Joint Optimization of Service Request Routing and Instance Placement in the Microservice System

Yinbo Yu[a], Jianfeng Yang[a,*], Chengcheng Guo[a], Hong Zheng[a], Jiancheng He[b]

[a]*School of Electronic Information, Wuhan University, Wuhan, 430072, China*
[b]*Shenzhen R&D Center, Huawei Technologies. Co., Ltd., Shenzhen, 518000, China*

Abstract

Microservice architecture is a promising architectural style. It decomposes monolithic software into a set of loosely coupled containerized microservices and associates them into multiple microservice chains to serve service requests. The new architecture creates flexibility for service provisioning but also introduces increased energy consumption and low service performance. Efficient resource allocation is critical. Unfortunately, existing solutions are designed at a coarse level for virtual machine (VM)-based clouds and not optimized for such chain-oriented service provisioning. In this paper, we study the resource allocation optimization problem for service request routing and microservice instance placement, so as to jointly reduce both resource usage and chains' end-to-end response time for saving energy and guaranteeing Quality of Service (QoS). We design detailed workload models for microservices and chains and formulate the optimization problem as a bi-criteria optimization problem. To address it, a three-stage scheme is proposed to search and optimize the trade-off decisions, route service requests into instances and deploy instances to servers in a balanced manner. Through numerical evaluations, we show that while assuring the same QoS, our scheme performs significantly better than and faster than benchmarking algorithms on reducing energy consumption and balancing load.

Keywords: Microservice, microservice chain, energy consumption, QoS, load balance, bi-criteria optimization

1. Introduction

As a new architecture style for provisioning services, microservice architecture (also called μService) is currently attracting significant attention. Traditionally, all the modules of an application are packaged and deployed as a monolith. However, this approach suffers from several issues regarding software updates, reliability, and scalability [1]. For instance, to update a small part of an application, the entire application needs to be redeployed. In contrast, μService refactors an application into a set of small, interconnected microservices. Each microservice is self-contained and can be developed, updated and deployed independently. The business logic is implemented by a series of microservices forming a *microservice chain* through remote API calls (e.g., REST or message queue). μService creates flexibility, reliability, and speed of software updating and service delivery. With the advent of container technologies such as Docker[1], the new architecture powers modern cloud applications.

However, along with the enhanced flexibility, come potentially increased hardware resource usage (e.g., CPU and memory) and request processing time due to service decomposition [2]. When we start building a μService system, resource allocation strategies are required to allocate into more microservice instances and also into more complex system management tools (e.g., load balancers and failover software) in such more elaborate system. Careless resource allocation can increase energy consumption. Moreover, a service request is processed across multiple instances through inter-process communication rather than language-level function calls in monolithic applications. Each instance may have unique request processing logic and processing capabilities depending on allocated computing resource. Hence, careless resource allocation, on the other hand, can increase the end-to-end request processing time of microservice chains (hereinafter referred to as *service time*) which is the major QoS concerned by service providers and users.

To reduce energy consumption and improve QoS, a *trade-off resource allocation optimization* problem is involved that more computing resources allocated to microservice instances can reduce service time, but at the cost of higher energy consumption, and vice versa. Even without considering energy consumption, inter-chain resource contention exists for their own performance [3], which can result in unbalanced QoS provided by different chains. Thus, careful resource allocation to avoid over/under-provisioning is deeply needed, but not trivial for μServices since they are being deployed at a signif-

* Corresponding author
Email addresses: yyb@whu.edu.cn (Yinbo Yu), yjf@whu.edu.cn (Jianfeng Yang), netccg@whu.edu.cn (Chengcheng Guo), zh@whu.edu.cn (Hong Zheng), hejiancheng@huawei.com (Jiancheng He)

[1] A popular container technology: https://www.docker.com/.

icant scale (e.g., Uber's application is composed of over 1000 instances) [4]. To this end, a critical problem needs to be addressed: *how can resources be efficiently allocated and balanced not only to reduce energy consumption but also to guarantee high QoS when creating and placing microservice instances and routing service requests?*

Such resource optimization problem for μServices is similar to the one involved in VM-based clouds which has attracted many solutions [5, 6]. Unfortunately, these solutions tend to be limited when being applied to μServices. The major difference between μServices and VM-based clouds is that μServices process users' requests by chain-oriented service provisioning. Thus, the resource optimization for μServices should be governed at the chain level by considering heterogeneous requests and inter-chain resource contentions [3]. Moreover, since vertical scaling for VMs (on-the-fly changing of assigned resources to VMs) is high-cost or not supported [7], few of them consider the detailed workload model inside a VM to scale vertically and they usually only optimize the number of running VMs (horizontal scaling) according to fluctuating workload. Similarly, such horizontal optimization is also followed in existing resource allocation solutions for μServices (e.g., [8, 3]). But container is a more lightweight virtualization technology and can support dynamic resource allocation inside containers with low operation costs [9]. Therefore, for such containerized μServices, efficient resource allocations should be redesigned at a fine-grained level (e.g., CPU cycle) rather than instance level, which has not been well studied.

In this paper, we study the resource allocation optimization for provisioning web services with μServices. We focus on the containerized μService system, in which a microservice *instance* is a container running the microservice. To achieve a fine-grained resource allocation, different from existing works, we model resource usage and system performance at the CPU cycle level in each microservice instance. The optimization problem is formulated as a biobjective optimization problem, with which we jointly optimize resource allocation to reduce both energy consumption and service time for provisioning services. To address this problem, we stratify it into three subproblems to reduce the space of decision searching, and then we propose LEGO, a three-stage scheme to find the optimal (or near-optimal) trade-off decisions for Load balancing, EnerGy saving and QoS assurance. Through extensive simulation experiments, we validate that LEGO significantly outperforms several state-of-the-art approaches that it can produce higher-quality trade-off decisions and achieve more well-balanced service request routing and microservice instance placement. Our main contributions are summarized as follows:

- We formulate the resource usage and service time patterns of microservice instances and chains by queueing theory and model energy consumption and QoS assurance as a bi-criteria resource allocation optimization problem.
- LEGO starts with our resource allocation algorithm based on multiobjective particle swarm optimization, which can produce excellent trade-off decisions within a few iterations to create microservice instances.
- We then propose an efficient heuristic request routing algorithm to route requests into these created instances in a balanced manner.
- Finally, LEGO deploys these instances into servers by our balance-aware instance placement algorithm, which can achieve high balancing performance within a low computational complexity.
- Our extensive simulation results show that compared to several widely-used optimization algorithms, when maintaining the same service time, LEGO achieves a significant performance on reducing overall energy consumption, routing requests and placing instances in a balance manner.

The rest of this paper is structured as follows. In Section 2, we discuss the related works. The system model and our optimization problem are presented in Section 3 and 4, respectively. Our three-stage scheme is illustrated in Section 5. We present experiments and evaluations in Section 6 and conclude the paper in Section 8.

2. Background

2.1. Motivating scenario

Consider a service provider that provides a cloud infrastructure with a set of microservices (MSs) as the μService system, in which each of MS is encapsulated in a container image. The provider provision services/applications in term of microservice chains (MSCs), which is composed of a set of ordered MS instances and these instances are interacted with REST requests. An API gateway is provided with a set of service APIs to receive service requests from front-end users and to route them to corresponding back-end MSCs [2]. That is, a service request received from a service API is processed by a MSC.

To provision services with desired QoS, the service provider is concerned with the service time provided by each MSC due to different request processing logic and processing time in each MS instance according to allocated computing resource. In addition, he also cares about the reduction of energy consumption for running these services. Finally, the reliability and availability of the system is also required to ensure which can be achieved by balancing load among multiple instances instead of setting in a single instance. The solution is easily affordable if only one of those objectives is considered. For instance, the service time can be minimized by deploying sufficient MS instances and each instance is allocated with sufficient computing resource. But this solution will also lead to high energy consumption. Those three objectives are likely to be mutually exclusive. If he considers all of them

together with the heterogeneity of MSCs and shared resources among MSCs, the solution is not such that easy.

2.2. Related work

Resource allocation optimization is a wide-studied research topic in the field of cloud computing [5, 6]. We start with a brief review of related research for common cloud computing and then discuss research activities for μServices. In addition, an accurate resource allocation decision requires a clear understanding of system workload behaviors. Thus, the related research on workload modeling is also presented.

Resource allocation: In the VM-based cloud computing, the objectives of resource allocation optimization mainly seeks to efficient and effective resource use within the constraints of Service Level Agreements. These works usually consolidate VMs into a set of physical servers to meet different objectives. Beloglazov *et al.* [10] addressed the VM consolidation problem by an online energy-aware allocation heuristics algorithm: placing VMs to servers by a modified best fit decreasing algorithm and then online selecting a minimum number of VMs for live migration to optimize the system running-time energy consumption. Liu *et al.* [11] designed an energy efficient algorithm based on ant colony optimization (ACO) to minimize the number of active servers for placing a set of VMs. Chen *et al.* [12] studied the trade-off problem between QoS and energy cost for on-demand autoscaling by a multiobjective ACO algorithm. More related research works are referred to read survey papers [5, 6].

Resource allocation for containerized μServices is very recent topic. Nonetheless, there are some studies along the literature but limited to a coarse optimization. Smet *et al.* [13] considered differences on functionality among microservice images to optimize instance placement and aimed to maximize the satisfied demand given the storage and delay constraints. Guerrero *et al.* [8] proposed a genetic algorithm based on the Non-dominated Sorting Genetic Algorithm-II to optimize container assignment to each microservice, in which each container has the same resource configuration. Niu *et al.* [3] proposed a Nash bargaining based approach to assigning launched containers to each type of microservice in a balanced manner with the objective of reducing the service time of each microservice chain. With the similar objective, Yu *et al.* [14] moved to use a directed acyclic graph based model to describe microservice interdependencies and proposed a fully polynomial-time approximation scheme to achieve load balancing for microservice application deployment. Samanta *et al.* [15] considered the resource sharing incentive problem for microservice-based edge clouds with limited resource. They design an online auction-based mechanism to reclaim resources in over-provisioning microservices and reallocate them under-provisioning microservices.

In addition to academic activities, several scaling tools for containerized μServices (e.g., Kubernetes[2] and AWS Auto Scaling[3]) have been also developed. These tools can support horizontal instance scaling and rely on a threshold-based method to install/remove containers to/from servers according to their CPU usage.

The main issue of these above works is that they ignored the difference between μServices and VM-based clouds: μServices allow more fine-grained resource management thanks to container technologies. It is possible to address these concerns in the motivating example by improving these works. However, they do an optimization at the instance level that instances are assigned with fixed resources and their resource allocation is processed through deleting or installing instances, similar to these solutions in VM-based cloud. In this paper, we allocate resource at the CPU cycle level and also consider dependencies among microservices. As measured in [9], without careful CPU cycle allocation to containers, CPU schedulers do not treat CPU access fairly in proportion to their arrival rate and thus may result in performance bottleneck issues to those containers with high CPU access requirements. Therefore, we further consider to optimize CPU cycle allocations inside instances for serving fluctuating service requests while satisfying demand QoS. Such approach can achieve more fine-grained optimization of the usage of resource than the instance-level or CPU-core-level one.

Workload modeling: To clearly analyze system workload, queueing theory has been widely applied to model workloads for efficient resource allocations in clouds [16, 17, 18, 19, 3]. For CPU, the single CPU workload was modeled with a G/G/1 queue [16] or a M/G/1/PS queue [17]. Dawoud *et al.* [18] further considered the model for multiple virtual CPUs in a VM with an M/M/c queue system. For web services, Pacifici *et al.* [19] used an M/M/1 queue to model the response time behavior of requests. For microservices, Niu *et al.* [3] utilized an M/G/1/PS queue to model message deliver among microservices through message queue (an asynchronous IPC protocol). In this paper, to conduct efficient resource allocation decisions, similar to [16, 17], we also apply queueing theory to modeling CPU workload behaviors inside microservice instances.

3. System Model

We now introduce our μService system model with the main terminology used to represent physical infrastructure, microservice, microservice chain, and energy consumption. The basic notations are explained in Table 1.

3.1. Physical Infrastructure

We consider a data center containing a set I of server nodes (SNs). For simplification, we assume that these SNs

[2] A popular container management tool: https://kubernetes.io/.

[3] https://aws.amazon.com/en/autoscaling/features/.

Table 1: Basic notations

I	the set of server nodes (SNs)
M	the set of microservice (MS) types
N	the set of service requests
S	the set of types of microservice chains (MSCs)
C	total available compute resource of a SN
c_m^t	thread resource need of MS $m \in M$
c_m^r	rigid resource need of MS $m \in M$
$\mathcal{U}_m$	the execution rate for handling a request in $m \in M$
s_n	the MSC for serving the service request $n \in N$
J_{mi}	the instance set of MS $m \in M$ deployed in SN $i \in I$
m_{ji}	the j-th instance of MS $m \in M$ deployed in SN $i \in I$
η_{mji}	# of threads in $m_{ji}, m \in M, j \in J_{mi}$
ρ_{mji}	the system service intensity in m_{ji}
$\overline{T}_{mji}$	the mean response time for processing a request
$\mathcal{P}_i$	the power rate of SN $i \in I$
U_i	the CPU utilization of SN $i \in I$

are homogeneous, each having identical physical resources (e.g., CPU, memory, and disk). We focus on optimizing the allocation of computing resource (i.e., CPU cycles) and leave other resources as a future research item for two reasons: 1) not like networking applications which are I/O intensive, microservices for web services are CPU-intensive for QoS assurance [20]; 2) CPU consumes the main part of entire system energy consumption [10, 21]. Let C denotes the total available CPU cycles of an SN for deploying microservice instances. We consider Docker as the container technology which adopts *cgroup*[4] to control CPU access with a CFS scheduler [22]. There is a CPU CFS scheduler period (hereafter denoted by $\mathcal{C}^*$), with which a n-core CPU can be sliced into $n * \mathcal{C}^*$ CPU cycles to schedule. With the *cgroup*, the computing resource of instances can be dynamically reallocated in a finer granularity according to the current workload [9].

3.2. Microservice

In this subsection, we introduce the resource requirements and request processing time, as well as their relationship in each microservice. There are M types of MSs in the system that can be provisioned for serving different requests. Let m_{ji} denotes the *j-th* MS instance $m \in M$ installed on an SN $i \in I$ and allocated with c_{mji} CPU cycles. Each type of MS $m \in M$ has unique functionality with a a unique REST API exposed to other MSs. Hence, each heterogeneous MS has different computing resource needs and response time for processing its API request. The resource needs c_{mji} can be divided into two types [23]: *rigid* resource need (c_m^r) denotes a constant load-independent fraction of a resource for instance running and does not scale with workloads; *fluid* resource need (c_m^f) specifies the fraction of a resource for the required performance under a specific workload (i.e., load-dependent).

[4] A user space primitive provided by Linux for CPU cycle scheduling: http://man7.org/linux/man-pages/man7/cgroups.7.html.

To model the relationship between these resource needs and response time, we assume that all MSs adopt an identical resource scheduling architecture (e.g., Jetty[5]) for receiving and processing REST requests, which is equipped with a FIFO queue to buffer incoming requests and is configured with a set of concurrent threads to process requests. Once a request coming into a MS, it first gets inserted to the tail of the queue and then wait for an available thread to process. Thus, the *response time* for a REST request is composed of the *queueing time* in the queue and the *processing time* in a thread. In such architecture, for an instance with fixed fluid resource and workload, there is a trade-off between the number of threads and the processing time that more threads there are, the slower the request is processed, and vice versa. This effect is because the system requires more resources to coordinate concurrent threads. Furthermore, although reducing concurrent threads can reduce the processing time, it can lead to a longer queueing time and waste of buffering resources. According to these characteristics, we model the conflicting relationship as follows.

First, we introduce a constant c_m^t to denote the required CPU cycles to finish processing a request within a thread. c_m^t varies among different types of MS due to unique request processing logic. Thus, c_{mji} can be formulated as follows:

$$c_{mji} = c_{mji}^f + c_m^r = \eta_{mji} c_m^t + c_m^r, \forall \eta_{mji} \in \mathbb{Z}_{>0}, \quad (1)$$

where η_{mji} is an integer variable and denotes the number of concurrent threads for processing requests in the MS instance m_{ji}. For an instance m, once given the available CPU cycles c_m and η_m, the CPU scheduler schedules CPU cycles to a thread each time. Thus, following the previous paper [18], we use an M/M/1 queue system to model the behavior of CPU cycle scheduling for each thread in an instance and formulate the processing time as follows:

$$\tau_{mji}(\eta_{mji}) = \frac{1}{\mathcal{U}_m - \eta_{mji}}, \forall \eta_{mji} \leq \lceil \mathcal{U}_m - 1 \rceil, \quad (2)$$

where $\mathcal{U}_m = {}^{\mathcal{C}^*}/_{c_m^t}$ is a constant and denotes the thread execution rate of handling requests.

Moreover, following existing literature [3, 19, 18] , we assume that the arrival rate of requests in MSs follows a Poisson distribution with an average *arrival rate* λ and requests are served by an MS at a Poisson process with an average *service rate* μ, where $\mu = {}^1/_{\tau(\eta)}$. Finally, we use an M/M/c queue system to model the entire behavior of requests processing in a MS instance which has a FIFO queue to buffer requests and multiple concurrent threads to process requests ($c = \eta$ is the number of concurrent threads). Hence, we have the following expression:

$$\rho_{mji} = \frac{\lambda_{mji}}{\eta_{mji}\mu_{mji}} = \frac{\lambda_{mji} c_m^t}{\eta_{mji}(\mathcal{C}^* - c_m^t \eta_{mji})}, \quad (3)$$

[5] A powerful web server: https://www.eclipse.org/jetty/.

where ρ_{mji} denotes the system *service intensity* (i.e., CPU utilization) of the instance m_{ji}. When $\rho > 1$, there will be a queuing up. According to [24], the probability that there is no request (P_0) in the system and a request has to wait (P_q) can be calculated by the following expression:

$$P_0 = \left[\left(\sum_{k=0}^{\eta-1} \frac{(\eta\rho)^k}{k!} \right) + \frac{(\eta\rho)^\eta}{\eta!(1-\rho)} \right]^{-1}, P_q = P_0 \frac{(\eta\rho)^\eta}{\eta!(1-\rho)}.$$

The mean number of requests hold in the system is:

$$L_s = \frac{\rho}{1-\rho} P_q + \frac{\lambda}{\mu} = \frac{\rho}{1-\rho} P_q + \tau(\eta)\lambda.$$

Hence, applying Little's law, the mean response time $\overline{T}$ in an MS can be calculated as follows:

$$\overline{T} = \frac{L_s}{\lambda} = \frac{(\eta\rho)^\eta \rho}{\eta!(1-\rho)^2 \lambda} P_0 + \tau(\eta). \tag{4}$$

3.3. Microservice Chain

We assume that the μService system provides $|S|$ types of service APIs for service requests, so that $|S|$ types of MSCs are provided. To model this system, let N denotes the number of distinct service requests, each of which $n = \{s_n, \lambda_n\}, n \in N, s_n \in S$ has the request type (s_n, i.e., the type of MSC) and a Poisson request arrival rate (λ_n); the *service time* of processing a request n, i.e., the time from when a request arrives at the first MS of the MSC s_n to when it ended after traversing all MSs in s_n.

To calculate the service time, we analyze the request processing procedures through a MSC with the M/M/c system. First, according to the Burke Theorem [25], suppose there is a Poisson stream of requests traversing an MS m with an arrival rate λ, $\forall m \in M$, it also departs from m as a Poisson process with the expectation of λ when the system is stable. Second, MSCs often share MSs to process requests. When two Poisson streams of requests from two chains c_1 (with the rate λ_1) and c_2 (with the rate λ_2) are served by a MS m simultaneously, the rate of request streams arrived at m follows a Poisson distribution with an expectation of $\lambda_1 + \lambda_2$. When leaving m, the combined request stream is then divided according to their identities and directly delivered to other MSs following their MSCs. Hence, we have the following lemma:

Lemma 1. *$\forall m \in M$, the output request stream of each chain that departs from an MS m follows a Poisson process with its original expectation of λ and thus, arrives at the downstream MS $m+1$ as the same rate λ.*

Therefore, we can depict the request serving model in each MS in Fig. 1. The service time is composed of the processing delay in each traversed MS and the network transmission delay in network links. Since the network delay between the two MSs has a similar value that relies on the link bandwidth allocation and its optimization is outside the scope of our paper, we only consider the processing delay in each MS. Hence, based on the above findings, the average service time $\overline{T}_n$ for serving a service request $n \in N$ by a MSC s can be calculated by the following expression:

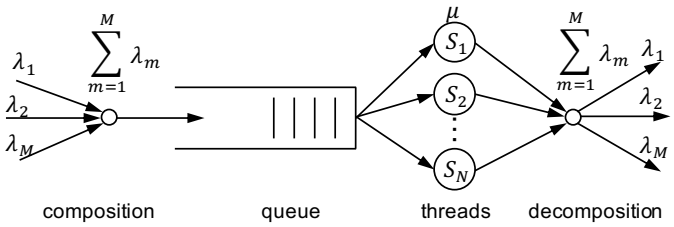

Figure 1: The M/M/c queueing model for a microservice.

$$\overline{T}_n = \sum_{m \in s_n} \overline{T}_m^n. \tag{5}$$

3.4. Energy Consumption

The energy consumption of a server $i \in I$ over a time period T is calculated by $E_i = \mathcal{P}_i T$, where $\mathcal{P}_i$ is the power rate of server i. Following existing literature [26, 21], we use a linear model to formulate the power rate as follows:

$$\mathcal{P}_i = \mathcal{P}^{idle} + (\mathcal{P}^{max} - \mathcal{P}^{idle}) * U_i, \tag{6}$$

where $\mathcal{P}^{idle}$ is the static power rate, whereby the energy is consumed by the host system and Docker daemon when the server is idle, i.e., without running containers and serving requests; $\mathcal{P}^{max}$ is the maximum power rate when the server is fully utilized; and U_i is the CPU utilization of an SN $i \in I$ and is calculated based on the resource model (1) as follows:

$$U_i = \left(\sum_{m \in M} \sum_{j \in J_{mi}} (c^f_{mji} \rho_{mji} + c^r_m) \right) / C, \tag{7}$$

where $\rho_{mji} < 1$.

4. Problem Formulation

Given the system model, we now define the **E**nergy- and **Q**oS-aware service request **R**outing and instance **P**lacement (**EQRP**) problem. Specifically, given the infrastructure resource I and a set of service requests N, we aim to find the minimum resource usage for deploying MS instances to reduce power rate, as well as the minimum service time of each MSC. This is a bi-criteria optimization problem in which the two objectives conflict with each other: allocating more resource can reduce service time, but at the cost of energy consumption.

4.1. Variables and Constraints

4.1.1. Decision Variables

We introduce several decision variables used in addressing **EQRP** as follows:

- $a_i \in \{0, 1\}, \forall i \in I$: a binary variable that assumes the value 1 if SN $i \in I$ is in active; otherwise, its value is 0;

- $|J_{mi}| \in \mathbb{Z}_{\geq 0}, \forall m \in M, \forall i \in I$: an integer variable that denotes the number of instances of MS m installed in SN i;
- $x_{s_n}^{mji} \in \{0,1\}, \forall m \in s_n, s_n \in S, n \in N$: a binary variable that assumes the value 1 if MSC s_n traverses an MS instance m_{ji}; otherwise, its value is 0;
- $\eta_{mji} \in \mathbb{Z}_{>0}, \forall j \in J_{mi}$: an integer variable that denotes the number of concurrent threads set in m_{ji}. Thus, the processing resource (c_{mji}) in the instance is allocated according to Eq. 1.

4.1.2. Capacity Constraint

An SN can deploy several MS instances with different types and resources. Eq. 8 ensures that all SN capacities are not violated.

$$c_i = a_i \sum_{m \in M} \sum_{j \in J_{mi}} c_{mji} < C, \forall i \in I. \tag{8}$$

4.1.3. Load Balance Constraint

To guarantee the reliability of service provisioning and physical infrastructure, we must balance the load among both homogeneous instances (i.e., the same MS type) and servers. We use the variance of load as the fairness index for load balancing as follows:

$$D = \frac{\sum U^2}{\sum n} - \left(\frac{\sum U}{\sum n}\right)^2 \leq \theta, \tag{9}$$

where U is the number of CPU cycles or CPU utilization and $\sum n$ is the total number of homogeneous instances or servers; θ is an acceptable threshold that reflects the maximum fluctuation degree of load distribution.

4.2. Objective Functions

EQRP problem has two objective functions as follows:
1) *Power rate*: The total power rate of the data center is defined by the following:

$$\mathcal{P}_{dc} = \sum_{i \in I} a_i \mathcal{P}_i. \tag{10}$$

Note that the data center's energy cost can be calculated with a penalty per consumed power watt.

2) *Service Time*: Due to inter-chain resource contention, we coordinate resources and minimize the service time of each MSC with the following objective function:

$$\mathcal{R}_{dc} = \sum_{n \in N} \left(\frac{\overline{T}_n - q_n^s}{q_n^s}\right)^2, \tag{11}$$

where q_n^s is the optimal service time of the MSC s:

$$q_n^s = \sum_{m \in \mathcal{V}^s} \frac{1}{\mathcal{U}_m} \tag{12}$$

4.3. Problem Description

We aim to jointly optimize these two objective functions and thus, we have the **EQRP** problem as follows:

$$\begin{aligned}
\min \quad & (\mathcal{P}_{dc}, \mathcal{R}_{dc}) && (13)\\
\text{s. t.} \quad & (8), (9) \\
& 0 < \eta_{mji} \leq \lceil \mathcal{U}_m - 1 \rceil, \forall j \in J_{mi} && (14)\\
& \rho_{mji} < 1, \forall m_j^i \in I && (15)\\
& |J_{mi}| \in \mathbb{Z}_{\geq 0}, \forall m \in M, \forall i \in I && (16)
\end{aligned}$$

where (14) denotes the stable condition of the M/M/1 queue system (i.e., $\rho = {}^{\eta}/_{\mathcal{U}} < 1$) for thread scheduling in an instance; (15) represents the stable condition of the M/M/c queue system. This is a bi-criteria decision optimization problem, which involves multiple trade-offs and is NP-hard [12].

5. Solution Scheme

In this section, we discuss our solution for the **EQRP** problem. Since it is a bi-criteria decision problem, there is no single decision but a set of trade-off decisions for the problem. These decisions (i.e., *optimality*) are commonly quantified by the *Pareto-dominance* relation, in which a decision vector $\vec{d_1}$ is Pareto-dominated by another decision vector $\vec{d_2}$, if and only if, (i) all the objective results achieved by $\vec{d_2}$ are better than or equivalent to those achieved by $\vec{d_1}$, and (ii) $\vec{d_2}$ achieves the result of at least one objective that is better than that achieved by $\vec{d_1}$. The set of all Pareto optimal decisions is called the **Pareto set** and the objective results of the Pareto set construct the **Pareto front**. We now describe our scheme to solve the bi-criteria optimization problem.

5.1. Three-stage Scheme

An NP-hard problem often contains a large search space. Taking the motivating scenario described in Section 2.1 for example, finding a suitable solution for service provision is difficulty since we need to consider these three concerns simultaneously. Such task will be more challenging as the size of service requests increasing.

In the above task, three sub-tasks are involved: *instance creation* which generates a set of MS instances and allocates resource (i.e., threads) to each instance; *request routing* which maps service requests to created instances; and *instance deployment* which deploys created instances into servers. Since the load balancing concern focuses on both balancing the service time promised by each MSC and balancing workload among servers, we find that once given a set of instances, the former can be achieved by mapping requests in a balance manner and the later can also be achieved by balanced instance deployment. Thus, this task with these three concerns can be decomposed into three stages that when creating instances, we only need to consider optimizing overall energy consumption and QoS when creating instances; after that, we route requests and

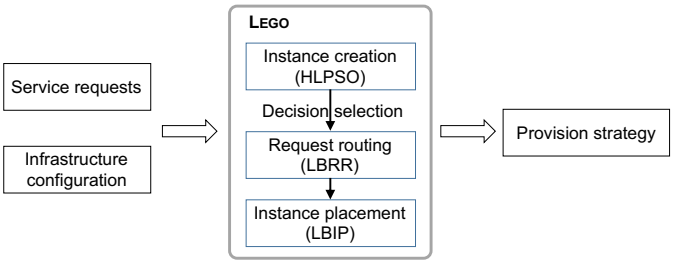

Figure 2: The workflow of our three-stage scheme.

deploy instances in a balance manner without violating the performance desired by the first stage. Addressing a problem in multi-stage can reduce its search space.

Hence, following the above workflow, we decompose the **EQRP** problem into three subproblems, so that reduce the search space of addressing **EQRP**, and propose a a three-stage scheme (LEGO) to address them. Fig. 2 illustrate the workflow of LEGO. Specifically, taking a set of service requests and infrastructure configurations, we address the **EQRP** problem in three stages as follows:

- *Instance creation*: for serving the service request set, we generate a set of MS instances with optimized number of instances and also optimized number of threads allocated in each instance.
- *Service request routing*: given the instance set, we route all service requests to them and balance the load among homogeneous instances without violating the desired performance provided by their allocated resources.
- *Instance placement*: we place all instances into a smallest number of servers with the objective of balancing both CPU allocation rate and utilization among these active servers.

5.2. Instance Creation

Since constraint (9) is used to coordinate the resources across MS instances for achieving a load balanced resource allocation, the optimal resource allocation is that homogeneous instances are configured with the same computing resource to serve requests with the same arrival rate. Thus, without considering the instance placement, the total power rate of all instances required for serving incoming requests can be calculated as follows:

$$\begin{aligned}\mathcal{P}'_{dc} &= \sum_{i\in I} a_i \left(\mathcal{P}^{idle} + (\mathcal{P}^{max} - \mathcal{P}^{idle}) * U_i\right) \\ &= \mathcal{P}^{idle} \left\lceil \left(\sum_{m\in M} |J_m| \left(c^t_m \eta_m + c^r_m\right) \right) / C \right\rceil + \\ &\frac{\mathcal{P}^{max} - \mathcal{P}^{idle}}{C} \left\{ \sum_{m\in M} \left(|J_m| \left(c^f_m \rho_m + c^r_m\right) \right) \right\} \\ &= \mathcal{P}^{idle} \left\lceil \left(\sum_{m\in M} |J_m| \left(c^t_m \eta_m + c^r_m\right) \right) / C \right\rceil + \\ &\frac{\mathcal{P}^{max} - \mathcal{P}^{idle}}{C} \left\{ \sum_{m\in M} \left(\frac{R_m {c^t_m}^2}{\mathcal{C}^* - c^t_m \eta_m} + |J_m| \, c^r_m \right) \right\},\end{aligned} \tag{17}$$

where homogeneous instances (m) have the same configuration (i.e., thread number η_m and arrival rate $R_m/|J_m|$); R_m is the total request arrival rate that needs to be served by MS m. Therefore, we have the optimization problem of instance creation as follows:

$$\begin{aligned} \min \quad & (\mathcal{P}'_{dc}, \mathcal{R}_{dc}) \\ \text{s. t.} \quad & (14), (15), (16). \end{aligned} \tag{18}$$

To address the above problem, we propose a metaheuristic algorithm based on multiobjective particle swarm optimization (MOPSO) to calculate the Pareto front. The particle swarm optimization (PSO) algorithm is a mature search algorithm based on swarm intelligence and is particularly suitable for MOPs, mainly because of its fast convergence speed [27, 28]. PSO starts with a population (i.e., swarm) of random candidate individuals, referred to as particles, each of which represents a potential decision. Each particle has two parameters, i.e., the *position* (X) and *velocity* (V). The position is associated with a fitness value, which is used to evaluate the quality of the particle. The velocity drives the movement of the position and reflects the socially exchanged information [27]. PSO then iteratively searches optimums from the search space. In each iteration, the velocity and position of each particle are updated from the last iteration as follows:

In this paper, we adopt an optimizer, namely, OMOPSO [27], which has been shown as the best MOPSO algorithm in [29]. Since the **EQRP** problem is a discrete problem, we introduce a rounding operator to ensure that the position of each particle is rounded to its nearest decimal integer. Moreover, a recognized shortcoming of MOPSO is that it may prematurely converge into local optimums [30]. That is, MOPSO may converge to a false Pareto front since it has no guaranteed convergence to the true Pareto front from an arbitrary initial state [30]. To address this issue, we design a **h**euristic swarm initia**l**ization strategy and propose our **PSO** algorithm based on OMOPSO, HLPSO.

To design a suitable initialization procedure than an arbitrary initialization, we find that given a $|J_m|$, for variable

η_m, the response time $\overline{T}_m$ is a convex function (detailed proof is referred to Appendix A). Hence, we can find a η_m that can make the response time $\overline{T}_m$ minimal via convex optimization algorithms, such as gradient descent or Newton's method. However, this means that each time we change $|J_m|$, we require computation time for performing gradient descent or Newton's method and calculating the derivative of $\overline{T}_m$. Thus, rather than directly calculating η_m, we still use a random η_m, but within a heuristic range. First, we limit $|J_m|$ in the range of $[w_l|N^m|, w_u|N^m|]$ to reduce the search space, where N^m is the set of requests that need to be served by the MS m, and w_l and w_u are the lower and upper boundaries. Starting from the minimum $|J_m|$ in the range, the initialization selects a $|J_m|$ in turn, to initialize the number of instances of each particle. Second, according to constraint (16), we constraint η_m as follows:

$$\frac{\mathcal{U} - \sqrt{\mathcal{U}^2 - 4\lambda}}{2} < \eta < \frac{\mathcal{U} + \sqrt{\mathcal{U}^2 - 4\lambda}}{2}.$$

With this range and a given $|J_m|$, we set the η_m as follows:

$$\eta_m = \frac{\mathcal{U}_m - \sqrt{\mathcal{U}_m^2 - 4\lambda_m}}{2} + r_b + r,$$

where r_b is a constant to reduce the destination to the optimal η_m and r is an integer random number to improve the diversity of the initialized decisions. With such heuristic initialization, HLPSO can start with the search space of the one that has a smaller response time and power rate calculated according to Theorem (1) and (2) without involving computational complexity. HLPSO then follows OMOPSO to search Pareto decisions iteratively. Finally, HLPSO will generate a Pareto front and we need to select one from the front as the decision for further deployment. The selection of a preferred decision depends on service providers' desires on the two objectives, i.e., they prefer a lower resource usage or lower service time. We will discuss our and existing selection approaches in our evaluation.

5.3. Service Request Routing

Given a Pareto decision, the next step is to route the service requests to these created instances in a balanced manner. Formally, given a set (J_m) of MS instances, the *request routing problem* routes each service request $n \in N^m$ to an instance $m_j, j \in J_m$ and balances the total arrival rate of the service requests (also called the serving rate) that must be served by each instance in J_m, since they are allocated the same resource by HLPSO. The request routing problem is similar to a bin packing problem, but with no upper bound. To address this problem, we design a load-balancing request routing (LBRR) algorithm, as shown in Alg. 1.

We first sort N^m in decreasing order by rate and use $R_m/(2|J_m|)$ as the upper bound rate to assign requests into $2|J_m|$ instances in turn (line 2-12) rather than $|J_m|$ instances, which can result in a better-balanced routing.

Algorithm 1: LBRR algorithm.

```
   Input  : N^m, |J_m|
1  sort N^m in descending order by rate;
2  K ← 2 * |J_m|;
3  r^avg ← R_m/K;
4  J_m ← J_m ∪ m(r^avg); // create an empty instance m(r^avg)
5  for n ← 1 to |N^m| do
6      if allocable (m, r_{N_n^m}) then
7          Route n by the new instance m;
8      else if |J_m| == K then
9          break;
10     else
11         J_m ← J_m ∪ m(r^avg);
12         Route n by m;
13 while true do
14     sort J_m in increasing order by rate;
15     if !allocable (J_m[1], r_{N_n^m}) then
16         for m ← J_m do
17             Route n by m;
18             n ← n + 1;
19             if n > |N^m| then break;
20         break;
21     else
22         for m ← J_m do
23             if !allocable (m, r_{N_n^m}) then break;
24             for remained n do
25                 if allocable (m, r_{N_n^m}) then
26                     Route n by m;
27                 else break;
28 sort J_m in descending order by rate;
29 for j ← 1 to |J_m| do
30     J_m[j] ← m_j ∪ m_{K-j+1}; //m_j, m_{K-j+1} ∈ J_m
31 J_m ←KKPA (J_m);
   Output: J_m
```

We sort these instances in decreasing order by rate and assign the remaining requests without the upper bound (line 13-27). These instances are then combined into J_m by a pair of the largest and smallest instances in J_m (line 28-31). After creating $|J_m|$ instances, we then leverage a popular number partitioning algorithm, Karmarkar and Karp heuristics (also called the KKPartition) [31], to further adjust the rate distribution among instances. Alg. 2 shows our KKParition-based adjustment algorithm, in which we first verify the max rate discrepancy Δ_{max} among instances with a threshold T^r, then continuously choose a pair of instances that have the max and min serving rate, and swap requests between them to balance their serving rate. The complexity of LBRR is $\mathcal{O}(|N_m|\log(|N_m|))$ since $|N_m|$ is larger than $|J_m|$. The worst-case running time is induced by the KKPA algorithm, whose complexity is $\mathcal{O}((\Delta_{max} - T^r)(|J_m| - 1) \cdot (n_1 + n_2)^{\alpha \log(n_1+n_2)})$, where n_1 and n_2 is the number of requests in two swapping instances. Given a fixed T^r, the more uneven the rate distribution among instances is, the more time the KKPA will take. Actually, since our routing strategy optimizes the

Algorithm 2: KKPA algorithm.

```
1  sort J_m in descending order by rate;
2  max ← 1, min ← |J_m|;
3  while true do
4      if Δ(x_max, x_min) < T^r then break;
5      if KKParition (x_max, x_min) then
6          sort the order of x_max and x_min in J_m;
7          min ← |J_m|; max ← 1;
8      else
9          min ← min − 1;
10         if min == 1 then break;
```

rate distribution by routing requests in $2|J_m|$ instances and then combining them, LBRR can achieve a smaller Δ_{max}, which controls the complexity in an acceptable range.

5.4. Microservice Instance Placement

Given a set S of servers with capacity C, and a set J of instances with different sizes, the problem of instance placement is to find the minimum number of servers to contain all the instances without exceeding the server capacity size, and to balance the load in terms of CPU occupation and utilization among these active servers. This problem is a bin packing problem, which is known to be NP-hard [32]. There exist many efficient packing algorithms (e.g., next-fit (NF), first-fit (FF) and first-fit decreasing (FFD)) to address this problem. However, these approaches may result in an unbalanced placement solution, since they are greedy to pack load without considering balancing the load of servers. While the balanced load among servers is necessary for the reliability of data centers, placing homogeneous instances into different servers (i.e., uniform placement) is also a necessary placement property, which can improve the availability of microservices.

To address these problems, we design our load-balancing instance placement (LBIP) algorithm as shown in Alg. 3. Since the instances with more CPU cycles can introduce higher utilization to a server, they should first be ensured a uniform placement rather than instances having fewer CPU cycles for load balance among servers. Hence, we first sort instances in descending order both by their CPU cycles and utilization and extract the first P^c percent of instances as the set J^P with corresponding MS set M^P (line 1-2). With $\mathcal{P}_M$ of a Pareto decision, we can obtain the minimum number of servers S (line 3). We then deploy the $\lfloor {}^{|N_m^P|}/_{|S|} \rfloor$ instances of the MS type $m \in M^P$ uniformly into the server set S (line 4-14). This step can reduce the complexity of the placement process into a constant time ($\mathcal{O}(|J^P|)$) and ensure the uniform distribution for these instances with more CPU cycles. Finally, we place the remaining instances into the less-loaded server achieved by `resortFirstServer`, which sorts the first server in increasing order according to its resource usage and thus, keeps the server with the smallest resource usage as the first server in S. The complexity of LBIP is $\mathcal{O}(|M||J_m|\log(|J_m|))$.

Algorithm 3: LBIP algorithm.

```
   Input  : J
1  sort J in descending order by CPU cycles and
     utilization;
2  {J^P, M^P} ← Extract(J, P^c);
3  |S| ← ⌈ ( Σ_{m∈M} Σ_{j∈J_m} c_mj ) / C ⌉;
4  for m ← M^P do
5      C_m ← c_m * ⌊ |J_m^P| / |S| ⌋;
6      for s ← 1 to |S| do
7          if s_s is null then s_s ← {};
8          c_s ← c_s − C_m;
9          for i ← ⌊ |J_m^P| / |S| ⌋ do
10             n ← (2|S| − 1) * ⌊ (i+1)/2 ⌋ + ⌊ i/2 ⌋;
11             if i is even then n ← n + s;
12             else n ← n − s;
13             s_s ← s_s ∪ {j_mi};
14             J_m ← J_m − {j_mi};
15 for m ← M do
16     if J_m ≠ ∅ then
17         for j ← J_m do
18             if !allocable (S[1], j) then
19                 open a new s in S with index 1;
20             S[1] ← S[1] ∪ {j};
21             resortFirstServer(S);
   Output: Extract all a_i and x_{s_n}^{mji} from S;
```

6. Experiments and Evaluations

Experimental evaluations are carried out in this section to validate the performance of Lego. By simulating a μService system with a real workload, we have considered the impact of different factors and conducted various quantitative experiments for our three algorithms and the entire three-stage scheme compared with other methods.

6.1. Simulation Setup

We simulate a data center with 150 homogeneous servers, each being assigned 8 CPU cores (i.e., $8\mathcal{C}^*$ CPU cycles) available for MS instance placement. We use the default CPU scheduler period [22], i.e., $\mathcal{C}^* = 100000$. We use the power consumption parameters ($\mathcal{P}^{idle} = 124$ watts and $\mathcal{P}^{max} = 219$ watts) for 8 CPU cores tested by [33] to model the server's energy consumption.

6.1.1. μService system

We follow the motivating scenario described in Section 2.1 and simulate a μServices system, in which $M = 40$ types of MSs encapsulated in container images are provided and $S = 100$ types of MSCs can be constructed for service requests. We follow the CPU cycle allocation range used in [9] to configure each type of MS in our simulation. For diversity, we use a lognormal distribution to determine each MS's CPU cycle requirements (i.e., c^t and c^r) in the range. The lognormal distribution is an efficient distribution to model the size distributions of various phenomena

[34, 35] and has a higher value for the upward distribution than a normal distribution. This distribution can reflect that most of MSs need more CPU cycles.

6.1.2. Workload data

In our simulations, similar to [3], we also use the online traffic activity of the world's top e-retailers in the U.S on Cyber Monday collected by Akamai [36]. The workload data were reported by Akamai's Retail Net Usage Index and Real User Monitoring. According to this fluctuating workload and the characteristics of the request rate tested in [37], we use a lognormal distribution ($N^L(\mu, \sigma^2)$) with different means (μ) and standard deviations (σ) to determine the size of the service request set and the request arrival rate of each service request.

6.1.3. Decision selection

To route requests and deploy instances into servers, choosing a suitable decision from a Pareto optimal decision set is still a problem. Here, we adopt a ε-constraint-based approach [38], in which ε is an upper threshold for $\mathcal{R}_{dc}$. We first extract the subset of decisions that have $\mathcal{R}_{dc}$ within ε and the minimum number of active servers. We then choose the decision from the subset with the minimum $\mathcal{R}_{dc}$. We use $|N|/50$ as ε. More selection approaches for yielding a single solution can be found in [38], e.g., the weighted sum method and lexicographic method.

6.1.4. State-of-the-art approaches

To validate the effectiveness of our proposed approach, we implemented the following algorithms for comparison:

- OMOPSO [27] and SMPSO [28]: Two popular and efficient multiobjective particle swarm optimizers [29]. SMPSO and our HLPSO are all developed based on OMOPSO, but the former extends OMOPSO by limiting the speed of the velocity update.
- NSGA-II [39]: A nondominated sorting genetic algorithm for MOPs, which can have excellent convergence speed due to its fast nondominated sorting approach.
- MOEA/D-DE [40]: A multiobjective evolutionary algorithm that decomposes an MOP into a number of scalar optimization subproblems and optimizes them simultaneously, which presents a better performance than NSGA-II on achieving the Pareto set in [40].
- Modified FFD, BFD and NFD for request routing: NFD, FFD, and BFD are three very straightforward packing algorithms. To route service requests, we first use $R_m/|J_m|$ (calculated by instance creation) as the instance's rate upper capacity and $|J_m|$ as the largest number of instances to pack requests. When there remains sufficient rate capacity in $|J_m|$ instances to route the next request, we sort all instances in ascending order by their serving rate and route remaining requests by sorted instances in turn. They have a running time of $\mathcal{O}(n + k)$ (NFD), $\mathcal{O}(n^2)$ (FFD) and $\mathcal{O}(n^2)$ (BFD), where n is the number of service requests, and k is the largest arrival rate.
- FFD, BFD, NFD, and a modified BFD (called MBFD) [10] for instance placement: FFD, BFD, and NFD are also very efficient for instance placement, which is also a packing problem. We also compare LBIP with MBFD, which uses the heterogeneity of resources by choosing the most power-efficient nodes firstly. MBFD sorts all instances in decreasing order of their CPU utilization and allocates each instance to a server that provides the least increase in power consumption due to this allocation. The complexity of MBFD is $\mathcal{O}(|M||J_m|\log(|J_m|))$.

The simulation and all algorithms were implemented in Java and evaluated on a desktop with Windows 10, a 3.60 GHz Intel Core i7 processor and 16 GB memory. We used these metaheuristic algorithms provided by the MOEA framework[6] and set the size of their population to 100 ($swarmSize$)and the size of their leader set (for storing leaders) to 200 ($leaderSize$).

6.2. Performance of proposed algorithms

We first evaluate the performance of three our proposed algorithms (HLPSO, LBRR, and LBIP) with existing algorithms, respectively. For the sake of a better understanding, the best and second-best results have dark (or bold font) and light gray backgrounds.

6.2.1. Instance Creation

First, we compare different approaches of instance creation on the quality of obtained results (often call approximated Pareto fronts) and convergence speed. To assess the quality of results, we use two unary indicators (**Hypervolume** I_{HV} [41] and **Additive Epsilon** $I_{\epsilon+}$ [42]) and a binary indicator (**C-metric** $C(A, B)$ [41]). While I_{HV} measures both the convergence and diversity of Pareto fronts, $I_{\epsilon+}$ and $C(A, B)$ measure the convergence.

Since I_{HV} and $I_{\epsilon+}$ rely on a reference set, but we have no ground truth of our problem as the reference, we performed these five approaches in 2000 iterations 3 times and extracted all nondominated decisions among all achieved results as the reference set. Table 2 shows the results, in which we performed all approaches with different numbers of iterations ($\mathcal{I}$=100, 500 and 1000). The better the quality of a front, the higher the resulting I_{HV} value is and the lower $I_{\epsilon+}$ is. $C(A, B)$ donates the percentage of the decisions in the approach B that are dominated by at least one decision in A. Thus, the higher the $C(A, B)$ value is, the better the quality of the front achieved by A, e.g., $C(A, B) = 1$ means that all decisions in A are better than those in B. Fig. 3 shows the approximations

[6] A Java library for developing and evaluating multiobjective evolutionary algorithms: http://moeaframework.org/.

Table 2: Quality of achieved approximated Pareto Fronts.

$\mathcal{I}$		HLPSO	NSGAII	MOEAD	OMOPSO	SMPSO
100	I_{HV}	9.89e-1	6.65e-1	6.32e-3	3.36e-1	4.43e-3
	$I_{\epsilon+}$	1.01e-2	3.34e-1	9.94e-1	6.63e-1	9.96e-1
	C(**HLPSO**,NSGA-II):C(NSGA-II,**HLPSO**)=1.0:0.0					
	C(**HLPSO**,MOEAD):C(MOEAD,**HLPSO**)=1.0:0.0					
	C(**HLPSO**,OMOPSO):C(OMOPSO,**HLPSO**)=1.0:0.0					
	C(**HLPSO,**SMPSO):C(SMPSO,**HLPSO**)=1.0:0.0					
500	I_{HV}	9.94e-1	8.02e-1	8.01e-1	7.99e-1	5.98e-1
	$I_{\epsilon+}$	5.98e-3	1.98e-1	1.99e-1	2.01e-1	3.99e-1
	C(**HLPSO**,NSGAII):C(NSGAII,**HLPSO**)=0.55:0.13					
	C(**HLPSO**,MOEAD):C(MOEAD,**HLPSO**)=1.0:0.0					
	C(**HLPSO**,OMOPSO):C(OMOPSO,**HLPSO**)=1.0:0.0					
	C(**HLPSO**,SMPSO):C(SMPSO,**HLPSO**)=1.0:0.0					
1000	I_{HV}	9.98e-1	9.99e-1	8.35e-1	8.35e-1	6.68e-1
	$I_{\epsilon+}$	2.64e-3	2.19e-3	1.65e-1	1.65e-1	3.32e-1
	C(**HLPSO**,NSGAII):C(NSGAII,**HLPSO**)=0.5:0.28					
	C(**HLPSO**,MOEAD):C(MOEAD,**HLPSO**)=1.0:0.0					
	C(**HLPSO**,OMOPSO):C(OMOPSO,**HLPSO**)=0.993:0.005					
	C(**HLPSO**,SMPSO):C(SMPSO,**HLPSO**)=1.0:0.0					

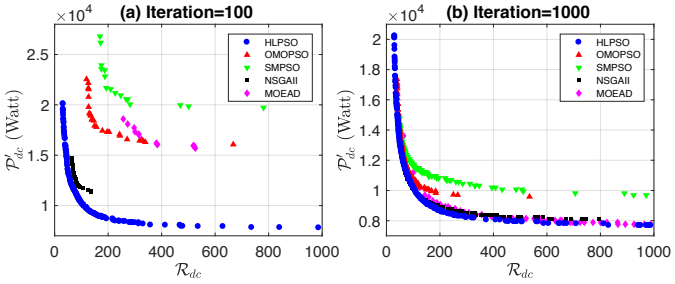

Figure 3: Obtained nondominated decisions under different numbers of iterations.

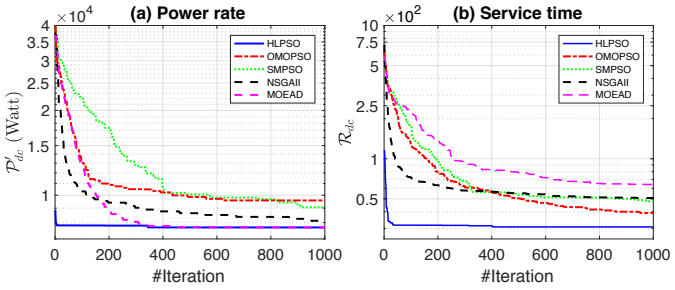

Figure 4: Convergence speed toward optimal results.

of Pareto front achieved by HLPSO and other approaches with $\mathcal{I}$=100 and 1000. These results show that our HLPSO outperforms other approaches.

We also considered the impact of different numbers of iterations on the quality of achieved results. In Table 2, the three indicators show that within a few iterations, HLPSO can achieve a better front than other approaches, and with more iterations, other approaches (like NSGA-II and OMOPSO) can achieve some decisions that equal or dominate some ones achieved by HLPSO. We also recorded the minimum value of each objective in each iteration, and these are presented in Fig. 4. We find that HLPSO can converge to the minimum value of each objective faster than other approaches.

These above results both illustrate that HLPSO outperforms other algorithms. The fundamental reason lies in that HLPSO leverages the characteristic of the EQRP problem to implement a heuristic strategy, which can generate a set of well-distributed initial decisions very close to the Pareto front.

6.2.2. Request Routing

To evaluate the effectiveness of the LBRR algorithm for service request routing, we compared LBRR with FFD, BFD, and NFD, and we also extended these three algorithms with the KKPA algorithm to optimize their results. Table 3 presents the comparison results of these algorithms with 8 types of evaluations. In each evaluation, we executed these algorithms 500 times. In each execution, we generated 1000 service requests, each of which had an arrival rate following a lognormal distribution with mean μ and standard deviation σ. We aimed to use $|J|$ instances to route these requests. To evaluate the balancing performance of each algorithm, we calculated the variance of the required serving rate among these instances routed by each algorithm and combined all the results of 500 executions as the final result. Considering the impact of μ, σ and $|J|$, we changed these factors in different evaluations.

Table 3 depicts that with the changed factors, our proposed approach can always achieve the minimum variance for request routing. FFD, BFD, and NFD are slightly faster than LBRR. However, they suffer from high variance since they only consider the rate constraints and do not consider the rate difference, which tends to fall into a local optimum. Different from these approaches, we used a smaller rate constraint to route service requests and considered the rate difference of created instances to combine these instances and balance the routed rate among them. Therefore, these operations can balance the rate of instances effectively and do not require much execution time towards a optimal result.

6.2.3. Instance Placement

Since HLPSO optimizes the number of servers and the power rate, the instance placement aims to deploy instances into these active servers and optimally balance their workload in terms of CPU cycles and utilization. We adopted different traffic workloads to create a set of instances by HLPSO and LBRR and evaluated the performance of LBIP with other approaches for placing different sets of instances. The detailed results are depicted in Table 4, where $|I^a|$ is the number of active servers.

The results show that our LBIP can obtain the minimum variance of the distribution both of CPU cycle allocation rate and utilization compared with the other four algorithms. FFD, BFD, and MBFD had the same result, but FFD was the fastest algorithm. The performance of NFD is unstable because it tends to open more servers than the result achieved by HLPSO. Different from the other methods, we consider the characteristic of created instances by HLPSO and LBRR that for homogeneous MS instances, they have the same processing resource needs but may be slightly different in CPU utilization due to different routed request rates. Hence, our approach first sorts all instances

Table 3: Performance of request routing algorithms (the execution time for routing requests and the balancing performance, i.e., the standard variance of the required serving rate among instances.). All results are the total statistic of 500 executions.

Parameters	Metrics	FFD	FFD+KKPA	BFD	BFD+KKPA	NFD	NFD+KKPA	LBRR
$\mu = 80, \sigma = 40$	Time (s)	0.112	0.558	0.046	0.489	0.037	0.613	0.432
$J = 40$	Variance	4.35e4	3.99e3	5.57e5	4.31e3	7.83e4	4.57e3	141.35
$\mu = 80, \sigma = 75$	Time (s)	0.115	0.325	0.046	0.263	0.034	0.398	0.217
$J = 40$	Variance	1.12e3	175.44	1.42e4	173.29	5.73e3	136.09	87.78
$\mu = 80, \sigma = 40$	Time (s)	0.167	0.622	0.059	0.533	0.037	0.741	0.524
$J = 80$	Variance	4.69e4	2.76e3	6.31e5	2.99e3	8.14e4	2.27e3	137.39
$\mu = 80, \sigma = 75$	Time (s)	0.128	0.411	0.055	0.339	0.034	0.451	0.272
$J = 80$	Variance	1.31e3	174.35	1.89e4	167.55	7.38e3	101.35	89.63
$\mu = 80, \sigma = 40$	Time (s)	0.227	0.721	0.072	0.622	0.044	0.826	0.622
$J = 120$	Variance	5.12e4	1.29e3	7.16e5	1.16e3	8.50e4	989.72	141.36
$\mu = 80, \sigma = 75$	Time (s)	0.260	0.534	0.073	0.415	0.052	0.600	0.321
$J = 120$	Variance	1.34e3	153.45	2.34e4	150.05	8.14e3	101.46	92.53
$\mu = 40, \sigma = 35$	Time (s)	0.108	0.268	0.041	0.212	0.032	0.300	0.138
$J = 40$	Variance	628.75	159.30	6.08e3	145.20	2.27e3	94.05	87.15
$\mu = 120, \sigma = 75$	Time (s)	0.112	0.530	0.051	0.469	0.030	0.642	0.480
$J = 80$	Variance	4.22e4	3.73e3	6.38e5	3.3e3	1.01e5	3.14e3	97.63

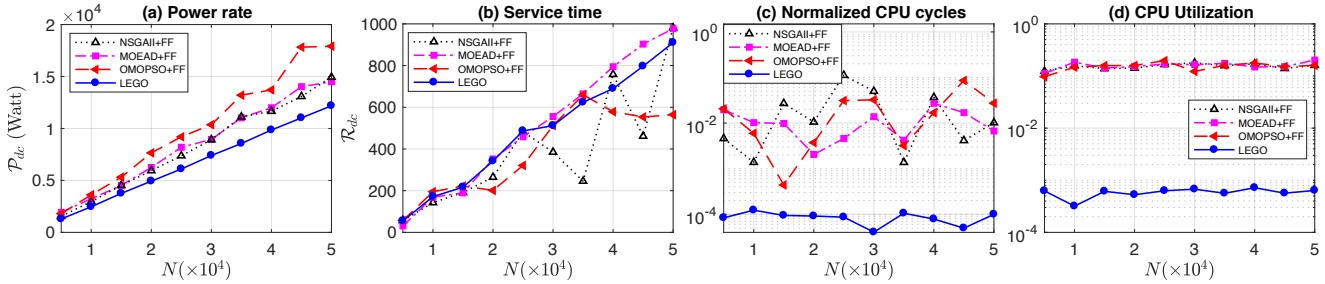

Figure 5: The resource allocation results with increasing number of service requests ($\mu = 80$, $\sigma = 75$ and $\mathcal{I}$=300). Besides comparing $\mathcal{P}_{dc}$ (a) and R_{dc} (b), we also show the distribution of CPU cycle allocation rate (c) and utilization (d) among active servers configured by these four algorithms.

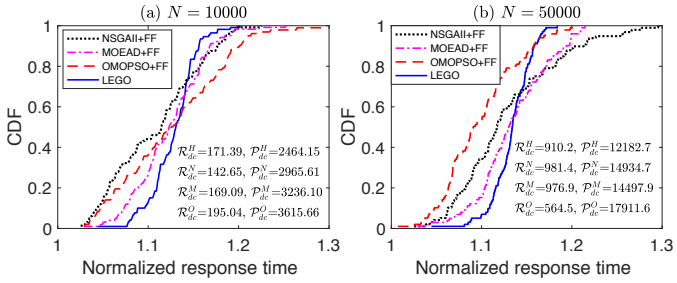

Figure 6: The CDF of normalized service time ($\overline{T}_n/q_n^s$).

in decreasing order both by their CPU cycles and utilization and uniformly deploys instances with large CPU cycles into the servers. Then, we deploy these instances with small CPU cycles with a best loaded strategy. Thus, our approach can efficiently balance the load among servers within a shorter running time.

6.3. Simulation Results

Given the performance experiments of our proposed algorithms, we further evaluated our LEGO scheme against other schemes. We chose NSGA-II, MOEA/D-DE, and OMOPSO to generate the Pareto set for instance creation and used the FFD-based algorithm to route requests and deploy instances due to its better performance than others. Following the workflow in Fig. 2, we integrated them as

Table 4: Performance of instance placement algorithms (the standard variance of the normalized CPU cycle allocation rate and CPU utilization among active servers (I^a)).

$\lvert I^a \rvert$	Metrics	NFD	FFD	BFD	MBFD	LBIP
7	Time (ms)	0.17	0.44	0.43	0.61	0.26
	CPU cycles	4.82e-3	7.63e-3	7.63e-3	7.63e-3	1.26e-4
	Utilization	0.103	0.104	0.104	0.104	1.00e-3
20	Time (ms)	0.19	0.45	0.62	1.62	0.99
	CPU cycles	1.23e-2	1.98e-2	1.98e-2	1.98e-2	1.05e-4
	Utilization	9.37e-2	9.66e-2	9.66e-2	9.66e-2	5.14e-4
33	Time (ms)	0.16	0.87	0.93	6.77	2.15
	CPU cycles	0.16	2.45e-3	2.45e-3	2.45e-3	1.12e-4
	Utilization	0.15	0.10	0.10	0.10	4.25e-4
45	Time (ms)	0.45	1.74	1.57	7.82	4.08
	CPU cycles	5.14e-3	1.74e-2	1.74e-2	1.74e-2	7.37e-5
	Utilization	9.26e-2	9.65e-2	9.65e-2	9.65e-2	5.49e-4
57	Time (ms)	0.31	2.36	2.46	12.47	6.66
	CPU cycles	0.118	1.74e-3	1.74e-3	1.74e-3	1.06e-4
	Utilization	0.127	9.51e-2	9.51e-2	9.51e-2	6.95e-4

NSGAII+FF, MOEAD+FF, and OMOPSO+FF. In our experiments, we considered the number of service requests ($|N|$), iterations of algorithms, and the arrival rate of service requests as three factors that may impact on these schemes' performance. We compared the entire power rate ($\mathcal{P}_{dc}$) of the data center, the objective function $\mathcal{R}_{dc}$ for the service time of all service requests, and balancing performance of the CPU cycle allocation rate and utilization among active servers.

6.3.1. Number of Service Requests

We considered the different sizes of the incoming service request set as the fluctuation of traffic workload. We performed these schemes for the traffic workload that varies from $|N| = 5000$ to $|N| = 50000$ service requests and their arrival rate follows $N^L(\mu, \sigma^2), \mu = 80, \sigma = 75$. Fig. 5 shows the resource allocation results with 300 iterations. We find that regardless of the number of service requests, LEGO always achieves the better performance; thus, it reduces the overall power rate by an average of 18.5% while keeping $\mathcal{R}_{dc}$ similar to that achieved by other schemes. We chose $|N| = 10000$ and $|N| = 50000$ to show the cumulative distribution function (CDF) of the normalized service time achieved by these four schemes (see Fig. 6). They have similar $\mathcal{R}_{dc}$, but the normalized service time among MSCs produced by LEGO are more balanced than others due to the higher balancing performance of our LLRR algorithms. Even some other schemes (e.g., OMOPSO+FF in Fig. 6 (b)) can produce a smaller value of $\mathcal{R}_{dc}$, but there exist some service times greater than the one produced by LEGO. Moreover, $\mathcal{R}_{dc}$ achieved by NSGAII+FF and OMOPSO+FF have a greater fluctuation than those achieved by LEGO and MOEAD+FF due to their slow convergence speed.

Fig. 5 (c) and (d) depicts the respective balancing performance of the CPU cycle allocation rate and utilization among servers. We calculated the standard variance of normalized allocated CPU cycles and CPU utilization among servers. We find that LEGO always achieves the best balancing performance (i.e., 97.94% in resource allocation and 99.58% in resource utilization on average) as the number of service requests increases. In addition, we also validated the overhead of our approach by computing the latency of the resource allocation process. As shown in Table 5, we can see that all four schemes have similar execution times for routing requests and deploying instances, and the overhead increases as the number of service requests increases.

6.3.2. Iterations of algorithms

Choosing a suitable number of iterations has a trade-off consideration between the running time and quality of achieved decisions. Fig. 4 has depicted that the convergence speed of HLPSO outperforms other approaches. Here, we further evaluated the impact of iterations on addressing the overall problem, in which the traffic workload has a fixed number of service requests ($|N| = 25000$). Fig. 7 shows the simulation results under different numbers of iterations. We observe that our LEGO can converge within approximately 200 iterations, which is better than others. As shown in Fig. (b), $\mathcal{R}_{dc}$ achieved by MOEAD+FF, NSGAII+FF, and OMOPSO+FF fluctuate as the number of iterations increases. The reasons for these results come from two aspects: one is that due to nonconvergence, the best decision that the methods searched changes as the number of iterations increases; the other is that due to the lower balancing performance of FFD, unbalanced request routing induces different $\mathcal{R}_{dc}$ than achieved by MOEAD, NSGAII, and OMOPSO. However, as the number of iterations increases, the other three schemes gradually achieve results similar to LEGO.

Table 5: Time consumption

$\|N\|$	LEGO	NSGAII+FF	MOEAD+FF	OMOPSO+FF
10000	10.32s	10.27s	10.45s	10.24s
25000	23.50s	23.32s	23.44s	23.59s
50000	47.00s	47.83s	47.05s	47.11s

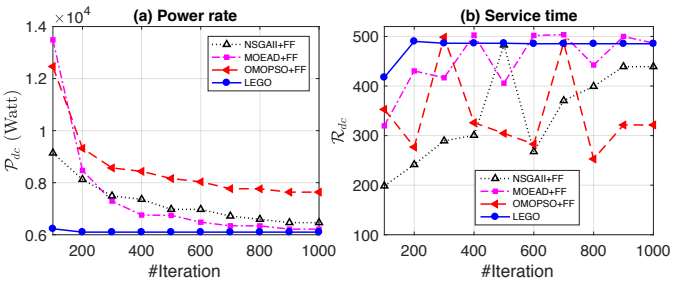

Figure 7: The resource allocation results with increasing number of iterations ($|N| = 25000$, $\mu = 80$ and $\sigma = 75$).

6.3.3. Arrival Rate of Requests

The fluctuation of traffic workload also reflects changes in the arrival rate of service requests. Thus, we tuned the arrival rate of all service requests from 1.0x to 1.5x (i.e., $\lambda = 1.5$x) and evaluated the performance of these four schemes with different numbers of incoming service requests. Fig. 8 shows the results. We find that the performance of LEGO is not affected by changes in the number of service requests and arrival rate and LEGO can achieve better power rates than others by an average of 15.5% and better balancing performance by an average of 95.52% on CPU cycle allocation rate and 99.53% on CPU utilization.

6.4. Summary

We have presented numerical experiment results of our proposed scheme on resource management for μService clouds. To summarize, our findings are as follows:

- For instance creation, our HLPSO can produce more near-optimal Pareto solutions and higher convergence speed than these widely used approaches (NSGA-II, MOEAD, and OMOPSO);
- For request routing, compared to FFD, BFD, and NFD, our LBRR can achieve the best solution of routing requests in a balance manner;
- For instance deployment, our LBIP can deploy instances into servers with a best loaded strategy, with which the load in these servers are balance efficiently, including CPU cycle allocation rate and utilization;
- For the overall service provision, our LEGO outperforms baseline heuristics significantly. It can reduce the overall energy consumption by an average of

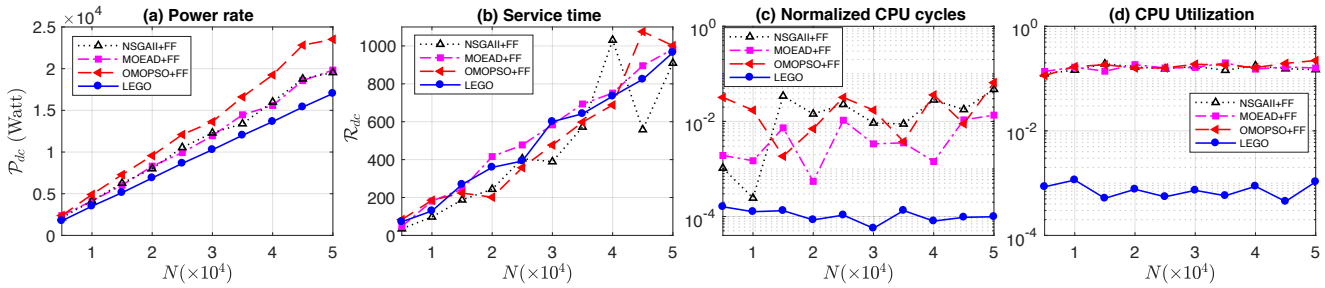

Figure 8: The resource allocation results with increasing number of service requests and $\lambda = 1.5$x ($\mu = 80$, $\sigma = 75$ and $\mathcal{I}$=300).

18.5% and achieve the better load balancing results by an average of 97.94% than these baseline heuristics.

7. Discussion and Future work

LEGO achieves a significant performance. However, it also has its limitations. Alleviating them will be important future work for application in production environments.

Poisson process. We follow Poisson process to model system behaviors in the μService system. In fact, the request arrivals in general follow a heavy-tailed (Pareto) distribution [43]. However, the first-come-first-serve message processing in a software environment leads to uniform Poisson-like distributions. The request service times are not always exponential, but also can be in a general distribution. However, the general distribution cannot make solution obtained in closed form, which the search for a suitable approximation is required [44]. Following existing literature, we simplify the system model with M/M/c model, which may result in inaccurate results. Therefore, modeling service time in the general distribution and designing lightweight approximation search algorithms are desired in future.

Hardware resource. In a cloud environment, optimizing allocation of hard resources is essential for cost saving and QoS guarantee. Our LEGO is the first step toward that goal, which focuses on optimizing allocation of computing resource. More different types of resources, like memory, disk, and I/O operations in servers, can be modelled and optimized. With the complete resource management, LEGO can be improved to achieve more accurate and practical results.

Network infrastructure. Currently, LEGO only focuses on CPU-intensive cloud applications. However, due to service decomposition, more frequent service interconnection are involved in the μService system. The interconnection may happen in the same server (processed by system kernel), among different servers (routed by switches and routers), and event across network domains (through public Internet). Those types of interconnection can introduce different delays to request routing. Thus, modeling this complex network environment is another point that can be take into account for improving LEGO, in which network bandwidth allocation and instance physical locations can be optimized.

Dynamic service provision. Services in the cloud are often provisioned on-demand. The μService system tends to provide more dynamic service provision than VM-based system, due to the increasing agility provided by containers. Therefore, we plan to investigate how to design an online version of LEGO which can incrementally serve frequently changing workloads.

Proactive traffic prediction. LEGO runs in an active manner. Involving proactive traffic prediction is a potential approach to enabling proactive service provision, so that reducing the latency of service delivery. Traffic prediction problem can be approached as a time series predication problem, which can be addressed by mature intelligent approaches, like statistical, rule-based, or deep learning. Therefore, combining these traffic prediction approaches with LEGO is our another plan for efficient planning and usage of resources in the cloud.

8. Conclusion

In this paper, we studied the service request routing and microservice instance deployment problem in the μService environment. We leveraged the characteristic of container-based μServices to model the μService system and formulate the resource allocation problem to be aware of energy consumption and the QoS assurance. We then proposed a three-stage approach (LEGO) to address this problem. In particular, the approach first searches and optimizes trade-off decisions of resource allocation for serving incoming requests. Then, the decision is performed to set up a set of microservice instances and route requests to those instances in a balanced manner. Finally, a minimal set of servers is activated to deploy these instances in a balanced manner. Our experiments show that, in contrast to several widely used approaches (NSGA-II, MOEAD, and OMOPSO), our approach produces better trade-off decisions with a higher balancing performance of request routing and instance placement for serving fluctuating service request workloads.

Acknowledgment

This work is supported by the Provincial Science & Technology Pillar Program of Hubei under Grant 2017AAA027, 2017AAA042 and 2017AHB048.

[1] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, M. Villari, Open issues in scheduling microservices in the cloud, IEEE Cloud Comput. 3 (5) (2016) 81–88.

[2] NGINX, Microservices From Design to Deployment, https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/, (accessed 5 October 2018).

[3] Y. Niu, F. Liu, Z. Li, Load Balancing across Microservices, in: INFOCOM, 2018, pp. 1–9.

[4] A. Panda, M. Sagiv, S. Shenker, Verification in the Age of Microservices, in: ACM HotOS, 2017, pp. 30–36.

[5] B. Jennings, R. Stadler, Resource management in clouds: Survey and research challenges, J. Network and Systems Management 23 (3) (2015) 567–619.

[6] Z.-H. Zhan, X.-F. Liu, Y.-J. Gong, J. Zhang, H. S.-H. Chung, Y. Li, Cloud computing resource scheduling and a survey of its evolutionary approaches, ACM Computing Surveys (CSUR) 47 (4) (2015) 63.

[7] L. M. Vaquero, L. Rodero-Merino, R. Buyya, Dynamically scaling applications in the cloud, ACM SIGCOMM Computer Commun. Review 41 (1) (2011) 45–52.

[8] C. Guerrero, I. Lera, C. Juiz, Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications, J. Supercomput 74 (7) (2018) 2956–2983.

[9] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, X. Fu, NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains, in: ACM SIGCOMM, 2017, pp. 71–84.

[10] A. Beloglazov, J. Abawajy, R. Buyya, Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing, Future Gener. Comput. Syst. 28 (5) (2012) 755–768.

[11] X.-F. Liu, Z.-H. Zhan, J. D. Deng, Y. Li, T. Gu, J. Zhang, An energy efficient ant colony system for virtual machine placement in cloud computing, IEEE Trans. Evol. Comput. 22 (1) (2016) 113–128.

[12] T. Chen, R. Bahsoon, Self-Adaptive Trade-off Decision Making for Autoscaling Cloud-Based Services, IEEE Trans. Serv. Comput. 10 (4) (2017) 618–632.

[13] P. Smet, B. Dhoedt, P. Simoens, Docker layer placement for on-demand provisioning of services on edge clouds, IEEE Trans. on Network and Service Management 15 (3) (2018) 1161–1174.

[14] R. Yu, V. T. Kilari, G. Xue, D. Yang, Load Balancing for Interdependent IoT Microservices, in: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, IEEE, 2019, pp. 298–306.

[15] A. Samanta, L. Jiao, M. Mühlhäuser, L. Wang, Incentivizing Microservices for Online Resource Sharing in Edge Clouds, in: IEEE International Conference on Distributed Computing Systems (ICDCS), 2019.

[16] T. Wood, P. J. Shenoy, A. Venkataramani, M. S. Yousif, et al., Black-box and Gray-box Strategies for Virtual Machine Migration., in: USENIX NSDI, Vol. 7, 2007, pp. 17–17.

[17] P. Xiong, Z. Wang, S. Malkowski, Q. Wang, D. Jayasinghe, C. Pu, Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach, in: ICDCS, 2011, pp. 571–580.

[18] W. Dawoud, I. Takouna, C. Meinel, Elastic virtual machine for fine-grained cloud resource provisioning, in: Global Trends in Comput. and Commun. Systems, Springer, 2012, pp. 11–25.

[19] G. Pacifici, M. Spreitzer, A. N. Tantawi, A. Youssef, Performance management for cluster-based web services, IEEE J. Sel. Areas Commun. 23 (12) (2005) 2333–2343.

[20] M. Elnozahy, M. Kistler, R. Rajamony, Energy conservation policies for web servers, in: USENIX USITS, 2003, pp. 8–8.

[21] J. A. Aroca, A. Chatzipapas, A. F. Anta, V. Mancuso, A measurement-based characterization of the energy consumption in data center servers, IEEE J. Sel. Areas Commun. 33 (12) (2015) 2863–2877.

[22] Limit a container's resources, https://docs.docker.com/config/containers/resource_constraints/, (accessed 2 August 2018).

[23] M. Stillwell, D. Schanzenbach, F. Vivien, H. Casanova, Resource allocation algorithms for virtualized service hosting platforms, J. of Parallel and distributed Comput. 70 (9) (2010) 962–974.

[24] J. Medhi, Stochastic models in queueing theory, Academic Press, 2002.

[25] P. J. Burke, The output of a queuing system, Operations research 4 (6) (1956) 699–704.

[26] X. Fan, W.-D. Weber, L. A. Barroso, Power provisioning for a warehouse-sized computer, in: ACM ISCA, 2007, pp. 13–23.

[27] M. R. Sierra, C. A. C. Coello, Improving PSO-based multi-objective optimization using crowding, mutation and-dominance, in: Springer Int. Conf. on Evol. Multi-Criterion Optimization, 2005, pp. 505–519.

[28] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. C. Coello, F. Luna, E. Alba, SMPSO: A new pso-based metaheuristic for multi-objective optimization, in: IEEE Symp. on Comput. intelligence in multi-criteria decision-making, 2009, pp. 66–73.

[29] J. J. Durillo, J. García-Nieto, A. J. Nebro, C. A. C. Coello, F. Luna, E. Alba, Multi-objective particle swarm optimizers: An experimental comparison, in: Springer Int. Conf. on Evol. Multi-Criterion Optimization, 2009, pp. 495–509.

[30] M. Reyes-Sierra, C. C. Coello, et al., Multi-objective particle swarm optimizers: A survey of the state-of-the-art, Int. J. of Comput. Intelligence Research 2 (3) (2006) 287–308.

[31] N. Karmarkar, R. M. Karp, The Difierencing Method of Set Partitioning, Tech. rep., UCB/CSD 82/113, Computer Science Division, University of California, Berkeley (1982).

[32] M. R. Garey, D. S. Johnson, Computers and intractability, Vol. 29, wh freeman New York, 2002.

[33] R. Morabito, Power consumption of virtualization technologies: an empirical investigation, in: IEEE/ACM UCC, 2015, pp. 522–527.

[34] W. J. Reed, M. Jorgensen, The double pareto-lognormal distributiona new parametric model for size distributions, Commun. in Statistics-Theory and Methods 33 (8) (2004) 1733–1753.

[35] N. C. Beaulieu, A. A. Abu-Dayya, P. J. McLane, Estimating the distribution of a sum of independent lognormal random variables, IEEE Trans. Commun. 43 (12) (1995) 2869.

[36] Akamai, Akamais 2015 Online Holiday Shopping Trends and Traffic Report for Europe and North America, https://goo.gl/FygHz7, (accessed 25 September 2018).

[37] D. Narayanan, A. Donnelly, A. Rowstron, Write off-loading: Practical power management for enterprise storage, ACM Trans. Storage 4 (3) (2008) 10.

[38] R. T. Marler, J. S. Arora, Survey of multi-objective optimization methods for engineering, Structural and multidisciplinary optimization 26 (6) (2004) 369–395.

[39] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.

[40] H. Li, Q. Zhang, Multiobjective optimization problems with complicated Pareto sets, MOEA/D and NSGA-II, IEEE Trans. Evol. Comput. 13 (2) (2009) 284–302.

[41] E. Zitzler, L. Thiele, Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach, IEEE Trans. Evol. Comput. 3 (4) (1999) 257–271.

[42] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, V. G. Da Fonseca, Performance assessment of multiobjective optimizers: An analysis and review, IEEE Trans. Evol. Comput. 7 (2) (2003) 117–132.

[43] A.-L. Barabasi, The origin of bursts and heavy tails in human dynamics, Nature 435 (7039) (2005) 207.

[44] H. Khazaei, J. Misic, V. B. Misic, Performance analysis of cloud computing centers using m/g/m/m+ r queuing systems, IEEE Trans. Parallel and Distributed Syst. 23 (5) (2012) 936–943.

[45] B. P. Chen, S. G. Henderson, Two issues in setting call centre staffing levels, Annals of operations research 108 (1-4) (2001) 175–192.

[46] W. Grassmann, The convexity of the mean queue size of the M/M/c queue with respect to the traffic intensity, Journal of

Applied Probability 20 (4) (1983) 916–919.

Appendix A. Theorem-proof

Theorem 1. *For $\eta \geq 2$, the response time $\overline{T}$ of an m MS instance is convex in its thread number η_m and decreasing in its instance number $|J_m|$.*

Proof. In [45], $\overline{T}$ was shown to be increasing and convex in λ_m. Since $\lambda_m = {}^{R_m}/_{|J_m|}$, $\overline{T}$ is decreasing in $|J_m|$.

The derivative of $\overline{T}(\rho)$ is calculated as follows:

$$\overline{T}'(\rho) = \frac{L_s'(\rho)\lambda - \eta L_s(\rho)(\mathcal{U} - \eta)}{\lambda^2}.$$

We then transform this expression to focus on η as follows:

$$\overline{T}'(\eta) = \frac{L_s'(\rho)\lambda - \eta L_s(\rho)(\mathcal{U} - \eta)}{\lambda} \frac{2\eta - \mathcal{U}}{\eta^2(\mathcal{U} - \eta)^2},$$

$$\overline{T}''(\eta) = \frac{2\rho L_s'(\rho) + \rho^2 L_s''(\rho)}{\eta^2 \lambda}.$$

According to [46], the derivatives of L_s in respect to ρ ($L_s'(\rho)$ and $L_s''(\rho)$) are all nonnegative. Thus, $\overline{T}''(\eta) > 0$. Now, we have proven the theorem. □

Theorem 2. *The CPU utilization U_m in an m MS instance is increasing both in η_m and in $|J_m|$.*

Proof. According to Eq. (17), we have U_m as follows:

$$U_m = \frac{R_m {c_m^t}^2}{\mathcal{C}^* - c_m^t \eta_m} + |J_m|\, c_m^r.$$

We then can obtain:

$$U'(\eta_m) = \frac{R_m {c_m^t}^3}{(\mathcal{C}^* - c_m^t \eta_m)^2} > 0, U'(|J_m|) = c_m^r > 0.$$

Thus, we have the above theorem. □