A Cooperative Coevolution Genetic Programming Hyper-Heuristic Approach for On-line Resource Allocation in Container-based Clouds

Boxiong Tan, Hui Ma, Yi Mei, and Mengjie Zhang,

Abstract—Containers are lightweight and provide the potential to reduce more energy consumption of data centers than Virtual Machines (VMs) in container-based clouds. The on-line resource allocation is the most common operation in clouds. However, the on-line *Resource Allocation in Container-based clouds (RAC)* is new and challenging because of its two-level architecture, i.e. the allocations of containers to VMs and the allocation of VMs to physical machines. These two allocations interact with each other, and hence cannot be made separately. Since on-line container allocation requires a real-time response, most current allocation techniques rely on heuristics (e.g. First Fit and Best Fit), which do not consider the comprehensive information such as workload patterns and VM types. As a result, resources are not used efficiently and the energy consumption is not sufficiently optimized. We first propose a novel model of the on-line *RAC* problem with the consideration of VM overheads, VM types and an affinity constraint. Then, we design a Cooperative Coevolution Genetic Programming (CCGP) hyper-heuristic approach to solve the *RAC* problem. The CCGP can learn the workload patterns and VM types from historical workload traces and generate allocation rules. The experiments show significant improvement in energy consumption compared to the state-of-the-art algorithms.

Index Terms—container-based clouds, container allocation, energy efficiency, genetic programming, hyper-heuristic

1 Introduction

Containers have long been used in PaaS clouds, such as Heroku [1] and OpenShift [2], for deploying applications. With the advent of web-based applications such as server-less and micro-service architectures, containers become even popular. Containers are ideal for providing a low overhead, isolated environment for application components. Container-based clouds not only allocate more applications than Virtual Machines-based clouds (see Fig. 1) by sharing Operating Systems (OSs), but also release the burden of application providers by managing the cloud resources with auto-scaling and migration techniques [3].

Although containers have numerous advantages compared to Virtual Machines (VMs), e.g. fast start-up time and low overhead, they suffer from security threats [4], [5] and performance interference (e.g. competition on I/O resources) [6], [7]. Therefore, when facing the diverse requirements from applications, e.g. various OSs and security levels, cloud providers use VMs to provide an extra level of isolation for allocating containers. Besides, Sharma et al. [8] found that nesting containers in VMs can also improve the performance of applications compared to running applications in VMs solely. Hence, a new two-level resource allocation [8]–[11] emerges where on the first level, containers need to be allocated to VMs. New VMs may be used if the existing VMs do not enough resources to host the containers. On the second level, the VMs allocated with containers need to be allocated to PMs.

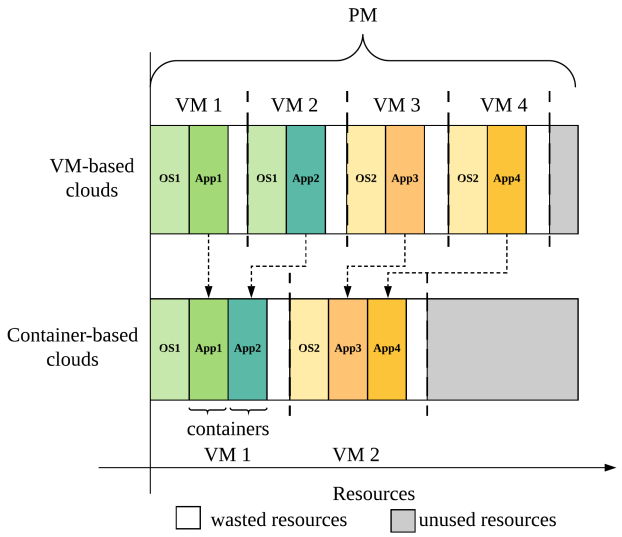


Fig. 1: Container-based clouds vs. VM-based clouds.

The container-based clouds brings new challenges to resource allocation. Resource allocation aims to minimize overall energy consumption in clouds by appropriately allocating resources (i.e. CPU and memories) to applications without overloading the PMs [12]. In a data center, allocation requests arrive from time to time. A cloud provider

• *School of Engineering and Computer Science, Victoria University of Wellington, PO Box 600, Wellington 6140, New Zealand. E-mail: {Boxiong.Tan, Hui.Ma, Yi.Mei, Mengjie.Zhang}@ecs.vuw.ac.nz*

Manuscript received x x, x; revised x x, x.

needs to allocate proper resource as requests arrived, which is called an on-line resource allocation problem. The problem of on-line *Resource Allocation in Container-based clouds (RAC)* is more difficult than resource allocation in VM-based clouds because of the finer granularity of resources. Both levels of allocation need to be optimized and the interaction between the allocations also makes it difficult to find the optimal solution.

Current works mostly employ rule-based approaches [13] to achieve fast and acceptable solutions. This is because the on-line *RAC* problem requires a real-time solution. Rule-based approaches are reactive approaches which response to arrivals of containers and the needs of container migration. Rule-based approaches are effective for solving *RAC* problem because they consider some features that are related to resource allocation at the time of container arrivals. Other approaches such as meta-heuristics-based ones [14], [15] are too slow for the on-line problem.

Current rule-based works have three major drawbacks. Firstly, some research focuses on container-based cloud where the allocation is a single-level problem [16]. These approaches cannot be applied in the container-based clouds. Secondly, the current rules are simple. They only consider simple features (e.g. residual resources of PMs) to make decisions. As a result, these rules cannot adapt to various workload patterns of applications [17] as well as different sets of VM types. The workload patterns of applications have been proven a critical factor to the resource allocation problems [18]. Therefore, the rules that ignore the patterns and VM types will lead to poor performance. The third drawback of the framework of AnyFit-based algorithms [19], [20] is that it always starts from allocating containers to existing VMs. Hence, it limits the decision and its performance.

In our previous works, we addressed the above issues by proposing a GPHH-based approach for the single-level allocation problem [17] and a hybrid GPHH with heuristics [21] for the *RAC* problem. However, previous studies learn a single-level allocation rule for allocating containers to VMs. On the other hand, the allocation of VMs to PMs is manually designed.

In this work, we propose a novel Cooperative Coevolutionary Genetic Programming (CCGP)-based approach that generates allocation rules for both levels simultaneously. Several reasons motivate us to propose the CCGP approach. Firstly, the CCGP is a learning algorithm that can train the allocation rules off-line, and then apply them to solve the on-line *RAC* problem. Secondly, CCGP trains multiple rules simultaneously by considering the interactions between them. CCGP has been successfully employed in many cooperative problems such as Dynamic Flexible Job Shop Scheduling (DFJSS) [22], multi-robot path planning [23] and multi-depot vehicle routing problem [24]. These combinatorial optimization problems have multiple cooperative stages which are similar with the on-line *RAC* problem. Thirdly, CCGP can learn the workload patterns by using the historical resource requirements of containers. Lastly, CCGP can easily generate reservation-based rules as shown in [21]. Therefore, CCGP is a promising technique for our problem.

The overall goal of this paper is to propose a CCGP for solving the on-line *RAC* problem. In particular, we will design a CCGP approach to automatically generate allocation rules for both containers–VMs level and VMs–PMs level to minimize the overall energy consumption. More specifically, we have the following objectives:

1) Propose a formal model of the on-line *RAC* problem with the consideration of VM overheads, VM types, and affinity constraint.
2) Design a terminal set including features of workloads and VM types for the CCGP approach.
3) Develop a CCGP approach for evolving the allocation rules for two levels simultaneously.
4) Evaluate our proposed approach by comparing it with human-designed rules and a state-of-the-art approach [21] on benchmark datasets;

The novelties of our proposed CCGP approach are demonstrated in the following aspects. CCGP uses a new representation that is able to represent two rules at the same time, with one for allocating containers to VMs and the other one for allocating VMs to PMs. In addition, CCGP is based on a cooperative coevolution (CC) framework that can coordinate multiple evolutionary algorithms to evolve simultaneously. To apply the general CC framework to solving the *RAC* problem, we designed novel problem-specific terminals, the attributes for PM selection, which are different from the terminals in GPHH [21].

The paper is organized as follows. Section 2 gives a background of our methodology and discusses related studies of the on-line *RAC* problem. Section 3 first presents the model of the *RAC* problem. Then, it introduces the proposed CCGP approach. Section 4 illustrates the experiment design, results, and analysis. Section 5 provides the analysis of human-designed rules and CCGP evolved rules. Section 6 summarizes the contributions and discusses the future works.

2 Literature Survey

This section gives a brief background of the hyper-heuristics, GPHH, and CCGP. Then, we discuss the related works of the on-line *RAC* problem from both perspectives of models and methods.

2.1 Hyper-Heuristics, GPHH, and CCGP

Hyper-heuristic is a learning method which searches in the heuristic space rather than the solution space [25]. Hyper-heuristic exploits the structure of a problem and uses the domain knowledge to automatically design heuristics for that problem. Although the domain knowledge is still provided by domain experts, human are freed from the difficulty of manual search for the best ways of combining potential components. Hyper-heuristic algorithms can be categorized into two groups: *selective* and *generative* [26]. Selective hyper-heuristics rank the best heuristics from a set of heuristics. Generative hyper-heuristics generate heuristics from a set of building blocks or domain knowledge given by domain experts. In our problem, we mainly focus on generative hyper-heuristics because we aim at generating effective rules for the *RAC* problem.

In recent years, genetic programming (GP) has become popular in generative hyper-heuristics known as GP Hyper-Heuristic (GPHH) [27]. GP can represent and evolve complex rules, and is therefore naturally suitable for heuristics generation.

GPHH uses GP as a search mechanism to automatically find computer programs for solving a specific task. GPHH optimizes computer programs in an iterative fashion. In the beginning, GPHH generates a population of individuals where each individual represents a computer program. Then the evolution starts. A fitness function is defined to evaluate these programs in each iteration. Then, top individuals will be retained in the next population. Other individuals will be modified by genetic operators, such as crossover and mutation, and added to the new population. The evolution stops after a predefined number of iterations.

CCGP combines GP with a cooperative framework [28], so that CCGP can simultaneously evolve multiple heuristics to solve a problem. CCGP maintains N sub-populations for generating N heuristics respectively.

GPHH has been successfully applied in a variety of problems. In Job Shop Scheduling (JSS) problems, GPHH has been widely used for evolving dispatching rules for various of JSS problems such as multi-objective JSS [29] the multi-task JSS [30], and the JSS with machine breakdown [31]. The generated rules outperform neural network techniques. For the bin packing problems, GPHH has been applied to evolve the rules for 1-dimension [32], 2-dimension [33], and 3-dimension [34] problems. In these cases, the generated rules outperform human-designed rules in terms of performance. Further, the automatic learning procedure greatly reduces the complexity of the heuristic-design process.

CCGP has been used for generating multiple cooperative heuristics. In [35], CCGP generates sequencing and routing rules for Dynamic Flexible JSS (DFJSS) [22], [36]. Similarly, Zhou et al. [37] employ CCGP to evolve machine assignment and job sequencing rules for a multi-objective DFJSS problem. CCGP is suitable for solving the *RAC* problem because it can design multiple cooperative rules and these rules adapt to the changing workload patterns.

2.2 Related Work

This section reviews the current studies on *Resource Allocation in Container-based clouds (RAC)* in terms of the problem model and methods. Then, we summarize their drawbacks which motivate us to improve the performance of *RAC*.

2.2.1 Problem Models

The *RAC* problem is NP hard [38], we need to model *RAC* problem as mathematical optimization problems that reflect the complexity of deployment but make the discovery of sufficiently good solution tractable. Researchers have simplified the problem model with different assumptions. For example, Zhang et al. [11] study resource allocation for applications without considering service arrive and departure time. Zhang et al. [39] assume all the applications will be hosted for a period of time. Many researchers [40], [41] find that live migration introduces high overhead and downtime. Wolke et al. [41] suggests that allocation could be performed periodically and treated as a static problem which focuses on container placement and therefore does not consider migration overhead. Other researchers [13] study resource allocation in clouds by focusing on container migrations, for which container migration overhead is considered in order to decide the time and the number of containers to migrate. Similar to the studies in [42], [43], our work studies resource allocation for applications when allocation request arrives. Applications would normally be hosted in clouds for much longer time than the time used by allocation process.

We summarize the existing models of *RAC* problem from the perspectives of objectives, dimensions of resources, and constraints. Existing studies for the *RAC* mainly have two objectives. The first one is from the perspective of cloud providers. They focus on minimizing the energy consumption of the used PMs [13], [42], [44], [45] or improving the utilization of resources [46]. Guan et al. [16] and Zhang et al. [47] consider not only the energy consumption but also the cost of the data exchange between containers. Fan et al. [46] focus on improving the utilization of PMs and load balancing between the VMs in the same PM. The other objective represents service providers. Guerrero et al. and Nardelli et al. [9], [48] aim to minimize the cost of the used resources. In our work, we focus on the energy consumption of cloud data centers.

Most of the existing studies [13], [43], [45] model the *RAC* problem as a vector bin packing problem [20]. They generally consider two dimensions of resources, i.e. CPU and memory. Another study [46] considers more resources such as local and remote disks.

As for the constraints, current studies generally consider two types of constraints. The first one is the resource constraint [13], [43], [45]–[47] where the total resource requirement of containers cannot exceed the capacity of the VMs and the total VM capacities on a PM cannot exceed the PM. The second one is that each container should only be allocated to one VM [45], [46].

Current studies mainly ignored three characteristics in the model of *RAC* problem. The first characteristic is the overall measurement of energy. Most of the existing studies evaluate allocations by measuring the temporal energy consumption at a certain time point [43]. However, the evaluation is not fair because the energy consumption of a data center is determined by the overall energy consumption of a given period [49]. For example, Fig. 2 shows the curves of energy consumption from two methods. Although energy consumption is the same at time t, the difference in the actual energy consumption during the entire period (the areas under the curves) can be huge. Therefore, to address this issue, we consider the *accumulated* energy consumption as a quality measure.

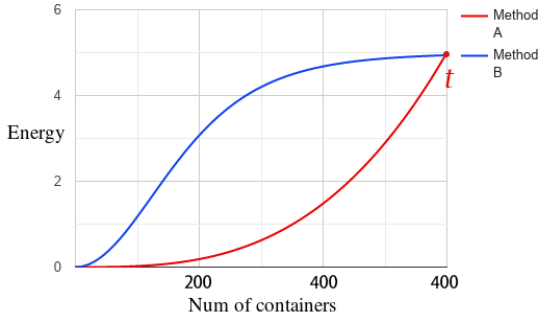


Fig. 2: The energy consumption at time t are same for method A and B.

The second limitation is that the existing works do not consider VM overheads. As a result, small VMs are often

selected for containers. However, a large number of small VMs leads to VM sprawl [50]. On the other hand, creating large VMs leads to unused VM resources [43]. This trade-off is the core issue in the VM creation problem. Some research considered the existence of overheads [42], [43], but they do not provide much analysis. We consider VM overheads in the model and Section 5.1 will provide an analysis of how the overheads affect energy consumption.

Additionally, no study considered affinity constraints of *RAC*. Affinity constraints define which containers can be co-allocated. Without the affinity constraint, all containers can be allocated directly to PMs. Containers require distinct Operating Systems (OSs) and software libraries. Therefore, not all containers can be consolidated into a single VM [51]. Therefore, we consider the requirement of OSs as the affinity constraints in this work.

In summary, existing works either use biased evaluation measure to evaluate container allocations, ignore VM overheads or affinity constraints. In this work, we will address the deficiencies of existing works in our problem model.

2.2.2 Existing allocation methods

For solving the on-line *RAC* problem, most research [13], [42], [43], [52] apply an AnyFit-based framework [19] with a human-design rule. Since the *RAC* problem has been considered as a vector bin packing problem with multiple resources, we need a rule to measure the residual resources. Many rules have been proposed to combine multiple resources into a single value [19].

Energy-aware BestFit [52] and *Least Full Host Selection* algorithm [13] essentially are the same algorithm. They use the same energy evaluate function (see in Eq. (2)). Therefore, these two approaches select PMs with the least CPU usage. Mann [43] applies six rules (such as sub, sum, and $product$) for the *RAC* problem. Wood et al. [53] propose a *volume* rule to allocate resources in VM-based clouds. They choose target PMs with the least $volume = \frac{1}{1-cpu} * \frac{1}{1-mem}$.

However, the performances of human-designed rules vary on different workload patterns [17]. To generate rules that can adapt to the given workload patterns, a GPHH approach is proposed [17]. The experiments have shown that the performance of the rules generated by GPHH outperformed human-designed rules in both balanced and unbalanced workload patterns.

In the *RAC* problem, most of the research employs AnyFit-based algorithms such as Best-Fit and First-Fit. AnyFit-based algorithms always select existing VMs until no VM is available. Then, they apply a simple heuristic such as a *Just-Fit* [43] or *Largest* [43] to create VMs. These simple heuristics may not lead to the optimal allocation at the end. Because they either create a large number of small VMs, which wastes the resources on VM overheads or create large but empty VMs.

To allow a more flexible allocation of containers, a reservation-based algorithm [20] considers not only the existing VMs but also new VMs. In particular, a GPHH approach is proposed to generate allocation rules to select the existing VMs as well as new VMs, while the First-Fit rule is used to allocate new VMs to PMs. Since the VM creation and selection are combined, the search space is more comprehensive and we have higher chances to find the optimal solution.

Hence, this work proposes a CCGP approach to simultaneously generate rules for both levels of resource allocation with the consideration of workload patterns and VM types. Specifically, for the container–VM level, we generate reservation-based rules so that it avoids the drawback of the Anyfit-based rules. The generated rules cooperate to achieve a near-optimal allocation of containers.

3 Methodology

This section introduces the proposed problem model and the CCGP approach.

3.1 Problem Model

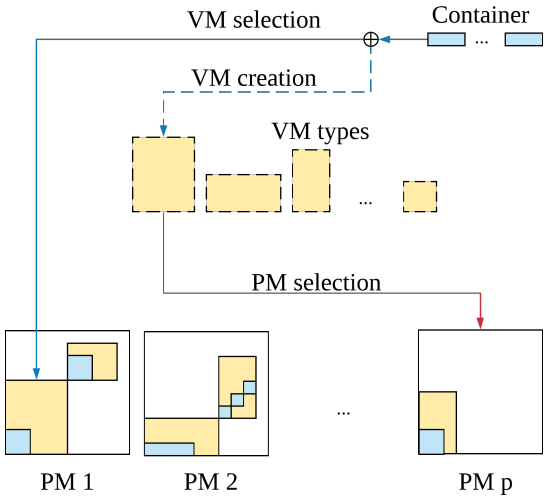


Fig. 3: The procedure of the on-line *RAC*.

The *Resource Allocation in Container-based Clouds (RAC)* problem is a task of allocating a set of containers to a set of VMs with various types, then allocate the created VMs to a set of PMs. Fig. 3 illustrates the allocation with three decision-making procedures: VM selection, VM creation, and PM selection. VM selection chooses an existing VM to allocate a container. VM creation selects a type of VM, creates a VM with the selected type and allocates the container to the new VM. The types of VM are defined by cloud providers. PM selection chooses an existing PM to allocate the new VM. If there is no available PM, a new PM will be created and the data center automatically allocates the new VM to the new PM. Since the PMs are homogeneous, no decision is needed for PM creation.

We show the notations used in our model and their descriptions in Table.1. In the *RAC* problem, a sequence of containers $\mathcal{C} = \{c_1, \ldots, c_n\}$ arrives to the cloud to be allocated. For the sake of simplicity, we assume they arrive at a constant pace, i.e. container c_i arrives at time i. Each container c_i has a CPU occupation $\zeta^{cpu}(c_i)$, a memory occupation $\zeta^{mem}(c_i)$, and the operating system $OS(c_i)$ for running it. There is a set of VM types $\Gamma = \{\tau_1, \ldots, \tau_m\}$ that can be selected to allocate the containers. Each VM type τ_j has a CPU capacity $\Omega^{cpu}(\tau_j)$ and a memory capacity $\Omega^{mem}(\tau_j)$. In addition, it has a CPU overhead $\pi^{cpu}(\tau_j)$ and

TABLE 1: Notation and description of the problem model

Notation	Description
c_i	a container of index i
τ_j	the VM type of a VM j
p_k	a PM of index k
x_{il}	An indicator of whether the container i is allocated to the l created VM
y_{lk}	An indicator of whether the lth created VM is allocated to the kth PM
z_{jl}	An indicator of whether the lth created VM is of type j
E	The energy consumption of the data center over the allocation period
E_{tk}	The energy consumption of the kth PM at time t
EP_k^{idle}, EP_k^{full}	The energy consumption when the kth PM is idle and fully used
$\zeta^{cpu}(c_i), \zeta^{mem}(c_i)$	The CPU and memory occupation of the ith container
$\Omega^{cpu}(), \Omega^{mem}()$	The CPU and memory occupation of a resource entity
$\pi^{cpu}(\tau_j), \pi^{mem}(\tau_j)$	The CPU and memory overheads of a VM type of τ_j
$OS(c_i)$	The operating system type of the ith container
μ_{tk}^{cpu}, μ_{tk}^{mem}	The CPU and memory utilization of a the kth PM at time t

memory overhead $\pi^{mem}(\tau_j)$, indicating the CPU and memory occupation for creating a new VM of that type. There is an unlimited set of PMs $\mathcal{P} = \{p_1, \dots, \}$ for allocating the created VMs. Each PM p_k has a CPU capacity $\Omega^{cpu}(p_k)$ and a memory capacity $\Omega^{mem}(p_k)$.

The *RAC* is subject to the following constraints:

1) Each container is allocated to one VM.
2) Each created VM is allocated to one PM.
3) For each created VM, the total CPU and memory occupations of the containers allocated to that VM does not exceed the corresponding VM capacity.
4) For each PM, the sum of the CPU and memory capacities of the VMs allocated on the PM does not exceed the corresponding PM's capacity.
5) For each created VM, the installed operating system must be the same as the required operating system of all the allocated containers.

The accumulated energy consumption over the allocation period is calculated as follows.

$$E = \sum_{t=1}^{n} \sum_{k=1}^{K} E_{tk}, \tag{1}$$

where E_{tk} is the energy consumption of the kth PM (K is the number of PM used) at time t.

E_{tk} is calculated as follows.

$$E_{tk} = E_k^{idle} + (E_k^{full} - E_k^{idle}) \cdot \mu_{tk}^{cpu}, \tag{2}$$

where E_k^{idle} and E_k^{full} indicate the energy consumption of the kth PM per time unit if it is idle and fully loaded, respectively. μ_{tk}^{cpu} indicates the CPU utilization level of the kth PM at time t. μ_{tk}^{cpu} is calculated as follows.

$$\mu_{tk}^{cpu} = \frac{\sum_{l=1}^{L} \left(\sum_{j=1}^{m} \pi^{cpu}(\tau_j) \cdot z_{jl} + \sum_{i=1}^{n} \Omega^{cpu}(c_i) \cdot x_{il} \right) \cdot y_{lk}}{\Omega^{cpu}(p_k)} \tag{3}$$

where x_{il}, y_{lk} and z_{jl} are binary decision variables, and L is the number of created VMs. x_{il} takes 1 if c_i is allocated to the lth created VM, and 0 otherwise. y_{lk} takes 1 if the lth created VM is allocated to the kth PM, and 0 otherwise. z_{jl} takes 1 if the lth created VM is of type j, and 0 otherwise.

Given the above mathematical notations, the *RAC* problem can be formulated as follows.

$$\min \ \sum_{t=1}^{n} \sum_{k=1}^{K} E_{tk}, \tag{4}$$

$$s.t. \ \sum_{l=1}^{L} x_{il} = 1, \ \forall \, i = 1, \dots, n, \tag{5}$$

$$\sum_{k=1}^{K} y_{lk} = 1, \ \forall \, l = 1, \dots, L, \tag{6}$$

$$\sum_{j=1}^{m} z_{jl} = 1, , \ \forall \, l = 1, \dots, L, \tag{7}$$

$$\sum_{i=1}^{n} \zeta^{res}(c_i) x_{il} \leq \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl}, \quad \forall \, l = 1, \dots, L, \ res \in \{cpu, mem\}, \tag{8}$$

$$\sum_{l=1}^{L} \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl} \leq \Omega^{res}(p_k), \quad \forall \, k = 1, \dots, K, \ res \in \{cpu, mem\}, \tag{9}$$

$$OS(c_{i_1}) = OS(c_{i_2}), \ \forall \ \sum_{l=1}^{L} x_{i_1 l} x_{i_2 l} = 1, \tag{10}$$

$$x_{il}, y_{lk}, z_{jl} \in \{0, 1\}, \tag{11}$$

where constraints (5) and (6) indicate that each container (or created VM) is allocated to exactly one created VM (or PM). Constraint (7) indicates that each created VM must belong to a type. Constraint (8) implies that the total occupation of all the containers allocated to each created VM does not exceed its corresponding capacity. Constraint (9) indicates that the total capacity of the created VMs allocated to each PM does not exceed its corresponding capacity. Constraint (10 means that the containers allocated to the same VM must have the same required operating system, which is the installed operating system on that VM. Constraint (11) defines the domain of the decision variables.

This model is used in the simulation of our experiments to evaluate the container allocation algorithms. The newly

introduced features, VM overheads, are also considered in our CCGP approach.

3.2 CCGP

This section describes the proposed Cooperative Coevolution Genetic Programming (CCGP) approach for the on-line *Resource Allocation in Container-based clouds (RAC)* problem. We first give an overview of CCGP and then introduce the representation of the allocation rules, the terminal set, and the fitness function. In the end, we describe the algorithm in details.

3.2.1 Overview

The on-line *RAC* problem involves three decision processes, VM selection, VM creation, and PM selection (see Section 3.1). We propose a CCGP approach to automatically generate rules for these processes. In our approach, we combine the VM selection and VM creation and uses a single rule to make both decisions. The other rule *PM selection* is also generated by the CCGP simultaneously.

To generate rules to solve the on-line *RAC* problem, we design the CCGP to search for the rules which adapt to the training data including various workload patterns and VM types. The training process is performed off-line and may take a long computation time (hours). The generated rules will then be used to solve on-line *RAC* problem.

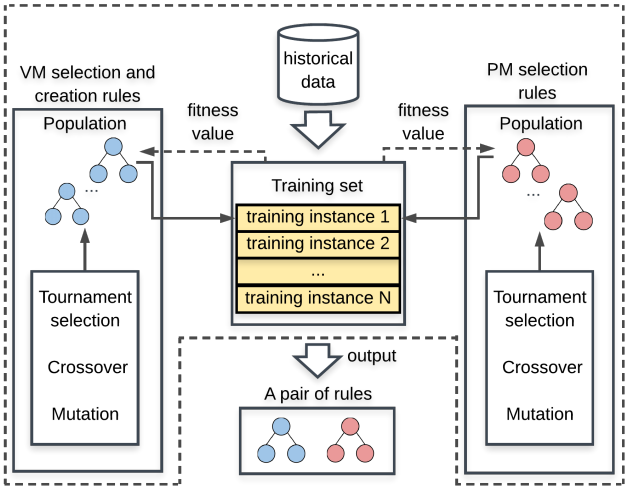


Fig. 4: The overview of the training process of CCGP.

Fig. 4 shows an overview of the training process of CCGP. The CCGP initializes two populations of rules randomly. Then, the rules are evolved cooperatively by the genetic (e.g. crossover and mutation) operators. A set of training instances are used for fitness evaluation. To evaluate a rule, first, the rule is combined with a collaborator from the other sub-population to form a pair of rules. Then, the pair of rules are applied to each training instance to generate an allocation solution. Finally, the average quality (i.e. accumulated energy consumption) of the allocation solutions is set to the fitness of the evaluated rules. By iteratively evaluating and modifying the rules, the population gradually searches in the space of rules. The performance keeps improving because only the best pairs of rules are kept to the next generation. This process of evaluation and modification continues until a predefined number of iterations is reached. Finally, the CCGP outputs a pair of rules with the best fitness value in the training process. The pair of rules will then be used to generate the allocation solution for any (unseen) *RAC* problem instance.

Notice that, CCGP is different from our previous GPHH approach [17] in the training process. CCGP evolves two populations of rules. In each generation, the best individuals of two populations are selected to cooperate with the rules from the other population, while, in GPHH, the individuals of cooperates with a predefine rule throughout the whole evolution process.

3.2.2 Representation, Terminal Set, and Function Set

The allocation rules act as a priority function which assigns a score to each candidate allocation decisions. With the score, we can decide which VM/PM to allocate the container/VM. The allocation rules are constructed by the features in the terminal set and function set.

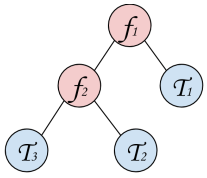


Fig. 5: The tree-based representation of a rule.

We use trees to represent rules (see Fig. 5). The benefit of using trees is that they can be easily interpreted as a prefix notation [54]. In addition, since the trees can grow, they can represent a complex relationship. Furthermore, trees are easy to be manipulated such as by pruning and reconstructing, therefore, we can easily evolve them to search in the rule space.

The nodes on Fig. 5 are drawn from the terminal set (the T nodes) and the function set (the f nodes). To generate sophisticated rules, we consider many features of the *RAC* problem and use them as a terminal set of GP trees. Table 2 describes the terminal and function sets that we used in CCGP. Note that the terminal sets for *VM selection and creation* rules and *PM selection* rules are different while the function set is the same. The design of the terminals follows the research [33] on 2D bin packing.

An illustration of the terminals is shown in Fig. 6. The *leftVmMem* and *leftVmCpu* are the remaining resources of a VM. They are calculated as subtracting the configuration resources of the VM by the overhead of that VM and the resources used by the containers running on the VM. The *leftPmMem* and *leftPmCpu* are the remaining resources of a PM. They are calculated as subtracting the configuration resources of the PM by the configuration resources of the VMs running on that PM. The protected % returns 1 when the denominator is 0.

As we can see from above, new features, the attributes of PMs, are considered in the CCGP approach in addition to those used in GPHH [17].

3.2.3 Fitness Function

To evaluate a pair of rules, we need to first apply the pair of rules on a training instance. Then, we use a fitness function

TABLE 2: Terminal and Function sets of CCGP

Symbol	Description
Attributes for VM selection and creation	
leftVmMem	remaining memory of a VM
leftVmCpu	remaining CPU of a VM
vmMemOverhead	memory overhead of a VM
vmCpuOverhead	CPU overhead of a VM
coCpu	container CPU requirement
coMem	container memory requirement
Attributes for PM selection	
leftPmMem	remaining memory of a PM
leftPmCpu	remaining CPU of a PM
vmMem	the configuration memory of a VM
vmCpu	the configuration CPU of a VM
Function set	+,-,×, protected %

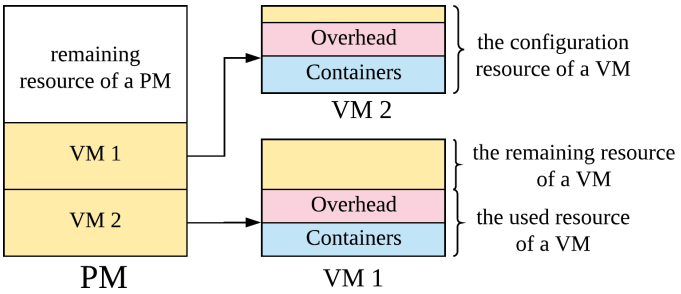


Fig. 6: Illustration of the features used in the terminal set

Algorithm 1: The procedure of the container allocation

Input : VM selection and creation rule vr, PM selection rule pr
Output: accumulated energy consumption AE

```
1  for a training instance in S do
2      AE = 0;
3      Initialize the data center;
4      for each container in C do
5          Filter the VMs with a conflicting OS with the
            current container;
6          vm = vmSelectionCreation(container, vr);
7          allocate(container, vm);
8          if vm is new then
9              add(vm, the list of VMs);
10             pm = pmSelection(vm, pr);
11             if pm is null then
12                 pm = pmCreation();
13                 add(pm, the list of PMs);
14             end
15             allocate(vm, pm);
16         end
17         AE += calculateEnergy(pm);
18     end
19 end
20 return AE;
```

to calculate its performance and represented as a fitness value.

Algorithm 1 shows the procedure of an allocation process. At the beginning, the training process randomly initializes a data center by generating a number of PMs and allocating a random number of containers into them. Then, a set of containers C arrives and are allocated one by one. The VMs with conflicting OS are filtered before the selection. The *VM selection and creation rule* vr is used to select an existing VM or create a new VM in *line 6*. The idea of the *VM selection and creation rule* is to include the new VM (one for each type) into the candidate VM list. If a new VM is created, the OS of the VM is set to the same as the container's OS requirement. Then, the PM selection rule pr is used to select among the existing PMs in *line 10*. If there is no available PM, *line 12* creates a new PM to host the VM. The training procedure outputs the accumulated energy consumption AE as introduced in Eq. (1).

The fitness function is then used to evaluate the pair of rules. The fitness function is designed as follows:

$$fitness = \frac{\widetilde{AE}}{N} \tag{12}$$

where $\widetilde{AE}$ is the normalized accumulated energy consumption of the allocation of a training instance. N is the number of containers. With $\frac{\widetilde{AE}}{N}$, we calculate the average accumulated energy consumption per container. The average AE is converging when the number of container increases. Therefore, it is easy for us to show the stableness of the rules.

We normalize the AE of a rule with the accumulated energy consumption of a benchmark rule (e.g. AE_b) using Eq. (13). The reason that we use normalized AE is that different training instances have major differences. It is unfair to use the aggregation of AE of all training instances to compare algorithms. In our experiments, the normalization is based on the benchmark rule *sub&Just-Fit/FF* which is explained in Section 4.1.2 i.e.

$$\widetilde{AE} = \frac{AE}{AE_{sub\&Just-Fit/FF}} \tag{13}$$

3.2.4 Algorithm

The proposed CCGP approach is described in Algorithm 2. It starts with the initialization of two sub-populations of P_{vr} and P_{pr} (*line 1*). Each sub-population contains N randomly generated rules. We apply the *ramped Half-and-Half* [54] in constructing trees to ensure the diversity in each sub-population.

The iteration of evaluation and modification begins at *line 4* and repeats until a predefined *maxGeneration* is reached. The gen is the counter of the iteration. We evaluate the rules in turns (the loop from *line 5* to *line 11*). At the beginning, each rule p_i^r from P_{vr} is paired with a representative rule $p_{rep}^{r'}$ from the P_{pr} (the loop from *line 6* to *line 10*). The best rule is defined as a representative of that sub-population. In the first generation, we randomly select a rule from P_{pr} as the representative. Then the pair of rules is evaluated in *line 7*.

In the evaluation stage, we apply the pair of rules to allocate a set of containers to a set of PMs. The detailed allocation process is shown in Algorithm 1. The allocation returns the accumulated energy consumption of AE. Then, we normalize the value (*line 8*) by Eq. (13) and calculates the fitness value (*line 9*) by Eq. (12). The information on containers and PMs are given by a training instance s^{gen}

Algorithm 2: CCGP for the on-line *RAC*

Input : A set of training instance S,
Terminal sets and function sets
Output: The best VM selection and creation rule,
The best PM selection rule

Initialize each sub-population P_r with $r = \{vr, pr\}$
$P_r \leftarrow \{p_1^r, p_2^r, \cdots, p_N^r\}$;
$gen \leftarrow 0$
while maxGeneration *is not reached* **do**
  for $r = vr \rightarrow pr$ **do**
    for $i = 1 \rightarrow N$ **do**
      $AE \leftarrow$ apply p_i^r and $p_{rep}^{r'}$ on the training instance s^{gen} where $r' \neq r$ (see Algorithm 1);
      calculate $\widetilde{AE}$;
      $p_i^r \leftarrow \frac{\widetilde{AE}}{N}$;
    end
  end
  for $r = vr \rightarrow pr$ **do**
    $p_r^{selected} \leftarrow$ TournamentSelection(p_r);
    $p_r \leftarrow$ genetic_operators($p_r^{selected}$);
  end
  $gen \leftarrow gen + 1$
end

from the training set. We switch to a different training instance in each generation to improve the generalization of the rules.

When all rules have been evaluated, we apply the tournament selection and genetic operators on two sub-populations. The tournament selection [54] guides the evolutionary process. The rules with higher fitness values have larger probabilities to be selected. Then, two genetic operators, crossover and mutation, are applied on the selected rules.

Crossover and mutation stochastically generate new solutions from the selected rules. The crossover randomly selects the branches on two selected rules and switch the branches. The mutation randomly selects a branch and replaces it with a randomly generated branch. After the modification by genetic operators, new rules are added to the new generation of the population. The tournament selection and genetic operators keep generating new rules until the new population has the same number of rules as before. Then, the next iteration starts.

4 EXPERIMENTS

To evaluate the performance of our proposed CCGP approach, we conduct experiments using two real-world datasets and compared with two state-of-the-art methods. This section first illustrates the experiment design including datasets, compared methods, and test instances, and then shows the results.

4.1 Experiment Design

The experiment compares our CCGP and other methods to demonstrate the effectiveness of CCGP. We evaluate the performance of CCGP approach with a variety of scenarios, with different complexities, i.e., 30 of VM types, 3 types of OS, and two real-world datasets. We use *evo/evo* to represent the two evolved rules by CCGP.

All algorithms were implemented in Java version 8 and the experiments were conducted on an i7-4790 3.6 GHz with 8GB of RAM running Linux Arch 4.14.15-1.

4.1.1 Simulation

We designed and implemented a simulator which is used in the training and testing the rules for container allocation. The purpose of the simulator is to test that whether the evolved rules can outperform the existing algorithms in the allocation tasks. Therefore, other features are ruled out to eliminate their effects. The rules are independent from the simulator and can be used in other data center simulators.

Below are the configurations of the simulator.

1) Containers arrive uniformly between [0, T];
2) Arrived containers must be allocated immediately;
3) Overload threshold of VM/PM is 100% of resource utilization;
4) No weight or priority of containers, which means containers are equally important;
5) Two sets of VMs types and homogeneous PMs (all PMs have the same initial resources);
6) Assume an infinite number of available VMs/PMs that can be used;

Each simulation allocates a set of containers into an initialized data center. The simulation starts with a randomly initialized data center, which contains a set of containers running on VMs/PMs. Then, a set of containers arrives at the data center one by one. A simulation uses four allocation rules, VM creation, VM selection, PM creation, and PM selection, to allocate the containers into the data center.

We design the data center initialization to simulate a real-world scenario in which PMs are running in different utilization levels. Random initialization can also help to train rules that are robust to a different initial state of data centers.

In a data center with an initial state, the container allocation procedure allocates containers to existing VMs or new VMs of some available types. The allocation procedure uses four allocation rules to allocate containers to VMs (existing or new), which are then allocated to PMs (existing or new). In particular, *PM creation* creates a PM with a fixed capacity for all experiments because we consider homogeneous PMs. The *PM creation* rule creates a PM when no existing PM is available. All other rules can be human-designed rules or evolved rules. Once all the containers are allocated, the simulator evaluates the performance of the data center, which is the accumulated energy consumption of all used PMs during the period of allocation.

To reliably measure the effectiveness of the rules, a large number of simulations (e.g. 30 to 50) are usually needed [55]. For training, we use 100 simulations in a rotating manner. That is, we switch to a new training instance at each generation. The purpose of switching simulation is to find good rules for VM selection and creation and PM selection, independent from training instances. This training method has been successfully applied in JSS problems [56]. For testing, we use 30 simulations. The testing result shows an average performance on the simulations.

4.1.2 Benchmark Algorithms

The sub&Just-Fit/FF rule is proposed in [43]. It applies BestFit with the *sub* rule to select the suitable VMs for containers. The *sub* rule is a way to measure the balance between multiple residual resources [57]. The *sub&Just-Fit/FF* rule aims to maximize the balance in order to achieve a better energy efficiency. In our problem, the *sub* rule can be represented as $|cpu - mem|$. The *Just-Fit* first sorts the VM according to a resource. In our case, we sort the resources according to residual memory because memory is the bottleneck in our datasets (see Section 4.1.3). Then, it creates the smallest VM that can satisfy the resource requirement of the container. In the second level, VMs are allocated to PMs with First Fit.

Hybrid GPHH Approach is proposed in our previous work [21]. It applies GPHH evolved rules for VM selection and creation. It uses First Fit to allocate newly created VMs to PMs. We will use *evo/FF* to represent the hybrid approach.

4.1.3 Dataset

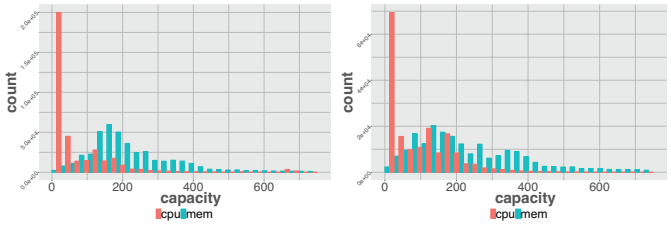
We design 12 scenarios for the experiments (see Table 3) which are divided into four groups. Each scenario has distinct numbers of OS from 3 to 5. For each scenario, we use 100 instances for training and 30 instances for testing. Each instance contains 2500 containers to be allocated. This number of containers is large enough for an algorithm to reach a stable status.

TABLE 3: Test instances

scenarios	number of OSs	VM types	workload patterns
scenario 1	3	synthetic VM types	AuverGrid trace
scenario 2	4	synthetic VM types	AuverGrid trace
scenario 3	5	synthetic VM types	AuverGrid trace
scenario 4	3	synthetic VM types	Bitbrains trace
scenario 5	4	synthetic VM types	Bitbrains trace
scenario 6	5	synthetic VM types	Bitbrains trace
scenario 7	3	real-world VM types	AuverGrid trace
scenario 8	4	real-world VM types	AuverGrid trace
scenario 9	5	real-world VM types	AuverGrid trace
scenario 10	3	real-world VM types	Bitbrains trace
scenario 11	4	real-world VM types	Bitbrains trace
scenario 12	5	real-world VM types	Bitbrains trace

To generate the instances, we use two real-world workload datasets – *AuverGrid trace* and *Bitbrains trace* [58]. The original workload trace files contain millions of lines of CPU and memory usage records of applications. For each dataset, we select the first 400,000 lines of records as the source files. Then, we filtered the records to exclude the containers that require more resources than the largest VM type (see Table 4 and Table 5). The last step is to randomly select the resource requirement of containers to construct an instance. Fig. 7 shows the distributions of CPU and memory requirements in two datasets.

For the configurations of PM and VMs, we use a quad-core PM size of [13200 MHz, 16000 MB] which has been used in [43]. We assume each core has 8 threads. To simulate real-world VM configuration, we use the information of VM types offered by Amazon EC2 in Table 4. Additionally, to generalize the VM configuration, we randomly generate



(a) AuverGrid trace **(b)** Bitbrains trace

Fig. 7: Resource usage frequency in the real-world datasets

10 VM types (see Table 5) where the values of CPU and memory are sampled from [0, 3300 MHz] and [0, 4000 MB] representing the capacity of one core.

TABLE 4: Real-world VM types

VM types	[CPU, Memory]	VM types	[CPU, Memory]
1	[206.25, 250]	11	[825, 2000]
2	[412.5, 500]	12	[1650, 250]
3	[825, 1000]	13	[1650, 500]
4	[1650, 2000]	14	[1650, 1000]
5	[412.5, 250]	15	[412.5, 937.5]
6	[412.5, 1000]	16	[825, 1875]
7	[825, 4000]	17	[1650, 3750]
8	[206.25, 500]	18	[412.5, 1312.5]
9	[412.5, 2000]	19	[825, 2625]
10	[412.5, 4000]	20	[2475, 2625]

VM types	[CPU (MHz), Memory (MB)]	VM types	[CPU, Memory]
1	[719, 2005]	6	[1311, 3238]
2	[917, 951]	7	[1363, 2634]
3	[1032, 1009]	8	[1648, 1538]
4	[1135, 3542]	9	[2047, 1181]
5	[1231, 1989]	10	[2100, 3013]

TABLE 5: Synthetic VM types

For the affinity constraint, we set up an Operating System (OS) constraint. Each container has a requirement of OS and can only be allocated to the VM which has the same OS installed. We simulate three scenarios where the number of OS increases from 3 to 5. The OS requirement of a container is generated from a distribution (see Table 6). We use this distribution to simulate a real-world market share of OS [59].

TABLE 6: OS distribution

number of OS	OS distribution (%)
3	50-30-20
4	62.5-17.5-15.5-4.5
5	17.9-45.5-23.6-10.5-2.6

All results have been tested with Wilcoxon signed-rank test between the rules from our CCGP and the existing rules. The significance level is set to $\alpha = 0.05$.

4.1.4 Parameter Settings

Table 7 shows the parameters that we used in all experiments. All the parameters follow the setting that has been commonly used in literature (e.g. [60]). The CCGP and the hybrid GPHH algorithms were implemented by ECJ [61].

TABLE 7: Parameter Settings

Parameter	Description
Initialization	ramped-half-and-half
Crossover/mutation/reproduction	80%/10%/10%
Maximum Depth	7
Number of generations	100
Sub-Population	512
Selection	tournament selection (size = 7)

4.2 Experiment Results

This section first shows the accumulated energy consumption comparison of the *sub&Just-Fit/FF* rule, the *evo/FF* rules and the *evo/evo* rules. Then, in the detailed results, we show the behaviors of these methods by examining their allocation procedures. We further look at the PM utilization and PM remaining resources to find out what causes the differences in these methods.

4.2.1 Overall Results

The comparison of the accumulated energy consumption among the three methods are shown in Table 8. We can see that the *evo/evo* rules have a major advantage over the *sub&Just-Fit/FF* rule and the *evo/FF* rules in all scenarios.

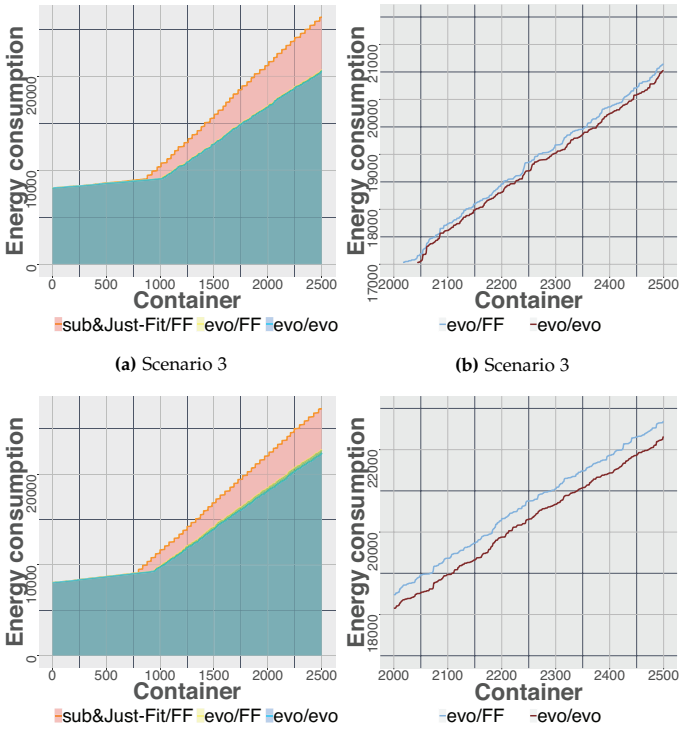
TABLE 8: Mean and standard deviation of the energy consumption (kwh) of 30 instances for 12 scenarios among the *sub&Just-Fit/FF* rule, the *evo/FF* rules, and the *evo/evo* rules.

	sub&Just-Fit/FF	*evo/FF*	*evo/evo*
scenario 1	3.75E7 ± 1.6E6	3.16E7 ± 2.3E6	**3.14E7 ± 2.3E6**
scenario 2	3.75E7 ± 1.6E6	3.16E7 ± 2.3E6	**3.15E7 ± 2.3E6**
scenario 3	3.77E7 ± 1.6E6	3.18E7 ± 2.3E6	**3.16E7 ± 2.3E6**
scenario 4	4.11E7 ± 1.7E6	3.39E7 ± 2.3E6	**3.38E7 ± 2.3E6**
scenario 5	4.11E7 ± 1.7E6	3.41E7 ± 2.3E6	**3.39E7 ± 2.3E6**
scenario 6	4.12E7 ± 1.7E6	3.42E7 ± 2.3E6	**3.40E7 ± 2.3E6**
scenario 7	3.72E7 ± 1.7E6	3.17E7 ± 2.3E6	**3.12E7 ± 2.4E6**
scenario 8	3.74E7 ± 1.7E6	3.18E7 ± 2.3E6	**3.13E7 ± 2.4E6**
scenario 9	3.74E7 ± 1.7E6	3.19E7 ± 2.3E6	**3.14E7 ± 2.4E6**
scenario 10	3.86E7 ± 2.0E6	3.42E7 ± 2.4E6	**3.37E7 ± 2.4E6**
scenario 11	3.91E7 ± 1.9E6	3.42E7 ± 2.4E6	**3.39E7 ± 2.4E6**
scenario 12	3.91E7 ± 1.9E6	3.47E7 ± 2.3E6	**3.44E7 ± 2.4E6**

4.2.2 Detailed Results

The *evo/evo* rules achieve good performance in all scenarios and we showed scenario 3 and 12 (see Fig. 8) because others have similar trends. The allocation procedure shows the increment of energy consumption while allocating containers. The left-hand sides are the comparisons of three methods. The right-hand sides are the zoom-in comparisons between the *evo/FF* rules and the *evo/evo* rules. The energy consumption of evolved rules is the average of 30 runs' results. In the beginning, the energy consumptions resulted from three methods increase slowly because containers are allocated to the free spaces in PMs. Since no new PM is created, the performances of all methods look the same (overlapping lines). Later on, the increments of energy consumption are different for three methods. The *evo/evo* rules are the slowest in terms of energy increment. Another noticeable pattern is that the turning point of the *evo/evo* rules is later than the *sub&Just-Fit/FF* rule. This means the *evo/evo* rules allocate more containers into the existing PMs than the *evo/FF* and *evo/evo*. Therefore, the *evo/evo* rules use a smaller number of PMs and the increment of energy consumption is slow.

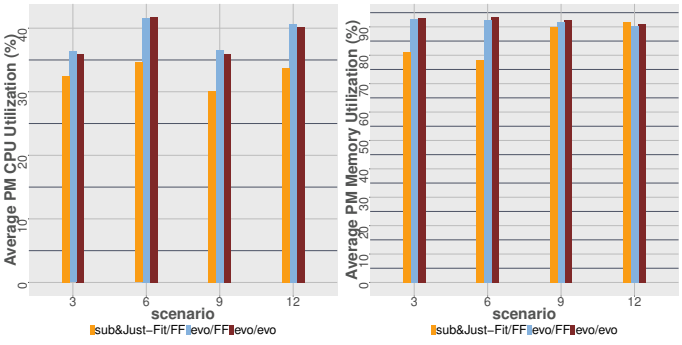
Fig. 8: Allocation process of simulation 0 from scenarios 3 and 12



(a) Scenario 3 **(b)** Scenario 3

(c) Scenario 12 **(d)** Scenario 12

To understand why the *evo/evo* rules has a slower increment of energy consumption compared to other rules, we show the CPU and memory utilization of four representative scenarios, i.e. 3, 6, 9, 12 in Fig. (9). The *sub&Just-Fit/FF* rule generates the lowest utilization in both CPU and memory among all scenarios except the memory utilization of scenario 12. Since the *sub&Just-Fit/FF* rule generally has a low resource utilization, it is not surprising that it uses more PMs and more energy consumption. To compare the *evo/FF* rules and the *evo/evo* rules, the *evo/FF* rules have better CPU utilization while the *evo/evo* rules have better memory utilization in all scenarios. As shown in Section 4.1.3, memory resource is the bottleneck in both real-world datasets. It is now clear that the *evo/evo* rules outperform *evo/FF* rules on the critical resource, e.g. memory. A remaining question is that, compared to the *evo/FF* rules, why *evo/evo* rules can obtain high utilization of memory?



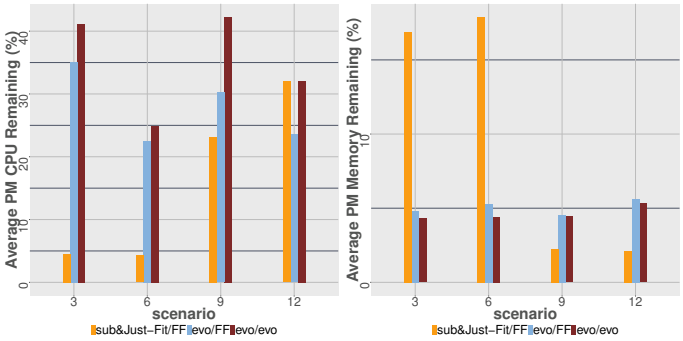
(a) PM CPU utilization **(b)** PM memory utilization

Fig. 9: PM resource utilization

To improve the utilization of resources, one can improve the utilization of VMs, reduce the PMs remaining resources (see Fig. 6), or both. Since the *evo/FF* rules and the *evo/evo* rules show difference only in the VM–PM level, we now focus on the reduction of PM remaining resources. The PM remaining resources are the idle resources in PMs which are affected by two factors, i.e. the number of VMs on the PM and the types of these VMs. The only way to reduce the PM remaining resource is constructing a combination of VMs which uses all or majority resources in a PM. To construct such a combination, the VM creation and PM selection rules must be used together.

From Fig. 10, the *evo/evo* rules have a higher remaining CPU and a lower remaining memory than the *evo/FF* rules do. It means that the *evo/evo* rules use the memory more effectively than the *evo/FF* rules on PMs. This is consistent with the figure of PM utilization shown in Fig. 9. The *evo/evo* rules achieve this because they are co-evolved while the *evo/FF* rules use an independent rule, i.e. First-Fit as the PM selection rule. Therefore, it is hard to construct a good combination of allocations because First-Fit always selects the first available PM. The detailed reason for why the *evo/evo* rules achieve a better memory utilization is explained in the Section 5.

To this end, we have shown the *evo/evo* rules achieve the lowest accumulated energy consumption. The *evo/FF* rules have slightly worse performance than the *evo/evo* rules while the *sub&Just-Fit/FF* rule has the worst performance.



(a) PM remaining CPU **(b)** PM remaining memory

Fig. 10: PM remaining resource

In summary, the experimental evaluations in this section show that the *evo/evo* rules achieve the best performance among three methods. In particular, the rules generated by CCGP can lead to better container to VM allocations and therefore lower PM remaining resources, comparing with other two methods.

5 Rule Analysis

This section further analyzes the drawbacks of the *sub&Just-Fit/FF* rule and shows how CCGP evolved rules make the allocation decisions.

5.1 VM Creation Behavior

We analyze the ratio of VM types and the quantity of VMs of three methods to show the patterns of VM types selection among three approaches. Fig. 11 illustrates the average ratio of VMs (bar chart) and the average quantity of VMs (pie chart) used by three methods in scenarios 3, 6, 9, 12. We these patterns, we found three facts. First, from the pie charts (see Fig. 11), we have seen that *sub&Just-Fit/FF* uses 2 to 4 times more VMs than the evolved rules. Second, we found that the most frequently used VM type by *sub&Just-Fit/FF* is **type 2** which is a small VM type. Third, from the PM utilization (see Fig. 9), the *sub&Just-Fit/FF* also generates the lowest utilization. From these facts, we infer that the *sub&Just-Fit/FF* leads to the VM sprawl.

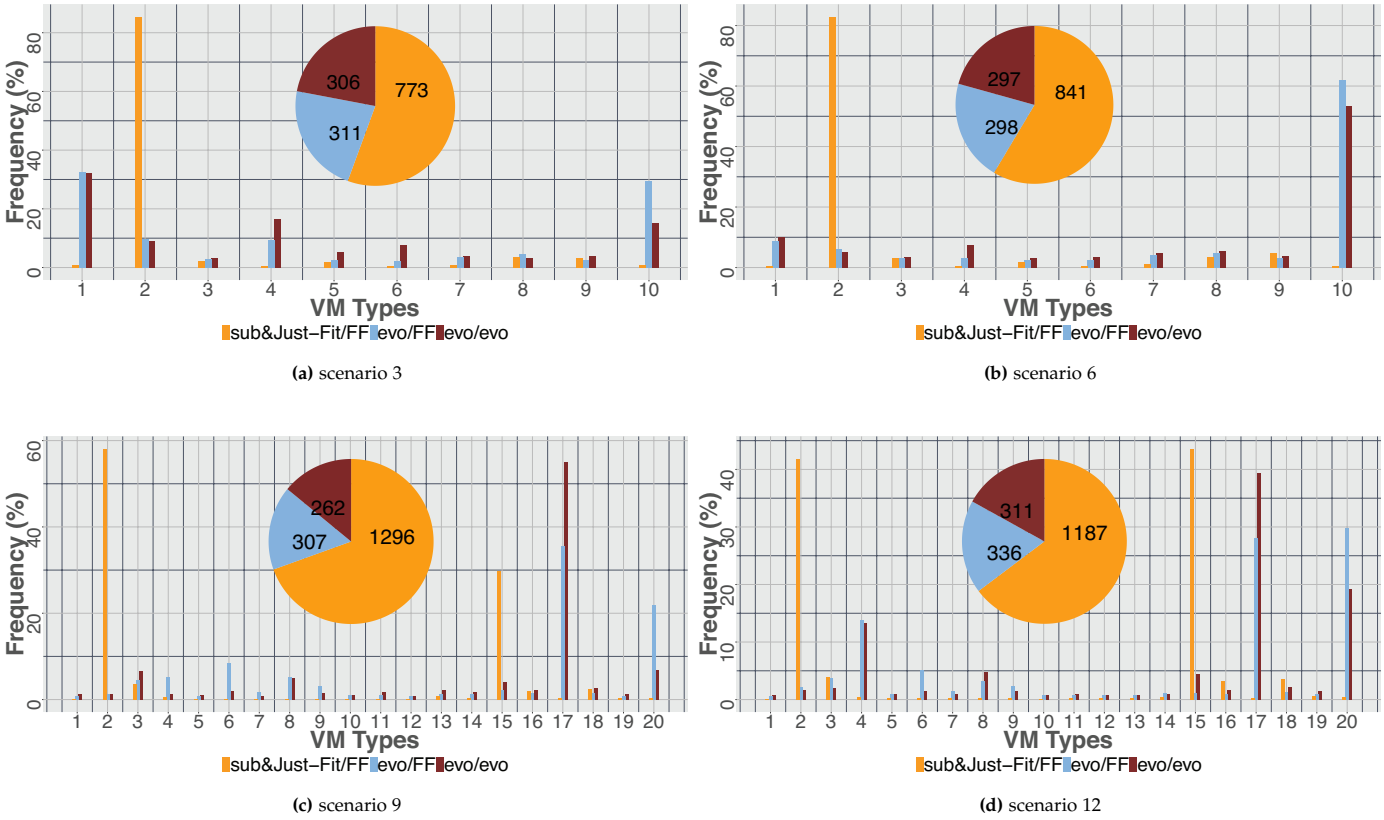
VM sprawl [50] is the major reason for the low utilization of data centers and the *sub&Just-Fit/FF* rule can lead to it. In a data center where VM sprawl occurs, PMs are filled with a large number of small VMs and most of them are low utilized. Therefore, the average of PM utilization is low, e.g. 15% to 20%. From the above patterns of VM types selection, the *sub&Just-Fit/FF* rule has created a large number of small VMs and has the lowest utilization among three methods which are the symptoms of VM sprawl.

To understand the consequence of VM sprawl, we observe the increment of VM wasted memory and memory overhead throughout the allocation process in Fig. 12. This figure shows that when applying the *sub&Just-Fit/FF* rule, memory is consumed by VM overheads and wasted quickly. VM wastes are the small resource segmentation inside VMs that will never be used. The fast accumulation of VM overheads and wastes are due to the vast number of VMs. Therefore, the actual resources used by containers are low when VM sprawl occurs. However, for the evolved rules, the wastes and overheads increase much slower than the *sub&Just-Fit/FF* rule.

The main reason that causes VM sprawl is that the *Just-Fit* rule only greedily considers the resource requirement of the current container. Since most containers have a small resource requirement (less than 100 in CPU and 200 in memory) (see Section 4.1.3). The *Just-Fit*, therefore, tends to create small VMs, e.g. **type 2** and **type 15**. In the scenarios of real-world VM types (scenarios 9 and 12), the *Just-Fit* might achieve a low PM remaining resources (see Fig. 10) because these VM types happen to be divisible (32 VMs can fill a PM). However, with a different set of VM configurations, e.g. synthetic VM types, the *Just-Fit* cannot construct a combination of VM types which uses PM resources efficiently.

On the other hand, the evolved rules can select a good combination of VMs with the given VM types to avoid VM sprawl. The evolved rules consider both the capacities of VMs and the residual resources on PMs. Therefore, they can create a combination of VM types so that PMs' resources are used more efficiently. For example, in scenario 3, evolved rules favor **type 1, 2, 4, 6, 10**. This is because the combination of these types of VM can easily achieve a high memory utilization of PMs. With the combination of **type 10** $\times$ 2, **type 1** $\times$ 3, and **type 6** $\times$ 1, the aggregated memory is 15279 MB which uses 95% of a PM's memory. This is the reason that the evolved rules remain stable in PM utilization and PM remaining regardless of the given set of VM types.

In summary, the *sub&Just-Fit/FF* rule causes VM sprawl by allocating too many small VMs. The *evo/evo* rules create VMs purposefully with the consideration of multiple factors, e.g. PM residual resources and VM types, so that they improve the utilization of PMs and successfully avoid VM sprawl.



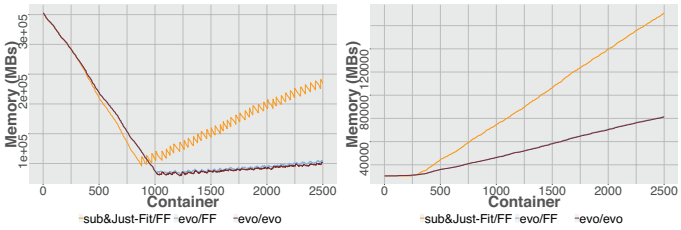
(a) scenario 3

(b) scenario 6

(c) scenario 9

(d) scenario 12

Fig. 11: The average frequency of VM types used by three algorithms



(a) average wasted memory

(b) average Memory overhead

Fig. 12: The average waste and overhead of memory in scenario 1, run 0.

$$((leftVmCpu-(leftVmCpu\times(leftVmCpu\times leftVmCpu)))\times ((leftVmMem\div(leftVmCpu+leftVmCpu))\times leftVmCpu))\div normalizedVmMemOverhead$$

$$\Downarrow \text{simplify}$$

$$10\times(leftVmCpu-leftVmCpu^3)\times leftVmMem$$

Fig. 13: The simplification of *Rule-15v*.

5.2 Structural Analysis of Evolved Rules

To better understand how the rules utilize the given features to decide the allocation of containers, we analyze an example of the *evo/evo* rules. We first manually simplifying the evolved rules, i.e. the VM selection and creation and PM selection rules. Then, we analyze the rules' behaviors by plotting them on a 3-D surface.

We select an *evo/evo* rule from scenario 9, run 15, called ***Rule-15***, and illustrate its performance. The reason that we select this rule is because the size of the rule is small and easy to explain. *Rule-15* achieves a better training performance than the *sub&Just-Fit/FF* rule, e.g. with 12748.55 vs. 14997.29 in fitness values. It also achieves a better test performance with an average of 3.39E7 Kwh vs. 4.12E7 Kwh. *Rule-15* consists of two sub rules, e.g. the *VM selection and creation* rule called ***Rule-15v*** and the *PM selection* rule called ***Rule-15p***.

We first simplify the *Rule-15v* rule so that we can study its behavior. As previously introduced (see Section3), the *VM selection and creation* rule *Rule-15v* has the functionalities of VM selection and VM type selection when creating a new VM. The switch between these functionalities is controlled by the feature of VM overhead. Specifically, in *Rule-15v* (see Fig. 13), in terms of VM selection, the *vmMemOverhead* becomes 0. Therefore, *Rule-15v* becomes a constant of 1 (as we applied the protected ÷). A constant means *Rule-15v* chooses the first VM which has enough resources. In other words, the VM selection of *Rule-15v* acts like First-Fit. In terms of VM creation, the *vmMemOverhead* is a constant of 0.0125 because it is a normalized number of 200 MB. Then, the rule can be simplified (see Fig. 13). Since the simplified rule has two variables, e.g. *leftVmCpu* and *leftVmMem*, we plot the rule on a 3-D surface.

The 3-D surface plot of *Rule-15v* (see Fig. 14) shows why the rule favors **type 17** and **type 20** VMs when creating a VM. The *leftVmMem* ranges in [0, 0.125] and *leftVmCpu* ranges in [0, 0.1875]. This is because **type 10** that owns the largest memory has 4000 MB memory which is normalized to 0.125 and **type 20** has the largest VM CPU (2475 MHz) which is normalized to 0.1875. We observe that the score

of *Rule-15v* is higher when both *leftVmMem* and *leftVmCpu* is getting larger. The score is more sensitive to *leftVmCpu* than *leftVmMem*. Applying *Rule-15v* on twenty VM types, we found that **type 17** generally obtains the highest score followed by **type 20**. This observation is consistent with the VM frequency shown in the last section (see Fig. 11, scenario 9).

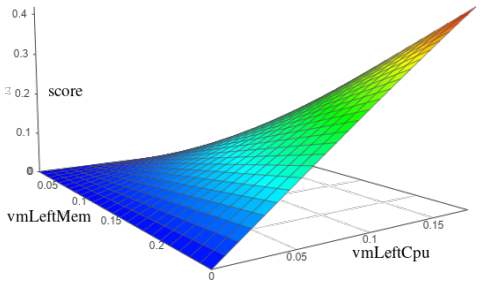
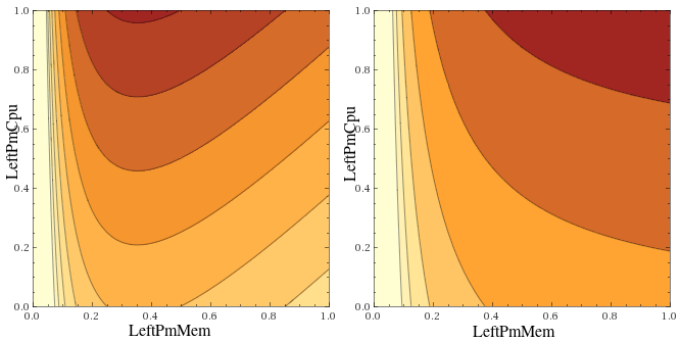


Fig. 14: The 3D and contour plot of the GP tree: $f = 10 \times (leftVmCpu - leftVmCpu^3) \times leftVmMem$, where x-axis is the *leftVmMem* and y-axis is the *leftVmCpu*

The *Rule-15p* allocates the VMs to achieve a good combination of VMs. The *Rule-15p* is $f = leftPmMem - leftPmCpu + vmCpu/leftPmMem$. If we allocate a **type 17** VM with *Rule-15p*, the *vmCpu* is replaced by 0.125. Fig. 15a shows a contour plot of using *Rule-15p* to allocate a **type 17** VM. The lighter area represents the region of a higher score. It shows that the rule prefers two types of PMs. The first type of PMs (appears on the right-bottom corner in the contour map) has high residual memory (more than 12800 MB) and low residual CPU. This type of PMs can allocate more VMs with large memory capacity and low CPU capacity. The other type of preferred PMs (appears on the left-bottom corner) has less or equal residual memory of 4000 MBs regardless of residual CPU. This means that *Rule-15p* tries to allocate the VM into a PM with the capacity just enough for one **type 17** VM. Similarly, if we allocate a **type 20** VM with *Rule-15p*, the *vmCpu* is replaced by 0.1875. Fig. 15b shows the preferred PMs should have low residual memory regardless of residual CPU capacity.

In summary, this section provides an analysis of a rule, generated by the CCGP approach. From the analysis, we can see how the rule leads to a better VM type selection and how to construct a good combination of VMs. Besides *Rule-15*, we have analyzed other rules and reach the same conclusion. Due to the page limit, we only described one rule.



(a) Allocate a **type 17** VM with *Rule-15p* **(b)** Allocate a **type 20** VM with *Rule-15p*

6 Conclusions and Future Work

This paper proposed a novel CCGP approach to solve the on-line *resource allocation in container-based clouds (RAC)* problem. In this work we have the following contributions. First, we propose a formal model for the on-line *RAC* problem with consideration of new features such as VM overheads, VM types, and an affinity constraint. Secondly, this work proposes a novel CCGP algorithm that cooperatively evolves rules for both containers–VMs and VMs–PMs levels of allocation. For this new approach, we design new features for the terminal set. Thirdly, we test the rules generated by CCGP on two real-world container datasets and two groups of VM settings. Experiment results show that the CCGP evolved rules (*evo/evo*) achieve significantly lower accumulated energy consumption than the human-designed rule (*sub&Just-Fit/FF*) and a state-of-the-art approach (*evo/FF*). Lastly, the analysis of rules shows some important insights for cloud providers. Specifically, this work mainly studies the influence of the historical workload patterns and the current status of the data center includes the resources status of VMs, PMs, and containers. This work shows that neglecting these features leads to the problem of VM sprawl. By using the rules that generated from CCGP, VM sprawl can be effectively avoid.

For cloud providers, our proposed CCGP approach provides several advantages for the on-line *RAC* problem. First of all, CCGP automatically designs allocation rules without human intervention. Secondly, the evolved rules have an explainable structure with cloud features interaction. The explainability provides insights for algorithm designers to understand how the interactions of cloud features reflect the information such as the historical workload patterns, VM types, and the status of VMs and PMs. The insights can help algorithm designers to develop more effective algorithms.

Future work can follow two directions. Firstly, we will consider the statistical features such as the mode of the resource requirement in improving the current CCGP algorithm. Second, we will apply clustering techniques as a preprocessing step to categorize containers, then apply ensemble techniques on the training process of CCGP. Specialized rules trained by ensemble technique may further improve the performance of the CCGP.

Acknowledgment

This work is supported in part by the New Zealand Marsden Fund with the contract numbers (VUW1510 and VUW1614), administrated by the Royal Society of New Zealand.

References

[1] "Build data-driven apps with fully managed data services," https://www.heroku.com/home, accessed: 2020-1-22.
[2] "The kubernetes platform for big ideas," https://www.openshift.com/, accessed: 2020-1-22.
[3] A. Kanso and A. Youssef, "Serverless: beyond the cloud," in *Proceedings of the 2nd International Workshop on Serverless Computing*, 2017, pp. 6–10.
[4] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "Containerleaks: Emerging security threats of information leakages in container clouds," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 237–248.

[5] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini, "Securing the infrastructure and the workloads of linux containers," in *IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2015, pp. 559–567.

[6] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, and C. A. De Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2015, pp. 253–260.

[7] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gvisor case study," in {*USENIX*} *Workshop on Hot Topics in Cloud Computing*, 2019.

[8] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *International Middleware Conference*, ser. Middleware. Association for Computing Machinery, 2016.

[9] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic provisioning of virtual machines for container deployment," in *International Conference on Performance Engineering Companion*. ACM, 2017, pp. 5–10.

[10] C. Prakash, P. Prashanth, U. Bellur, and P. Kulkarni, "Deterministic container resource management in derivative clouds," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 79–89.

[11] R. Zhang, A.-m. Zhong, B. Dong, F. Tian, and R. Li, "Container-VM-PM architecture: A novel architecture for docker container placement," in *International Conference on Cloud Computing*. Springer, 2018, pp. 128–140.

[12] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE Transactions on services computing*, vol. 3, no. 4, pp. 266–278, 2010.

[13] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A Framework and Algorithm for Energy Efficient Container Consolidation in Cloud Data Centers," in *International Conference on Data Science and Data Intensive Systems*. IEEE, 2015, pp. 368–375.

[14] B. Tan, H. Ma, and Y. Mei, "Novel genetic algorithm with dual chromosome representation for resource allocation in container-based clouds," in *International Conference on Cloud Computing*. IEEE, 2019, pp. 452–456.

[15] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *Journal of Grid Computing*, vol. 16, no. 1, pp. 113–135, 2018.

[16] X. Guan, X. Wan, B. Y. Choi, S. Song, and J. Zhu, "Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers," *IEEE Communications Letters*, vol. 21, no. 3, pp. 504–507, 2017.

[17] B. Tan, H. Ma, and Y. Mei, "A genetic programming hyper-heuristic approach for online resource allocation in container-based clouds," in *AI: Advances in Artificial Intelligence*. Springer, 2018, pp. 146–152.

[18] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," in *International Conference on Big Data*. IEEE, 2017, pp. 2884–2892.

[19] D. S. Johnson, "Fast algorithms for bin packing," *Elsevier Journal of Computer and System Sciences*, vol. 8, no. 3, pp. 272 – 314, 1974.

[20] E. G. Coffman Jr., J. Csirik, G. Galambos, S. Martello, and D. Vigo, *Bin Packing Approximation Algorithms: Survey and Classification*. Springer, 2013, pp. 455–531.

[21] B. Tan, H. Ma, and Y. Mei, "A hybrid genetic programming hyper-heuristic approach for online two-level resource allocation in container-based clouds," in *Congress on Evolutionary Computation*. IEEE, 2019.

[22] F. Zhang, Y. Mei, and M. Zhang, "A new representation in genetic programming for evolving dispatching rules for dynamic flexible job shop scheduling," in *European Conference on Evolutionary Computation in Combinatorial Optimization*. Springer, 2019, pp. 33–49.

[23] R. Kala, "Multi-robot path planning using co-evolutionary genetic programming," *Expert Systems with Applications*, vol. 39, no. 3, pp. 3817—-3831, 2012.

[24] F. B. de Oliveira, R. Enayatifar, H. J. Sadaei, F. G. Guimarães, and J.-Y. Potvin, "A cooperative coevolutionary algorithm for the multi-depot vehicle routing problem," *Elsevier Expert Systems with Applications*, vol. 43, pp. 117—-130, 2016.

[25] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: a survey of the state of the art," *Springer Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[26] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, "A classification of hyper-heuristic approaches: Revisited," in *Handbook of Metaheuristics*. Springer, 2019, pp. 453–477.

[27] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward, *Exploring Hyper-heuristic Methodologies with Genetic Programming*. Springer, 2009, pp. 177–201.

[28] M. A. Potter and K. A. D. Jong, "Cooperative coevolution: An architecture for evolving coadapted subcomponents," *MIT Press Journals Evolutionary Computation*, vol. 8, no. 1, pp. 1—-29, 2000.

[29] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 2, pp. 193–208, 2014.

[30] J. Park, Y. Mei, S. Nguyen, G. Chen, and M. Zhang, "Evolutionary multitask optimisation for dynamic job shop scheduling using niched genetic programming," in *AI: Advances in Artificial Intelligence*. Springer, 2018, pp. 739—-751.

[31] ——, "Investigating the generality of genetic programming based hyper-heuristic approach to dynamic job shop scheduling with machine breakdown," in *Artificial Life and Computational Intelligence*. Springer, 2017, pp. 301—313.

[32] E. K. Burke, M. R. Hyde, and G. Kendall, "Evolving bin packing heuristics with genetic programming," in *Parallel Problem Solving from Nature*. Springer, 2006, pp. 860—869.

[33] E. K. Burke, M. Hyde, G. Kendall, and J. Woodward, "A genetic programming hyper-heuristic approach for evolving 2-d strip packing heuristics," *IEEE Transactions on Evolutionary Computation*, vol. 14, no. 6, pp. 942–958, 2010.

[34] S. Allen, E. K. Burke, M. Hyde, and G. Kendall, "Evolving reusable 3d packing heuristics with genetic programming," in *Conference on Genetic and Evolutionary Computation*. ACM, 2009, pp. 931–938.

[35] D. Yska, Y. Mei, and M. Zhang, "Genetic programming hyper-heuristic with cooperative coevolution for dynamic flexible job shop scheduling," in *European Conference on Genetic Programming*. Springer, 2018, p. 306–321.

[36] F. Zhang, Y. Mei, and M. Zhang, "Surrogate-assisted genetic programming for dynamic flexible job shop scheduling," in *AI 2018: Advances in Artificial Intelligence*. Springer, 2018, pp. 766–772.

[37] Y. Zhou, J. Yang, and L. Zheng, "Hyper-heuristic coevolution of machine assignment and job sequencing rules for multi-objective dynamic flexible job shop scheduling," *IEEE Access*, vol. 7, pp. 68–88, 2019.

[38] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali, "Approximation and online algorithms for multidimensional bin packing: A survey," *Computer Science Review*, vol. 24, pp. 63–79, 2017.

[39] X. Zhang, T. Wu, M. Chen, T. Wei, J. Zhou, S. Hu, and R. Buyya, "Energy-aware virtual machine allocation for cloud with resource reservation," *Journal of Systems and Software*, vol. 147, pp. 147–161, 2019.

[40] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck, "Resource management in a containerized cloud: Status and challenges," *Journal of Network and Systems Management*, pp. 1–50, 2019.

[41] A. Wolke, M. Bichler, and T. Setzer, "Planning vs. dynamic control: Resource allocation in corporate clouds," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 322–335, 2016.

[42] Z. A. Mann, "Resource optimization across the cloud stack," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 169–182, 2018.

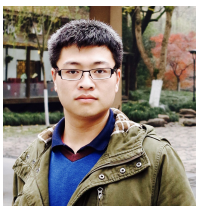
[43] Z. Á. Mann, "Interplay of virtual machine selection and virtual machine placement," in *European Conference on Service-Oriented and Cloud Computing*, vol. 9846. Springer, 2016, pp. 137–151.

[44] C. Lin, J. Chen, P. Liu, and J. Wu, "Energy-efficient core allocation and deployment for container-based virtualization," in *International Conference on Parallel and Distributed Systems*. IEEE, 2018, pp. 93–101.

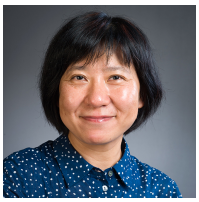
[45] T. Shi, H. Ma, and G. Chen, "Energy-aware container consolidation based on pso in cloud data centers," in *Congress on Evolutionary Computation*. IEEE, 2018, pp. 1–8.

[46] C. Fan, Y. Wang, and Z. Wen, "Research on Improved 2D-BPSO-Based VM-Container Hybrid Hierarchical Cloud Resource Scheduling Mechanism," in *International Conference on Computer and Information Technology*. IEEE, 2016, pp. 754–759.

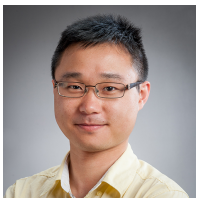
[47] D. Zhang, B. Yan, Z. Feng, C. Zhang, and Y. Wang, "Container oriented job scheduling using linear programming model," in *International Conference on Information Management*. Association for Information Systems, 2017, pp. 174–180.
[48] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications," *Springer The Journal of Supercomputing*, vol. 74, no. 7, pp. 2956—2983, 2018.
[49] M. De Cauwer, D. Mehta, and B. O'Sullivan, "The temporal bin packing problem: An application to workload management in data centres," in *International Conference on Tools with Artificial Intelligence*. IEEE, 2016, pp. 157–164.
[50] V. Sarathy, P. Narayan, and R. Mikkilineni, "Next generation cloud computing architecture: Enabling real-time dynamism for shared distributed physical infrastructure," in *International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*. IEEE, 2010, pp. 48–53.
[51] J. Chen, K. Chiew, D. Ye, L. Zhu, and W. Chen, "Aaga: Affinity-aware grouping for allocation of virtual machines," in *International Conference on Advanced Information Networking and Applications*. IEEE, 2013, pp. 235–242.
[52] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Elsevier Future Generation Computer Systems*, vol. 28, no. 5, pp. 755 – 768, 2012.
[53] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, "Sandpiper - Black-box and gray-box resource management for virtual machines." *Elsevier Computer Networks*, vol. 53, no. 17, pp. 2923–2938, 2009.
[54] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Springer Statistics and Computing*, vol. 4, no. 2, pp. 87–112, 1994.
[55] S. Nguyen, M. Zhang, and K. C. Tan, "Surrogate-assisted genetic programming with simplified models for automated design of dispatching rules," *IEEE Transactions on cybernetics*, vol. 47, no. 9, pp. 2951–2965, 2017.
[56] T. Hildebrandt, J. Heger, and B. Scholz-Reiter, "Towards improved dispatching rules for complex shop floor scenarios: A genetic programming approach," in *Conference on Genetic and Evolutionary Computation*. ACM, 2010, pp. 257–264.
[57] M. Mishra and A. Sahoo, "On theory of VM placement: Anomalies in existing methodologies and their mitigation using a novel vector based approach," in *International Conference on Cloud Computing*. IEEE, 2011, pp. 275–282.
[58] S. Shen, V. van Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *International Symposium on Cluster, Cloud and Grid Computing*. IEEE/ACM, 2015, pp. 465–474.
[59] "Revenue share of global server operating system market in 2015, by operating system," https://www.statista.com/statistics/639574/worldwide-server-operating-system-market-share/, accessed: 2018-10-05.
[60] Y. Mei, M. Zhang, and S. Nyugen, "Feature selection in evolving job shop dispatching rules with genetic programming," in *Conference on Genetic and Evolutionary Computation*. ACM, 2016, pp. 365–372.
[61] L. Sean, "A java-based evolutionary computation research system," https://cs.gmu.edu/~eclab/projects/ecj/, accessed: 2018-10-05.



Boxiong Tan is a Ph.D. candidate at the School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand. He received B.E degree from Guangzhou University and MCS degree from Victoria University of Wellington in 2011 and 2016 respectively.



Hui Ma received her B.E. degree from Tongji University and her B.S. (Hons.), M.S. and Ph.D. degrees from Massay University. She is currently a Senior Lecturer in Software Engineering at Victoria University of Wellington. Her research interests include service computing, conceptual modeling, database systems, resource allocation in clouds, and evolutionary computation in combinatorial optimization.



Yi Mei (M'09) is a lecturer at the School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand. He received his BSc and PhD degrees from University of Science and Technology of China in 2005 and 2010, respectively. His research interests include evolutionary computation in scheduling, routing and combinatorial optimization, as well as evolutionary machine learning, hyper-heuristics, genetic programming, feature selection and dimensional reduction.

Mengjie Zhang (M'04-SM'10) received the B.E. and M.E. degrees from Artificial Intelligence Research Center, Agricultural University of Hebei, Hebei, China, and the Ph.D. degree in computer science from RMIT University, Melbourne, Australia, in 1989, 1992, and 2000, respectively.

Mengjie is a Fellow of Royal Society of New Zealand, and has been serving as an Associated Editor for over ten international journals (including IEEE Transactions on Evolutionary Computation, IEEE Transactions on Cybernetics, Evolutionary Computation Journal, and IEEE Transactions on Emergent Topics in CI) and as a Reviewer of over 30 international journals. He has been serving as a Steering Committee Member and a Program Committee Member for over 100 international conferences. He has supervised over 50 postgraduate research students. He is the Chair of the IEEE CIS Intelligent Systems and Applications Technical Committee, an Immediate Past Chair of the IEEE CIS Evolutionary Computation Technical Committee, a Vice-Chair of the IEEE CIS Task Force on Evolutionary Computer Vision and Image Processing, a Vice-Chair of the IEEE CIS Task Force on Evolutionary Computation for Feature Selection and Construction, a member of IEEE CIS Task Force of Hyper-heuristics, and the Founding Chair for IEEE Computational Intelligence Chapter in New Zealand.