UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

# Optimizing Service Placement for Microservice Architecture in Clouds

Hu, Y.; de Laat, C.; Zhao, Z.

*Published in:*
Applied Sciences

*DOI:*
10.3390/app9214663

Link to publication

*License*
**CC BY**

*Citation for published version (APA):*
Hu, Y., de Laat, C., & Zhao, Z. (2019). Optimizing Service Placement for Microservice Architecture in Clouds. *Applied Sciences*, *9*(21), [4663]. https://doi.org/10.3390/app9214663

**General rights**
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

**Disclaimer/Complaints regulations**
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

*UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)*

Download date: 26 Feb 2021

applied sciences

MDPI

*Article*

# Optimizing Service Placement for Microservice Architecture in Clouds

**Yang Hu [1,2], Cees de Laat [1] and Zhiming Zhao [1,*]**

[1] Informatics Institute, Faculty of Science, University of Amsterdam, 1098 XH Amsterdam, The Netherlands; y.hu@uva.nl (Y.H.); delaat@uva.nl (C.d.L.)
[2] School of Computer Science, National University of Defense Technology, Changsha 410073, China
* Correspondence: z.zhao@uva.nl

Received: 9 September 2019; Accepted: 28 October 2019; Published: 1 November 2019

**Abstract:** As microservice architecture is becoming more popular than ever, developers intend to transform traditional monolithic applications into service-based applications (composed by a number of services). To deploy a service-based application in clouds, besides the resource demands of each service, the traffic demands between collaborative services are crucial for the overall performance. Poor handling of the traffic demands can result in severe performance degradation, such as high response time and jitter. However, current cluster schedulers fail to place services at the best possible machine, since they only consider the resource constraints but ignore the traffic demands between services. To address this problem, we propose a new approach to optimize the placement of service-based applications in clouds. The approach first partitions the application into several parts while keeping overall traffic between different parts to a minimum and then carefully packs the different parts into machines with respect to their resource demands and traffic demands. We implement a prototype scheduler and evaluate it with extensive experiments on testbed clusters. The results show that our approach outperforms existing container cluster schedulers and representative heuristics, leading to much less overall inter-machine traffic.

**Keywords:** service placement; cluster scheduling; network optimization; resource management; microservice architecture; cloud computing

## 1. Introduction

Microservice architecture is a new trend rising fast for application development, as it enhances flexibility to incorporate different technologies, it reduces complexity by using lightweight and modular services, and it improves overall scalability and resilience of the system. In the definition (Microservices: https://martinfowler.com/tags/microservices.html), the microservice architectural style is an approach to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms (e.g., HTTP resource API). The application then is composed of a number of services (service-based application) that work cohesively to provide complex functionalities. Due to the advantages of microservices architecture, many developers intend to transform traditional monolithic applications into service-based applications. For instance, an online shopping application could be basically divided into product service, cart service, and order service, which can greatly improve the productivity, agility, and resilience of the application. However, it also brings challenges. When deploying a service-based application in clouds, the scheduler has to carefully schedule each service, which may have diverse resource demands, on distributed compute clusters. Furthermore, the network communication between different services needs to be handled well, as the communication conditions significantly influence the quality of service (e.g., the response time of

a service). Ensuring the desired performance of service-based applications, especially the network performance between the involved services, becomes increasingly important.

In general, service-based applications involve numerous distributed and complex services which usually require more computing resources beyond single machine capability. Therefore, a cluster of networked machines or cloud computing platforms (e.g., Amazon EC2 (Amazon EC2: https://aws.amazon.com), Microsoft Azure (Microsoft Azure: https://azure.microsoft.com), or Google Cloud Platform (Google Cloud Platform: https://cloud.google.com)) are generally leveraged to run service-based applications. More importantly, containers are emerging as the disruptive technology for effectively encapsulating runtime contexts of software components and services, which significantly improves portability and efficiency of deploying applications in clouds. When deploying a service-based application in clouds, several essential aspects have to be taken into account. First, services involved in the application often have diverse resource demands, such as CPU, memory and disk. The underlying machine has to ensure sufficient resources to run each service and at the same time provide cohesive functionalities. Efficient resource allocation to each service is difficult, while it becomes more challenging when the cluster consists of heterogeneous machines. Second, services involved in the application often have traffic demands among them due to data communication, which require meticulous treatment. Poor handling of the traffic demands can result in severe performance degradation, as the response time of a service is directly affected by its traffic situation. Considering the traffic demands, an intuitive solution is to place the services that have large traffic demands among them on the same machine, which can achieve intra-machine communication and reduce inter-machine traffic. However, such services cannot all be co-located on one machine due to limited resource capacities. Hence, placement of service-based applications is quite complicated in clouds. In order to achieve a desired performance of a service-based application, cluster schedulers have to carefully place each service of the application with respect to the resource demands and traffic demands.

Recent cluster scheduling methods mainly focus on the cluster resource efficiency or job completion time of batch workloads [1–3]. For instance, Tetris [4], a multi-resource cluster scheduler, adapts heuristics for the multi-dimensional bin packing problem to efficiently pack tasks on multi-resource cluster. Firmament [5], a centralized cluster scheduler, can make high-quality placement decisions on large-scale clusters via a min-cost max-flow optimization. Unfortunately, these solutions would face difficulties for handling service-based applications, as they ignore the traffic demands when making placement decisions. Some other research works [6,7] concentrate on composite Software as a service (SaaS) placement problem, which try to minimize the total execution time for composite SaaS. However, they only focus on a set of predefined service components for the application placement. For traffic-aware scheduling, relevant research solutions [8,9] are proposed to handle virtual machine (VM) placement problem, which aims to optimize network resource usage over the cluster. However, these solutions rely on a certain network topology, while most of existing cluster schedulers are agnostic to network topology. In particular, it is hard to get the network topology information when deploying a service-based application on a virtual infrastructure.

In this paper, we propose a new approach to optimizing the placement of service-based applications in clouds. The objective is to minimize the inter-machine traffic while satisfying multi-resource demands for service-based applications. Our approach involves two key steps: (1) The requested application is partitioned into several parts while keeping overall traffic between different parts to a minimum. (2) The parts in the partition are packed into machines with multi-resource constraints. Typically, the partition can be abstracted as a minimum k-cut problem; the packing can be abstracted as a multi-dimensional bin packing problem. However, both are NP-hard problems [10,11]. To address these problems, we first propose two partition algorithms: *Binary Partition* and *K Partition*, which are based on a well designed randomized contraction algorithm [12], for finding a high quality application partition. Then, we propose a packing algorithm, which adopts an effective packing heuristic with traffic awareness, for efficiently packing each part of an application partition into machines. Finally, we combine the partition and packing algorithm with a resource demand threshold

to find an appropriate placement solution. We implement a prototype scheduler based on our proposed algorithms and evaluate it on testbed clusters. The results show that our scheduler outperforms existing container cluster schedulers and representative heuristics, leading to much less overall inter-machine traffic.

## 2. Problem Formulation

In this section, we formulate the placement problem of service-based application, and introduce the objective of this work. The notation used in the work is presented in Table 1.

**Table 1.** Notation and Description.

| Notation | Description |
|---|---|
| $\mathcal{M}$ | Set of heterogeneous machines in the cluster: $\mathcal{M} = \{m_1, m_2, ..., m_M\}$ |
| $M$ | Number of the machines: $M = \|\mathcal{M}\|$ |
| $\mathcal{R}$ | Set of resource types: $\mathcal{R} = \{r_1, r_2, ..., r_R\}$ |
| $R$ | Number of the resource types: $R = \|\mathcal{R}\|$ |
| $V_i$ | Vector of available resources on machine $m_i$: $V_i = (v_i^1, v_i^2, ..., v_i^R)$ |
| $v_i^j$ | Amount of resource $r_j$ available on machine $m_i$ |
| $\mathcal{S}$ | A service-based application which is composed by a set of services: $\mathcal{S} = \{s_1, s_2, ..., s_N\}$ |
| $N$ | Number of services in the application: $N = \|\mathcal{S}\|$ |
| $D_i$ | Vector of resource demands of service $s_i$: $D_i = (d_i^1, d_i^2, ..., d_i^R)$ |
| $d_i^j$ | Amount of resource $r_j$ that service $s_i$ demands |
| $\mathbb{T}$ | Matrix of communication traffic between services: $\mathbb{T} = [t_{ij}]_{N \times N}$ |
| $t_{ij}$ | Traffic rate from service $s_i$ to service $s_j$ |
| $\mathbb{X}$ | A placement solution: $\mathbb{X} = [x_{ij}]_{N \times M}$, where $x_{ij} = 1$ if service $s_i$ is to be placed on machine $m_j$, otherwise $x_{ij} = 0$ |

### 2.1. Model Description

We consider a cloud computer cluster is composed of a set of heterogeneous machines $\mathcal{M} = \{m_1, m_2, ..., m_M\}$, where $M = |\mathcal{M}|$ is the number of machines. We consider $R$ types of resources $\mathcal{R} = \{r_1, r_2, ..., r_R\}$ (e.g., CPU, memory, disk, etc.) in each machine. For machine $m_i$, let $V_i = (v_i^1, v_i^2, ..., v_i^R)$ be the vector of its available resources, where the element $v_i^j$ denotes the amount of resource $r_j$ available on machine $m_i$.

In infrastructure as a service (IaaS) model or container as a service (e.g., Amazon ECS) model, users would specify the resource demands of VMs or containers (e.g., a combination of CPU, memory, and storage) when submitting deployment requests. Thus, the resource demands are known upon the arrival of service requests. We consider a service-based application is composed of a set of services $\mathcal{S} = \{s_1, s_2, ..., s_N\}$ that are to be deployed on the cluster, and $N = |\mathcal{S}|$ is the number of services. For service $s_i$, let $D_i = (d_i^1, d_i^2, ..., d_i^R)$ be the vector of its resource demands, where the element $d_i^j$ denotes the amount of resource $r_j$ that the service $s_i$ demands. Let matrix $\mathbb{T} = [t_{ij}]_{N \times N}$ denote the traffic between services, where $t_{ij}$ denotes the traffic rate from service $s_i$ to service $s_j$.

We model a placement solution as a 0–1 matrix $\mathbb{X} = [x_{ij}]_{N \times M}$. if service $s_i$ is to be deployed on machine $m_j$, it is $x_{ij} = 1$. Otherwise, it is $x_{ij} = 0$.

### 2.2. Objective

To achieve a desired performance of service-based applications, a scheduler should not only consider the multi-resource demands of services but also the traffic situation between services.

As services, especially data-intensive services, often need to transfer data frequently, the network performance would directly influence the overall performance. Considering the network dynamics, the placement of different services of an application is crucial for maintaining the overall performance, particularly when unexpected network latency or congestion occurs in the cluster. Given the traffic situation, the most intuitive solution is to place the services that have high traffic rate among them on the same machine so that the co-located services can leverage the loopback interface to get a high network performance without consuming actual network resources of the cluster. However, such services cannot all be co-located on one machine due to limited resource capacities. Thus, with the resource constraints, we try to find a placement solution to minimize the overall traffic between services that are placed on different machines (inter-machine traffic) while satisfying multi-resource demands of services, so that the objective of this work can be formulated as:

$$\textbf{Minimize} \quad \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{p=1}^{M}\sum_{\substack{q=1\\q\neq p}}^{M} t_{ij}\cdot x_{ip}\cdot x_{jq} \tag{1}$$

$$\textbf{Subject to:} \quad \sum_{j=1}^{M} x_{ij} = 1 \ (\forall i \in \{1,2,...,N\}) \tag{2}$$

$$\sum_{i=1}^{N} x_{ij}\cdot d_i^k \leq v_j^k \ (\forall j \in \{1,2,...,M\}, \forall k \in \{1,2,...,R\}) \tag{3}$$

$$x_{ij} \in \{0,1\} \ (\forall i \in \{1,2,...,N\}, \forall j \in \{1,2,...,M\}) \tag{4}$$

Equation (2) guarantees that each service is placed on a machine. Equation (3) guarantees that resource demands on a machine do not exceed its resource capacities. Equation (1) expresses the goal of this work.

## 3. Minimum K-Cut Problem

As a service-based application typically cannot be placed on one machine, an effective partition of the set of services involved in the application is necessary during the deployment. After partition, each subset of the services should be able to be packed into a machine, which means the machine has sufficient resources to run all the services in the subset. Considering the traffic rate between different services, the quality of the partition is crucial for the application performance. To tackle this problem, we first discuss the minimum k-cut problem to understand the problem's complexity.

Let $G = (V, E)$ be an undirected graph, where $V$ is the node set, and $E$ is the edge set. In the graph, each edge $e_{u,v} \in E$ has a non-negative weight $w_{u,v}$. A k-cut in graph $G$ is a set of edges, which when removed, partition the graph into $k$ disjoint nonempty components $G' = \{G_1, G_2, ..., G_k\}$. The minimum k-cut problem is to find a k-cut of minimum total weight of edges whose two ends are in different components, which can be computed as:

$$\sum_{i=1}^{k-1}\sum_{j=i+1}^{k-1}\sum_{\substack{u\in G_i\\v\in G_j}} w_{u,v} \tag{5}$$

A minimum cut is a simply minimum k-cut when $k = 2$. Figure 1 shows an example of a minimum cut of a graph. There are 2 cuts shown in the figure, and the dash line is a minimum cut of the graph, as the total weight of edges cut by the dash line is the minimum of all cuts. Given a service-based application, we can represent it as a graph, where the nodes represent services and the weights of edges represent the traffic rate. Specifically, the traffic rate from service $s_i$ to service $s_j$ and the rate from service $s_j$ to service $s_i$ are represented as two edges respectively in the graph. Hence, finding a minimum k-cut of the graph is equivalent to partitioning the application into $k$ parts while keeping

overall traffic between different parts to a minimum. However, for arbitrary $k$, the minimum k-cut problem is NP-hard [10].

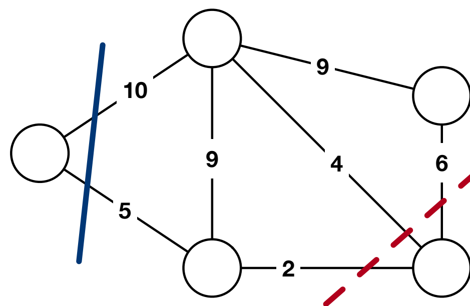



**Figure 1.** An example of a minimum cut (dash line).

Different from developing a deterministic algorithm, Karger's algorithm [12] provides an efficient randomized approach to find a minimum cut of a graph. The basic idea of the Karger's algorithm is to randomly choose an edge $e_{u,v}$ from the graph with probability proportional to the weight of edge $e_{u,v}$ and merge the node $u$ and node $v$ into one (called edge contraction). In order to find a minimum cut, the algorithm iteratively contracts the edge which are randomly chosen until two nodes remain. The edges that remain at last are then output by the algorithm. The pseudocode is shown in Algorithm 1.

**Algorithm 1:** Contraction Algorithm ($k = 2$)

**Input:** $G = (V, E)$
**Output:** A cut of $G$
1 **while** $|V| > 2$ **do**
2 choose an edge $e_{u,v}$ with probability proportional to its weight;
3 $G \leftarrow G/e_{u,v}$ ; // contract edge $e_{u,v}$
4 **end**
5 **return** the cut in $G$;

Figure 2 shows an example process of the contraction algorithm ($k = 2$). The algorithm iteratively merges two nodes of the chosen edge, and all other edges are reconnected to the merged node. For a graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, Karger [12] argues that the contraction algorithm returns a minimum cut of the graph with probability $\Omega(1/n^2)$. Therefore, if we perform the contraction algorithm independently $n^2 \log n$ times, we can find a minimum cut with high probability; if we do not get a minimum cut, the probability is less than $\Omega(1/n)$. For minimum k-cut, the contraction algorithm is basically the same, except that it terminates when k nodes remain (change $|V| > 2$ to $|V| > k$ in Algorithm 1) and returns all the edges left in the graph $G$. Similarly, the contraction algorithm returns a minimum k-cut of the graph with probability $\Omega(1/n^{2k-2})$. If we perform the algorithm independently $n^{2k-2} \log n$ times, we can obtain a minimum k-cut with high probability. Regarding the time complexity, the contraction algorithm can be implemented to run in strongly polynomial $O(m\log^2 n)$ time [12].

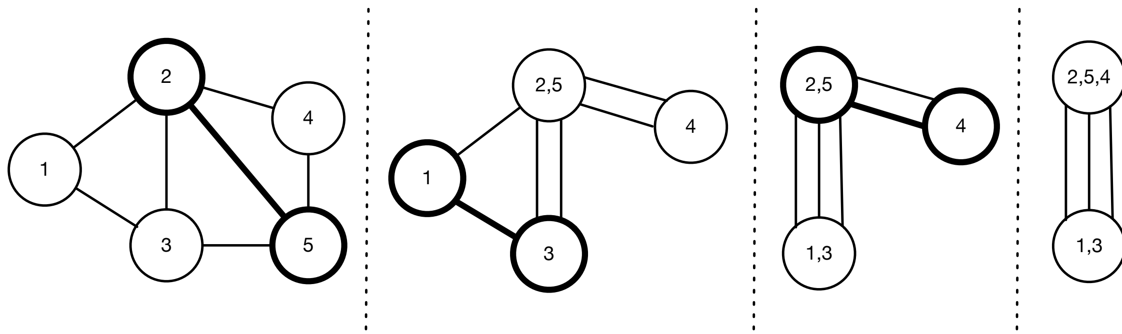



**Figure 2.** An example process of the contraction algorithm ($k = 2$).

## 4. Placement Algorithm

In this section, we describe the algorithms we proposed in this work. The goal of our algorithms is to find a placement solution to minimize inter-machine traffic while satisfying multi-resource demands. The key design of our approach includes: (1) application partition based on contraction algorithms, (2) heuristic packing with traffic awareness, and (3) placement finding with threshold adjustment.

### 4.1. Application Partition

In order to make the values of different resources comparable to each other and easy to handle, we first normalize the amount of available resources on machines and the resources that services demands to be the fraction of the maximum ones. We define the term $v_{max-j}$ to be the maximum amount of available resources $r_j$ on a machine.

$$v_{max-j} = \max_{i \in \{1,2,...,M\}} \left(v_i^j\right) \tag{6}$$

Then the vector $V_i$ of available resources on machine $m_i$ and the vector $D_i$ of resource demands of service $s_i$ are normalized as:

$$V_i = (\frac{v_i^1}{v_{max-1}}, \frac{v_i^2}{v_{max-2}}, ..., \frac{v_i^R}{v_{max-R}}) \tag{7}$$

$$D_i = (\frac{d_i^1}{v_{max-1}}, \frac{d_i^2}{v_{max-2}}, ..., \frac{d_i^R}{v_{max-R}}) \tag{8}$$

After normalization, we start partitioning the service-based application. The key question we ask first is how many parts the application is partitioned into. Considering multi-resource demands of different services, we introduce a threshold $\alpha$ to determine the number of parts when performing partition algorithms. The threshold $\alpha$ denotes the upper bound of the resource demands of partitioned parts, which means we perform partition algorithms continuously until the total resource demands from each part do not exceed $\alpha$ or no part contains more than one service. With a threshold $\alpha \in [0, 1]$ (as the resource demands have been normalized), it assures that each part after partition can be packed into a machine. Figure 3 shows an example of an application partition with threshold $\alpha = 0.5$. In the figure, the total CPU demands and memory demands from each part do not exceed 0.5. Given a threshold $\alpha$, we propose two partition algorithms: *binary partition* and *k partition*, which are based on the contraction algorithm.

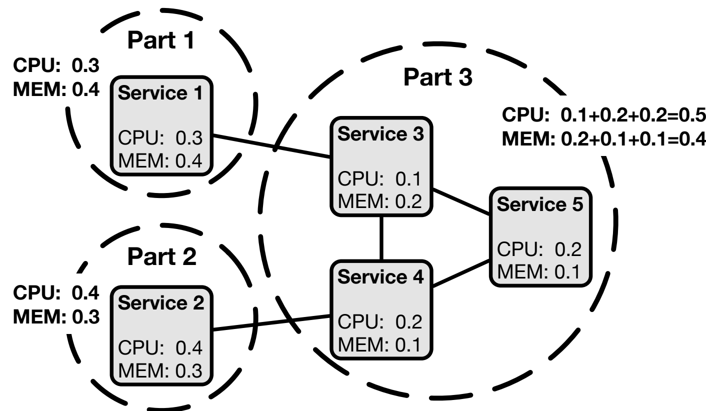



**Figure 3.** An example of an application partition with threshold $\alpha = 0.5$.

#### 4.1.1. Binary Partition

The idea of the binary partition algorithm is to continuously perform binary partition on the application until the resource demands from each part do not exceed $\alpha$ or no part contains more than one service. The pseudocode is shown in Algorithm 2. The basic process can be described as follows. The algorithm continuously checks the resource demands of each part in current application partition $P$. The initial partition is $P = \{\mathcal{S}\}$ where the entire application is treated as one part. If the total resource demands of a part $\mathcal{S}_i$ in $P$ exceeds the threshold $\alpha$ and part $\mathcal{S}_i$ contains more than one service, the part is selected to be partitioned into 2 parts (binary partition). It first constructs a graph $G = (V, E)$ based on $\mathcal{S}_i$, where the nodes represent services and the weights of edges represent the traffic rate. As mentioned in Section 3, if we repeatedly perform the contraction algorithm many times, we can obtain a minimum cut with high probability. Considering both the partition quality and the partition speed, we choose to perform the contraction algorithm $n$ times in our algorithm (in offline manner, it can be set to run $n^2 \log n$ times to get a minimum cut with high probability). Then, according to the minimum cut $G_{min}$, we get from the contraction algorithm, it partitions the $\mathcal{S}_i$ into two parts $\{\mathcal{S}_x, \mathcal{S}_y\}$. This process would be repeatedly performed until the resource demands from each part do not exceed threshold $\alpha$ or no part contains more than one service.

#### 4.1.2. K Partition

The idea of the k partition algorithm is to directly partition the application into $k$ parts. By iteratively increasing $k$, it terminates when the resource demands from each part do not exceed $\alpha$ or no part contains more than one service. The pseudocode is shown in Algorithm 3. The basic process can be described as follows. The algorithm first constructs a graph $G = (V, E)$ based on the application $\mathcal{S}$ and then continuously checks the resource demands of each part in current application partition $P$ where $P = \{\mathcal{S}\}$ initially. If the total resource demands of a part $\mathcal{S}_i$ in $P$ exceeds the threshold $\alpha$ and part $\mathcal{S}_i$ contains more than one service, it increases $k$, which is the number of partitioned parts. As mentioned in Section 3, in order to obtain a minimum k-cut with high probability, we have to perform the contraction algorithm independently $n^{2k-2} \log n$ times. However, the time complexity increases exponentially with $k$, which is prohibitively high. Thus, we make the time complexity consistent with the binary partition algorithm by sacrificing some probability of finding a minimum k-cut. It also performs the contraction algorithm $n$ times. Then, according to the minimum k-cut $G_{min}$ we get from the contraction algorithm, it partitions the application into $k$ parts $P = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_k\}$. Similarly, this process would be repeatedly performed until the resource demands from each part do not exceed threshold $\alpha$ or no part contains more than one service.

**Algorithm 2:** Binary Partition

```
Input: service-based application S, threshold α
Output: a partition of the application P = {S_1, S_2, ..., S_N'}, N' is number of parts after
        partition
1  P ← {S};
2  while exists part S_i in P that the total resource demands exceed α, and part S_i contains more than one
   service do
3      P ← P − {S_i};
4      Construct a graph G = (V, E) based on S_i;
5      n ← |V|;
6      G_min ← G;
7      t ← 0;
8      repeat
9          Perform the contraction algorithm (k = 2) to get a cut G';
10         G_min ← min(G_min, G') ;                    // Store the smaller cut in G_min
11         t ← t + 1;
12     until t > n;
13     Get a partition {S_x, S_y} of part S_i according to G_min;
14     P ← P ∪ {S_x, S_y};
15 end
16 return P;
```

**Algorithm 3:** K Partition

```
Input: service-based application S, threshold α
Output: a partition of the application P = {S_1, S_2, ..., S_N'}, N' is number of parts after
        partition
1  P ← {S};
2  Construct a graph G = (V, E) based on S;
3  n ← |V|;
4  k ← 1;
5  while exists part S_i in P that the total resource demands exceed α and part S_i contains more than one
   service do
6      G_min ← G;
7      k ← k + 1;
8      t ← 0;
9      repeat
10         Perform the contraction algorithm until k nodes remain to get a k-cut G';
11         G_min ← min(G_min, G') ;                    // Store the smaller k-cut in G_min
12         t ← t + 1;
13     until t > n;
14     Get a partition {S_1, S_2, ..., S_k} of the application S according to G_min;
15     P ← {S_1, S_2, ..., S_k};
16 end
17 return P;
```

### *4.2. Heuristic Packing*

Given a partition of the application, the algorithm here is to pack each part into the heterogeneous machines. Without considering the traffic rate, the problem can be formulated as a classical

multi-dimensional bin packing problem, which is known to be NP-hard [11]. When there are a large amount of services involved in the application, it is infeasible to find the optimal solution in polynomial time. Considering the time complexity and packing quality, we adopt two greedy heuristics in our packing algorithm: *Traffic Awareness* and *Most-Loaded Heuristic*. The algorithm is shown in Algorithm 4.

**Algorithm 4:** Heuristic Packing

**Input:** partition of the application $P = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_{N'}\}$, vectors of available resources on each machine $\{V_1, V_2, ..., V_M\}$

**Output:** a placement solution $X$

1. Calculate vectors of resource demands of each part as: $\{D'_1, D'_2, ..., D'_{N'}\}$;
2. $X \leftarrow [x_{ij} = 0]_{N' \times M}$;
3. **for** $i \leftarrow 1;\ i \leq N';\ i \leftarrow i+1$ **do**
4.     $tf \leftarrow 0; ml \leftarrow 0; y \leftarrow 0$;
5.     **for** $j \leftarrow 1;\ j \leq M;\ j \leftarrow j+1$ **do**
6.         **if** *part* $\mathcal{S}_i$ *can be packed into machine* $m_j$ **then**
7.             $tf_j \leftarrow \sum t_{uv}$;<br>            `/* Calculate the total traffic rates between part` $\mathcal{S}_i$ `and machine` $m_j$`, for any service` $s_u$ `in` $\mathcal{S}_i$ `and any service` $s_v$ `packed into machine` $m_j$ `before */`
8.             $ml_j \leftarrow \sum_{k=1}^{R} \frac{d'^k_i}{v^k_j}$;<br>            `/* Calculate the load situation between the vector of resource demands from part` $\mathcal{S}_i$ `and the vector of available resources on machine` $m_j$ `*/`
9.             **if** $tf_j > tf$ **then**
10.                 $tf \leftarrow tf_j; ml \leftarrow ml_j; y \leftarrow j$;
11.             **end**
12.             **else if** $tf_j == tf$ *and* $ml_j > ml$ **then**
13.                 $tf \leftarrow tf_j; ml \leftarrow ml_j; y \leftarrow j$;
14.             **end**
15.         **end**
16.     **end**
17.     **if** $y == 0$ **then**
18.         **return** *null*;
19.     **end**
20.     **else**
21.         $V_y \leftarrow V_y - D'_i$;
22.         $x_{iy} \leftarrow 1$;
23.     **end**
24. **end**
25. **return** $X$;

In order to find a best possible machine for part $\mathcal{S}_i$ , the algorithm calculates two matching factors: $tf$ and $ml$. For machine $m_j$, the factor $tf$ is the sum of the traffic rate between the services in part $\mathcal{S}_i$, and the services have been determined to be packed into machine $m_j$ before. The factor $ml$ is a scalar value of the load situation between the vector of resource demands from part $\mathcal{S}_i$ and the vector of available resources on machine $m_j$. Assuming $D'_i$ is the resource demand vector of part $\mathcal{S}_i$ and $d'^k_i$ is

the amount of resource $r_k$ part $\mathcal{S}_i$ demands, it is $ml = \sum_{k=1}^{R} \frac{d'^k_i}{v^k_j}$. The higher $ml$ is, the more loaded the machine. The idea of this heuristic is to improve the resource efficiency by packing the part to the most loaded machine. As our main goal is to minimize the inter-machine traffic, the algorithm is designed to first prioritize the machines based on the factors of $tf$. If the factors of $tf$ are the same, it then prioritizes the machines based on the factors of $ml$. Consequently, if all parts in the partition can be packed into machines, the algorithm returns the placement solution. Otherwise, it returns $null$.

### *4.3. Placement Finding*

As we discussed before, in order to partition the application, the threshold $\alpha$ is required by the algorithm. However, giving an appropriate deterministic threshold $\alpha$ is difficult, as it cannot guarantee that the algorithm can find a placement solution through the randomized partition and the heuristic packing under a certain threshold $\alpha$. Intuitively, the higher threshold $\alpha$ results in less parts in the partition, which leads to less traffic rate between different parts. Thus, we introduce a simple algorithm to find a better threshold $\alpha$ by enumerating from large to small. The algorithm is shown in Algorithm 5. At the beginning, the value of $\alpha$ is 1.0. To adjust the thresholds, we set a step value $\Delta$, and the default value is 0.1, which can be customized by users. In each iteration, with the threshold $\alpha$, the algorithm first partitions the given application $\mathcal{S}$ based on the binary partition algorithm or k partition algorithm. Note that the algorithm records the latest partition results to avoid multiple repeated partition. It then tries to pack all parts in the partition into machines based on the heuristic packing algorithm to find a placement solution for the application.

**Algorithm 5:** Placement Finding

**Input:** service-based application $\mathcal{S}$, vectors of available resources on each machine $\{V_1, V_2, ..., V_M\}$

**Output:** a placement solution $X$

```
1  X ← [x_ij = 0]_{N×M};
2  α ← 1.0;
3  Δ ← 0.1;
4  while α ≥ 0.0 do
5  |   P ← Binary_Partition(S, α);
   |   /* Or P ← K_Partition(S, α);                    */
6  |   X' ← Heuristic_Packing(P, {V_1, V_2, ..., V_M});
7  |   if X' ≠ null then
8  |   |   Calculate X according to X' and P;
9  |   |   return X;
10 |   end
11 |   α ← α − Δ;
12 end
13 return null;
```

Next, we discuss the time complexity of the algorithm we proposed. We assume the number of services is $n$; the number of edges in the service graph is $m$ (i.e., the number of the traffic rates $t_{ij} > 0$); the number of machines is $M$. For a service-based application, it can be partitioned up to $n$ parts. For each partition, we perform the contraction algorithm $n$ times, and the time complexity of the contraction algorithm is $O(m\log^2 n)$. As we record the latest partition results to avoid multiple repeated partition, the time complexity of the overall partition is $O(n^2 m\log^2 n)$. To the heuristic packing, the time complexity is $O(nM + n^2)$ as the overall time complexity of calculating the factor $tf$ is $O(n^2)$. Let $C = \frac{1}{\Delta}$ denote the number of iterations. The overall time complexity of the proposed algorithm is $(n^2 m\log^2 n + CnM + Cn^2)$.

## 5. Evaluation

We implement a prototype scheduler using python, which is based on our proposed algorithms, for deploying service-based applications on container clusters. In the experiments, we evaluate our scheduler in testbed clusters of ExoGENI [13] experimental environment.

### 5.1. Experimental Methodology

**Cluster.** We create two different testbed clusters in ExoGENI for experiments. For the first cluster, we use 30 homogeneous VMs with 2 CPU cores and 6 GB RAM. Considering the heterogeneity, we use 10 VMs with 2 CPU cores and 6 GB RAM and 10 VMs with 4 CPU cores and 12 GB RAM for the second cluster. The homogeneous cluster has 30 VMs, and the heterogeneous cluster has 20 VMs, but the total resource capacity is the same.

**Workloads.** In order to evaluate the proposed algorithms in different scenarios, we use synthetic applications in the experiments. Considering the scale of the testbed cluster, we yield service-based applications which are composed by 64, 96, and 128 services. For the size of 64, the CPU demand of each service is uniformly picked at random from [30,100] where 100 represents 1 CPU core, and the memory demand is picked at random from [100,300] where 100 represents 1 GB RAM. For the size of 96, the CPU demand is picked at random from [20,67], and the memory demand is picked at random from [67,200]. For the size of 128, the CPU demand is picked at random from [15,50], and the memory demand is picked at random from [50,150]. According to these ranges, the total resource demands of different application sizes are roughly the same. For each application size, we generate 10,000 instances for testing. As the work [14] shows that the log-normal distribution produces the best fit to the data center traffic, we choose to generate the traffic demands between services with the probability 0.05 (ensure that application graph is connected), and the traffic rate follows a log-normal distribution (mean = 5 Mbps, standard deviation = 1 Mbps).

**Implementation.** We implement all proposed algorithms in our prototype scheduler, where the contraction algorithm is based on the parallel implementation [12]. As we proposed two algorithms for application partition, there are two kinds of configuration. BP-HP is based on binary partition (BP) and heuristic packing (HP). KP-HP is based on k partition (KP) and heuristic packing.

**Baselines.** As we mentioned above, many research efforts have been devoted to the composite Software as a service (SaaS) placement problem [6,15]. However, they target at the placement for a certain set of predefined service components. More importantly, these metaheuristic based approaches often take minutes or even hours, particularly for large-scale clusters, to generate a placement solution, which would face difficulties for an online response. Another research work focuses on traffic-aware VM placement problem [16,17]. However, these solutions rely on a certain network topology, while our approach is agnostic to network topology. Thus, we choose to compare our scheduler with the following schemes:

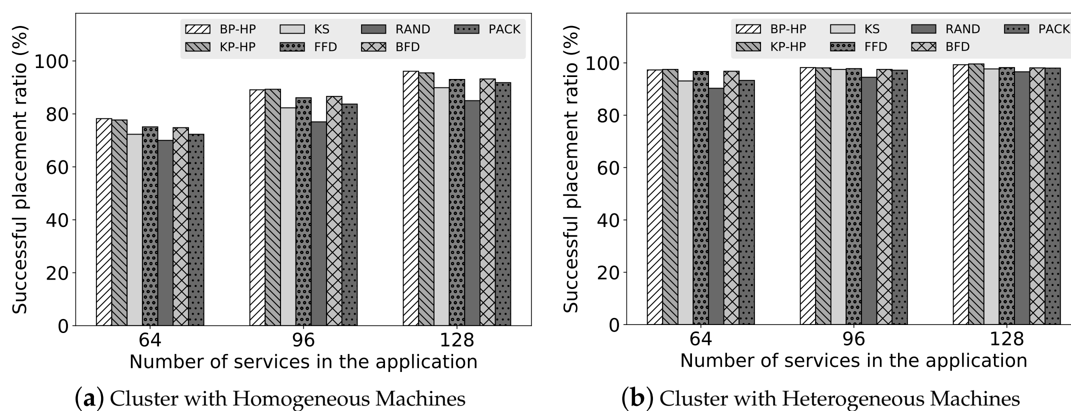
- Kubernetes Scheduler (KS): the default scheduler in Kubernetes [18] container cluster tends to distribute containers evenly across the cluster to balance the overall cluster resource usage. Specifically, we add a soft affinity (i.e., pod affinity in Kubernetes) to the services that have traffic between them, as the scheduler would try to place the services which have affinity between them on the same machine.
- First-Fit Decreasing (FFD): it is a simple and commonly adopted algorithm for the multi-dimensional bin packing problem [19]. FFD operates by first sorting the services in decreasing order according to a certain resource demand and then packs each service into the first machine with sufficient resources.
- Best-Fit Decreasing (BFD): it places a service in the fullest machine that still has enough capacity. BFD operates by first sorting the machines in decreasing order according to a certain resource capacity and then packs each service into the first machine with sufficient resources.
- Multi-resource Packer (PACK): the idea of this heuristic [4] is that it schedules the services in increasing order of alignment between resource demands of services and resource availability of

machines (i.e., dot product between the vector of resource demands and the vector of available resources).

- Random (RAND): it randomly picks a service in the application and then packs it into the first machine with sufficient resources.

*5.2. Comparison with Baselines*

Figure 4 shows the successful placement ratio of different schemes over two clusters. The successful placement of an application is that the algorithm can find a placement solution to place all the involved services, so the ratio is the number of successfully placed applications to the number of all requested applications. We observe that RAND performs worst, as it has no heuristic to pack the services. FFD and BFD perform better than KS and PACK because KS mainly focuses on balancing the resource utilization over the cluster, and PACK focuses on the alignment between resource demands and resource availability. FFD and BFD have been demonstrated as effective algorithms for multi-dimensional bin packing problems [20]. BP-HP performs comparably to KP-HP, and they both slightly outperform other schemes in this evaluation. This is mainly because the iterative partition and packing with different thresholds improve the probability of finding a placement solution. Moreover, the packing algorithm can pack services tightly due to the most-loaded heuristic. The results of the homogeneous cluster also show that the successful placement ratio increases when the number of services increases. As the total resource demands of the applications in different sizes (different number of services) are roughly the same, the less number of services results in larger resource demands of each individual service, which easily causes the resource fragmentation problem in the placement. Compared to the homogeneous cluster, the successful placement ratio is much higher in the heterogeneous cluster. As the machines have larger resource capacity in the heterogeneous cluster, it is easier to pack services constrained by multiple resources.

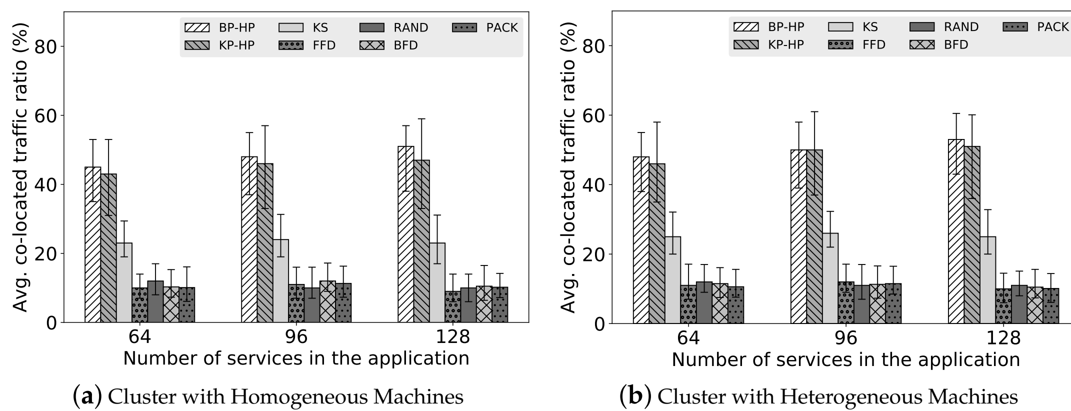



(**a**) Cluster with Homogeneous Machines (**b**) Cluster with Heterogeneous Machines

**Figure 4.** Comparison of successful placement ratio of different schemes.

Next, we evaluate the traffic situation of different schemes. In the evaluation, we only compare the applications whose all services are placed on the cluster by different algorithms. Figure 5 shows the average co-located traffic ratio of different schemes, and the error bars represent the maximum and minimum ratio. The co-located traffic is the traffic between the services that are placed on the same machine, so the ratio is the amount of co-located traffic to the amount of all traffic. For minimizing inter-machine traffic, the higher the co-located traffic ratio is, the better the placement solution is. To be specific, we present the co-located traffic ratio in Table 2. We observe that BP-HP and KP-HP significantly outperform the baselines. For the cluster with homogeneous machines, BP-HP improves average co-located traffic ratio by 24.8% to 38.1%; KP-HP improves the ratio by 22% to 35.3%. For the cluster with heterogeneous machines, BP-HP improves average co-located traffic ratio by 24.7% to 39.6%; KP-HP improves the ratio by 23.4% to 38.3%. FFD, BFD, PACK, and RAND perform poorly as they only focus on packing the services, without considering the traffic rate. As we set the affinity to the

services that have traffic between them in KS, KS tries to put the affinity services on the same machine. However, KS ignores the concrete traffic rate when making placement decisions. Regarding BP-HP and KP-HP, we find that BP-HP performs slightly better and more stable than KP-HP, but KP-HP may find a better solution in some cases (according to the error bars). In contrast, KP-HP also easily returns a worse solution. This is mainly because BP-HP performs the contraction algorithm to find a minimum cut with probability $\Omega(1/n^2)$; KP-HP performs the contraction algorithm to find a minimum k-cut with probability $\Omega(1/n^{2k-2})$ which is much less than the BP-HP. Thus, the performance of KP-HP varies widely in the experiments. Nevertheless, benefiting from the partition that strives to co-locate the large traffic demands and the traffic-aware packing, BP-HP and KP-HP both can effectively reduce inter-machine traffic for deploying service-based applications on computer clusters.




(**a**) Cluster with Homogeneous Machines (**b**) Cluster with Heterogeneous Machines

**Figure 5.** Comparison of average co-located traffic ratio of different schemes.

**Table 2.** Co-located traffic ratio (%) of different schemes.

| Scheme | Homogeneous Cluster | | | Heterogeneous Cluster | | |
|---|---|---|---|---|---|---|
| | Avg. | Min. | Max. | Avg. | Min. | Max. |
| BP-HP | 48.1 | 35.2 | 56.9 | 50.3 | 37.9 | 60.5 |
| KP-HP | 45.3 | 30.8 | 59.1 | 49.0 | 35.2 | 61.1 |
| KS | 23.3 | 17.1 | 31.4 | 25.6 | 20.4 | 32.8 |
| FFD | 10.0 | 6.5 | 15.8 | 10.9 | 6.2 | 17.1 |
| BFD | 10.9 | 6.4 | 17.2 | 11.1 | 7.3 | 16.6 |
| PACK | 10.5 | 6.1 | 16.3 | 10.7 | 6.1 | 16.5 |
| RAND | 10.6 | 6.2 | 16.7 | 11.0 | 6.8 | 16.8 |

### *5.3. Impact of Threshold $\alpha$*

In this section, we discuss the impact of threshold $\alpha$ on the service-based application placement. To illustrate, we fix the threshold $\alpha$ by using BP-HP on the cluster with homogeneous machines. Figure 6 shows the successful placement ratio with different values of threshold $\alpha$. For instance, BP-HP can find a placement solution for 77% of the applications with 64 services when $\alpha = 0.5$. We observe that the successful placement ratio decreases when the value of threshold $\alpha$ increases in general, and few applications can be successfully placed when $\alpha > 0.7$. Higher threshold $\alpha$ leads to less parts and larger average resource demands of parts in the partition, so it becomes harder to pack them into machines with multi-resource constraints. To understand the impact on the network traffic, Figure 7 shows the results of average co-located traffic ratio for each value of threshold $\alpha$, and the error bars represent the maximum and minimum ratio. It explicitly demonstrates that the co-located traffic ratio increases more when $\alpha$ is larger. However, larger threshold $\alpha$ increases the difficulty of packing the applications. Thus, we try to find an appropriate threshold $\alpha$ by enumerating from large to small in the proposed algorithms.

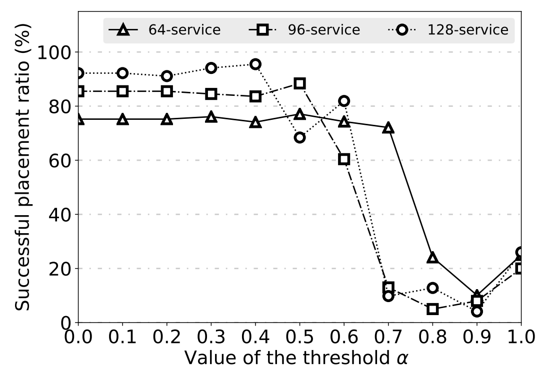



**Figure 6.** Successful placement ratio on the homogeneous cluster by using BP-HP with different values of threshold $\alpha$.

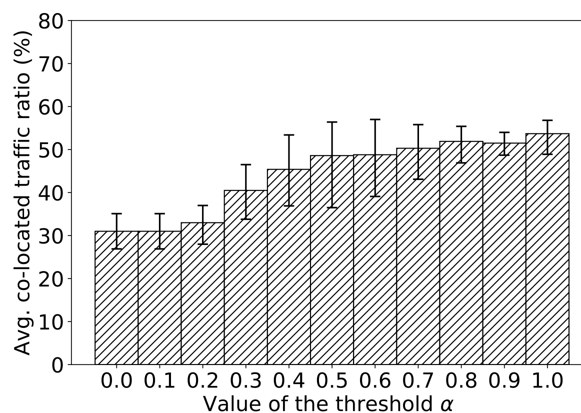



**Figure 7.** Average co-located traffic ratio on the homogeneous cluster by using BP-HP with different values of threshold $\alpha$.

### 5.4. Overhead Evaluation

In this section, we evaluate the overhead by measuring the algorithm runtime and compare it with KS and RAND. In order to fairly compare the algorithm runtime, we also implement the scheduling algorithm of KS in Python, which is the same with other schemes. We conduct this experiment on a dedicated server with Intel Xeon E5-2630 2.4 GHz CPU and 64 GB memory. Figure 8 shows the results of the average algorithm runtime of different schemes for the heterogeneous cluster (the homogeneous cluster is similar), and the error bars represent the maximum and minimum algorithm runtime. RAND incurs little overhead, as it is a very simple algorithm. Compared with RAND, KS is a bit complex, as KS has multiple predicated policies and priorities policies to filter and score machines, such as handling the affinity between services. BP-HP and KP-HP are more complicated than the baselines, and have obviously higher overhead. We also observe that the difference between the maximum and minimum algorithm runtime is quite large, as the algorithm runtime heavily depends on the value of threshold $\alpha$. In the algorithm, higher threshold $\alpha$ results in less iterations, and lower threshold $\alpha$ causes more iterations. Nevertheless, BP-HP and KP-HP can respond in seconds for different application sizes. Especially, for the application with less than 100 services, BP-HP and KP-HP can respond in sub-second time, which is acceptable for online scheduling. Moreover, the most time consuming part of the proposed algorithms is application partition, which means there would be no big difference of the algorithm runtime for large-scale clusters with the same number of services. We believe that the proposed algorithms can also effectively handle the placement problem on large-scale clusters.

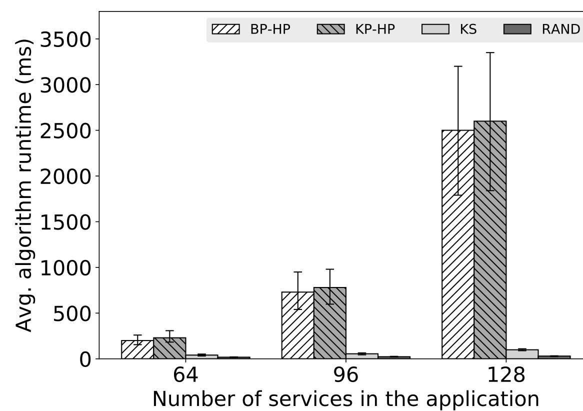



**Figure 8.** Average algorithm runtime of different schemes for the heterogeneous cluster.

## 6. Related Work

As the microservice architecture is emerging as a primary architectural style choice in the service oriented software industry [21], many research efforts have been devoted to the analysis and modeling of microservice architecture [22–24]. Leitner et al. [25] proposed a graph-based cost model for deploying microservice-based applications on a public cloud. Balalaie et al. [26] presented their experience and lessons on migrating a monolithic software architecture to microservices. Amaral et al. [27] evaluated the performance of microservices architectures using containers. However, the performance of service placement schemes received little attention in these works.

Software as a Service (SaaS) is one of the most important services offered by cloud providers, and many works have been proposed for optimizing composite SaaS placement in cloud environments [15]. Yusoh et al. [6] propose a genetic algorithm for the composite SaaS placement problem, which considers both the placement of the software components of a SaaS and the placement of data of the SaaS. It tries to minimize the total execution time of a composite SaaS. Hajji et al. [7] adopt a new variation of PSO called Particle Swarm Optimization with Composite Particle (PSO-CP) to solve the composite SaaS placement problem. It considers not only the total execution time of the composite SaaS but also the performance of the underlying machines. Unfortunately, they target at the placement for a certain set of predefined service components, which has limitations to handle a large number of different services. In addition, plenty of research has been proposed to optimize service placement in edge and fog computing [28,29]. Mennan et al. [30] proposed a service placement heuristic to maximize the bandwidth allocation when deploying community networks micro-clouds. It uses the information of network bandwidth and node availability to optimize service placement. Different from it, we consider the constraints of multiple resources rather than just network bandwidth to minimize the inter-machine traffic while satisfying multi-resource demands of service-based applications. Carlos et al. [31] presented a decentralized algorithm for the placement problem to optimize the distance between the clients and the most requested services in fog computing. They assume there are unlimited resources in cloud computing and try to minimize the hop count by placing the most popular services as closer to the users as possible. In contrast, our work focuses on the overall network usage of the cloud underlying cluster, which is modeled as a set of heterogeneous machines.

In recent years, a number of research works have been proposed in the area of VM placement with traffic awareness for cloud data centers [16,17]. Meng et al. [8] analyze the impact of data center network architectures and traffic patterns and propose a heuristic approach to reduce the aggregate traffic when placing VM into the data center. Wang et al. [9] formulate the VM placement problem with dynamic bandwidth demands as a stochastic bin packing problem and propose an online packing algorithm to minimize the number of machines required. However, they only focus on optimizing the network traffic in the data center, without considering the highly diverse resources requirements of the virtual machines. Biran et al. [32] proposed a placement scheme to satisfy the traffic demands of the VMs while meeting the CPU and memory requirements. Dong et al. [33] introduced a

placement solution to improve network resource utilization in addition to meeting multiple resource constraints. They both rely on a certain network topology to make placement decisions. Besides, many research efforts have been devoted to the scheduling and partitioning on heterogeneous systems [34,35]. Different from them, our work is agnostic to the underlying network topology, which aims to minimize the overall inter-machine traffic on the cluster.

## 7. Conclusions

In this paper, we investigated service placement problem for microservice architecture in clouds. In order to find a high quality partition of service-based applications, we propose two partition algorithms: *Binary Partition* and *K Partition*, which are based on a well designed randomized contraction algorithm. For efficiently packing the application, we adopt most-loaded heuristic and traffic awareness in the packing algorithm. By adjusting the threshold $\alpha$ which denotes the upper bound of the resource demands, we can find a better placement solution for service-based applications. We implement a prototype scheduler based on our proposed algorithms and evaluate it on testbed clusters. In the evaluation, we show that our algorithms can improve the ratio of successfully placing applications on the cluster while significantly increasing the ratio of co-located traffic (i.e., reducing the inter-machine traffic). In the overhead evaluation, the results show that our algorithms incur some overhead but in an acceptable time. We believe that the proposed algorithms are practical for realistic use cases. In the future, we will investigate problem-specific optimizations to improve our implementation and consider resource dynamics in the placement to adapt more sophisticated situations.

**Author Contributions:** Conceptualization, Y.H.; methodology, Y.H.; software, Y.H.; formal analysis, Y.H.; investigation, Y.H. and Z.Z.; resources, C.d.L.; writing—original draft preparation, Y.H. and Z.Z.; writing—review and editing, Y.H.; supervision, Z.Z. and C.d.L.

**Funding:** This research has received funding from the European Union's Horizon 2020 research and innovation program under grant agreements 643963 (SWITCH project), 654182 (ENVRIplus project), 676247 (VRE4EIC project), 824068 (ENVRI-FAIR project) and 825134 (ARTICONF project). The research is also supported by the Chinese Scholarship Council.

**Acknowledgments:** The authors thank the anonymous reviewers for their thoughtful comments on this paper.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---|---|
| SaaS | Software as a Service |
| VM | Virtual Machine |
| BP | Binary Partition |
| KP | K Partition |
| HP | Heuristic Packing |
| KS | Kubernetes Scheduler |
| FFD | First-Fit Decreasing |
| RAND | Random |

## References

1. Hu, Y.; Wang, J.; Zhou, H.; Martin, P.; Taal, A.; de Laat, C.; Zhao, Z. Deadline-Aware Deployment for Time Critical Applications in Clouds. In Proceedings of the European Conference on Parallel Processing, Santiago de Compostela, Spain, 28 August–1 September 2017; Springer: New York, NY, USA, 2017; pp. 345–357.
2. Koulouzis, S.; Martin, P.; Zhou, H.; Hu, Y.; Wang, J.; Carval, T.; Grenier, B.; Heikkinen, J.; de Laat, C.; Zhao, Z. Time-critical data management in clouds: Challenges and a Dynamic Real-Time Infrastructure Planner (DRIP) solution. *Concurr. Comput. Pract. Exp.* **2019**, e5269. [CrossRef]

3. Hu, Y.; Zhou, H.; de Laat, C.; Zhao, Z. Ecsched: Efficient container scheduling on heterogeneous clusters. In Proceedings of the European Conference on Parallel Processing, Turin, Italy, 27–31 August 2018; Springer: New York, NY, USA, 2018; pp. 365–377.
4. Grandl, R.; Ananthanarayanan, G.; Kandula, S.; Rao, S.; Akella, A. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Comput. Commun. Rev.* **2015**, *44*, 455–466. [CrossRef]
5. Gog, I.; Schwarzkopf, M.; Gleave, A.; Watson, R.N.; Hand, S. Firmament: Fast, centralized cluster scheduling at scale. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 99–115.
6. Yusoh, Z.I.M.; Tang, M. A penalty-based genetic algorithm for the composite SaaS placement problem in the cloud. In Proceedings of the IEEE Congress on Evolutionary Computation, Barcelona, Spain, 18–23 July 2010; pp. 1–8.
7. Hajji, M.A.; Mezni, H. A composite particle swarm optimization approach for the composite saas placement in cloud environment. *Soft Comput.* **2018**, *22*, 4025–4045. [CrossRef]
8. Meng, X.; Pappas, V.; Zhang, L. Improving the scalability of data center networks with traffic-aware virtual machine placement. In Proceedings of the 2010 IEEE INFOCOM, San Diego, CA, USA, 15–19 March 2010; pp. 1–9.
9. Wang, M.; Meng, X.; Zhang, L. Consolidating virtual machines with dynamic bandwidth demand in data centers. *Infocom* **2011**, *201*, 71–75.
10. Goldschmidt, O.; Hochbaum, D.S. Polynomial algorithm for the k-cut problem. In Proceedings of the 1988 29th Annual Symposium on Foundations of Computer Science, White Plains, NY, USA, 24–26 October 1988; pp. 444–451.
11. Woeginger, G.J. There is no asymptotic PTAS for two-dimensional vector packing. *Inf. Process. Lett.* **1997**, *64*, 293–297. [CrossRef]
12. Karger, D.R. Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm. *SODA* **1993**, *93*, 21–30.
13. Baldin, I.; Chase, J.; Xin, Y.; Mandal, A.; Ruth, P.; Castillo, C.; Orlikowski, V.; Heermann, C.; Mills, J. Exogeni: A multi-domain infrastructure-as-a-service testbed. In *The GENI Book*; Springer: New York, NY, USA, 2016; pp. 279–315.
14. Benson, T.; Anand, A.; Akella, A.; Zhang, M. Understanding data center traffic characteristics. In Proceedings of the 1st ACM Workshop on Research on Enterprise Networking, Barcelona, Spain, 21 August 2009; ACM: New York, NY, USA, 2009; pp. 65–72.
15. Huang, K.C.; Shen, B.J. Service deployment strategies for efficient execution of composite SaaS applications on cloud platform. *J. Syst. Softw.* **2015**, *107*, 127–141. [CrossRef]
16. Alicherry, M.; Lakshman, T. Network aware resource allocation in distributed clouds. In Proceedings of the 2012 IEEE INFOCOM, Orlando, FL, USA, 25–30 March 2012; pp. 963–971.
17. Kliazovich, D.; Bouvry, P.; Khan, S.U. DENS: Data center energy-efficient network-aware scheduling. *Clust. Comput.* **2013**, *16*, 65–75. [CrossRef]
18. Hightower, K.; Burns, B.; Beda, J. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*; O'Reilly Media, Inc.: Champaign, IL, USA, 2017.
19. Ajiro, Y.; Tanaka, A. Improving packing algorithms for server consolidation. In Proceedings of the International CMG Conference, San Diego, CA, USA, 2–7 December 2007; Volume 253.
20. Stillwell, M.; Schanzenbach, D.; Vivien, F.; Casanova, H. Resource allocation algorithms for virtualized service hosting platforms. *J. Parallel Distrib. Comput.* **2010**, *70*, 962–974. [CrossRef]
21. Thönes, J. Microservices. *IEEE Softw.* **2015**, *32*, 116. [CrossRef]
22. Cerny, T.; Donahoo, M.J.; Trnka, M. Contextual understanding of microservice architecture: Current and future directions. *ACM SIGAPP Appl. Comput. Rev.* **2018**, *17*, 29–45. [CrossRef]
23. Hasselbring, W.; Steinacker, G. Microservice architectures for scalability, agility and reliability in e-commerce. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 243–246.
24. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*; Springer: New York, NY, USA, 2017; pp. 195–216.

25. Leitner, P.; Cito, J.; Stöckli, E. Modelling and managing deployment costs of microservice-based cloud applications. In Proceedings of the 9th International Conference on Utility and Cloud Computing, Shanghai, China, 6–9 December 2016; ACM: New York, NY, USA, 2016; pp. 165–174.
26. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Migrating to cloud-native architectures using microservices: An experience report. In Proceedings of the European Conference on Service-Oriented and Cloud Computing, Taormina, Italy, 15–17 September 2015; Springer: New York, NY, USA, 2015; pp. 201–215.
27. Amaral, M.; Polo, J.; Carrera, D.; Mohomed, I.; Unuvar, M.; Steinder, M. Performance evaluation of microservices architectures using containers. In Proceedings of the 2015 IEEE 14th International Symposium on Network Computing and Applications, Cambridge, MA, USA, 28–30 September 2015; pp. 27–34.
28. Wu, Z.; Lu, Z.; Hung, P.C.; Huang, S.C.; Tong, Y.; Wang, Z. QaMeC: A QoS-driven IoVs application optimizing deployment scheme in multimedia edge clouds. *Future Gener. Comput. Syst.* **2019**, *92*, 17–28. [CrossRef]
29. Chen, X.; Tang, S.; Lu, Z.; Wu, J.; Duan, Y.; Huang, S.C.; Tang, Q. iDiSC: A New Approach to IoT-Data-Intensive Service Components Deployment in Edge-Cloud-Hybrid System. *IEEE Access* **2019**, *7*, 59172–59184. [CrossRef]
30. Selimi, M.; Cerdà-Alabern, L.; Freitag, F.; Veiga, L.; Sathiaseelan, A.; Crowcroft, J. A lightweight service placement approach for community network micro-clouds. *J. Grid Comput.* **2019**, *17*, 169–189. [CrossRef]
31. Guerrero, C.; Lera, I.; Juiz, C. A lightweight decentralized service placement policy for performance optimization in fog computing. *J. Ambient Intell. Humaniz. Comput.* **2019**, *10*, 2435–2452. [CrossRef]
32. Biran, O.; Corradi, A.; Fanelli, M.; Foschini, L.; Nus, A.; Raz, D.; Silvera, E. A stable network-aware vm placement for cloud systems. In Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid), Ottawa, ON, Canada, 13–16 May 2012; pp. 498–506.
33. Dong, J.; Jin, X.; Wang, H.; Li, Y.; Zhang, P.; Cheng, S. Energy-saving virtual machine placement in cloud data centers. In Proceedings of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Delft, The Netherlands, 13–16 May 2013; pp. 618–624.
34. Brandolese, C.; Fornaciari, W.; Pomante, L.; Salice, F.; Sciuto, D. Affinity-driven system design exploration for heterogeneous multiprocessor SoC. *IEEE Trans. Comput.* **2006**, *55*, 508–519. [CrossRef]
35. Hu, Y.; Zhou, H.; de Laat, C.; Zhao, Z. Concurrent container scheduling on heterogeneous clusters with multi-resource constraints. *Future Gener. Comput. Syst.* **2020**, *102*, 562–573. [CrossRef]

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).