that $w = w_1 + w_2$. Then, if x is r-approximate both with respect to w_1 and with respect to w_2, for some r, then x is also an r-approximate solution with respect to w.*

Proof. Let x^* denote an optimal solution with respect to w, and let x^i denote an optimal solution with respect to w_i where $i \in \{1, 2\}$. Then,

$$\begin{aligned}
w \cdot x &= w_1 \cdot x + w_2 \cdot x \\
&\geq r \cdot (w_1 \cdot x^1) + r \cdot (w_2 \cdot x^2) \\
&\geq r \cdot (w_1 \cdot x^* + w_2 \cdot x^*) \\
&= r \cdot (w \cdot x^*) ,
\end{aligned}$$

and we are done. □

Surveys on the local ratio technique can be found in [3] and in [18].

IV. APPROXIMATION ALGORITHMS

In this section we present a 12-approximation algorithm for CONALLOC. Our algorithm is designed and analyzed using the local ratio technique. It computes three solutions: (i) a 2-approximation for a-wide requests, and (ii) a 2-approximation for b-wise requests. (iii) a 8-approximation for narrow requests. The classification of the containers' requests into three groups separates the narrow requests from the rest (a-wide or b-wide requests), thus allowing us to achieve a better approximation constant overall.

We also discuss the possibility of using dynamic programming in order to handle non-narrow requests in the case where $k = O(1)$. Together with the 8-approximation algorithm this leads to a 9-approximation algorithm in this case.

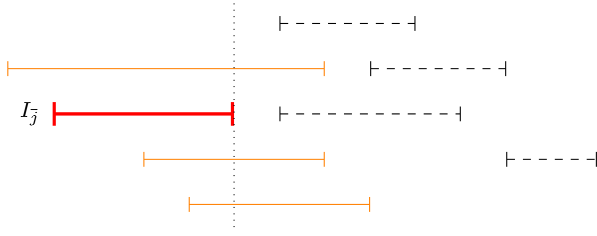

Figure 1. An example showing the time intervals of container requests. The thick red line represents $I_{\bar{j}}$, the solid orange lines represent the time intervals of requests whose time interval intersects $I_{\bar{j}}$, and dashed intervals represent the rest of the time intervals. w_1 assigns non-zero weights to $\bar{j}$ and to the requests whose time intervals are represented by solid orange lines.

A. Algorithm for a-Wide and b-Wide Container Requests

In this section we design an approximation algorithm for a-wide container requests, i.e., for the set W^a. Later on we show that a similar algorithm works for b-wide requests.

Let $j, \ell \in W^a$ be two requests such that $I_j \cap I_\ell \neq \emptyset$. In this case c_j and c_ℓ cannot be mapped to the same machine since $a_j + a_\ell > 1$. More formally, given a solution $\mathcal{S} = \{S_1, \dots, S_k\}$, it cannot be that $j, \ell \in S_i$, for some i. It follows that S_i must be a pairwise non-intersecting subset of requests, for every i. Hence, the number of requests which contains time t is at most k, for any given time t.

Let $\bar{j}$ be a container request with the earliest finish time. i.e., $f_{\bar{j}} \leq f_j$, for every $j \in W^a$. Define the following weight function

$$w_1(j) = w(\bar{j}) \cdot \begin{cases} 1 & j = \bar{j} \\ \frac{1}{k} & j \neq \bar{j}, I_j \cap I_{\bar{j}} \neq \emptyset \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

See example in Fig. 1.

The next lemma shows that a $\bar{j}$-maximal solution is 2-approximate with respect to the weight function w_1.

Lemma IV.1. *Let $\mathcal{S}^*$ be an optimal solution with respect to W^a and w_1. Also, let $\mathcal{S}$ be an $\bar{j}$-maximal solution. Then, $w_1(\mathcal{S}) \geq \frac{k}{2k-1} w_1(\mathcal{S}^*)$.*

Proof. If $\bar{j}$ appears in $\mathcal{S}^*$, then it may also contain at most $k-1$ additional requests that contain $f_{\bar{j}}$. Hence,

$$w_1(\mathcal{S}^*) \leq w_1(\bar{j}) + (k-1)\frac{w_1(\bar{j})}{k} = (2 - \tfrac{1}{k})w(\bar{j}) \ .$$

Otherwise, if $\bar{j}$ is not a part of $\mathcal{S}^*$, we have that $w_1(\mathcal{S}^*) \leq k\frac{w_1(\bar{j})}{k} = w_1(\bar{j})$.

On the other hand, if $\bar{j}$ is contained in $\mathcal{S}$ we have that $w_1(\mathcal{S}) \geq w_1(\bar{j})$. Otherwise, $\mathcal{S}$ contains k requests whose time interval contain $f_{\bar{j}}$. Hence, $w_1(\mathcal{S}) \geq k\frac{w_1(\bar{j})}{k} = w_1(\bar{j})$.

It follows that $w_1(\mathcal{S}) \geq w_1(\bar{j}) \geq \frac{k}{2k-1} w_1(\mathcal{S}^*)$. ☐

We use the above weight function in our local ratio algorithm (Algorithm 1). The algorithm is named *Cradle* which stands for **C**ontainers **R**esource **A**llocation in **D**ynamic **CL**oud **E**nvironments. The initial call is *Cradle*(W^a, w).

We discuss the implementation details and the running time of *Cradle*. If we order the requests by their finish times, then it takes $O(1)$ time to find $\bar{j}$ in each recursive call. Hence, each recursive call can be implemented with a running time of $O(n)$. Since there are at most $2n$ recursive calls, the total running time is $O(n^2)$. A faster $O(n \log n)$ implementation can be obtained using a sweep-line approach as was done in [3].

Algorithm 1 – *Cradle* (C, w)

1: **if** $C = \emptyset$ **then return** $(\emptyset, \dots, \emptyset)$
2: **if** there exists j such that $w(j) \leq 0$ **then**
3: **return** *Cradle*$(C \setminus \{j\}, w)$
4: Let $\bar{j}$ be the container request whose finish time is earliest in J, and let w_1 be the weight function from (3). Define $w_2 = w - w_1$.
5: $\mathcal{S} \leftarrow$ *Cradle*(C, w_2)
6: **if** there exists i such that $S_i \cup \{\bar{j}\}$ is pair-wise non-intersecting **then**
7: $S_i \leftarrow S_i \cup \{\bar{j}\}$
8: **return** $\mathcal{S}$

It remains to bound the approximation ratio.

Theorem IV.2. *Algorithm* Cradle *is a $(2 - \frac{1}{k})$-approximation algorithm for an instance containing only a-wide requests.*

Proof. We use induction on the number of recursive calls to show that the approximation ratio of Algorithm *Cradle* is $2 - \frac{1}{k}$.

In the base case $C = \emptyset$, and therefore the empty solution is optimal.

For the inductive step, we assume that the recursive call returns a $(2 - \frac{1}{k})$-approximate solution. There are two options. If the recursive call was made in Line 3, then the solution returned is $(2 - \frac{1}{k})$-approximate with respect to $C \setminus \{j\}$ and w. Since $w(j) \leq 0$, it is also $2 - \frac{1}{k}$-approximate with respect to C and w. Otherwise, if the recursive call was made in Line 5, then the solution returned is $(2 - \frac{1}{k})$-approximate with respect to C and w_2. In addition, since $\mathcal{S}$ is $\bar{j}$-maximal, by Lemma IV.1 it is $(2 - \frac{1}{k})$-approximate with respect to C and w_2. Hence, by the local ratio lemma it is $(2 - \frac{1}{k})$-approximate with respect to C and w as well. ☐

Now we turn to handle b-wide requests. Let $j, \ell \in W^b$ be two requests such that $I_j \cap I_\ell \neq \emptyset$. In this case c_j and c_ℓ cannot be mapped to the same machine since $b_j + b_\ell > 1$. Hence, given a solution $\mathcal{S} = \{S_1, \dots, S_k\}$, it cannot be that $j, \ell \in S_i$, for some i. If follows that similar arguments work for b-wide requests, and thus we have the following result.

Theorem IV.3. *Algorithm* Cradle *is a $(2 - \frac{1}{k})$-approximation algorithm for an instance containing only b-wide requests.*

B. Algorithm for Narrow Container Requests

In this section we use a version of Algorithm *Cradle* in order to obtain an 8-approximation algorithm for instances containing only narrow requests. (Recall that $a_j, b_j \leq \frac{1}{2}$, for every narrow request j.)

As before, let $\bar{j}$ be a container with the earliest finish time, i.e., $f_{\bar{j}} \leq f_j$, for every $j \in N$. Define the following weight function

$$w_1(j) = w(\bar{j}) \cdot \begin{cases} 1 & j = \bar{j} \\ \frac{a_j + b_j}{2k - a_{\bar{j}} - b_{\bar{j}}} & j \neq \bar{j}, I_j \cap I_{\bar{j}} \neq \emptyset \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

The next lemma shows that a $\bar{j}$-maximal solution is 8-approximate with respect to the weight function w_1.

Lemma IV.4. *Let $\mathcal{S}^*$ be an optimal solution with respect to N and w_1. Also, let $\mathcal{S}$ be an $\bar{j}$-maximal solution. Then, $w_1(\mathcal{S}) \geq \frac{w_1(\mathcal{S}^*)}{8}$.*

Proof. First, assume that $\bar{j} \in S_i^*$, for some i. In this case, S_i may contain additional requests whose total CPU requirements is $1 - a_{\bar{j}}$ and whose total memory requirements is $1 - b_{\bar{j}}$. Moreover, the other $k - 1$ machine may also be fully used. Hence, in this case

$$\begin{aligned} w_1(\mathcal{S}^*) &\leq w_1(\bar{j}) + w_1(\bar{j}) \cdot \frac{1}{2k - a_{\bar{j}} - b_{\bar{j}}} \cdot \sum_{j \in \cup_i S_i, j \neq \bar{j}} (a_j + b_j) \\ &\leq w_1(\bar{j}) + w_1(\bar{j}) \cdot \frac{1}{2k - a_{\bar{j}} - b_{\bar{j}}} \cdot (2k - a_{\bar{j}} - b_{\bar{j}}) \\ &= 2w_1(\bar{j}) \ . \end{aligned}$$

Otherwise, if $\bar{j}$ is not a part of $\mathcal{S}^*$, we have that

$$\begin{aligned} w_1(\mathcal{S}^*) &\leq w_1(\bar{j}) \cdot \frac{1}{2k - a_{\bar{j}} - b_{\bar{j}}} \cdot \sum_{j \in \cup_i S_i} (a_j + b_j) \\ &\leq w_1(\bar{j}) \cdot \frac{2k}{2k - 1} \\ &\leq 2w_1(\bar{j}) \ . \end{aligned}$$

As for $\mathcal{S}$, if $\bar{j}$ is contained in $\mathcal{S}$ we have that $w_1(\mathcal{S}) \geq w_1(\bar{j})$. Otherwise, the load $\mathcal{S}$ places either on the CPU or on the memory of each of the k machines prevents $\bar{j}$ to be placed. Hence,

$$\begin{aligned} w_1(\mathcal{S}) &= w_1(\bar{j}) \cdot \frac{1}{2k - a_{\bar{j}} - b_{\bar{j}}} \cdot \sum_i \sum_{j \in S_i} (a_j + b_j) \\ &\geq w_1(\bar{j}) \cdot \frac{1}{2k} \cdot \sum_i \max \left\{ \sum_{j \in S_i} a_j, \sum_{j \in S_i} b_j) \right\} \\ &\geq w_1(\bar{j}) \cdot \frac{1}{2k} \cdot \sum_i \min \left\{ 1 - a_{\bar{j}}, 1 - b_{\bar{j}} \right\} \\ &\geq w_1(\bar{j}) \cdot \frac{1}{2k} \cdot \sum_i 0.5 \\ &\geq \frac{w_1(\bar{j})}{4} \end{aligned}$$

It follows that $w_1(\mathcal{S}) \geq w_1(\bar{j}) \geq \frac{w_1(\mathcal{S}^*)}{8}$. □

We use *Cradle* with the weight function given in (4) instead of the one given in (3). The initial call is *Cradle*(N, w). The running time of *Cradle* remains $O(n^2)$.

Theorem IV.5. *Algorithm* Cradle *is a* 8*-approximation algorithm for an instance containing only narrow requests.*

Proof. The proof is similar to the proof of Theorem IV.2, where Lemma IV.4 is used instead of Lemma IV.1. □

C. Approximation Algorithm for General Instances

In this section we give a 12-approximation algorithm for general instances using the algorithms for a-wide, b-wide, and narrow requests.

Theorem IV.6. *There exists a* 12*-approximation algorithm for* ConAlloc.

Proof. Let $\mathcal{S}^*$ be an optimal solution. Let

$$\begin{aligned} \mathcal{S}_a^* &= (S_1^* \cap W^a, \ldots, S_k^* \cap W^a) \\ \mathcal{S}_b^* &= (S_1^* \cap W^b, \ldots, S_k^* \cap W^b) \\ \mathcal{S}_N^* &= (S_1^* \cap N, \ldots, S_k^* \cap N) \ . \end{aligned}$$

Observe that at least one of the following option must hold: (i) $w(\mathcal{S}_a^*) \geq \frac{1}{6} w(\mathcal{S}^*)$, (ii) $w(\mathcal{S}_b^*) \geq \frac{1}{6} w(\mathcal{S}^*)$, or (iii) $w(\mathcal{S}_N^*) \geq \frac{2}{3} w(\mathcal{S}^*)$. Let $\mathcal{S}_a$, $\mathcal{S}_b$, and $\mathcal{S}_N$ be the solutions computed by *Cradle* for W^a, W^b, and N. By Theorems IV.3 and IV.5 it follows that one of these solutions must be 12-approximate. □

D. Improving the Approximation Ratio

In this section we present a dynamic programming algorithm for computing an optimal solution for an instance containing only non-narrow container requests. Combining it with the 8-approximation algorithm for narrow requests, leads to a 9-approximation algorithm for the general case. Our dynamic programming algorithm is based on the algorithms for the Bandwidth Allocation Problem by Chen et al. [7] and by Calinescu et al. [6].

Consider a set of container requests W containing only non-narrow requests. That is, $W = W^a \cup W^b$.

Theorem IV.7. *Let $k = O(1)$. There is a polynomial time algorithm that, given an instance containing only non-narrow requests, computes an optimal solution.*

The proof of the next theorem is similar to the proof of Theorem IV.6.

Theorem IV.8. *There exists a* 9*-approximation algorithm for* ConAlloc *for the case where* $k = O(1)$.

The proofs of both theorems were omitted for lack of space.

V. Experimental Evaluation

To evaluate the new algorithm's performance, we use Google's cluster data, which contains requests of Linux container jobs during one month[8]. These container requests include the CPU and the memory requirements, together with the start and end times of the execution. Note that the requests have different ratios between CPU and memory. For example,

[8]https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md

one request for a Linux container can be 4vCPUs and 4GB of memory, and another can be 2vCPU and 4GB of memory. These may result either in a 1:1 or a 1:2 CPU to memory ratios, accordingly.

We used the real-world data and normalized the demands for CPU and memory. To test the performance in different load situations, we sampled (uniformly at random) the data space to produce multiple scenarios. For example, we used the same data to test the performance of the various algorithms when we have a total of 1000 virtual machines or a total of 100 virtual machines. The available resources in the latter case are much lower; thus, they demonstrate a higher load on the system.

We use the term *load factor* to describe the ratio between the overall sum of required resources for all the container requests in the tested set and the amount of available resources (i.e., the number of VMs, k, times the time horizon). Since we deal with a multidimensional problem, we use the average of the requirements as the demand of a single container request. We note that the requirements are typically in the same magnitude in our data set. For example: a CPU load factor of a data set can be 900 and the memory load factor of the same data set can be 700. For example, if the normalized request of a container is 0.1 CPU, and 0.3 memory, and it is executed over 10 seconds, then the total normalized demand is $(0.3+0.1)/2 \cdot 10 = 2$ sec. The load factor is then the sum of the demand of all containers requests divided by k times the overall time of the test. More formally, define

$$\rho \triangleq \frac{\sum_{j \in C} \frac{a_j + b_j}{2} \cdot (f_j - s_j)}{k \cdot T_{\text{Horizon}}} \;, \tag{5}$$

where $T_{\text{Horizon}} = \max\{T\} - \min\{T\}$ is the overall time duration of the execution. The algorithm's quality on a given input sequence is measured by its profit, where the goal is to maximize it. When containers are not associated with a specific profit (as is the case for our data set), we use the overall number of scheduled items as the profit. Thus we measure the performance as the percentage of scheduled containers algorithm A achieves compared to the total number of containers in the input sequence.,

$$\mathcal{P}(A) \triangleq 100 \cdot \frac{\text{\# containers scheduled by ALG}}{\text{\# containers in } C} \;.$$

In order to compare *Cradle*$^+$ to existing best practices results, we use the Spread and Bin-Packing algorithms, which are currently known best practices when it comes to Kubernetes containers scheduling[9]. The Bin Packing algorithm schedules container requests by scheduling as many as possible requests to the first virtual machine and moves on to the next virtual machine as soon as the current machine's resources, either CPU or memory, can not satisfy the following request. Spread algorithm schedules container requests by scheduling a container's request on a virtual machine and serving the subsequent request on the next indexed virtual machine. While Bin packing makes the best effort to pack as many container requests on a single virtual machine, spread tries to load balance the requests on different virtual machines.

[9]https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes/

Both algorithms have operational and economic justification: Bin packing can minimize the required amount of virtual machines when scheduling containers, which can help save costs for the users or the cloud provider, for example, in testing and development environments. However, Spread can reduce the density of containers on each virtual machine and allow the higher performance of a container in a micro-services architecture deployments.

In order to get different series of requests with different load factors for the simulation based evaluation, we sampled the real-world-data uniformly at random to get multiple series of requests. For each such set, we used Equation 5 and computed for each relevant load the appropriate number of server.

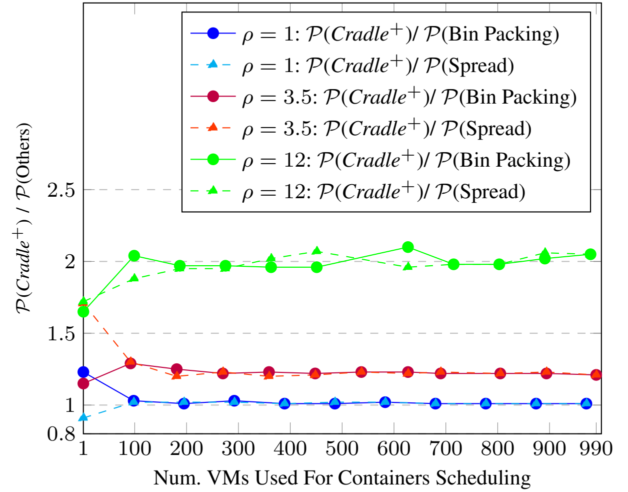

Figure 2. *Cradle*$^+$ Performance Comparisons to Others

Fig. 2 depicts the ratio between the performance of *Cradle*$^+$ and the currently deployed heuristics. One can observe that *Cradle*$^+$ outperforms these heuristics, especially in high load situations (e.g., $\rho = 12$). In such high load situations, the scheduling system receives many requests from containers such that the overall resource request is much higher than the available resources. Since the distribution of the requests over time varies, the actual performance of an algorithm is not determined solely by the load. Further, one can observe that the algorithm's preference remains high even when the number of VMs (or Kubernetes nodes) increases and becomes very large, which indicates that *Cradle*$^+$ scales up well.

We note that both Bin packing and Spread are online in nature while *Cradle*$^+$ is offline. Yet, during high load *Cradle*$^+$ successfully handles more than 100% container requests along the time of tests. The reason for *Cradle* to serve more requests as the load increases is as follows: *Cradle* uses the weight function described in Subsection IV-B. This function causes *Cradle* to prioritize narrower containers having higher utility. This means that the scheduling scheme of *Cradle* is more dense compared both to Bin packing and Spread when it comes to populating more narrower containers. Containers with wide

dimensions and high utility are thus prioritized by *Cradle* and the overall utility increases.

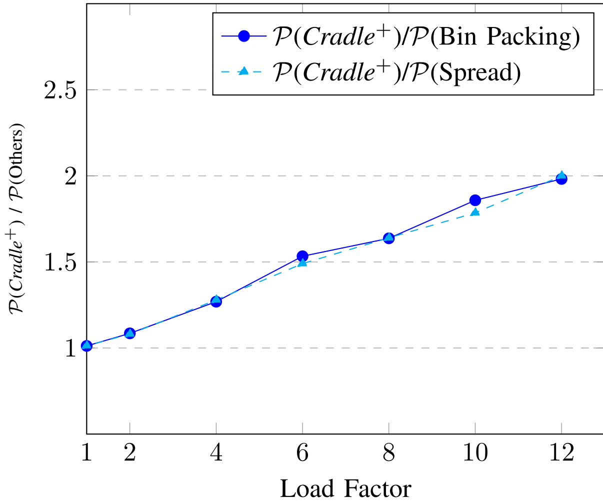

Figure 3. Cradle's Performance vs. Bin-Packing's and Spread's

To better understand the effect of the load, we depict in Fig. 3 the ratio between *Cradle*$^+$ performance and the performance of the other heuristics as a function of the load. *Cradle*$^+$ serves more container requests compared to the other two heuristics as the load increases. When $\rho = 1$, the overall amount of requested resources is roughly equal to the available resources, and thus it is possible to satisfy almost all requests. As the load increases, the ratio between *Cradle*$^+$ performance and the other heuristics performance increases until it gets to 2 when the load is 12.

In Fig. 4 we depict the actual performance of all three algorithms. One can see that starting with a load of 1, *Cradle*$^+$ outperforms the other heuristics. When the load increases, even by a small amount, say to 2, there is already a performance gap between the algorithms where *Cradle*$^+$ serves about 10% more containers than others. Finally, cloud operators can reduce their infrastructure by a similar factor if they use *Cradle*.

A natural question to ask is what is a reasonable working load for container scheduling systems. When resources are scarce, demand is high; a cloud provider may still offer different service levels, for example, premium service Vs. Best effort one. Thus it makes sense to operate cloud environments at high loads. However, in most common cases, cloud operators would target a moderately high load that will allow them to serve almost all requests.

For this reason, we further analyzed *Cradle*$^+$'s performance in situations where it schedules 50% to 80% of the arriving containers. The results of this analysis are presented in Fig. 5. While all algorithms perform similarly at around 100% load, the degradation in performance of *Cradle*$^+$ is much more moderate than Bin-Packing and Spread. The latter is because, in this situation, the algorithm has a choice regarding the exact containers to schedule, and *Cradle*$^+$ can then use the logic to effectively select more requests compared to other algorithms as later on demonstrated in Fig. 6.

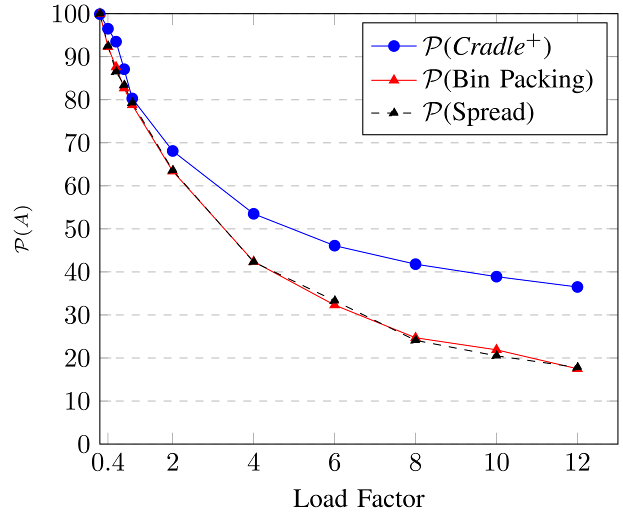

Figure 4. Cradle's Performance vs. Others

Our simulations examined a spectrum of load factors and their matching performance results. However, in the figures below, we emphasize specific anchor numbers we believe are meaningful for comparing the different algorithms.

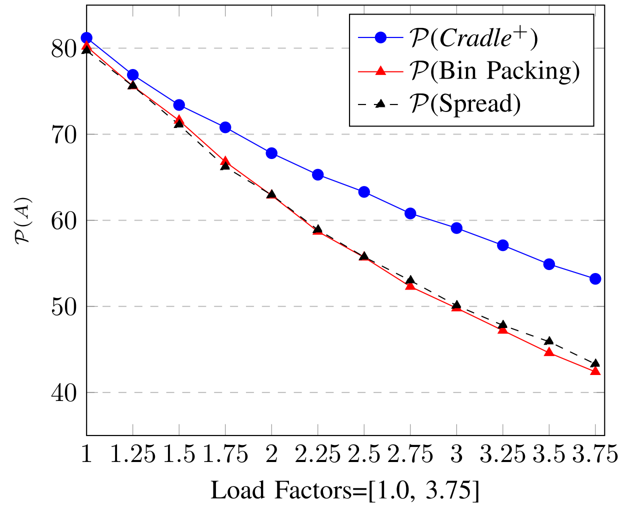

Figure 5. Cradle's Performance Under Load Factors: 1.0-3.75

As explained in the previous sections, *Cradle* gives priority to requests that require fewer resources. Hence, even when scheduling the same number of containers, *Cradle* uses fewer resources than the currently deployed alternative. Hence *Cradle*$^+$ typically uses fewer resources as shown below for *Load Factors* higher than one compared to Bin-Packing and Spread. As can be seen, *Cradle*$^+$ can be twice more effective when scheduling container requests when $\rho = 12$. In such situations *Cradle*$^+$ uses half of the available resources and still manages to double the performance as noted above since it first serves narrow container requests.

Our extensive simulation results indicate that *Cradle*$^+$ does a better job of utilizing the available resources. The gap

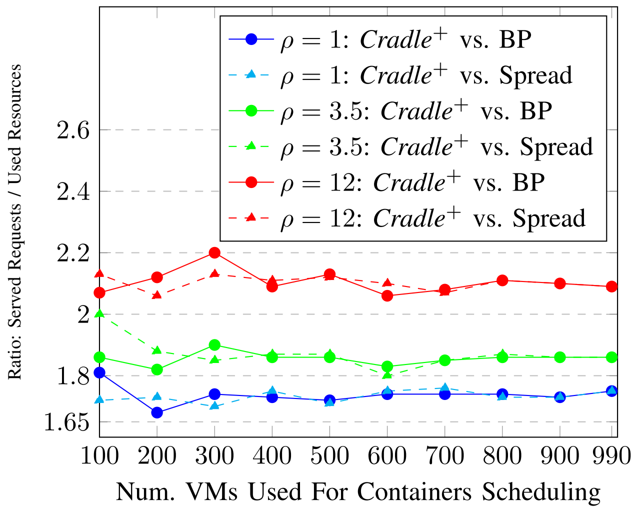

Figure 6. *Cradle*$^+$'s Utility-To-Resources Effectiveness

concerning the commonly deployed heuristics increases when resources are scarce, and the exact choice of the containers has a more significant impact. As we pointed out, cloud providers can utilize this advantage and reduce their infrastructure costs without lowering their SLA.

VI. Discussion

In this paper, we introduced *Cradle*$^+$, a new container scheduling algorithm designed to handle dynamic workloads when resources are scarce. We provide a theoretical analysis that proves a constant approximation ratio. Moreover, our experimental results demonstrate that *Cradle*$^+$ outperforms currently used heuristics, especially when the load on the system is high.

One possible way to further improve *Cradle*$^+$ is to use its *weight function* property. An appropriate use of this weight function, that allows prioritizing user requests, can significantly improve performance.

A vital aspect left for future work is using the insights we gained through the analysis and generating an online solution that maintains *Cradle*'s advanced properties.

References

[1] A. Anagnostopoulos, F. Grandoni, S. Leonardi, and A. Wiese. A mazing $(2+\epsilon)$-approximation for unsplittable flow on a path. In *25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 26–41, 2014.

[2] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. *J. ACM*, 48(5):1069–1090, 2001.

[3] R. Bar-Yehuda, K. Bendel, A. Freund, and D. Rawitz. Local ratio: A unified framework for approxmation alglrithms in memoriam: Shimon even 1935-2004. *ACM Comput. Surv.*, 36(4):422–463, 2004.

[4] R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–46, 1985.

[5] P. S. Bonsma, J. Schulz, and A. Wiese. A constant-factor approximation algorithm for unsplittable flow on paths. *SIAM J. Comput.*, 43(2):767–799, 2014.

[6] G. Calinescu, A. Chakrabarti, H. J. Karloff, and Y. Rabani. Improved approximation algorithms for resource allocation. In *9th International Integer Programming and Combinatorial Optimization Conference*, volume 2337 of *LNCS*, pages 401–414, 2002.

[7] B. Chen, R. Hassin, and M. Tzur. Allocation of bandwidth and storage. *IIE Transactions*, 34:501–507, 2002.

[8] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu. Caching is hard - even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[9] A. Chung, J. W. Park, and G. R. Ganger. Stratus: cost-aware container scheduling in the public cloud. In *ACM Symposium on Cloud Computing*, pages 121–134, 2018.

[10] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 1980.

[11] F. Grandoni, T. Mömke, A. Wiese, and H. Zhou. A $(5/3+\epsilon)$-approximation for unsplittable flow on a path: placing small tasks into boxes. In *50th Annual ACM Symposium on the Theory of Computing*, pages 607–619, 2018.

[12] L. Li, J. Chen, and W. Yan. A particle swarm optimization-based container scheduling algorithm of docker platform. In *4th International Conference on Communication and Information Processing*, pages 12–17, New York, NY, USA, 2018. ACM.

[13] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang. A new container scheduling algorithm based on multi-objective optimization. *Soft Computing*, 22(23):7741–7752, 2018.

[14] Y. Mao, Y. F. S. Gu, S. Vhaduri, L. Cheng, and Q. Liu. Resource management schemes for cloud native platformswith computing containers of docker and kubernetes. In *arXiv preprint*, page arXiv:2010.10350, 2020.

[15] K. Mason, M. Duggan, E. Barrett, J. Duggan, and E. Howley. Predicting host cpu utilization in the cloud using evolutionary neural networks. In *Future Generation Computer Systems*, volume 86, pages 162–173. ELSEVIER, 2018.

[16] T. Menouer. Kcss: Kubernetes container scheduling strategy. In *The Journal of Supercomputing*, pages https://doi.org/10.1007/s11227–020–03427–3, 2020.

[17] T. Menouer and P. Darmon. New scheduling strategy based on multi-criteria decision algorithm. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2019*, pages 101–107, 2019.

[18] D. Rawitz. c. In T. F. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*, chapter 6. Chapman and Hall/CRC, 2018.

[19] S. ur Rehman Baig, W. Iqbal, J. L. Berral, A. Erradi, and D. Carrera. Adaptive prediction models for data center resources utilization estimation. In *16th IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT*, volume 16, 2019.

[20] Z. Zheng and N. B. Shroff. Online multi-resource allocation for deadline sensitive jobs with partial values in the cloud. In *35th Annual IEEE INFOCOM*, pages 1–9, 2016.

[21] R. Zhou, Z. Li, and C. Wu. Scheduling frameworks for cloud container services. *IEEE/ACM Trans. Netw.*, 26(1):436–450, 2018.