A Genetic Programming Hyper-heuristic Approach for online Resource Allocation in Container-based Clouds

Boxiong Tan, Hui Ma, Yi Mei

Victoria University of Wellington, Wellington, New Zealand,
{Boxiong.tan, Hui.ma, Yi.mei}@ecs.vuw.ac.nz

Abstract. The key reason for the popularity of application containers in clouds is its ability to deploy and run applications without launching an entire virtual machine (VM) for each application. Not only does the container support a fast deployment of applications but also its potential to reduce the energy consumption of data centers. However, with the goal of energy reduction, it is more much difficult to optimize the allocation of containers than traditional VM-based clouds because of the finer granularity of resources. Specifically, for online resource allocation, existing methods have shown poor performance in dealing with different scenarios (balanced and unbalanced resources). In this paper, we first compare three human-design heuristics and show they cannot handle balanced and unbalanced resources scenarios well. Second, we propose a learning-based algorithm: genetic programming hyper-heuristic (GPHH) to automatically generate a suitable heuristic for allocating containers in an online fashion adapt to the environment. The experimental results show that the proposed GPHH managed to evolve much better heuristics than the human-designed ones in terms of energy consumption in a range of cloud scenarios.

Keywords: container-based clouds, container allocation, energy consumption, genetic programming, hyper-heuristic

1 Introduction

A container-based cloud [1] is a promising new technology for both software and cloud computing industries. Compared to traditional virtual machine (VM)-based clouds, container-based clouds use a lightweight virtualization technology — containers as the basic resource management unit [1]. For application providers (e.g Google, Apple), the major benefit of container-based clouds is the support of agile software development including developing, testing, and fast deploying applications. Besides software providers, containers are also beneficial for cloud providers because they can potentially reduce the energy of data centers [2]. Energy reduction is achieved by deploying more applications in fewer physical machines (PMs).

Although container-based clouds have better energy efficiency, the side-effect is that with the additional level of containers, the complexity for allocating

both containers and VMs is much higher than solely managing VMs. In other words, in a container-based cloud, a typical PM may host multiple VMs with different operating systems. Each VM hosts multiple containers. This box-inside-box structure forces us to break down the container allocation process into two levels: containers to VMs and VMs to PMs. For a given set of containers, the decision variables include the allocation of containers, VMs, and the type of the used VMs.

Because of the high complexity, this work considers a simplified structure and focuses on the challenge of the container allocation problem. Many cloud providers (e.g. Amazon) skip VM level and deploy containers directly to PMs. Moreover, we consider an online container allocation in which the request come in real time, and the information of each request (e.g. CPU and memory demand) is unknown until the request arrives. Similar to job shop scheduling problems [3], which can be categorized in offline and online scenarios, online scenario allocates a container instantly when a request comes. In contrast, offline scenario allocates a batch of queuing containers.

Existing approaches [4] treat the container allocation problem as an online vector bin packing problem [5]. They [4] apply AnyFit-based algorithms with human-designed greedy rules such as *sub* and *sub* (detailed discussed in Section 2.2). We argue that the goal for resource allocation in clouds is to minimize the accumulated energy consumption instead of the cutting-point energy. It is critical to consider the order of creating new PMs when allocating containers [6]. Therefore, the container allocation problem can be treated as a scheduling task.

Existing human-designed rules, therefore, may not be suitable for the scheduling task. First, they are designed based on the simple assumptions such as balanced resources leads to fewer bins (see Section 2.2). However, these assumptions are only true for special scenarios (e.g balanced resources requirement). Since we cannot make any assumption of resource requirement, it is easy to infer that these rules cannot handle all scenarios very well. Second, these rules have rather navie structure (see Section 2.2). Their structures cannot capture the nature of the online scheduling problem. It is difficult for a human expert to design an effective scheduling rule because of the large scale of resource interactions.

To address the drawbacks of human-design rules and the high design difficulty, we propose a learning algorithm: Genetic programming-based hyper-heuristic (GPHH) to automatically design scheduling rules using the information of a data center. Hyper-heuristic is a search method which explores the heuristic space instead of the solution space [7]. GPHH is developed based on an evolutionary computation search method called genetic programming. GPHH has been successfully applied for many combinatorial optimization problems such as dynamic job shop scheduling (JSS) problem [3], storage location assignment problem [8], and vehicle routing problem. Noticeably, GPHH has been applied for solving single-dimensional bin packing problems [9]. Our container allocation problem can be simplified to a multi-dimensional bin packing problem. To apply GPHH in the container allocation problem, we need to develop new terminal sets and fitness function.

In this paper, our contributions are:

- We compare the widely used human-designed greedy rules: *sub*, *sub*, and random in container allocation. This comparison provides an important insights: the human-designed rules do not perform well in all scenarios. Therefore, we need to develop a learning algorithm to automatically generate rules to adapt all scenarios.
- We develop a GPHH approach for generating rules for online container allocation. We show that using the rules generated by GPHH, the accumulated energy consumption is smaller than using human-design rules in all scenarios.

2 Background

This section first formally defines the container allocation problem. Then, we introduce the human-designed rules for online container allocation.

2.1 Problem Description

The container allocation problem can be described as, for a given set of t containers, each of which arrives at a time i, $0 <= i <= t$, the aim of container allocation is to allocate containers to PMs so that during the period of time of allocation, the accumulated energy consumption of PMs are minimized. In the following, we present a formal model of the problem including optimization objective, decision variables, and constraints.

Assuming that we have t containers to be allocated into p Physical Machines (PMs). Each container $i \in \{1, \cdots, t\}$ has a CPU demand A_i and a memory occupation M_i. Each PM $j \in \{1, \cdots, p\}$ has a CPU capacity PA_j and a memory capacity PM_j. A PM can host multiple containers. We consider homogeneous PMs where all PMs have the same size of CPU capacity and memory.

The overall objective is to minimize the accumulated energy consumption of the data center when a data center finishes all the allocations. We describe this objective in a top-down manner: first, we illustrate the accumulated energy equation as in Eq. 1.

$$E = \sum_{i=1}^{t} \sum_{j=1}^{p} P_j \cdot [u_{cpu}(j) > 0] \tag{1}$$

Equation 1 sums up the energy consumption of all the PMs during the allocation of t containers. Energy consumption P_j is the energy consumption of a PM j. $[u_{cpu}(j) > 0]$ returns 1 if $u_{cpu}(j) > 0$ (i.e. PM j is active), and 0 otherwise.

P_j is determined by a widely used energy model [10] (see Equation. 2).

$$P_j = \alpha \cdot P^{max} + (1 - \alpha) \cdot P^{max} \cdot u_{cpu}(j) \tag{2}$$

Equation 2 calculates the energy consumption P_j of a PM j. This equation assumes the energy consumption consists of two parts where the first part $\alpha \cdot$

P^{max} of a PM represents the energy consumption when the PM is idle. The second part $(1-\alpha) \cdot P^{max} \cdot u_{cpu}(j)$ represents the fluctuate energy consumption related to the current state of CPU utilization $u_{cpu}(j)$ defined in Equation. 3.

The CPU and memory utilization of PM j are denoted as $u_{cpu}(j)$ and $u_{mem}(j)$. They can be calculated as the Eq. 3 and Eq. 4.

$$u_{cpu}(j) = \sum_{i=1}^{t} (x_j^i \cdot A_i) \tag{3}$$

$$u_{mem}(j) = \sum_{i=1}^{t} (x_j^i \cdot M_i) \tag{4}$$

Where x_j^i is a binary value (e.g. 0 and 1) denoting whether a container i is allocated on a PM j.

Constraints To satisfy the performance requirement which is often defined by Service Level Agreements (SLAs), a container can be allocated on a PM if and only if the PM j has enough resources required by the container (Eq. 5 and Eq. 6).

$$\sum_{i=1}^{t} x_j^i \cdot A_i \leq PA_j \quad \forall j \in \{1, \cdots, p\} \tag{5}$$

$$\sum_{i=1}^{t} x_j^i \cdot M_i \leq PM_j \quad \forall j \in \{1, \cdots, p\} \tag{6}$$

Equation 7 indicates that each container can only be deployed once.

$$\sum_{i=1}^{t} x_j^i = 1 \quad \forall j \in \{1, \cdots, p\} \tag{7}$$

In the online version of the problem, we need to allocate a container to a PM before the next container comes.

2.2 Human-designed Rules for online Container Allocation

AnyFit algorithms [11] are greedy-based algorithms. They allocate one item each time and do not create new bins until the existing bins cannot hold the item. AnyFit algorithms use a human-designed rule for evaluating an allocation. For example, in one-dimensional bin packing, the objective is to fill bins. We define the rule as a bin with fewer residual resources earns a higher score (e.g. in best-fit). In multi-dimensional bin packing, however, it is not obvious how to combine multiple resources into a single number so that all dimensional resources can be effectively used. Mann's [4] applied six rules (such as *sub*, *sum*, and *product*) for container allocation. However, the different effects of these rules have not been

shown. Therefore, it motivates us to explore the effectiveness of the most used rules: *sum* and *sub* in solving the problem of container allocation problem.

Sum is the most commonly used rule in multi-dimensional bin packing. It can be represented as $resourceA+resourceB$ in the two-dimensional case. Resources A and B are the residual resources of a chosen bin after the item has been allocated. The two resources are normalized into between 0 and 1. The smaller the function result, the better the candidate bin. This heuristic tries to minimize the residual resources in all dimensions. It is based on a simple assumption that less residual resource results in fewer number of used bins.

However, *sum* rule may lead to an imbalanced resource allocation which leads to more bins. For example, given four two-dimensional items A(1, 6), B(2, 4), C(7, 4), D(4, 2) and the size of bin is (10, 10). It is obvious that these items can be put into two bins as (A, C) and (B, D). However, if we group (A, B), the extremely unbalanced resources of (3, 10) will lead to an extra bin: (A, B), (C), (D).

Sub, therefore, is designed to maintain the balance in a bin. It can be represented as $|resourceA - resourceB|$. Similar to *sub*, we prefer a smaller function value. *sub* rule tries to minimize the difference between the two resources. With the assumption of balanced resource allocation can lead to fewer bins, much research [12] has applied *sub* rule.

In summary, first, as we consider the container allocation problem as a scheduling problem instead of a bin packing problem, the performance of these simple rules has not been well studied. Second, because of their simplicity, we believe they cannot fully capture the complex behavior of diverse resource requirements and temporal effect. Therefore, these two reasons motivate us to investigate a learning method: GPHH using the information from a data center to generate rules.

Next section presents our proposed GPHH method for automatically generating rules for container allocation problem.

3 Genetic Programming Hyper-heuristic (GPHH) for Generating Rules for online Container Allocation

This section describes the framework that we used to investigate the performance of GPHH for online container allocation problem (see Figure 1). First of all, we describe the training and testing process. Secondly, we describe the GPHH-related features such as fitness function, representation, and terminals.

3.1 Simulation, Training, and Testing Processes

Both training and testing processes rely on the simulation of container allocation for evaluating the quality of rules (see Figure 1). The simulation includes two parts: data center initialization and container allocation.

Data center initialization is a critical step in container allocation experiments. The major reason is that the effects of different allocation rules only show differences when there exists multiple allocation candidates (PMs). As mentioned

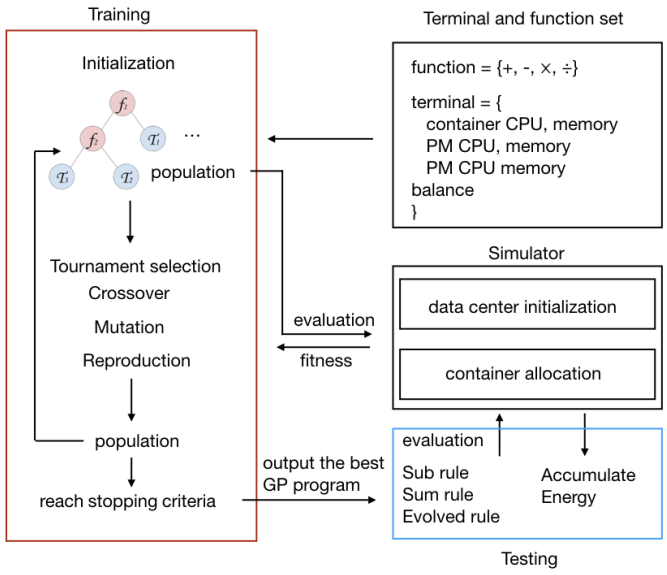


Fig. 1: The overview of GPHH training and testing process

in Section 2.2, the AnyFit-based algorithms only create a new bin unless all existing bins have been filled. Therefore, an empty data center will create-and-fill PMs one by one. If all container allocation rules have only one candidate PM, they perform similarly.

Container allocation process allocates a number of containers into the initialized data center. In order to simulate the online allocation, each request must be executed immediately without the knowledge of future containers. For each container, the simulator first filters out the PMs which cannot satisfy the demand of container. Then, the simulator chooses the best PM according to an evolved rule wrapped by the BestFit framework (see Algorithm 1). The evolved rule works similarly as human-design rules – assign scores to candidate PMs. The features of the data center (described in next section) will be used as the terminal set in GPHH. During the allocation, the simulator evaluates the accumulated energy consumption according to Eq 1.

GPHH training process intends to construct a rule using the features of the data center. During the training process (details in [7]), GPHH initializes a predefined size of the population. Each individual in the population is an allocation rule represented by a tree structure. Then, individual rules will be used in the simulation on the training set (see Section 4.2). The evaluation assigns a fitness value to a rule. The fitness value is calculated by a fitness function (see next section). After evaluation, the population will go through one of the genetic operators: crossover, mutation, and reproduction according

Algorithm 1: BestFit framework for the evolved rules

```
Input  : container, A list of available PMs, features of the data center
Output: The best PM
BestPM = nil;
bestFitness = nil;
while PM_i in PMs do
    fitness =evolvedRule(container, PM_i, features);
    if fitness > bestFitness then
        bestFitness = fitness;
        BestPM =PM_i;
    end
    i = i + 1;
end
return BestPM;
```

to a predefined probability. The genetic operators modify the tree structure as a mean to explore the search space. A new population of rules are generated and then evaluated. This loop ends until a stopping criterion is met. We define the stopping criteria as the maximum number of generations. This number is usually based on empirical observation of convergence of the training process. That is, in the training process, GPHH converges when the fitness value stops improving.

The testing process runs rules on the test sets. GPHH is a stochastic algorithm which employs a degree of randomness as part of its logic (in genetic operators). Therefore, in order to rule out the influence of randomness, we test the GPHH evolved allocation rules with the following procedure. For every test scenario, we generate 30 evolved rules with different random seeds. Each test of the evolved rule is called a single run. After obtaining the results of 30 runs, the accumulated energy of each test instance is first normalized with the benchmark *sub* rule (see Eq. 8). Then, for each instance, we calculate the average normalized accumulated energy from 30 runs. Lastly, we applied the paired Wilcoxon test to calculate the statistic significance between the evolved rules and the benchmark rules.

$$normalized\ E_{evolve} = \frac{E_{evolve}}{E_{sub}} \tag{8}$$

3.2 Function, Terminal Sets, and Fitness function

Function and Terminal sets are the building block of evolved rules. Function set contains the arithmetic operators which combines the raw features into high-level features. In this work, our function set includes $\{+, -, \times\}$ and protected $\div$ which returns 1 when divided by 0.

Terminal set includes the features of the data center. Ideally, these features should have a heavy impact on the container allocation decision. However, we do not know which feature may affect the allocation decision marking. Motivated

by Burke's research [9], we design five terminals. The first two includes the CPU and memory requirement of a container. The following two terminals are the residual CPU and memory from a PM. They are calculated as the current PM's resources subtract the resource requirement of the container. Furthermore, we define a balance factor for PM. The factor can be represented as residual memory / residual CPU, if CPU > memory. Otherwise, residual CPU / residual memory. We did not use the subtraction between resources because the division keeps the consistency in the BestFit framework (see Alg 1) with the objective value – the larger the better.

We calculate the fitness function (Eq. 9) with two steps. The first step, for each test instance (including a number of containers to be allocated), we calculate the average accumulated energy consumption. It divides accumulated energy by the number of containers. The second step calculates the average accumulated energy consumption among all test instances. It sums up the average accumulated energy and divides by the number of test instances. The fitness value represents the average increase in energy consumption after allocating a container. Therefore, it is free from the bias of the randomize initial data center.

$$fitness\ function = \frac{\sum_{k=1}^{test\ instances} \frac{E}{t}}{test\ instances} \tag{9}$$

4 Experiments

This section describes the experiment purpose, design, and results. Experiment purpose states the hypothesis that we intend to verify and the research questions that we intend to answer. Experiment design illustrates the settings for test instances, data center initialization, and the GPHH. Lastly, we show and analyze the results in the experiment result section.

4.1 Experiment Purpose

The purpose of testing rules on three scenarios (listed in Table 1) is that, firstly, we intend to verify the hypothesis that *sub* rule can lead to less energy consumption than the *sum* and random rules by balancing the resources (see Section 2.2). This purpose can be achieved by testing rules on the three scenarios.

Secondly, based on the previous result, if *sub* does save energy, does it work consistently well for all scenarios? Here, we roughly categorize the resource requirement and PMs as balanced and unbalanced. Balanced resources mean the value of two resources is equal.

Thirdly, if the hypothesis is not true, can we use GPHH in different scenarios? What features can be used to construct a better rule? This can be achieved by observing and comparing results. We can infer which feature has an effect on the performance.

In summary, we design three scenarios for two objectives: testing whether existing rules can work well in all scenarios and discovering useful features for rules.

Table 1

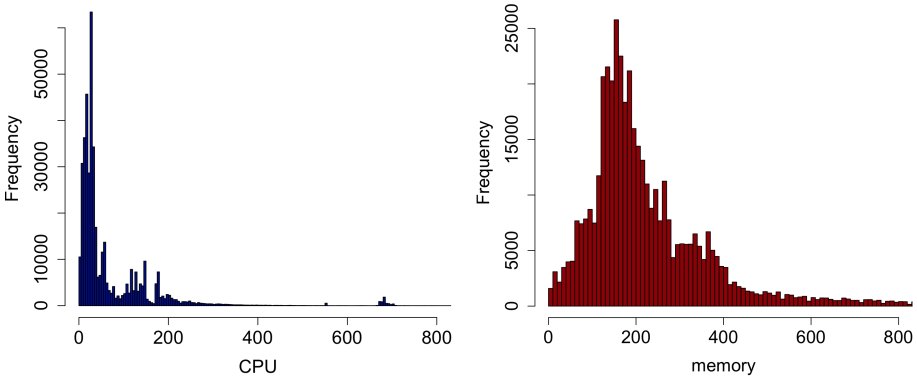
scenarios	dataset	PM size (CPU[Mhz], Memory[MB])
unbalanced containers and PMs	real-world dataset	(3300, 4000)
balanced containers and unbalanced PMs	synthetic dataset	(3300, 3300)
balanced containers and PMs	synthetic dataset	(3300, 3300)

4.2 Experiment Design

Each of the three scenarios includes 100 of test instances. They are splitted equally into training and testing set. Each test instance consists of 200 containers to be allocated.

The scenario with unbalanced containers and PMs is the most common scenario in the real world because of the diverse applications. We use a real-world dataset (AuverGrid trace [13]) to generate test instances of unbalanced container scenario. To analyze resource workload distribution, we present two histograms (Figure 2) of the distribution of CPU and memory usage of the dataset. We can observe that the most frequent memory usage is on average three times as much as the CPU usage. For container generation, we randomly choose pairs of CPU and memory from the dataset with both resource requirement less than or equal to the maximum capacity.

Fig. 2: Resource usage frequency in real world dataset



(a) CPU usage frequency

(b) Memory usage frequency

Balanced containers are generated from an exponential distribution with the rate $\lambda = 0.004$ in both CPU and memory. The reason for using this rate is because we want to have similar size of containers between the synthetic dataset and real-world dataset. From the empirical study, we found the average number of containers allocated to a PM is 15 containers in real-world dataset. With the $\lambda = 0.004$, it gives us a similar average number of containers allocated to a PM.

In order to compare the performance between *sub* and sum, we add a random rule. The random rule will choose a random available PM instead of the best one according to a fitness function. We intend to compare *sub* and sum with the random rule as a baseline. Hence, we can identify which rule performs badly in which scenario.

For initialization of a data center, we randomly generate 4 to 8 running PMs. Each VM will host at least one container. In addition, we use the corresponding dataset as the test cases for generating the initial containers in PMs.

For GPHH, we use the population size of 1024. The number of generation is 100. For crossover, mutation, and reproduction, we use 0.8, 0.1, and 0.1 respectively. We use tournament selection and the size of the tournament is 7.

4.3 Experiment Results

In all three scenarios, the evolved rules show significant advantages than the other three rules (Table 2 to 4).

Table 2 shows the Win-Draw-Loss of the unbalanced containers and PMs dataset among four algorithms. *sub* rule is significantly worse than all the other rules including the random selection. Evolved rules dominate *sub* and random, but is better than *sum* with a small margin.

Table 2: Win-Draw-Loss table for real world scenarios.

	evo	sub	sum	random
evo	Nah	**49-0-1**	**30-0-20**	**45-0-5**
sub	1-0-49	Nah	2-1-47	5-2-43
sum	20-0-30	47-1-2	Nah	40-4-6
random	5-0-45	43-2-5	6-4-40	Nah

To investigate the reason for the poor performance of *sub* rule, we follow the allocation procedure. We observe that *sub* rule always try to keep resource balance among all PMs. However, because the frequency of extremely unbalancing containers is high in the real-world dataset when an unbalanced container comes, it is very likely that the balanced PMs have no room for the container. In contrast, the *sum* tries to fill the PM one by one starting with the most full PM. Therefore, it is likely that there are available PMs for the unbalanced container. In summary, the balance of resource is not suitable for an unbalanced resource scenario.

In balanced containers and unbalanced PMs scenario(Table 3), there is no statistic difference between *sub*, *sum* and random rules. In balanced containers and PMs (Table 4), evolved rules dominate other rules. Both *sub* and *sum* are significantly better than the random rule.

For balanced container scenarios, the performance of *sub* rule has a great improvement. It is because the frequency of unbalanced containers is low. Therefore, *sub* rule can balance the load in all PMs. The likelihood of creating new PM is much lower than the unbalanced scenario.

To explain the goodness of evolved rules, Figure 3 shows the energy consumption of the data center while allocating 200 containers with four rules. The

Table 3: unbalanced PMs

	evo	sub	sum	random
evo	Nah	**44-0-6**	**41-0-9**	**43-0-7**
sub	6-0-44	Nah	20-1-29	30-0-20
sum	9-0-41	29-1-20	Nah	33-1-16
random	7-0-43	20-0-30	16-1-33	Nah

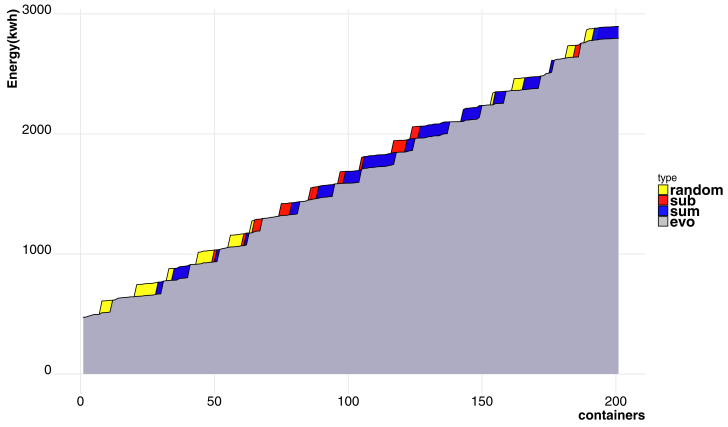
Table 4: balanced containers and PMs

	evo	sub	sum	random
evo	Nah	**43-0-7**	**38-0-12**	**46-0-4**
sub	7-0-43	Nah	23-0-27	32-0-18
sum	12-0-38	27-0-23	Nah	35-0-15
random	4-0-46	18-0-32	15-0-35	Nah

initial energy consumption are the same because of the same initialized data center. With the allocation processing, *random* rule (yellow) creates a new PM which incurs the sudden increase of energy while other rules can still allocate containers to existing PMs.

Similarly, during the allocation, *sub*, *sum*, and *random* rules create new PMs earlier than the evolved rule. Although, in most cases, all four rules create the same number of PMs (not in this case), evolved rules always allocate more containers to the existing PMs. That is, evolved rules put off the creation of new PMs. This is the main reason that the evolved rule has the least accumulated energy consumption.

Fig. 3: The energy consumption of allocating 200 containers with four algorithms (from run 15, test case 5)



5 Conclusion

This paper has two contributions. First, we show that exising rules for container allocation do not perform well in dealing with real-world resource requirement and PM. Second, in order to automatically generate rules using the information of data centers, we develop a new GPHH approach for container allocation.

Experiments show that the evolved rules perform significantly better the human-designed rules in all scenarios. The advantage of evolved rules shows that GPHH is a promising technique for automatically generating rules for various

scenarios in data centers. Furthermore, the experiments also prove that the basic features that we designed for constructing a container allocation rules are useful.

In future work, we will investigate the container allocation in a general architecture where multiple VMs are allocated in PMs. The use of VM has two reasons, firstly, different sizes of VMs can further improve the energy consumption of the data center. Secondly, with multiple VMs, a PM can host containers with different operating system requirements. The major difficulty is the decision of the size of a newly created VM.

References

1. D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
2. Z. . Mann, "Resource optimization across the cloud stack," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 169–182, 2018.
3. J. Branke, S. Nguyen, C. W. Pickardt, and M. Zhang, "Automated design of production scheduling heuristics: A review," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 110–124, 2016.
4. Z. Á. Mann, "Interplay of virtual machine selection and virtual machine placement," in *Lecture Notes in Computer Science*. Cham: Springer, 2016, vol. 9846 LNCS, pp. 137–151.
5. W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive resource provisioning for the cloud using online bin packing," *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2647–2660, 2014.
6. M. D. Cauwer, D. Mehta, and B. O'Sullivan, "The temporal bin packing problem: An application to workload management in data centres," in *28th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2016, pp. 157–164.
7. 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," *Journal of the Operational Research Society, Springer*, vol. 64, no. 12, pp. 1695–1724, 2013.
8. J. Xie, Y. Mei, A. T. Ernst, X. Li, and A. Song, "A genetic programming-based hyper-heuristic approach for storage location assignment problem," in *IEEE Congress on Evolutionary Computation (CEC)*, Beijing, China, 2014, pp. 3000–3007.
9. E. K. Burke, M. R. Hyde, and G. Kendall, "Evolving bin packing heuristics with genetic programming," in *Parallel Problem Solving from Nature*. Berlin, Heidelberg: Springer, 2006, pp. 860–869.
10. X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," *ACM SIGARCH Computer Architecture News*, vol. 35, no. June, p. 13, 2007.
11. D. S. Johnson, "Fast algorithms for bin packing," *Journal of Computer and System Sciences, Elsevier*, vol. 8, no. 3, pp. 272 – 314, 1974.
12. Y. Gao, H. Guan, Z. Qi, Y. Hou, and L. Liu, "A multi-objective ant colony system algorithm for virtual machine placement in cloud computing." *Journal of Computer and System Science, Elsevier*, vol. 79, no. 8, pp. 1230–1242, 2013.
13. S. Shen, V. v. Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 465–474.