An Evolutionary Computation Approach to Resource Allocation in Container-based Clouds

by

Boxiong Tan

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2020

Abstract

A container-based cloud is a new trend in cloud computing that introduces more granular management of cloud resources. Compared with VM-based clouds, container-based clouds can further improve energy efficiency with a finer granularity resource allocation in data centers. The current allocation approaches for VM-based clouds cannot be used in container-based clouds. The first reason is that existing research lacks appropriate models that can represent the interaction of allocation features. Many critical features, such as VM overhead, are also not considered in the current models. The second reason is that current allocation approaches do not perform well to the three frequently encountered allocation scenarios, off-line allocation, on-line allocation, and multi-objective allocation. Current approaches for these scenarios are mostly based on greedy heuristics that can be easily stuck at local optimum, or meta-heuristics that consider a simplified one-level allocation problem. Evolutionary Computation (EC) is particularly good at solving combinatorial optimization problems for both off-line and on-line scenarios. The overall goal of this thesis is to propose an EC approach to the three allocation scenarios in order to improve the performance of container-based clouds. Specifically, we aim to optimize energy consumption in all the scenarios. An additional objective, availability of application, is considered for the multi-objective scenario.

First, this thesis investigates two promising representations, vector-based and group-based. We proposes two novel vector-based (e.g., *Single-chromosome Genetic Algorithm (SGA)* and *Dual-chromosome GA (DGA)*) and a group-based GA approaches for the off-line allocation scenario. Corresponding genetic operators and decoding processes are also developed

and evaluated. Two contributions have been made. Firstly, a novel off-line model has been proposed based on current models with additional features. It can be used to evaluate allocation algorithms. Secondly, two types of problem representation, vector-based and group-based, are investigated and three novel approaches are proposed. The three approaches are compared with state-of-the-art approaches. The results show that all solutions produced by these approaches are better than the state-of-the-art approaches and group-based GA is the best approach.

Second, this thesis proposes a novel genetic programming hyper-heuristic (GPHH) and a cooperative coevolution (CCGP)-based approach for the on-line allocation scenario. These hyper-heuristic methods can automatically generate allocation rules. For GPHH-based approach, we develop a novel training procedure to generate reservation-based rules for allocating containers to VM instances. For the CCGP-based approach, we introduce a new terminal set and develop a training procedure to generate allocation rules for two-level allocations. We analyze both human-designed rules and generated rules to provide insights for algorithm designers. Two contributions have been proposed for the on-line problem. First, the on-line model for the on-line allocation scenario is developed. Second, a novel terminal set and training procedures are developed. The automatically generated heuristics perform significantly better than the manually designed heuristics.

Third, this thesis proposes a multi-objective approach that generates a set of trade-off solutions for the cloud providers to choose from. Our novel approach, namely *Nondominated Sorting-Group GA (NS-GGA)*, combines the group-based representation and the NSGA-II framework. The experimental results are compared with existing approaches. The results show that our proposed *NS-GGA* approach outperforms all other approaches. We propose two novelties. The first novelty is the multi-objective model including objectives of energy consumption and availability. The second novelty is the *NS-GGA* approach that combines the group-based represen-

tation with NSGA-II. The allocation solutions found by *NS-GGA* dominate the solutions found by other existing approaches.

Acknowledgments

I would like to thank my supervisors, A/Prof Hui Ma, Dr Yi Mei, and Prof Mengjie Zhang, for their patient guidance and support during the course of my Ph.D study. A/Prof Hui Ma is always good to talk to. Dr Yi Mei challenged me to dig deeper into the problems. Prof Mengjie Zhang gave his feedback on my articles and thesis fast and helpful. I am also grateful for the financial support from the Marsden Fund (VUW1510 and VUW 1614) over the past three years.

I am especially grateful to Qu Ying for being a teacher, a good listener, and my closest friend. Thank you for spending hours and hours teaching me writing. Thank you for listening to my frustrations when I was stumbled. This thesis cannot be done without your encouragement and help.

Thank my friends in Evolutionary Computation Research Group, Alexandre Sawczuk da Silva, Chen Wang, John Park, Bach Hoai Nguyen, Victoria Huang, FangFang Zhang and so many unlisted, for discussing, sharing ideas, reviewing my writings. I would also like to thank Mengge Hao, Shenbo Xuan, Ying Zhu, and Mingming Zhong, for your warm friendship.

Last but not least, I would like to thank my family for their great support and understanding. Thank you to my parents who give me the opportunity to pursue this degree and have always been there for me.

List of Publications

- **B. Tan**, H. Ma and Y. Mei, "A NSGA-II-based approach for service resource allocation in Cloud", IEEE Congress on Evolutionary Computation (CEC), 2017, pp. 2574-2581.
- **B. Tan**, H. Ma and Y. Mei, "A Genetic Programming Hyper-heuristic Approach for Online Resource Allocation in Container-Based Clouds". AI: Advances in Artificial Intelligence, 2018, pp. 146-152.
- **B. Tan**, H. Ma and Y. Mei, "A Hybrid Genetic Programming Hyper-Heuristic Approach for Online Two-level Resource Allocation in Container-based Clouds", IEEE Congress on Evolutionary Computation (CEC), 2019, pp. 2681-2688.
- **B. Tan**, H. Ma and Y. Mei, "Novel Genetic Algorithm with Dual Chromosome Representation for Resource Allocation in Container-Based Clouds", IEEE International Conference on Cloud Computing (CLOUD), 2019, pp. 452-456.
- **B. Tan**, H. Ma, Y. Mei and M. Zhang, "A Genetic Programming Cooperative Coevolution Hyper-Heuristic Approach for Resource Allocation in Container-based Clouds", IEEE Transaction on Cloud Computing, 2019, second around of revision.
- **B. Tan**, H. Ma, Y. Mei, "A Group Genetic Algorithm for Resource Allocation in Container-Based Clouds". Evolutionary Computation in Combinatorial Optimization (EvoCOP), 2020, pp. 180-196.

- **B. Tan**, H. Ma and Y. Mei, "An NSGA-II-based Approach for Multi-objective Micro-service Allocation in Container-based Clouds". IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). 2020, to appear.

Contents

Figures

Chapter 1

Introduction

1.1 Problem Statement

Cloud computing has become the pillar of software industry by offering on-demand computing resources (e.g., storage and computing) [27]. Clouds are essentially data centers that use virtualization technologies, e.g., Virtual Machines (VMs) and containers, to separate servers (Physical Machines (PMs)) into smaller units and lease them to *cloud users* [88]. Clouds bring numerous benefits to *cloud users*. For example, Google deploys its applications on clouds, and anyone who has access to the Internet could use them from anywhere in the world. Clouds release the burden of purchasing and maintaining hardware resources. Hence, the cost of operating applications has decreased because *cloud users* only need to pay for the resources that they rent. In addition, clouds guarantee that their services can be accessible 99.99% of the time [14] and dynamically adjust the capacity for applications when handling the fluctuation of workloads. With the development of cloud technology, many new types of clouds, e.g., container-based clouds, have emerged to provide more profits for both *cloud users* and *cloud providers*.

Container-based clouds [110] or Container-as-a-Service (CaaS) [166] is a recent development in this area. Accoriding to a survey [1] in 2019,

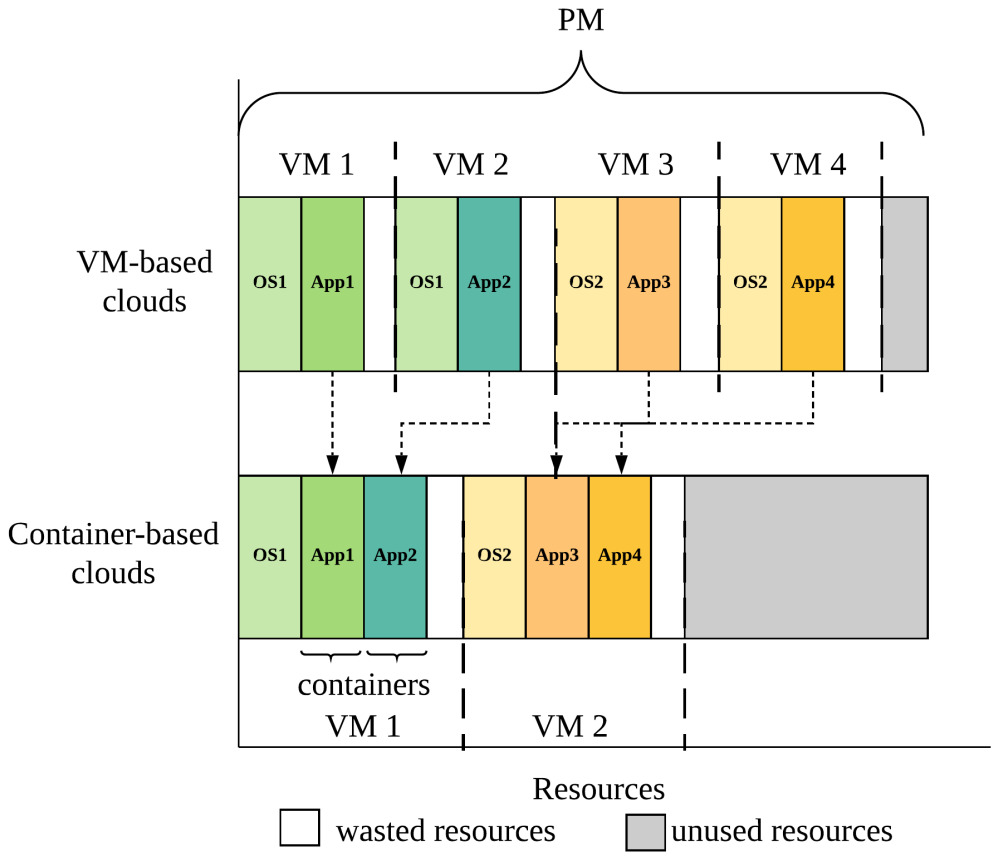


Figure 1.1: Compared to VM-based, the number of VM can be reduced in container-based cloud because containers are co-allocated to VMs.

87% of IT companies have adopt container technology and this percentage was 55% in 2017. There is an urgent need to container-based clouds. Different from traditional VM-based clouds that resources are managed with VMs, container-based clouds use containers as the basic resource management unit. Containers [157] are a new virtualization technology for allocating physical resources to applications. Compared to VMs, containers incur less overhead (e.g., CPU and memory) by sharing Operating Systems (OSs). For example, in Figure 1.1, the PM instance in container-based clouds uses two OSs instead of the four OSs needed by the VM-

based clouds. However, containers suffer from the drawbacks of performance interference (e.g., competition on I/O resources) [229] and security treats [76]. Therefore, container-based clouds use VMs to provide an additional level of isolation for security.

Container-based clouds are beneficial for both stakeholders of clouds, e.g., *cloud users* and *cloud providers*. From the *cloud users'* perspective, container-based clouds are suitable for allocating large scale cloud-native applications [111], e.g., micro-services [149] and server-less architectures [21]. This is because of the flexibility of allocation, the auto-scaling, and high speed in the delivery of enhancements [60]. From the *cloud providers'* perspective, containers provide the potential to reduce the energy consumption of data centers. By using auto-scaling [125] and migration [65] technologies, containers are resource units with finer granularity, which makes their management flexible. Hence, to fulfill the *cloud providers'* need for energy reduction, resource allocation is a critical task in container-based clouds.

Resource allocation is a crucial procedure to effectively utilized resources in data centers. In cloud computing, resource allocation [135] denotes "the process of distributing resources economically between competing groups of programs or users". Resource allocation determines the type, quantity, and placement of resources [238]. The goal for resource allocation is to maximize the profit without violating constraints such as Service Level Agreement (SLA) [197]. Specifically, unlike VM-based clouds which have a one-level structure where VM instances are directly allocated to PM instances, *resource allocation in container-based clouds (RAC)* involves a two-level structure [150]. On the first level, containers need to be allocated to VM instances. New VM instances may be used if the existing VM instances do not provide enough resources to host containers. On the second level, new VM instances need to be allocated to PM instances. In these processes, more variables, such as the placement of containers and the types of new VM instances, need to be determined. Therefore, VM-based allocation ap-

proaches cannot be directly applied in container-based clouds. Existing resource allocation approaches for container-based clouds are mainly meta-heuristics [84,119] and rule-based [133,166,245]. They either only consider one level of allocation, containers to VM instances, or using simple rules to allocate resources. Hence, it is urgent to propose resource allocation approaches for container-based clouds to use cloud resources effectively. Without effective allocation algorithms, container-based clouds may suffer from low resource utilization ranging from 10% to 50% [253]. The extra PM instances consume enormous amounts of energy every year, and this significantly lowers clouds' profit while also emitting a large amount of carbon dioxide into the environment.

Among the numerous scenarios of allocation [88] in clouds, the following three should be further discussed. To maintain a highly utilized data center, some of the allocation scenarios are particularly important and need to be addressed urgently in container-based clouds. One of the most frequently encountered scenarios is **off-line resource allocation** (also called **static** resource allocation). Off-line scenarios [213] allocate a set of containers simultaneously. During the allocation, no migration is done with workload changes [68]. The off-line scenario frequently occurs during the allocation of new applications, e.g., initial application allocation [97, 213]. In this scenario, allocation performance, measured by resource utilization or energy consumption, is the single objective for this task. That is because a large number of allocations can hugely increase the energy consumption of the data center.

On-line resource allocation [213] (also called **dynamic** resource allocation) refers to the immediate allocation of a container. Clouds rely on the on-line allocation to adjust the allocation of applications dynamically. For example, this type of allocation is used in the procedure of container migration, where a container is moved from a PM instance, which is PM overloading or machine breakdown. The container needs to be allocated to another PM instance immediately. Hence, the data center could main-

tain a load-balancing, highly utilized status.

Another important scenario is the **multi-objective resource allocation [238]**, where the relationship between containers is considered. For example, containers are often the components for large-scale applications; their connections and relationships need to be considered. Therefore, in addition to energy consumption, extra objectives, such as availability, are frequently considered in this scenario. As the availability is the most concerned requirement of application users according to IBM's survey [12]. These scenarios bring challenges and require specific designs to cope with their characteristics.

Different approaches are needed for handling the distinct characteristics of each previously discussed resource allocation scenario. The **off-line resource allocation** is a two-level optimization problem. Each level of allocation can be simplified as a vector bin packing problem, which is *NP-complete problems* [245]. This means that a polynomial-time algorithm for solving the problem is not known. Moreover, two levels of allocation need to be determined simultaneously because of their interaction. In addition to energy consumption, the **on-line resource allocation** only has a short decision time to solve the problem once a new container arrives at the cloud. In the on-line problem, the resource requirement of a container is unknown in advance. The **multi-objective resource allocation** problem requires more objectives to be optimized independently. Moreover, when the objectives are conflicting with each other, it is difficult to handle the trade-offs.

Current studies solve the *RAC* problem mainly with two types of method, rule-based approaches [102, 121, 134, 166] and meta-heuristics approaches [84, 119]. The rule-based approaches use rules to make allocation decisions, each targeting a sub-problem of allocation (e.g., containers to VM instances, and VM instances to PM instances). These rules are mostly based on greedy algorithms such as AnyFit-based heuristics [50], e.g., First-Fit, Best-Fit, and other human-designed heuristics [228]. Therefore, they are

easily stuck in local optimal solutions. Besides, these rules consider too few features, which lead to poor generality. For example, their performance varies without considering VM types. As Wolke et al. [226] suggested, rule-based approaches are only useful in some scenarios, such as container migration, because of the short decision-making time. However, they are not appropriate for other scenarios, such as initial off-line container allocation, which has relaxed decision times. The existing metaheuristics are mostly focused on the OS-container allocation problem, which is a one-level allocation problem. Hence, these approaches cannot solve the two-level allocation problems.

Evolutionary Computation (EC) [236] is particularly useful for solving combinatorial optimization problems. EC uses a population of solutions to iteratively search in the solution space. The population-based approaches can avoid becoming stuck in local optimum solutions, and ECs can easily incorporate domain-specific knowledge when addressing real-world problems [69]. Additionally, EC can be used to adapt solutions to dynamic changes in the environment [69]. EC has been successfully applied in many academic and real-world problems, such as Job Shop Scheduling [47, 90, 152], Vehicle Routing [32, 115], and Arc Routing problems [18, 139]. *RAC* is also a combinatorial optimization problem that shares many similarities with the previously mentioned problems. Therefore, in this thesis, we propose EC-based approaches to solve the resource allocation problem in container-based clouds. However, it is challenging to design effective EC-based algorithms for the resource allocation problem. To do this, we need to design representations and specific genetic operators so that EC algorithms can be adapted to the problem. We consider the use of EC to solve the resource allocation problem for the three scenarios in container-based clouds.

This research aims to improve the performance in container-based clouds by using EC algorithms. Specifically, this work focuses on three representative resource allocation scenarios: off-line allocation, on-line allocation,

and multi-objective allocation.

1.2 Motivation

Existing works [178, 198, 206] adopt models from VM-based clouds and use them to solve resource allocation problems in container-based clouds. However, the adaptation of these models is not fully compatible. Using the models from VM-based clouds can lead to inefficiency because of lacking critical features and decision variables. The core difference between VM-based and container-based clouds is that, in VM-based clouds, *cloud users* decide the quantity and types of VM instances and allocate applications inside these VM instances. *Cloud providers* only allocate VM instances to PM instances without considering the utilization of VM instances. In container-based clouds, all of the above decisions are made automatically according to the requirements of the received applications. Hence, additional variables and features, e.g., the placement of containers in VM instances, the types of VM instances, and the overheads of VM instances, are needed to reflect the interactions of two-level allocation in container-based clouds. Thus, a new two-level model where the first level allocates containers to VMs and the second level allocates VMs to PMs must be proposed.

Previous approaches to resource allocation in VM-based clouds also can lead to inefficiency if they are applied directly in container-based clouds. For the off-line resource allocation problems, some studies [83] simplify the two-level allocation problem by using one type of VM for all containers. Other works [85, 119, 185, 220] simplify the two-level allocation as one-level by directly allocating containers to PM instances. Although the allocation problem is easier to solve with these simplifications, the current approaches are not flexible enough. For example, a variety of affinity constraints on containers, such as security and Operating System, cannot be implemented on the same PMs. Containers with different affinity re-

quirements must be allocated to separate PMs. This causes low utilization of PMs. Besides, the computational time of Integer Linear Programming (ILP)-based approaches [83] grows exponentially as the problem size increases. Therefore, it is infeasible to apply an ILP approach in large allocation problems (more than 1000 containers).

For the on-line resource allocation problem, current works have three major drawbacks. Firstly, similarly to the previous problem, some research focuses on the simplified one-level problem [83]. Secondly, current works mostly employ rule-based approaches [166] to achieve fast and acceptable solutions, and these human-designed rules only consider simple features (e.g., residual resources of PMs) to make decisions. As a result, the performance of these rules varies on various workload patterns of applications, as well as different VM settings. The workload patterns of applications have been proven a critical factor to the resource allocation problems [126]. Therefore, the rules that ignore the patterns and VM types will lead to poor performance. The third drawback is found from a widely used framework of AnyFit-based algorithms [51,98]. AnyFit-based algorithms are often applied to on-line bin packing problems. In this problem, AnyFit-based algorithms always start from allocating containers to existing VMs without considering new VMs, which limits the decision space.

1.3 Research Goals

The overall goal of this research is to improve the performance (e.g., energy efficiency and availability of applications) of resource allocation in container-based clouds by using EC algorithms. More specifically, this work applies EC algorithms to deal with the three resource allocation scenarios, off-line allocation, on-line allocation, and multi-objective allocation.

1. Off-line allocation deals with a set of applications being allocated to unused PM instances to minimize the used PM instances. This is the

most common scenario in a data center, e.g., the initial allocation of applications. We design a new two-level allocation model and Genetic Algorithms (GAs) to solve the problem. In particular, we will investigate the performance of two representations of the problem, vector-based and group-based, and compare their performances when applying them in an EC algorithm.

- *Propose a model for the resource allocation in container-based clouds problem.* The detailed two-level allocation procedure is first defined. The relationship between overheads, types of VMs, and numbers of VMs are introduced in this new model. The decision variables, constraints, and objectives are formally defined.

- *Propose a vector-based representation approach for the off-line resource allocation problem.*

  The research aims to investigate the performance of vector-based representation approaches. The vector representation has been successfully applied in a variety of cloud resource allocation tasks [101] with EC algorithms. It is easy to encode a solution in the vector form and use existing genetic operators to find near-optimal solutions. GA is a population-based search algorithm [215] that has been widely applied in the combinatorial optimization problems.

  Two reasons motivate us to develop vector-based approaches. Firstly, existing vector-based approaches are designed for one-level optimization problems [105, 119, 164] and therefore they cannot be directly used to solve our problem. We study the performance of two-level vector-based approach in this thesis. Secondly, the effectiveness of different decoding techniques is unknown.

- *Propose a group-based representation approach for the off-line resource allocation problem.*

  We also investigate the group-based representation, as it can directly

represent the grouping of containers and VM instances. Hence, it does not require a decoding process that affects the quality of solutions. Besides, the group representation is intuitive facilitates the incorporation of domain knowledge in the genetic operators in order to accelerate the search process.

2. On-line allocation requires dynamically allocating applications. For example, it requires an immediate allocation of applications for PM overloading and machine breakdown [28]. The computational time should be as short as possible for a single allocation decision. Therefore, current solutions are rule-based [133, 166]. However, Manually designing heuristics/rules to include all features related to the problem is not practical. Therefore, current rules only consider simple features, and this leads to poor performance. A genetic programming [22] hyper-heuristic (GPHH) can learn patterns from historical workload records of applications off-line and automatically generate new heuristics. A Cooperate Coevolution GP (CCGP) integrates GPHH with a cooperative coevolution framework that allows GPHH to generate rules for two-levels of allocation. These generated heuristics can be used to solve the on-line problem. Hence, we list three sub-objectives as follows:

- *Propose an on-line model for the resource allocation in container-based clouds problem.* The first sub-objective is to propose an on-line model for the problem. This on-line model is used in the training and test processes in order to evaluate the performance of allocation algorithms.

- *Propose a GPHH approach combined with heuristics for the on-line resource allocation problem.*

  Since it is challenging to evolve both levels of rules in the first place, we develop a GPHH approach to generate allocation rules for allocating containers to VM instances. For the allocation of VM instances to PM instances, we apply human-designed heuristics. The main purpose is to develop a GPHH to evolve reservation-based

rules [51,98] instead of AnyFit-based rules [51,98] for allocating containers to VM instances.

- *Propose a CCGP approach for the on-line resource allocation problem.*

  The third sub-objective is to propose a CCGP approach to generate all the allocation rules that are needed, including *VM selection*, *VM creation*, and *PM selection*, to solve the resource allocation problem. This sub-objective adopts the method of generating reservation-based rules in the previous sub-objective. New training procedures and terminal sets are proposed.

3. Multi-objective allocation considers the interconnections among the applications. Therefore, multiple optimization objectives are commonly considered. We design the allocation model with additional objectives to capture the real-world requirements. Meanwhile, we design an EC-based approach to solve the multi-objective allocation problem.

- *Propose a multi-objective model for the resource allocation in container-based clouds problem.* The first sub-objective is to propose a multi-objective model. Interconnections of containers, variables, constraints, and objectives are proposed.

- *Propose an EC approach for the multi-objective resource allocation problem.* The second sub-objective is to develop a multi-objective EC algorithm to solve the problem. This includes developing a representation and genetic operators that are suitable in the problem with conflicting objectives.

1.4 Major Contributions

This thesis proposed four major contributions to the area of resource allocation in cloud computing:

- This thesis proposes three variations of the *RAC* model corresponding to three resource allocation scenarios in container-based clouds: an off-line, an on-line, and multi-objective resource allocation models. New features, such as VM overheads, VM types, and affinity constraints, are included in the new models. These models can either be used in the off-line algorithms or used in simulations to evaluate the performance of algorithms.

- This thesis proposed three GA-based approaches, e.g., *SGA*, *DGA*, and *GGA-RAC*, for the off-line resource allocation. These three approaches are developed based on two types of representation, i.e., vector-based and group-based representations. Corresponding genetic operators and decoding processes are also developed and compared. The three approaches are compared with state-of-the-art algorithms to determine which is most suitable for the off-line allocation scenario.

- This thesis proposed two hyper-heuristic approaches, i.e., a Genetic Programming Hyper-Heuristic-based approach (*GPHH-RAC*) and Cooperative Coevolution Genetic Programming-based approach (*CCGP-RAC*), for the on-line resource allocation. Since this is the first time that GPHH and CCGP are applied on the *RAC* problem, the first contribution of the two approaches is the training procedure for generating reservation-based rules. The reservation-based rules can simultaneously decide whether a container is allocated to an existing VM instance or a new VM instance with the selected type. The second contribution is the evolution of two cooperative rules to solve the two-level problem. The experiment results show that automatically generated rules from both *GPHH-RAC* and *CCGP-RAC* outperform the state-of-the-art, manually designed rules. We also provide some insightful analysis of human-designed rules and generated rules. These insights are informative for future studies to design

algorithms.

- This thesis proposes a multi-objective approach, e.g., *NS-GGA*, for the multi-objective resource allocation. *NS-GGA* adopts a group-based representation and an NSGA-II framework. We develop novel genetic operators that can handle the trade-offs between conflicting objectives. The experiments on real-world datasets show that by comparing the proposed approach with three state-of-the-art algorithms: *FF&BF/FF*, *Spread*, and a vector-based *NS-DGA* approach. The proposed *NS-GGA* outperforms all other approaches in both objectives. In addition, *NS-GGA* provides a set of solutions that have a trade-off between energy consumption and availability.

1.5 Organization of Thesis

- Chapter 1: Introduction

  The problem statement, research goal, motivation, thesis contributions, publications, and the thesis organization are presented in this chapter.

- Chapter 2: Literature Review

  This chapter presents a background of cloud resource management, virtualization technologies, and resource allocation strategies. In addition, it presents a literature review of off-line, on-line, and multi-objective resource allocation. It highlights the main limitations and current challenges.

- Chapter 3: Genetic Algorithms for Off-line Resource Allocation in Container-based Clouds (RAC)

  This chapter proposes a novel model for the off-line resource allocation problem in container-based clouds. Three novel genetic algorithm-based approaches, which based on two representations, are proposed

to solve the allocation problem. The performance of three proposed approaches is compared with the benchmark algorithms using a real-world dataset.

- Chapter 4: Genetic Programming for On-line Resource Allocation in Container-based Clouds (RAC)

  This chapter proposes an on-line model for the on-line allocation problem. Then, two novel hyper-heuristics approaches are proposed, a Genetic Programming Hyper-heuristic-based approach (*GPHH-RAC*), and a Cooperative Coevolution GP-based approach (*CCGP-RAC*). The performance of the two algorithms is evaluated through simulations.

- Chapter 5: Evolutionary Multi-objective Optimization for Resource Allocation in Container-based Clouds (RAC)

  This chapter proposes a novel model for the multi-objective resource allocation problem in container-based clouds. It also proposes a novel *NS-GGA* approach to solve this problem. The performance is evaluated with a real-world dataset and compared with algorithms used in the industry.

- Chapter 6: Conclusions

  In this chapter, the conclusions and findings in each chapter are presented and summarized. The chapter also describes the main future research directions arising from the contributions of this work.

  The connections between the major contribution chapters in this thesis is shown in Figure 1.2.

  The off-line *RAC* model in Chapter 3 provides the functionality to evaluate the problem. This model is also used and modified to fit the on-line *RAC* and multi-objective *RAC* problems in Chapter 4 and 5. Meanwhile, two vector-based approaches and a group-based approach are proposed. Chapter 4 develops two GP-based approaches,

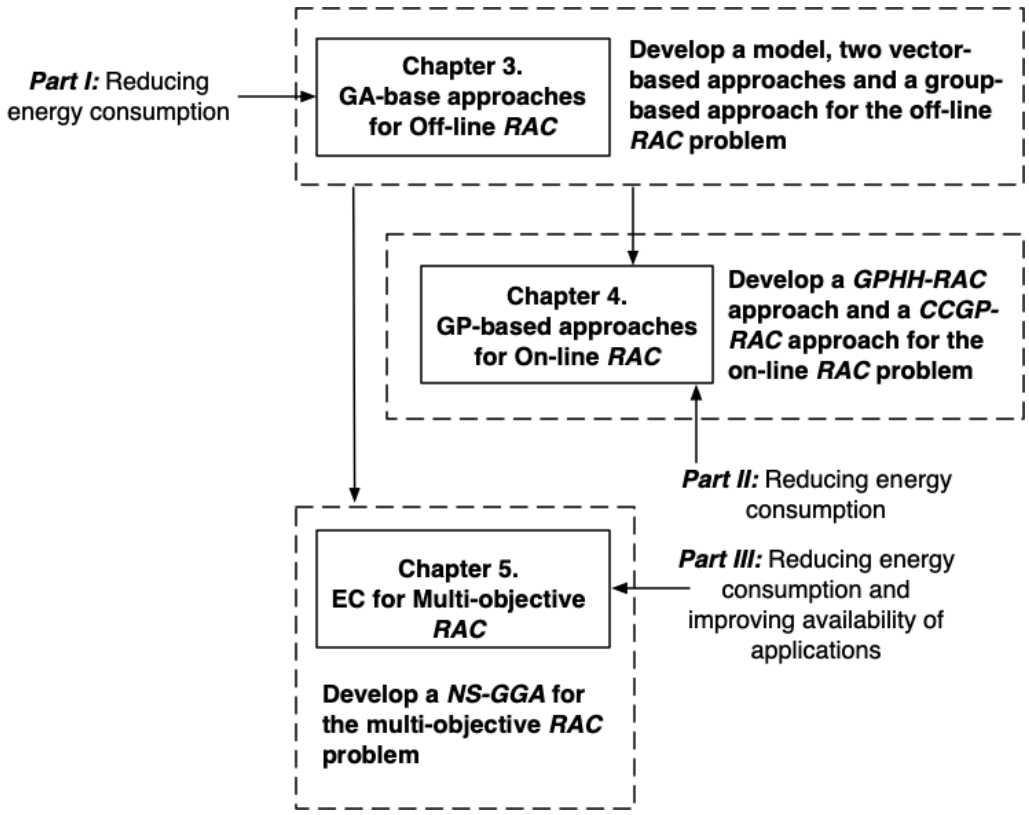


Figure 1.2: The connection between major contributions chapters in the thesis.

e.g., a *GPHH-RAC* approach and a *CCGP-RAC* approach, for the on-line *RAC* problem. Chapter 5 investigates the trade-off between energy consumption and the availability of applications in a multi-objective problem. A multi-objective approach *NS-GGA* is developed to improve both energy consumption and the availability of applications.

Chapter 2

Literature Review and Background

This chapter introduces the fundamental concepts of resource allocation in cloud computing, evolutionary computation (EC), and related works. We separate the background section into two parts. Section 2.1, introduces the concepts of cloud computing. Section 2.2 explains the methods of evolutionary computation that we intend to use to solve the allocation problem. Then, Section 2.3 reviews the related works. Section 2.3.1 reviews the existing problem models for the research problem. Section 2.3.2 and 2.3.3 review the studies on off-line and on-line scenarios of the allocation problems. Section 2.4 concludes the findings in the literature and positions our research in the field.

2.1 Fundamental Concepts of Cloud Resource Allocation

In this section, we discuss the roles in cloud computing and service models. Then, Section 2.1.2 discusses the goal of resource allocation. Section 2.1.3 introduces two widely used virtualization technologies to achieve

this goal, virtual machines, and containers and discusses their differences. We then introduce the problem studied in the thesis, the resource allocation problems in container-based clouds. Section 2.1.4 discusses the workload types in resource allocation problems. Section 2.1.5 explains the abstract problem of resource allocation and further categorizes the problems into two scenarios, off-line and on-line allocation scenarios because they must be resolved with different methods. Section 2.2 introduces the overview of evolutionary computation (EC). We explain a few EC algorithms in detail, e.g., genetic algorithm, NSGA-II, and Genetic Programming.

2.1.1 An Overview of Cloud Computing

Cloud computing is a computing model that offers a network of servers to their clients in an on-demand fashion. According to NIST's definition [141], *"cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* Hence, the primary functionality of cloud computing is to provide facilities and resource management to *cloud users*.

Cloud computing involves three stakeholders (see Figure 2.1). *Cloud providers* build data centers and maintains the servers at the data centers. To use these remote servers, *cloud users* (e.g., an application provider), can deploy and access their applications (e.g., Endnote, Google Drive) in these servers from anywhere in the world. Once the applications are deployed, *end users* can use them without installing on their local computers. *Cloud providers* charge fees from *cloud users* for using the infrastructure. *Cloud users* charge fees from *end users* for using applications. To provide cloud services, three fundamental service models define the responsibilities of stakeholders.

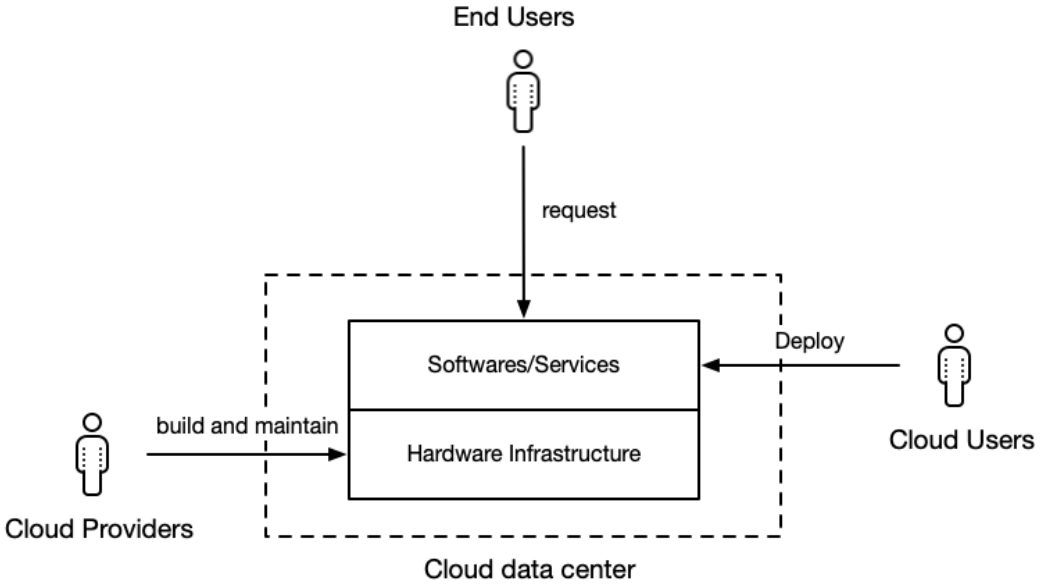


Figure 2.1: Stakeholders of cloud computing adapted from [97]

Software as a Service (SaaS) *Cloud users* develop applications so that *End users* access and use these applications. The applications can be deployed in private or public cloud data centers. In the following content, we use the term *clouds* to represent *cloud data centers*. Deploying applications in private clouds is the traditional software development paradigm. *Cloud users* manage their facilities and resource management of applications, e.g., adding, removing servers, install software libraries. When applications are deployed in public clouds, *cloud providers* take care of resource management. Different cloud applications, e.g., Google Drive, Gmail, and Netflix, are examples of SaaS.

Infrastructure as a Service (IaaS) *Cloud providers* offers resources (e.g. virtual machines (VMs)) to *cloud users*. The abstract of physical computing resources, such as CPU, memory, and network, are packed into VMs and hid from *cloud users*. Amazon Elastic Compute Cloud (EC2) [3] is one of the examples of IaaS.

To manage the cloud resources, *cloud users* and *cloud providers* have different responsibilities. *Cloud users* estimate the resources that they need and rent several VM instances of certain types. After that, *cloud users* manages the VM instances as remote servers. *Cloud providers* do very little management inside the VM instances. Instead, *cloud providers* manage VM instances as whole resource units, e.g., monitoring the usage of VM instances, adding and deleting VM instances (i.e., horizontal auto-scaling service).

One of the significant disadvantages of IaaS in terms of resource allocation is that *cloud users* tend to reserve more resources for ensuring the Quality of Service (QoS) at peak hours [52]. The overly reserved resources lead to low resource utilization of Physical Machine (PM) instances in clouds during the non-peak hours.

Platform as a Service (PaaS) Unlike IaaS, PaaS offers a platform of development, deployment, and automatic resource management to *cloud users*. Since the cloud platform supports the full cycle of software development, *cloud users* can focus on the functionalities of the software. Google AppEngine [9] is an example of PaaS.

PaaS cloud provides resource management, including resource provisioning, allocation, and auto-scaling, and offers a set of APIs that allows *cloud users* to deploy their applications. PaaS clouds avoid the disadvantage of the separated responsibilities between *cloud providers* and *cloud users*. Therefore, it can potentially improve resource utilization than IaaS clouds.

Also, in recent years, a set of new PaaS clouds has emerged. As an extension of PaaS, Function-as-a-Service (FaaS) [8] allows *cloud users* to develop, run, and manage a set of functions instead of developing a monolithic application. Requests of the functions from *end users* are allocated and processed in clouds automatically. Since no virtual or physical server is needed to run an application, FaaS is also a subset of server-less com-

puting [21]. Amazon Lambda [6] is an implementation of FaaS clouds. Container-as-a-Service [167] uses containers and VMs as resource management units. CaaS is particularly good at managing micro-service-based applications. With each container hosts a micro-service instance, CaaS can handle large scale of containers. CaaS is also named container-based clouds, which will be discussed in Section 2.1.3. Red Hat OpenShift [10] and Amazon Container Service [5] are both the implementations of CaaS.

Clearly, cloud techniques are becoming more automatic and intelligent to handle the deployment and delivery of applications. This could release the burden of resource management away from *cloud users*. Hence, this requires *cloud providers* to use advanced techniques to handle the resource allocation in clouds to not only meet the requirements of *cloud users* but also maximize the profit of clouds.

2.1.2 Cloud Resource Allocation

Cloud resource allocation is subset of cloud resource management [224] which broadly includes all manipulations of cloud resources including resource provisioning [194], resource scheduling [195], etc. Resource allocation [135] denotes as "the process of distributing resources economically between competing groups of programs or users". That is, resource allocation refers to the process of selecting the number of resources for applications and deciding the physical locations of these resources in cloud data centers. The aims of cloud resource allocation can be generally categorized into two groups, satisfying Service Level Agreement (SLA) and minimizing energy consumption of data centers.

SLA

SLA [38] is a blueprint that defines the cloud service quality parameters that required by *cloud users*. These parameters, also called Quality of Service (QoS), such as availability that defines the time that an application

is available, e.g., 99% uptime means *cloud users* will be unable to access the application for no more than about 3.65 days per year. Other QoS parameters [7], such as maximum response times, worse case recovery, are different by *cloud providers*.

SLA is also a contract that sets up QoS constrains the *cloud providers* need to satisfy while providing services. Otherwise, penalties are enforced. Hence, to satisfy these SLA constraints, *cloud providers* apply various methods to improve the service quality, such as using clusters of computers to offer high-availability or distributing workload requests to multiple data centers [38]. Other methods are using algorithms to optimize the allocation resources in data centers so that the QoS is satisfied. The resource allocation problem is first modeled and formulated into optimization problems. The QoS parameters are modeled as the optimization objectives [162] (e.g. availability) or the constraints of the problem [162] (e.g. resource constraint).

Energy Consumption of Data centers

The major expense of *cloud providers* is the energy consumption [182]. According to [182], both the cooling system and PM instances account for 40% of energy consumption. To minimize energy consumption of data centers, two approaches can be used. First is the energy reduction of cooling system. This is mainly for the design of data centers. Another approach is server consolidation. This approach reduces the energy consumption of used servers. The current energy efficiency of servers is low on average [253]. This thesis focuses on the server consolidation methods.

The main cause of low energy efficiency is the low utilization of PM instances, which accounts for 20 to 30% on average. A low utilized PM instance, e.g., 15% of the CPU capacity, also consumes 70% of the energy of its peak time. Hence, *cloud providers* aim to maximize the utilization of PM instances to reduce the energy consumption of data centers.

Cloud providers apply virtualization technologies so that large PM instances are fragmented into smaller units which can be to allocated to

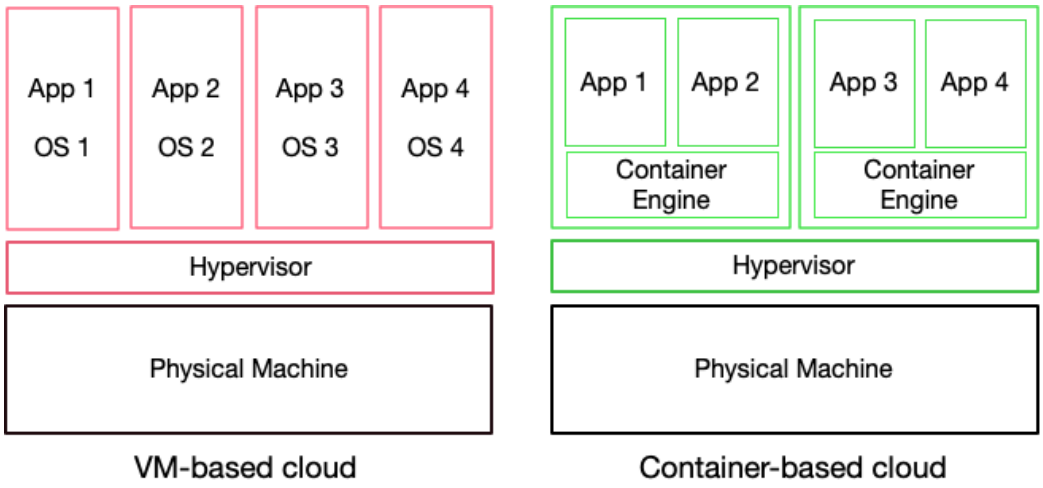


Figure 2.2: VM-based and Container-based virtualization adapted from [165]

multiple users and released.

2.1.3 Virtualization Technologies

Cloud providers use virtualization technologies [208] to achieve finer granularity resource management than the traditional way of allocating an entire PM instance to a single user. Virtualized management partitions the resources of a PM instance (e.g., CPU, memory, and disk) into several independent units and allocates applications into these units. The most common units are VMs and containers. The following sections illustrate two classes of virtualization (see Figure 2.2): *VM-based* and *container-based virtualization*.

VM-based Virtualization

A VM-based virtualization has three-layers of structure: PM-Hypervisor-VM (see Figure 2.2 left-hand side). An underlying PM instance provides the hardware resources such as CPU and memory. Hypervisors [147], or

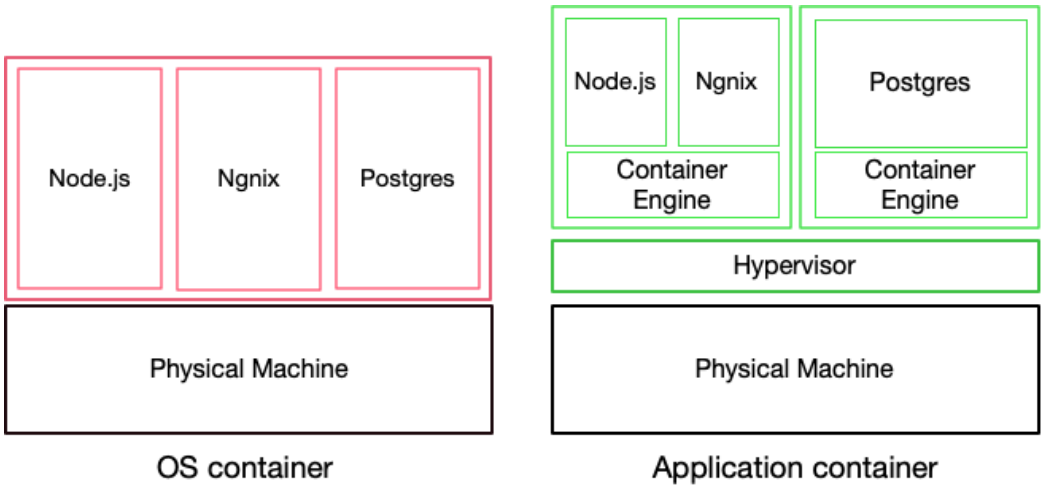


Figure 2.3: A comparison between OS container and Application container adapted from [168]

the Virtual Machine Monitors (VMMs) are on the software layer on top of a PM instance. A hypervisor accesses to a PM instance's resources so that VM instances can share resources of the PM instance. A VM instance is the basic resource management unit. A VM instance has its resource capacities include CPU, memory, bandwidth, etc. Each VM instance can be accessed remotely and managed as a server. Among multiple VM instances in the same PM instance, isolation of PM's resources is ensured by the hypervisor. To simplified the resource allocation in clouds and satisfy the requirement of *cloud users*, *cloud providers* normally provide various types of VM with distinct capacities [4]. Some implementations of VM-based hypervisor such as Xen [23], KVM [104], and VMware ESX [216] dominate this field in recent years. On top of a hypervisor, VM instances are the resource management units. A VM instance allows an independent Operating System (OS) to run on it.

Container-based Virtualization

Traditional container virtualization, e.g., OS containers [187], has been used in big data processing platforms as the computing units for over a decade [136, 219]. In recent years, new container platforms, e.g., Docker, introduce application containers [89] to facilitate the development of applications.

An OS containers (Figure 2.3 left-hand side) have an OS installed and host multiple applications from different *cloud users*. These applications are segmented and isolated. An OS container is suitable for deploying a large number of applications that share the same OS kernel. On the other hand, when various applications require different OSs, OS containers' performance is similar to VMs. Three implementations of OS containers: OpenVZ, Google's control groups, and namespace are widely used in Google and Facebook.

Application containers [29] (Figure 2.3 right-hand side) have a many-to-one mapping relationship with a VM instance. A single application runs in an application container. Major implementations such as Docker, Rocket, and Kubernetes [29] are prevalent in the software industry. In comparison with OS containers, application containers are much more flexible in terms of software development and deployment. With application containers, each application can be deployed separately on different machines. These containers can be vertically scaled up to add capacities. Hence, application containers are more suitable for modern cloud-native architectures such as micro-services and server-less because these architectures are highly distributed and loosely coupled.

Comparison between Container-based and VM-based Virtualization

This section compares VM-based and container-based virtualization [59, 66, 230] in terms of the key characteristics of resource allocation.

Containers have mainly three advantages over VMs. Firstly, contain-

ers are lightweight, which means they generate much less overhead than a VM hypervisor [66]. Secondly, containers share OSs. Therefore, they can reduce the overheads on OSs. Thirdly, containers naturally support vertical scaling [53] while VMs do not. Vertical scaling means a container can dynamically adjust its resources under the host's resource constraint. This feature offers fine granularity management of resources.

Although containers have numerous advantages compared to Virtual Machines (VMs), they suffer from security threats [76, 138] and performance interference (e.g. competition on I/O resources) [229, 237]. Therefore, when facing the diverse requirements from applications, e.g., various OSs and security levels, *cloud providers* can use VM instances to provide an extra level of isolation for containers. Hence, a new two-level resource allocation [150, 173, 187, 245] emerges where on the first level, containers need to be allocated to VM instances. New VM instances may be used if the existing VM instances do not have enough resources to host the containers. On the second level, the new VM instances need to be allocated to PM instances. This novel cloud architecture is called container-based clouds.

Container-based Clouds

In recent years, a new type of cloud, container-based clouds [110] have emerged that apply both containers and VMs as the resource management unit. With these two virtualization technologies, i.e., VMs and containerization, complement each other, container-based clouds have the potential to achieve a high resource utilization as well as high security over the applications.

Resource allocation in container-based clouds (RAC) brings new challenges. First, the existing resource allocation tools and approaches cannot deal with heterogeneous resources [136]. For example, a cluster management tool, Swarmkit [142] can be used to manage a cluster of docker containers. However, Swarmkit cannot allocate containers into heterogeneous VMs

or PMs. The heterogeneity of VM/PM configurations, including different resource sizes, types, Operating Systems, etc. Second, the resource allocation in container-based clouds involves two-levels of allocation that is extremely difficult. Each level of allocation is a vector bin packing problem, which is NP-hard [50]. The allocations at the two levels interact with each other. For example, when allocating containers to VM instances, the selection of different VM types impact the allocations in the second level. Hence, ideally, the two-level allocations should be done simultaneously to find the global optimal solution.

Hence, the popularity of container-based clouds and the difficulty of the *RAC* problem motivate us to develop novel algorithms to solve the *RAC* problems in different scenarios.

2.1.4 Workload Types

Workload [39] is defined as the resource consumption of a computational job that completed by a computational unit in a given time. The type of application significantly affects the resource allocation techniques [155]. This section summarizes five types of commonly encountered workload types in clouds.

Web Applications

Web applications are long-term workloads meaning that they continuously run for days before releasing from servers [88]. Once web applications are deployed, they constantly receive requests from *end users*. Two distinct allocation scenarios require different allocation strategies. The first scenario is at the beginning of deployment. A set of application web application with their predefined resource requirement is initially deployed to a data center. Since the information of web applications are known, this scenario is considered as an off-line allocation [226]. After deployment, the resource requirement is continuously changing with the number of

incoming requests. Migration strategies are used when the host servers are overloading. This scenario is often considered as an on-line allocation [103]. Reactive rules are usually used to re-allocate web applications in order to avoid the competing of resources.

The optimization objectives of web applications can be separated into two perspectives. From the *cloud users'* perspective, the objectives include cost of used VMs/PMs, and a variety of QoS requirements such as availability [78,85,220]. From the *cloud providers'* perspective, energy consumption is often considered.

Bag of Tasks

Bag-of-tasks (BoT) refers to those applications which have a set of independent jobs that can be processed in parallel [86], e.g., massive search such as key breaking. In clouds, BoT is charged for complete allocation slots (e.g., 1 hour) of computing resources regardless of whether the tasks are being executed [86]. Traditional algorithms such as Min-Min, Max-Min [71], and Sufferage [129] are used to schedule these tasks. Recent approaches [113, 199] assumes that the task execution time is known to optimize the allocation.

Makespan is a common objective of BoT scheduling tasks [86]. Besides, preparation of the input data before executing is necessary for BoT tasks.

Big Data Applications

Big data applications involve a massive amount of data being collected and processed [137]. Generally, big data applications involve high volume, diverse variety of data, and require a high velocity to process [251]. To be specific, two categories of applications, batch processing and workflow applications, are examples of big data applications.

Batch Processing Batch processing applications, such as Page Rank and TeraSort, utilize distributed frameworks like MapReduce [56] to process a batch of computational tasks. One of the famous implementation of MapReduce, Hadoop, first employed First-In-First-Out (FIFO) scheduler and then implemented the Fair scheduler [241]. However, these schedulers do not consider the energy efficiency of the data center. Mashayekhy [137] developed two energy-aware scheduling algorithms, EMRSA-I and EMRSA-II, which can significantly improve energy efficiencies and satisfied the SLA requirements.

Workflow Applications Workflow applications, such as scientific workflows [80], are commonly represented as a directed acyclic graph (DAG). The optimization objective of workflows is usually the execution time of applications or the processing capacities [158]. The allocation of workflow jobs is an NP-complete problem in general form [209]. Hence, heuristics and meta-heuristics are proposed to solve the problems. Deelman et al. [58] develop a framework named Pegasus. Other systems, such as GridFlow [40] and ICENI [73], are also based on heuristics. Evolutionary Algorithms, such as GA [240], PSO [158] based approaches, are also proposed to solve workflow allocation problems and show better performance than the traditional heuristics.

This thesis considers the workload type of web applications. Firstly, since web applications are long-term tasks, they may take months to be released. Secondly, our research can be extended to other types of workload by considering the release time or the due date.

2.1.5 Allocation Scenarios and Problems

Since the focus of the thesis is the *RAC* problem, this section introduces the abstract of the *RAC* problem as a vector bin packing problem; and two allocation decision scenarios, the off-line scenario, and on-line scenario [226].

Vector Bin Packing Problem

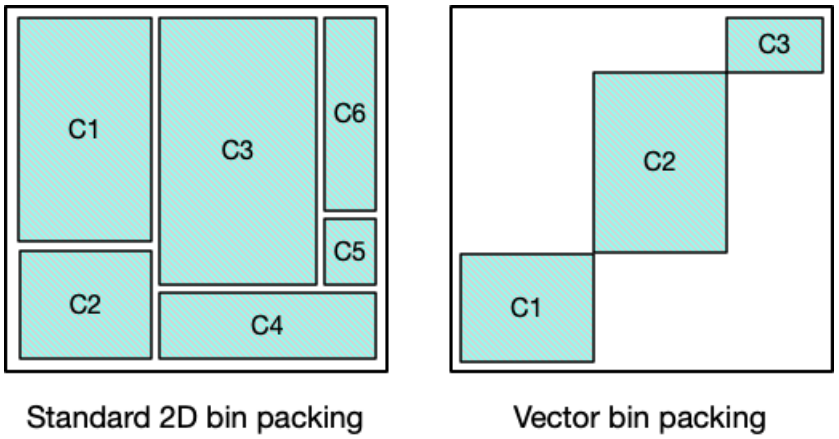


Figure 2.4: A comparison between standard bin packing and vector bin packing

Resource allocation in clouds [159], including *RAC* problem, can be simplified as a vector bin packing problem [45]. The resources, such as CPU, memory, and storage of resource units, are considered as the dimensions of the problem. Different from the standard bin packing problem (see Figure 2.4), the vector bin packing problem considers the resources that have been used by other items.

Since vector bin packing is an NP-hard problem [45]. Current studies develop heuristics and meta-heuristics to solve the problems. First Fit Decreasing (FFD) is a widely used heuristic [16, 190]. Pandit et al. [159] propose a simulated annealing-based approach that searches for the near-optimal solutions in the discrete solution space. Meta-heuristics, such as genetic algorithm (GA) [225], Particle Swarm Optimization [233], have shown great potential to solve the problem.

The *RAC* problem can be treated as a two-level vector bin packing problem. The first level allocates containers to VM instances, and the second level allocates VM instances to PM instances. Hence, the *RAC* problem is even challenging to solve. Additionally, two allocation scenarios,

off-line allocation, and on-line allocation, have their different characteristics and require different methods to solve.

Off-line Allocation Scenario

Off-line scenario allocates a set of applications in planning time to achieve a global optimal solution. In real-world cloud allocation, the initial allocation of applications [145, 201] and periodic re-allocation of applications [64, 227] are two typical examples of the off-line scenario. The initial allocation of applications occurs when a set of new applications needs to be allocated [97]. The periodic re-allocation of applications occurs when the utilization of PM instances decreases to a certain level, which triggers the re-allocation process [97]. Then, the re-allocation process moves the existing applications from one PM instance to another to achieve high resource utilization. The moving step is called migration [114]. Since the applications are constantly arrived and released, *cloud providers* often set a utilization threshold [120] or periodically [227] re-allocate the existing applications that running in the PM instances.

On-line Allocation Scenario

On-line scenario allocates applications one by one as they arrive at clouds. Hence, the allocation decision must be made in real-time. In clouds, on-line allocation happens in multiple scenarios such as machine breaking [75], overloading [28], allocation with real-time priority [44]. All these scenarios require emergent allocations. Therefore, a real-time decision is necessary. Since time is a critical factor in the on-line problem, the off-line methods usually cannot be used because they take a long time to search for the optimal solution. Also, the on-line allocation only allocates a small number of applications at a time. It is difficult to find the optimal allocation solution for the period considered.

2.2 Fundamental Concepts of Evolutionary Computation

Evolutionary Computation (EC) algorithms [20] are artificial intelligence algorithms that are inspired by biological mechanisms of evolution, social interactions and swarm intelligence. They are famous for strong searchability because they have several distinguishing characteristics, such as the use of a population-based search, the stochastic search, and heuristics embedded with domain knowledge [31]. EC algorithms have been applied successfully to solve a variety of real-world problems [232], especially combinatorial optimization problems. This section mainly reviews three EC algorithms that are applied to our work, *genetic algorithms (GAs)*, *Non-dominated Soring GA II (NSGA-II)*, and *genetic programming (GP)*.

2.2.1 Genetic Algorithms (GAs)

GA is a population-based search algorithm [215] that originally developed by Holland [91]. In GA, the solution of a problem is encoded as a chromosome or an individual. GA searches for the best solution by iteratively changing the values in the chromosomes so that the solution is explored.

A general GA procedure begins with an initialization of a population of chromosomes. The initialization generates a population of solutions that widely spread across the solution space. The purpose of initialization is to start the search in a good position by obtaining knowledge through sampling. It followed by an iterative process called generation. In each generation, the fitness value of chromosomes is evaluated according to a defined fitness function. Generally, a fitness value represents the quality of a solution. Then, genetic operators such as mutation and crossover are applied to the solutions so that they are modified. As a search mechanism, these operators evolve solutions to explore the search space on different distances. New generations of solutions are then evaluated by the fitness

function. This evolutionary procedure ends when a predefined generation number or a satisfactory level of fitness has been reached.

GA-based algorithms are suitable for off-line combinatorial optimization problems. Although GA-based algorithms do not guarantee the global optimal solution, they usually can find near-optimal solutions within a feasible amount of time. However, GA-based algorithms still take too long than on-line problems require. Therefore, this thesis considers using a GA-based algorithm in the off-line *RAC* problems.

Representation for Resource Allocation Problems

For evolution computation, the design of representation is a key issue because the quality of the representation of a problem can decide the landscape of the studied problem [19]. It is difficult to design the representation because it must be able to correctly represent the problem search space. The represetation is tightly coupled with the operators that act upon it. The operators must be able to manipulate an instance with different distances so that it can control the search speed and strategy.

This section reviews some of the representations for resource allocation problems. The common representations in the literatures include *vector-based representation* [87,93] and *group-based representation* [61].

Vector-based Representation Traditionally, GA employs vector-based representation and encode solutions of a problem into a vector of integers or binary values [101,183]. For resource allocation problems, the vector-based representation is an indirect representation, and the individuals must be decoded into a solution.

In the literature, the vector-based representation has been successfully applied to a variety of cloud resource allocation tasks [87,101]. Specifically, Phan et al. [164] develops a vector-based representation to allocate computation tasks to VM instances. Klein el al. [105] proposes a SanGA approach for service composition problem. Rahimi et al. [177] proposes MAPCloud

to allocate mobile applications on 2-tier cloud architecture. Their Simulated Annealing-inspired approach is applied to the vector-based representation.

There are mainly two advantages of applying vector-based representation. Firstly, it is easy to encode the solution in a vector form. Both binary and permutation representations have been developed for similar problems [101]. Secondly, with various existing genetic operators, GA is able to search the solution space stochastically. The stochastic search provides a general search pattern without any background knowledge of the problem. Hence, GA can be easily applied to a variety of problems.

On the other hand, the disadvantages are also observed. Since the *RAC* problem is a grouping problem (bin-packing problem [101]), the vector-based representation requires a decoding method to transform an individual into an allocation solution. The quality of the decoding method affects the effectiveness of the search. Another disadvantage is also caused by the decoding process that it is hard to apply domain knowledge to the operators. Although GA can search the solution stochastically, domain knowledge can accelerate the search. However, with vector-based representation, the search process is in the genotype space, while the solutions are evaluated in the phenotype space. Hence, the design of domain-specific operators is difficult.

Group-based Representation Group-based representation is a direct representation for the resource allocation problems because these problems are normal variations of bin packing problems [124].

Group-based GA (GGA) was proposed by Falkenauer [61] to solve the bin packing problem. *GGA* overcomes a significant defect, the redundant encoding problem, in the ordering GA [169]. The ordering GA uses a vector-based representation, and the decoding process highly relies on items rather than the numbering of groups. For example, using two letters A and B to represent distinct groups, AAB and BBA are two solutions.

However, a decoding process may decode these two solutions with the same meaning – the first two items are in the same group, and the third item is in another group. Therefore, these two solutions are redundant. *GGA* proposes a variable-length representation to solve the issue of redundancy. The new crossover, mutation, and inversion operators directly operate on groups instead of items. Later on, Quiroz-Castellanos [175] embeds heuristics into the algorithm to speed up the search process.

GGA has been successfully applied to solve many bin packing problems such as ordering batch problems in warehouse [106], VM allocation problem [82, 99, 123, 235], and assembly line balancing problem [184]. Group representation is also widely used for those similar problems, such as Job shop scheduling problems [47].

Compared to vector-based representation, group-based GA has two advantages. Firstly, the group representation does not require a decoding process, which affects the quality of solutions. Secondly, group-based representation has a variable length. Hence, no computational resource is wasted on the unused part of a chromosome. Thirdly, since the group representation is a direct mapping of the allocation solution, it is easy to apply domain knowledge for designing heuristics. Hence, the operators can be more efficient than the stochastic search.

The effectiveness of GA highly depends on the representation of studied problems. Hence, this thesis will investigate which representation is more suitable for the *RAC* problem.

2.2.2 Hyper-Heuristics, GPHH, and CCGP

Hyper-Heuristics

Hyper-heuristic is a learning method which searches in the heuristic space rather than the solution space [36]. Hyper-heuristic exploits the structure of a problem and uses domain knowledge to design heuristics for that problem automatically. Although domain experts still provide domain

knowledge, the human is 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* [36]. 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.

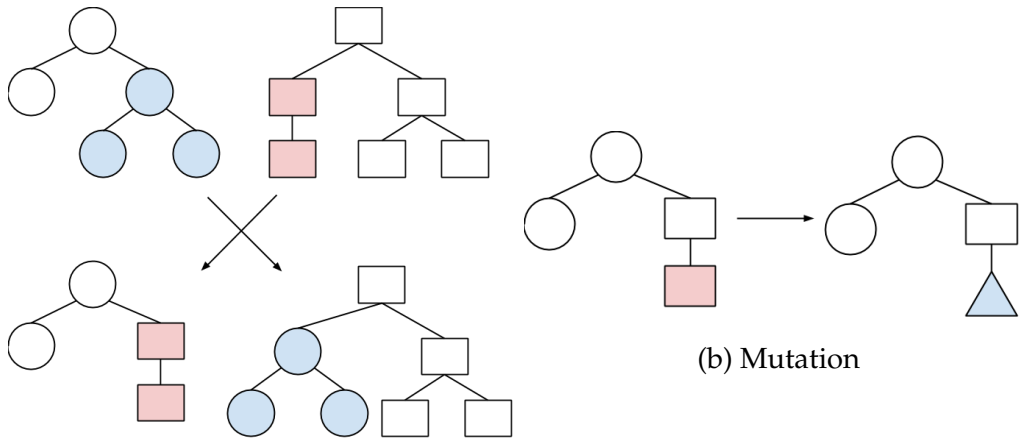
Hyper-heuristics have been used on bin packing problems, and they have been proven to outperform than simple heuristics (e.g., First Fit, Best Fit). López-Camacho et al. [124] state that hyper-heuristics perform equally or better than the best single heuristics. This is beneficial for real-world applications because the choice of best heuristics varies according to different instances and environments. Thus, it is unknown which heuristic is the best in advance. Therefore, the computational cost of using a generated hyper-heuristics is lower than applying all heuristics on the problem and using the best result. Sotelo-Figueroa et al. [196] develop a micro-differential evolution algorithm for evolving bin packing heuristics with an indirect representation. Burke et al. [34, 35, 37] propose hyper-heuristics for variations of bin packing problems, including 1-D, 2-D, and Strip Packing problems. Sim et al. [192, 193] develop lifelong learning hyper-heuristics to the 1-D bin packing problem by using the artificial immune system.

Genetic Programming Hyper-Heuristics

Genetic programming [108] is an evolutionary computation technique, inspired by biological evolution, to find computer programs for solving a specific task automatically. In a GP population, each individual represents a computer program with a tree. In each generation, these programs are evaluated by a predefined fitness function, which accesses the performance of each program. Then, individuals will go through several genetic operators such as selection, crossover, and mutation.

Crossover and mutation (see Figure 2.5) stochastically generate new

solutions from the selected individuals. The crossover randomly selects the branches on two selected individuals 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 individuals until the new population has the same number of individuals as before. Then, the next iteration starts.



(a) Crossover

(b) Mutation

Figure 2.5: Crossover and mutation

The major difference between GA and GP is the representations they used. Each GP individual is represented as a tree with variant depth instead of a string. This representation is particularly suitable for a program. For example, a GP individual is shown in Figure 2.6 which is a program x + max(y $\times$ 2, -2). The variables {x, y}, and constraint {-2, 2} are called terminals of the program. The arithmetic operations {+, $\times$, max } are called functions in GP. A GP individual is a specific combination of elements in a terminal set and a functional set. In order to observe the relationship between a function and its subtrees, the GP programs are usually presented to human users by using the $prefix$ notation similar to a Lisp expression,

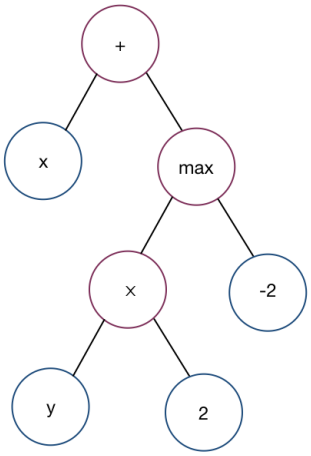


Figure 2.6: GP program that represents x + max(y × 2, -2)

for example, x + max(y × 2, -2) can be expressed as (+ (x (min (× y 2) -2))).

GPHH has been successfully applied to 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 [152] the multi-task JSS [161], and the JSS with machine breakdown [160]. The generated rules outperform neural network techniques. For the bin packing problems, GPHH has been applied to evolve the rules for 1-dimension [35], 2-dimension [34], and 3-dimension [17] 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.

Cloud resource allocation usually has extra constraints such as multi-dimensional resources, migration costs, heterogeneous PMs. These constraints make the cloud resource allocation problems much harder than bin packing problems [132]. Therefore, traditional bin packing approaches such as First-Fit Decreasing, Best Fit, cannot perform well in this context. GPHH, therefore, is a promising technique that can be used to generate heuristics under multiple constraints automatically.

CCGP combines GP with a cooperative framework [152], so that CCGP

can simultaneously evolve multiple heuristics to solve a problem. CCGP maintains N sub-populations for generating N cooperative heuristics respectively. In [239], CCGP generates sequencing and routing rules for Dynamic Flexible JSS (DFJSS) [243, 244]. Similarly, Zhou et al. [250] employ CCGP to evolve machine assignment and job sequencing rules for a multi-objective DFJSS problem.

2.2.3 Non-dominated Sorting GA-II (NSGA-II)

NSGA-II was proposed by Deb et al. [57] in 2002, and it is a widely used multi-objective algorithm. The procedure is similar to GA with additional operators for handling the non-dominated solutions. Initially, the algorithm randomly generates a population of N solutions. Then the population is sorted with a fast non-dominated sorting. Solutions in the population are ranked with its non-domination level, where level 1 contains the best solutions, level 2 means the second-best, and so on. The algorithm applies a binary tournament selection, a crossover, and mutation operators that are similar to GA's. After generating a set of offspring, the new offspring and original population are combined. Now the population has twice the number of solutions. Then, the population will be sorted again, and the top N solutions are preserved to the next generation.

NSGA-II proposes two innovative operators: fast non-dominated sorting and crowding distance comparison-based diversity preservation. The fast non-dominated sorting has a complexity of $O(MN^2)$ where M is the number of objectives, N is the population size. Hence, the method can quickly sort the solutions and group them into different levels of the front. Crowding distance is a method to estimate the density of solutions surrounding a particular solution in the population [57]. The distance is calculated by averaging the distance of two solutions on either side of a solution along with each of the objectives (see Figure 2.7).

NSGA-II has been used in many real-world multi-objective problems

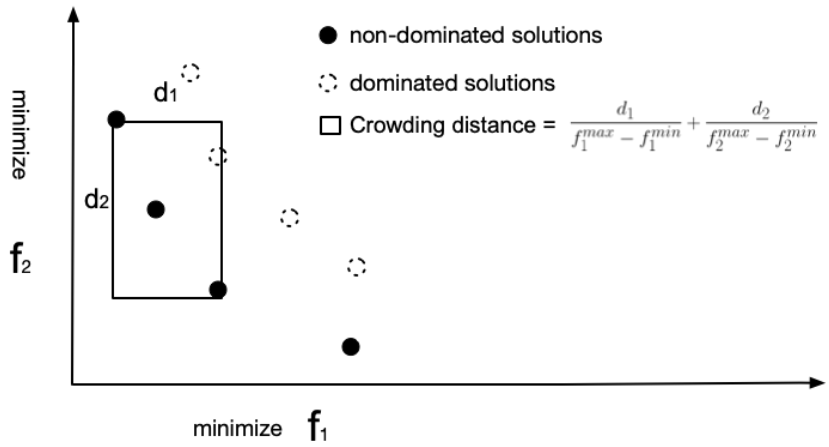


Figure 2.7: Crowding distance, adapted from [48], where f_1, and f_2 are two optimization objectives to be minimized.

in cloud computing, such as web service composition [217, 218], web service allocation [127], task scheduling in clouds [207], and resource allocation in clouds [171, 200, 248].

Multi-objective Performance Metrics

The performance of a multi-objective algorithm is evaluated by several performance metrics. These metrics consider mainly three aspects of a set of solutions, the convergence, the diversity, and the number of solutions. The convergence indicates the closeness between the solutions and the theoretical Pareto optimal front. Diversity denotes the distribution of solutions. Over fifty different methods [180] have been proposed to evaluate these performance metrics. Among assorted methods, we introduce the hyperVolume indicator [252] and IGD [214] because they are widely applied to the multi-objective studies and we will apply these two methods in this thesis.

HyperVolume indicator [181, 252] is a measure used in evolutionary multi-objective optimization. The indicator reflects the volume enclosed by a solution set and a reference point. The hyperVolume mainly measures

the convergence of the solutions. A larger HyperVolume value indicates a better solution set.

The IGD [214] is short for inverted generational distance [211] as a way of estimating how far the elements in the true Pareto front are from those in the non-dominated set produced by an algorithm. IGD calculates the sum of the distances from each point from the true Pareto front to the nearest point from the non-dominated set produced by an algorithm. In other words, IGD measures the coverage of the true Pareto front from the solutions from an algorithm. The lower the IGD, the better quality the solution is.

2.3 Related Work

Related work discusses the modeling of the *RAC* problem and resource allocation in the off-line, on-line, and multi-objective scenarios. Furthermore, we summarize the related works of using EC algorithms in the combinatorial optimization problems that are similar to our problems.

2.3.1 RAC Problem Models

We summarize the existing models of *RAC* problem from the perspectives of objectives, dimensions of resources, and constraints.

Existing studies for the *RAC* focusing on objectives either from *cloud providers* or *cloud users*. From the perspective of *cloud providers*. Their priority is minimizing the energy consumption of the used PM instances [117, 134, 166, 191] or improving the utilization of resources [62]. Guan et al. [83] and Zhang et al. [242] consider not only energy consumption but also the cost of the data exchange between containers. Fan et al. [62] focus on improving the utilization of PMs and load balancing between the VM instances in the same PM instance. From *cloud users'* perspective, minimizing the cost and maximizing the QoS are their concerns [85, 150].

Most of the existing studies [133,166,191] model the *RAC* problem as a vector bin packing problem [51]. They generally consider two dimensions of resources, i.e., CPU and memory. Another study [62] 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 [62, 133, 166, 191, 242] where the total resource requirement of containers cannot exceed the capacity of the VM instances and the total VM capacities on a PM cannot exceed the PM's capacity. The second one is that each container should only be allocated to one VM instance [62,191].

Currently, researchers have simplified the problem model with different assumptions. For example, Zhang et al. [245] study resource allocation for applications without considering application arrival and departure time. Zhang et al. [246] assume all the applications will be hosted for a period of time. Many researchers [128, 226] find that live migration introduces high overhead and downtime. Wolke et al. [226] suggest that allocation could be performed periodically and treated as an off-line problem that focuses on container placement and, therefore, does not consider migration overhead. Other researchers [166] 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.

Overall, current studies mainly ignored three characteristics in the model of *RAC* problem. The first limitation is that the current works do not consider VM overheads. As a result, small VM instances are often selected for containers. However, a large number of small VM instances leads to VM sprawl [186]. On the other hand, creating large VM instances leads to unused VM resources [133]. This trade-off is the core issue in the *VM creation* problem. Some researchers consider the existence of overheads [133,134], but they do not provide much analysis.

Secondly, they do not considered affinity constraints of *RAC*. Affinity

constraints define which containers can be co-allocated, e.g., security reasons, distinct OS requirements. Without the affinity constraint, all containers can be allocated directly to PM instances. Containers require distinct Operating Systems (OSs) and software libraries. Therefore, not all containers can be consolidated into a single VM [46]. Therefore, we need to consider the requirement of OSs as the affinity constraints in this work.

The third characteristic is that, for the on-line problem, the overall quality of allocation does not consider energy consumption over a period of time. Most of the existing studies evaluate allocations by measuring the temporal energy consumption at a certain time point [133]. However, the evaluation cannot represent the energy efficiency of a cloud because the energy consumption of a data center is determined by the overall energy consumption of a given period [55]. For example, Figure. 2.8 shows the curves of energy consumption from two methods. Although the 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 should consider the *accumulated* energy consumption as a quality measure.

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

2.3.2 Off-line Resource Allocation in Clouds

Off-line resource allocation, also called static resource allocation, allocates a set of applications in the planning stage. The optimization objectives of the off-line problems focus on energy consumption and QoS. We group related studies into four categories. Among these problems, the *RAC* problem is similar to the allocation of containers and VMs to PMs.

- Allocation of VMs to PMs

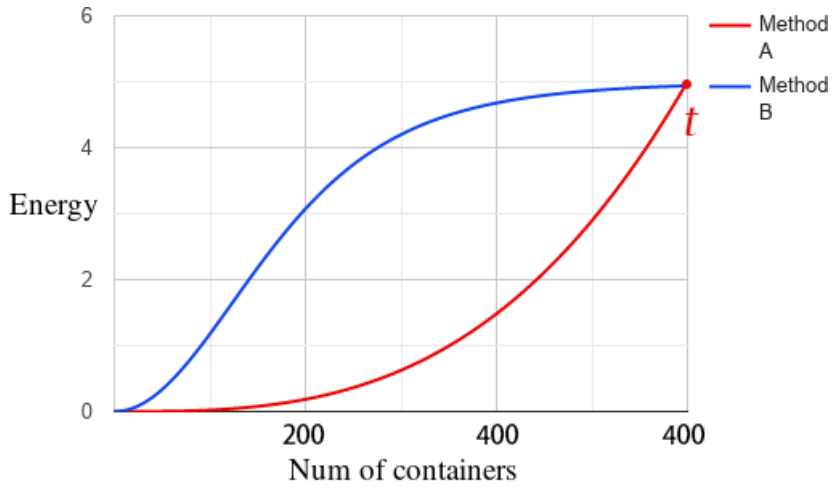


Figure 2.8: The energy consumption at time t are same for method A and B.

- Allocation of containers to PMs
- Allocation of containers to VMs
- Allocation of containers and VMs to PMs

Allocation of VMs to PMs

Existing literature has studied the VM allocation problem for over a decade. Many studies were conducted to investigate the allocation of VM problem [131, 188, 210]. We discuss some of the important work in this field.

Traditional approaches model the problem as variants of vector bin packing problems and apply heuristics and mathematical programming to solve them. For instance, studies [179, 197, 221] uses mathematical programming such as Integer Linear programming (ILP), or Mixed Integer Linear programming (MIP) to solve the VM allocation problem. Wang and Xia [221] develop a MIP algorithm for solving large-scale VM allocation problem under a *non-linear* power consumption model. They first use a linear function to approximate the cubical function. Then, they use the

Gruobi MIP solver to solve the relaxed linearized problem. Further, they apply an iterative rounding algorithm to obtain the near-optimal solution. Speitkamp et al. [197] apply an LP-relaxation-based heuristics and analyze the historical workload patterns. The main problem of the ILP-based approach is that the scale of the allocation problem is constrained because the ILP takes an infeasible time to find the optimal solution. Hence, the present works applied relaxation methods, which also do not guarantee to find the optimal solution.

Most of the works propose extensions of greedy-based heuristics such as First-Fit Decreasing (FFD) [228, 247], Best-Fit, Best-Fit Decreasing [26], etc. However, although greedy-based approaches but they cannot guarantee to find the optimal solution. Beloglazov et al. [26] consider the utilization of VM instances and the candidate PM instances. They propose a modified Best-Fit Decreasing to solve the problem. Zhang et al. [247] propose a heterogeneity-aware algorithm using FFD and its variations. Lin et al. [118] propose two heuristics, a round robin-based and a hybrid approach of round-robin and FF. All of the approaches mentioned above focus on the reduction of energy consumption of clouds. An obverse drawback of these heuristics is that they lead to a local optimal solution due to their greedy feature.

In order to avoid the long computation time and premature solution, meta-heuristics [225,234] are often applied to solve the VM allocation problems. Wilcox et al. [225] also propose a reordering GA approach. They use an indirect representation [176], which represents a packing solution as a sequence of items. In order to transform the sequence into a packing, they applied an ordering operator, which, in essence, is the First-Fit algorithm. This design naturally avoids an infeasible solution. Therefore, there is no need for constraint handling. Xiong and Xu [234] propose a PSO based approach to solve the problem. Their major contribution is using a total Euclidean distance δ to represent the distance between current resource utilization and the optimal resource utilization (see Eq. 2.1) where d is

the dimension of resources, u_j^i is the current resource utilization of j in a PM instance i, $ubest_i$ is the predefined optimal resource utilization (e.g. 70% CPU utilization). Another contribution is their representation used in PSO. They represent the allocation of each VM instance to a PM instance as a probability and let particles search through the indirect solution space.

$$\delta = \sum_{i=1}^{n} \sqrt{\sum_{j=1}^{d} (u_j^i - ubest_i)^2} \tag{2.1}$$

Allocation of Containers to PMs

Heuristics are proposed to solve the allocation of containers to PMs. Docker Swarm [143] and Google Kubernetes [2] have been widely used as the container allocation tools. Docker Swarm allocates containers to VMs with a round-robin algorithm [206] and a Spread algorithm [2]. Spicuglia et al. [198] propose an OptiCA framework that allocates containers to PM instances. Their goal is to optimize the performance of big-data applications without considering energy consumption. Raj et al. [178] minimize the energy consumption under SLA constraints. They propose simple heuristics that are based on First Fit Decreasing.

Different from the above methods, a distributed agent-based system [154] has been proposed to allocate workflow tasks in container-based clouds. The agents (containers) calculate their benefit functions and use different greedy-based strategies based on the geographic information. The system adjusts continuously and eventually reaches Nash Equilibrium when all agents stop switching their allocation.

Allocation of Containers to VMs

Different from the previous two categories, the allocation of containers to VM instances does not focus on reducing energy consumption. Instead, they mostly improve application response time and VM utilization.

Kozhirbayev et al. [110] try to reduce the number of used VM instances to reduce the cost. Zhou et al. [249] propose a framework for both on-line and off-line container allocation. Beaumont et al. . [24] consider variants of the allocation problems and analyze their complexities. These variants are grouped into different situations, one single container, several grouped containers, and several independent containers. They proposed a set of heuristics and compared their performance.

Allocation of Containers and VMs to PMs

Most researchers tend to develop ILP or propose heuristics to solve the problem. Guan et al. [83] consider one type of VM, and each PM instance is filled with ten VM instances. Then, they propose an ILP-based approach to allocate containers. However, only considering one type of VM is not only inflexible to meet different resource requirements but also leads to the waste of resources. Furthermore, allocating a large number of VM instances lead to huge overheads. Nardelli et al. [151] propose the Elastic provisioning of Virtual machines for Container Deployment (EVCD) to allocate containers to VM instances. They propose a problem model that aims at optimizing QoS attributes instead of energy consumption. They also apply an ILP to solve the problem and compared it with traditional approaches such as round-robin. In their experiment, they consider the allocation as a periodic allocation problem, and two types of VM are used. From the perspective of the above two approaches, it is known that, without any relaxation method, the computational time of ILP-based approaches grows exponentially with the increase of the problem size. Hence, their approaches cannot handle large scale problems.

Heuristics have been proposed to solve the *RAC* problem. Zhang et al. [245] propose to use traditional bin packing heuristics to solve the two-level allocation problem. They apply a BestFit approach to select VM instances for containers and select PM instances for new VM instances. For *VM creation*, the smallest type of VM should be selected when no VM in-

Table 2.1: Container allocation studies are categorized based on their approaches.

Allocation Scenario	ILP and MLP	Greedy-Heuristics	Meta-Heuristics
VMs to PMs	[179,197,221]	[26,118,228,247]	[225,234]
Containers to PMs and containers to VMs		[2,24,154,178,206,249]	
Containers and VMs to PMs	[83,151]	[133,166,245]	

stance is available.

In Mann's work [133], they consider the overheads of VM instances and model the overhead as a constant value of CPU utilization, but he stated that more sophisticated models could be more realistic. In order to prove the interaction of two-levels of allocation, Mann uses a fixed VM allocation algorithm and test a series of VM selection algorithms such as simple selection [74], Multiple selections, Maxsize, Consolidation-friendly. Mann discovers that the final energy consumption varies with the selection algorithms. Mann claims that the performance is better when VM selection has more knowledge of the PM instances' capacity. However, Mann's study only focuses on the partial allocation with fixed VM allocation algorithm. The answer to "How these two-levels of placement interact ?" is still undiscovered. Piraghaj et al. [166] develop a set of heuristics such as Random Host Selection, Least Full Host Selection, and Correlation Threshold Host Selection. These algorithms are also based on greedy heuristics.

2.3.3 On-line Resource Allocation in Clouds

For solving on-line allocation problems, most research applied rule-based approaches in order to make fast allocation decisions.

Allocation of VMs to PMs

Beloglazov et al. [25] apply an AnyFit-based framework [98] and consider the problem as a vector bin packing problem with multiple resources. These resources are combined into a single value in order to decide which PM instances are suitable for which VM instances [98]. [25] proposes *Energy-aware* BF to select PM instances with the least CPU usage. Wood et al. [228] propose a *volume* rule to allocate VM instances. They choose target PM instances with the least value of $volume = \frac{1}{1-cpu} * \frac{1}{1-mem}$.

Forsman et al. [70] propose two distributed migration strategies to balance the load in a system. *The push* strategy is applied to overloaded PM instances; it attempts to migrate *One* VM instance at a time to less loaded PM instances. *The pull* strategy is applied to underutilized PM instances to request workloads from heavier loaded PM instances. Each of the strategies is executed on each PM instance as an intelligent agent. These intelligent agents (e.g., PMs) share their status with each other through a communication protocol. Forsman's approach has several interesting features. First, they apply an adaptive high-load threshold (e.g., 0.7 of overall CPU utilization) so that it considers the environment changes. Second, they use an EWMA algorithm to reduce the unnecessary migration because EWMA [92] is useful in smoothing out variations in the average load. Third, they apply entropy to model the load distribution. The entropy method is also applied to some previous approaches [112, 174]. In this thesis, we design allocation algorithms for a centralized allocation system.

Xiao et al. [231] propose an algorithm based on evolutionary game theory. Their approach has two contributions. First, they build a quadratic energy model for the energy consumption of PM instances and a linear model for the energy consumption of migration. Second, they propose an algorithm based on Multiplayer random evolutionary game theory to solve the on-line problem. In their approach, VM instances are mapped into players that take part in the evolutionary game. In each iteration, all

players choose their best feasible action, i.e., players migrate to the PMs, which can minimize energy consumption. Some players randomly choose PM instances in order to avoid being stuck at a local optimum. Xiao compared their approach with giving simple bin-packing heuristics: First Fit, Best Fit Increasing, Best Fit Decreasing, Greedy, and Load Balance rule. The solutions show their approach can improve energy consumption significantly, especially in the scenario where the distributions of VMs are very centralized.

Allocation of Containers and VMs to PMs

Piraghaj et al. [166] propose a framework for container-based resource management, including three steps, analyzing resources to trigger migration, deciding which containers to migrate, and placing the container to a VM instance. In the third step, Piraghaj applies three heuristics: FF, Random, and Least Full. However, this work only reports that their approach can reduce the number of VM instances but does not mention how to reduce the number of PM instances by migrating VM instances. Therefore, this work does not consider the interactions between two levels: VMs and PMs.

Extending Piraghaj's work, Gholipour [79] develops a joint VM and container multi-criteria migration decision (JVCMMD) policy which consider the interactions between containers and VMs. JVCMMD policy includes 8 sub-policies for the entire migration process of the existing containers and VMs. The process determines which host is overload, which container and VM should be migrated, and the destination of container and VM migration. Since, no new container is coming and container are keep releasing as they finished, VM creation rule is not considered.

Unlike previously introduced reactive approach, Liu et al [122] applies linear regression to make prediction of PMs' status and applies rule-based algorithms to allocate containers. The experiment results show that rule-based algorithms with prediction can improve the energy efficiency than

the reactive rules.

For the *RAC* problem, most of the research employs AnyFit-based algorithms such as BF and FF. AnyFit-based algorithms always select existing VM instances until no VM instance is available. Then, they apply a simple heuristic such as a *Just-Fit* [133] or *Largest* [133] to create VM instances. These simple heuristics may not lead to the optimal allocation at the end because they either create a large number of small VM instances, which wastes the resources on VM overheads or create large but low-utilized VM instances.

2.3.4 Multi-objective Resource Allocation in Clouds

Studies [84, 94, 119, 121, 185, 220] discuss the multi-objective problems and optimize objectives such as energy consumption, communication cost between containers, and availability of applications. However, these multi-objective approaches could only be applied to OS-container architecture, where containers are allocated to PM instances directly. Most of these works consider the relationship between containers. Both [94, 121] consider the data transmission. Guerrero et al. [84] consider the reliability of micro-service allocation. Besides energy consumption, various objectives are considered. [84, 94, 119] consider the load balancing of PM instances. Network overheads [84, 119] or transmission [121] have also frequently been used as the optimization objectives.

Heuristics are used in these studies. Liu et al. [121] use a weight-sum function to measure the score of each node and select the PM instance with the highest score to allocate a container. Hu et al. [94] develop a heuristics with two key ideas *Resource Utilization Threshold* and *Dot-Product* heuristic. They adjust the distribution of containers on the PM instances based on the threshold of utilization. The *Dot-Product* heuristic measures the similarity of containers and PM instances in terms of resource demand and resource capacity. Containers are allocated to the most suitable PM instances.

EC methods have been dominating in multi-objective resource allocation. Studies [63, 67, 77, 84, 100, 119, 235] focus on the multi-objective VM allocation problems. Guerrero et al. [84] propose an NSGA-II-based approach to solve the problem, and it outperforms the approaches used in Kubernetes. [67, 77, 100, 119] propose ant colony algorithm-based approaches. Both Gao et al. [77] and Ferdaus et al. [67] use a vector algebra complimentary resource utilization model proposed by Mishra [146]. They consider three resources CPU, memory, and network I/O with two objectives: minimizing power consumption and resource wastage. They apply the *Resource Imbalance Vector* to capture the imbalance among three resources. Meanwhile, they use a linear energy consumption function to capture the relationship between CPU utilization and energy [63]. Their solution is compared with four algorithms: Max-Min Ant System, a greedy-based approach, and two FFD-based methods. The results show that their proposed algorithm has much less resource wastage than other algorithms.

Xu and Fortes [235] propose a multi-objective VM allocation approach with three objectives: minimizing total resource wastage, power consumption, and thermal dissipation costs. They apply an improved grouping genetic algorithm (GGA) with fuzzy multi-objective evaluation. The wastage is calculated as differences between the smallest normalized residual resource and the others. They also applied a linear power model to estimate the power consumption [116]. They conduct experiments on synthetic data and compare it with six traditional approaches, including FFD, BFD, and single-objective grouping GA. The results showed superior performance than other approaches.

2.3.5 EC Algorithms in Combinatorial Optimization

This section reviews a number of EC algorithms in two combinatorial optimization fields: cloud computing scheduling and job shop scheduling (JSS). We review these two fields because of two reasons. Firstly, though,

these problems have some fundamental differences from the *RAC* problem but share some similarities. These problems deal with task allocation in both off-line and on-line scenarios. For example, JSS has many on-line variations, such as the dynamic JSS problem [223]. Secondly, the EC algorithms that have been applied to these problems use some techniques, e.g., permutation-based crossover operator, which can potentially inspire us to design problem-specific operators for solving our problem.

Cloud Computing Resource Allocation

Traditional cloud computing resource allocation includes four categories [87]: cloud brokering and service placement are typically off-line problems; server load balancing and cloud capacity planning are on-line problems. EC algorithms have been applied to problems in each of these categories. This section will briefly review approaches.

In cloud brokering problem, a *cloud broker* is an intermediary between *cloud users* and *cloud providers* to estimate the resource requirements from *cloud users* and choose resources from *cloud providers*. The objective of cloud brokering is to minimize the cost for *cloud users* as well as guarantee the quality of service (QoS) of *cloud users'* applications.

Frey et al. [72] propose a GA approach (CDOXploer) for finding near-optimal cloud deployment architectures and runtime reconfiguration rules for software. Their approach takes into account VM types, the number of VM instances, and the scaling policies of VM instances. CDOXplorer minimizes response times, costs, and SLA violations in order to satisfy the requirement of *cloud users*. In comparison with Frey's centralized approach, Iturriaga et al. [95] propose an Evolutionary Algorithm (EA) with distributed sub-populations. In each sub-population, they applied a Simulated Annealing (SA) method to find the local optimal solution. It is shown that the proposed algorithm outperforms the reference list scheduling algorithms in both computation time and performance (maximizing the profit of a broker).

In the service placement problem, *cloud users* aim to optimize the costs and performance (e.g., QoS) of their services. Compared with the cloud brokering problem, service placement focuses less on resource allocation but focuses more on the location selection. Cloud users need to decide the locations of services and the number of services [87].

Tan et al. [202] propose an aggregation approach with binary PSO to solve the service placement problem with two objectives: minimizing cost and minimizing the response time. They find that the single-objective algorithm can only provide one solution for each run. This single-objective algorithm is suitable to use when *cloud users* have preferences of different QoS. Therefore, they develop an NSGA-II-based approach [205] that uses Pareto front approach to find a set of non-dominated solutions. They conclude that multi-objective evolutionary algorithms are suitable for the service placement problem.

Capacity planning problems estimate the future load of VM instances and PM instances before allocating resources. The primary objectives are to optimize the QoS while minimizing the cost of users. Different from the above problems, the capacity planning problem is often treated as an on-line problem. Kousiouris et al. [107] propose an artificial neural (ANN) network-based framework to predict the load of a GNU Octave system. They use a GA to create the structure of ANN, and the ANN is encoded using a bit-string representation. The algorithm can be trained off-line and test on-line. Therefore, it is well-suit for an on-line problem.

Job Shop Scheduling

Job shop scheduling (JSS) problems [172] dispatch a set of jobs to machines. A job goes through a predetermined sequence of operations in order to finish its tasks. Each machine can only process a specific operation. Therefore, it needs to make intelligent decisions to schedule these jobs in order to finish processing jobs before their due dates.

JSS problems can also be divided into on-line (dynamic) and off-line

(static). In off-line JSS problems, all properties of jobs and machines are known in advance. In on-line JSS problems, properties of jobs are unknown. This section focuses on the on-line JSS problems because they are similar to our on-line *RAC* problem.

On-line JSS problems often apply small heuristics such as dispatching rules [33] because dispatching rules have short reaction times and can quickly deal with unforeseen changes in an on-line event. For example, an SPT (shortest processing time) selects the job with the shortest processing time waiting at available machines. Apart from simple dispatching rules, composite dispatching rules (CDRs) [96] combine several simple dispatching rules to achieve higher performance. It is difficult to design dispatching rules for on-line JSS problems because, first, no single dispatching rule is more effective than others for all JSS problem instances. Second, a real-world dynamic JSS problem is changing over time, e.g., machines are added and removed. Therefore, designing dispatching rules often requires domain knowledge.

To design dispatching rules more effectively, researchers use a hyper-heuristic technique to generate dispatching rules automatically. Specifically, a great number of hyper-heuristic approaches to the JSS problem uses GPHH as previously introduced. GPHH represents dispatching rules with a tree-based representation, and these dispatching rules can be interpreted as priority-based dispatching rules. GPHH approaches generally outperform manually designed dispatching rules for both off-line and on-line JSS problems [33].

The on-line JSS problem and the on-line *RAC* problem share many similarities. First of all, they are both on-line problems that require a fast decision. Second, they both dispatch jobs to PM instances. The differences are that the on-line JSS problem usually does not care about the resources requirement of a job. The on-line *RAC* problem considers not only the resource requirement of the containers but also the remaining resources in PM instances. Another difference is that the jobs in on-line JSS go through

a route of PM instances while the containers in on-line *RAC* problem stay at PM instances. The similarities inspire us to apply the GPHH approach to the on-line *RAC* problem.

In conclusion, EC-based algorithms have been widely used in resource allocation problems in clouds for both off-line and on-line scenarios. Specifically, meta-heuristics, such as GA, are more suitable for off-line scenarios, and hyper-heuristics, such as GPHH, are suitable for on-line scenarios.

2.4 Summary and Thesis Scope

This chapter introduced the main concepts of cloud resource allocation, Evolutionary Computation (EC), and reviewed the recent studies on resource allocation in clouds. We discuss the limitations of existing work on two allocation decision scenarios.

We summarize the drawbacks of current studies and challenges on the *RAC* problem from four perspectives. **Firstly**, the current models of the *RAC* lack some critical features. Current *RAC* models are mostly extended from previously VM-based clouds. Hence, the VM-related features, such as VM overhead, affinity constraints, are not included. These features are critical because they heavily affect decision making. Therefore, a new model that includes these features needs to be developed. **Secondly**, the *RAC* problem is a new and difficult (NP-hard). Hence, EC is the most promising technique for the off-line scenario. The difficulties for developing EC methods are the design of problem representation and genetic operators. Currently, two types of representation, vector-based and group-based, are potentially suitable for the *RAC* problem. Their effectiveness needs to be discovered. **Thirdly**, manually designed heuristics are widely used for the on-line *RAC* problem. However, manually designed heuristics cannot consider the complex interaction of features. Hence, these heuristics are often simple. A related field of Job Shop Scheduling (JSS) provides a promising technique of GPHH. GPHH is a hyper-heuristic

technique that can automatically generate heuristics based on the features and historical workload datasets. However, the employment of GPHH on *RAC* is difficult because the *RAC* is a two-level allocation problem. **Fourthly**, the multi-objective *RAC* also requires the development of multi-objective EC methods. Hence, we introduce the thesis scope.

Table 2.2: The thesis scope

Characteristic	Thesis scope
Virtualization	Containerization
System resources	CPU and Memory
Service Model	Platform-as-a-Service, Container-as-a-Service
Cloud Architecture	Container-based clouds
Resource Management Unit	Heterogeneous Container, VMs, and homogeneous PMs
Goal	Minimize energy consumption and other objectives
Workload type	Web service
Allocation Scenarios	Off-line, on-line, and multi-objective scenarios
Methods	Evolutionary Algorithms

This thesis (see Table 2.2) investigates EC approaches to help *cloud providers* improving the performance of resource allocation in container-based clouds problems. The thesis considers heterogeneous containers, VMs, and homogeneous PMs as the resource units. In addition, CPU and memory are being considered. The major objective is to minimize the energy consumption of data centers. Other objectives, such as availability, are also considered. This thesis focuses on long-term workload types such as web service. Evolutionary algorithms are proposed in order to solve these combinatorial optimization problems in three main allocation scenarios, i.e., off-line single-objective, on-line single-objective, and off-line multi-objective problems.

Chapter 3

Genetic Algorithms for Off-line Resource Allocation in Container-based Clouds (RAC)

3.1 Introduction

The purpose of this chapter is to design an effective vector-based and group-based representations of GA for solving the off-line *RAC* problem. As discussed in Chapter 2.2.1, vector-based [105,164,177] and group-based representations [82, 99, 123, 235] for various resource allocation problems have been proposed in the literature. Moreover, each representation has its advantages. However, the exising representations cannot be directly used in the *RAC* problem because they have the structure for one-level allocation. In order to design effective GA-based approaches for the off-line *RAC* problem, this chapter proposes both vector-based and group-based GAs and compare them with existing approaches.

In order to develop GA-based approaches for the *RAC* problem, multiple objectives need to be done. We first propose a model for the *RAC* problem. Such a model is used to evaluate the performance of algorithms for solving the problem. Then, we develop problem-specific operators for

each representation. Finally, we evaluate the effectiveness of the proposed approaches and compare to the state-of-the-art algorithms with real-world datasets.

1. To propose an off-line model for the *RAC* problem. The new problem model represents the two-level allocation problem and includes new features such as VM overheads and new constraints.
2. To propose new vector-based GA approaches for the *RAC* problem, including initialization strategy, decoding methods, genetic operators, and constraint handling methods.
3. To propose a new group-based GA approach for the *RAC* problem, including initialization strategy, genetic operators, and constraint handling methods.
4. To evaluate the performance of these proposed approaches in order to identify their advantages and limitations.

3.2 Chapter Organization

The remainder of this chapter is organized as follows. Section 3.3 presents the off-line *RAC* model used in this chapter. Section 3.4 illustrates the off-line allocation process and our assumptions. Then, Section 3.5 introduces the vector-based GA approaches. The first approach, a single-chromosome GA (*SGA*) is described in Section 3.5.1. In Section 3.5.2, we introduce another vector-based GA, a dual-chromosome GA (*DGA*). Section 3.6 presents a distinct representation of group-based GA (*GGA-RAC*). To evaluate the proposed algorithms, we conduct a series of experiments in Section 3.7 and illustrate the pros and cons of the proposed algorithms. Section 3.9 summarizes this chapter.

3.3 Off-line *RAC* Model

The off-line *RAC* problem is a task of allocating a set of containers to a set of VM instances with various types, then allocate the created VM instances to a set of PM instances (see Figure 3.1). The allocation process involves four decision-making processes. *VM selection* chooses an existing VM instance to allocate a container. In the meanwhile, the data center can also create a new VM instance to allocate the container. *VM creation* selects a type of VM, and creates a VM instance with the selected type. Then it allocates the container to the new VM instance. Cloud providers define the types of VM. Likewise, *PM selection* chooses an existing PM instance to allocate the new VM instance. *PM creation* is used to select types of PM if the PMs are also heterogeneous, e.g., there are different types of PM to be used. In this research, we consider homogeneous PM, which means there is one type of PM; therefore, no need to decide types for *PM creation*.

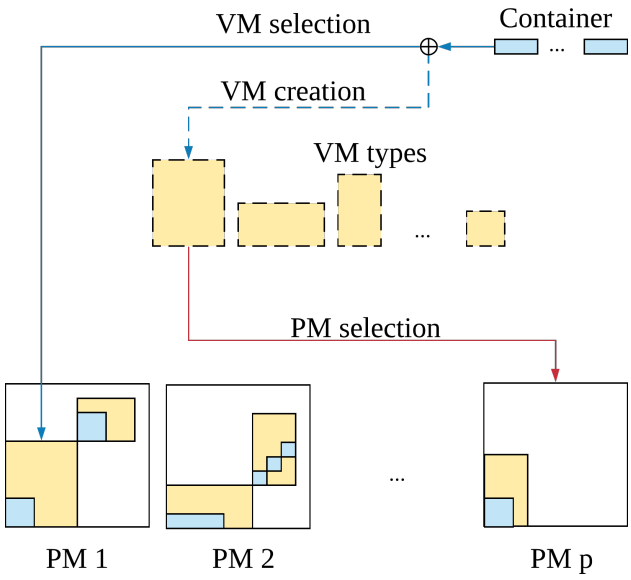


Figure 3.1: An illustration of the *RAC* problem.

We now define a formal model for the off-line *RAC* problem. Assume a set of containers $\mathcal{C} = \{c_1, \ldots, c_n\}$ arrives to the cloud to be allocated.

Table 3.1: Notation and description of the problem model

Notation	Description
c_i	a container of index i
τ_j	the VM type of a VM instance j
ψ_i	the OS type of a container i
p_k	a PM instance of index k
x_{il}	An indicator of whether the container i is allocated to the l VM instance
y_{lk}	An indicator of whether the l th VM instance is allocated to the kth PM instance
z_{jl}	An indicator of whether the l th VM instance 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 instance at time t
EP_k^{idle}, EP_k^{full}	The energy consumption when the kth PM instance 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
$\mu_{tk}^{cpu}, \mu_{tk}^{mem}$	The CPU and memory utilization of a the kth PM instance at time t

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. A set of OS types $\Psi = \{\psi_1, \ldots, \psi_o\}$ that can be required by the containers. 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 $\Omega^{mem}(\tau_j)$. Also, it has a CPU overhead $\pi^{cpu}(\tau_j)$ and memory overhead $\pi^{mem}(\tau_j)$, indicating the CPU and memory occupation for running a new VM instance of that type. There is an unlimited set of PM instances $\mathcal{P} = \{p_1, \ldots, \}$ for allocating the created VM instances. Each PM instance p_k has a CPU capacity $\Omega^{cpu}(p_k)$ and a memory capacity $\Omega^{mem}(p_k)$.

The off-line container allocation problem is subject to the following constraints:

1. Each container is allocated to one VM instance.
2. Each created VM instance is allocated to one PM instance.
3. For each created VM instance, the total CPU and memory occupations of the containers allocated to that VM instance does not exceed

the VM instance's capacity.

4. For each PM instance, the sum of the CPU and memory capacities of the VM instances allocated on the PM instance does not exceed the PM instance's capacity.
5. For each container, it must be allocated to a VM instance which has installed the same OS.

The energy consumption of all the PM instances is calculated as follows:

$$E = \sum_{k=1}^{K} E_k, \tag{3.1}$$

where E_k is the energy consumption of the kth PM instance (K is the number of PM instance used).

E_k is calculated as follows:

$$E_k = E_k^{idle} + (E_k^{full} - E_k^{idle}) \cdot \mu_k^{cpu}, \tag{3.2}$$

where E_k^{idle} and E_k^{full} indicate the energy consumption of the kth PM instance per time unit when it is idle and fully loaded, respectively. The energy model is proposed by Fan [63] and has been widely used in research of VM allocation [54].

μ_k^{cpu} indicates the CPU utilization level of the kth PM instance. μ_k^{cpu} is calculated as follows.

$$\mu_k^{cpu} = \frac{\sum_{l=1}^{L} \left(\sum_{j=1}^{m} \pi^{cpu}(\tau_j) \cdot z_{jl} + \sum_{i=1}^{n} \Omega^{cpu}(c_i) \cdot x_{il} \right) \cdot y_{lk}}{\Omega^{cpu}(p_k)}, \tag{3.3}$$

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

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

$$\min \sum_{k=1}^{K} E_k, \tag{3.4}$$

$$s.t. \sum_{l=1}^{L} x_{il} = 1, \ \forall\, i = 1, \ldots, n, \tag{3.5}$$

$$\sum_{k=1}^{K} y_{lk} = 1, \ \forall\, l = 1, \ldots, L, \tag{3.6}$$

$$\sum_{j=1}^{m} z_{jl} = 1, , \ \forall\, l = 1, \ldots, L, \tag{3.7}$$

$$\sum_{i=1}^{n} \zeta^{res}(c_i) x_{il} \leq \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl}, \quad \forall\, l = 1, \ldots, L, \ res \in \{cpu, mem\}, \tag{3.8}$$

$$\sum_{l=1}^{L} \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl} \leq \Omega^{res}(p_k), \quad \forall\, k = 1, \ldots, K, \ res \in \{cpu, mem\}, \tag{3.9}$$

$$OS(c_i) = OS(c_j), \ \forall l = 1, \cdots, L. \ \sum_{l=1}^{L} x_{il} x_{jl} = 1, \tag{3.10}$$

$$x_{il}, y_{lk}, z_{jl} \in \{0, 1\}, \tag{3.11}$$

where constraints (3.5) and (3.6) indicate that each container (or the created VM instance) is allocated to exactly one created VM instance (or a PM instance). Constraint (3.7) indicates that each created VM instance must belong to a VM type. Constraint (3.8) implies that the total occupation of all the containers allocated to each created VM instance does not exceed the capacity of the VM instance. Constraint (3.9) indicates that the total capacity of the created VM instances allocated to each PM instance does not exceed its corresponding capacity. Constraint (3.10) means that the containers allocated to the same VM instance must have the same required OS, which is the installed OS on that VM instance. Constraint (3.11) defines the domain of the decision variables.

The next section explains the allocation process and our assumptions in detail.

3.4 The Off-line RAC Process and Assumptions

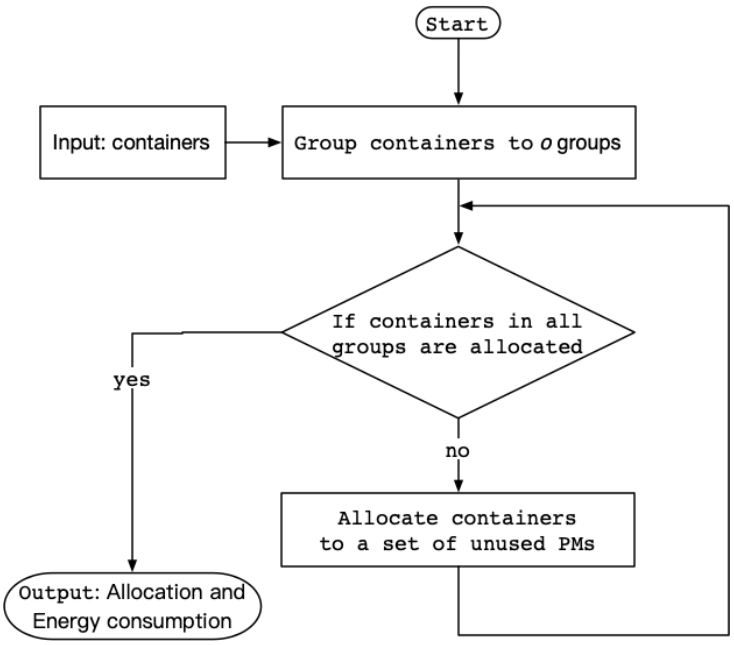


Figure 3.2: The flowchart of the off-line *RAC* process.

This section explains the allocation process of the off-line *RAC* process (see the flowchart in Figure 3.2) and our assumptions. In the beginning, we first apply a preprocessing technique that groups the incoming containers into o groups, where o is the total number of OS types. Then, we allocate each group of containers into a set of unused PM instances using an off-line allocation algorithm. Finally, we evaluate the allocation by computing the energy consumption of used PM instances.

The preprocessing of grouping the applications according to their OS requirements is a common procedure in the industry. The primary reason is that traditional clouds allocate containers into a cluster of bare-metal

PM instances (No virtualization). It is straightforward to map groups of containers into clusters of PM instances.

This research also adopts the preprocessing for three reasons. Firstly, since the off-line *RAC* allocates a large number of containers at each time. A large number of containers in different OS groups can be combined to achieve high utilization in PM instances. Secondly, OS constraint is a hard constraint, which must be satisfied by allocation solutions. There is not much benefit to allocate a mixed set of containers with different OS requirements. Thirdly, the computational complexity of an allocation algorithm increases dramatically with considering an additional variable of OS. Based on these reasons, we also adopt the preprocessing technique.

The grouping procedure of containers is shown in Algorithm 1. For each container c_i, find its OS requirement ψ_o and allocate it to the corresponding groups.

Algorithm 1: Group containers according to their OS requirements

Input : a set of containers
Output: j groups of containers

```
for each container c_i do
    for each OS types ψ_o do
        if OS(c_i) = ψ_o then
            assign the container to group o;
        end
    end
end
```

The next section explains the proposed vector-based GAs for the off-line *RAC* problem.

3.5 Vector-based GA

GA [215] has been successfully applied to various combinatorial optimization problems over its fifty-years of history [81]. Not only does GA overcome the shortcoming of greedy-based heuristics (e.g., First-Fit) that are easily stuck at local optimum but also GA has a controllable computational time. However, the key challenge of applying GA to solve the problem is the design of the representation of solutions and genetic operators that can evolve solutions [130]. A good representation narrows the search space, and good operators can accelerate the searching for near-optimal solutions.

This section investigates the effectiveness of vector-based GA by developing two distinct vector-based representations and corresponding operators.

3.5.1 Single-Chromosome GA (SGA) Approach

This section introduces the design of the first vector-based GA approach, which includes representation, genetic operators, the fitness function, and the procedure of the algorithm. In the following content, we will use *SGA* as the short name of single-chromosome GA.

Single-Chromosome Representation

In this approach, we use a single vector to represent a solution of *RAC* problem called a single-chromosome representation (to distinguish from the dual-chromosome representation in Section 3.5.2). The representation shown in Figure 3.3 includes a vector of integers. An individual is divided into cells (separated with dotted lines). Each cell denotes the allocation of a container to a VM, e.g. the index and the type of the selected VM. The second level of allocation, VM instances to PM instances, is decided by a decoding process.

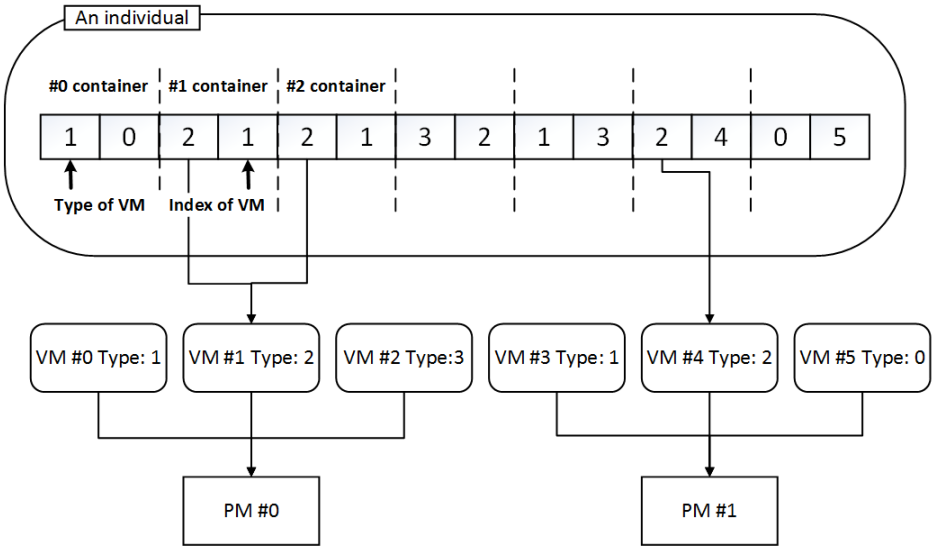


Figure 3.3: An example of *SGA* representation of *RAC* solution

The representation consists of several paired cells (separated with dotted lines). Each pair of cells represents the allocation of a container. The order of containers follows the same order in the given *RAC* task. The integer in the first cell represents the type of VM instance, and the second integer denotes the index of the VM instance. The type of the VM instance is selected from a given set of VM types. The index of a VM instance has no particular order. However, the numbers of indexed must be consecutive. For example, it is invalid that, in an individual, containers are allocated to #0, #1, #3 VMs. Since #2 should be allocated before #3 VM instance is created. In Figure 3.3, the example allocates seven containers. Container #0 is allocated to VM instance #0 with type 1. Container #1 and #2 are both allocated to VM instance #1 with type 2.

For the second-level of allocation, VM instances to PM instances, a decoding process is applied. The process decodes an individual starting from the VM instance with #0 index. It finds the VM instance of the next index and allocates VM instances one by one to a PM instance until the PM instance cannot host the current VM instance. Then, a new PM instance is

created, and the remaining VM instances are allocated to the new PM instance. Essentially, this decoding process uses a Next-Fit (NF) heuristic.

Clearly, specifically designed operators are needed to manipulate chromosomes. Therefore, based on this representation, we further developed initialization and mutation methods.

Initialization

The initialization (see Algorithm 2) is designed to generate a diverse population. For each individual, it first generates a vector with a length of twice the number of containers (*line 2*). Then, for each pair of the entries (the allocation for each container), it follows several steps (from *line 4* to *line 17*). It first tries to find an existing VM that can host c_i. If there exists a VM instance that can host the container, then allocate the container by assigning the type of the VM instance to the first entry (VM type) and assigning the index of the VM instance to the second entry (VM index). If no existing VM instance can host the container, a new VM instance is created.

From *line 10* to *line 14*, a new VM instance is created. First, it iterates the VM type list and finds the first VM type τ_f that can host the container. Then, it randomly generates a VM type τ_k that has more or equal capacity than τ_f. Finally, it allocates the container to the new VM instance and updates the VM counter.

Mutation

We design a mutation operator (see Algorithm 3) to manipulate the types of VM. The mutation operator has two functions. First function re-allocates containers to existing VMs from *line 13* to *line 15*. The second function re-allocates containers to new VM instances from *line 6* to *line 12*. The steps of the creation of a new VM instance is similar to the steps in the initialization.

Since a container can be randomly allocated to an existing VM instance,

Algorithm 2: Initialization

Input : A set of types of VM τ, A set of containers C,
Output: A population of individuals P

```
for each individual p do
    Generate a vector with the length of |C| × 2;
    vmCounter ← 0;
    for Each container c_i do
        if an existing VM v_j that can host the container c_i then
            First entry of the i pair ← τ_{v_j};
            Second entry of the i pair ← j;
        end
        else
            Find the first VM type τ_f that has enough resource to
             host c_i;
            Randomly generate a type of VM τ_k that has more or
             equal capacity than τ_f ;
            First entry of the i pair ← τ_k;
            Second entry of the i pair ← vmCounter;
            vmCounter ← vmCounter + 1;
        end
        P ← p;
    end
end
return a population P;
```

the mutation operator can lead to two types of invalid solution. The first invalid type of solutions contains a non-consecutive index of VM instance. That means the indexes of VM instances are not continuous. For example, VM instances with indexes, #0, #1, #3, but without #2. The second type of invalid solution is VM overloading, which means one or more VM in-

stances host more containers than their resource capacities. At this stage, a repair operator is needed to solve the above issues.

Algorithm 3: Mutation operator

Input : An individual p, a set of types of VM τ, a set of containers C, a set of existing VM instances V, a mutation rate β,

Output: An individual p

```
for each container c_i do
    Randomly generate a number u from [0, 1];
    if u < β then
        vmCounter ← count(V);
        Randomly select j from [0, vmCounter];
        if j = vmCounter then
            find the first type of VM τ_f that can host the container;
            Randomly generate a type τ_k with ζ^cpu(τ_k) >= ζ^cpu(τ_f)
             and ζ^mem(τ_k) >= ζ^mem(τ_f);
            First entry of the i pair ← τ_k;
            Second entry of the i pair ← vmCounter;
            vmCounter ← vmCounter + 1;
        end
        else
            First entry of the i pair ← τ_{v_j};
            Second entry of the i pair ← j;
        end
        Repair(p);
    end
end
return the individual p;
```

Repair Operator

The repair operator (see Algorithm 4) has two functions to fix violations in individuals after mutation. The first function fixes the issue of non-consecutive VM indexes. The repair operator re-assigns indexes to the individual according to each group of containers that are allocated to the same VM instance. To achieve this, the repair operator first constructs lists of VM instances. The lists include VM instances and their containers (*line 2* to *line 5*). Then, it re-assigns VM indexes to the individual (*line 10*) if the VM instance has enough resources to host the container.

In the meanwhile, the repair operator also fixes the overloading of VM instances. During the re-assigning of VM indexes, the operator also checks whether a required VM instance has enough resources to host the container (*line 8*). If a VM instance cannot host the container, the container is added to a $leftContainerList$. Later on, from *line 17* to *line 19*, these containers are re-allocated with a rule *FF&C/FF*. The rule attempts to allocate a container with First-Fit (FF). If no existing VM instance is available, it creates a VM instance with its required VM type (as defined in the first entry of container i in the vector) and allocates the new VM instance to PM instances with *FF*. Thus, the individual remains a valid solution.

Fitness Function

We use Eq 3.12 as the fitness function to evaluate individuals of a population in terms of the overall energy consumption of all used PM instances. Lower fitness indicates a better individual.

$$Fitness\ E = \sum_{k=1}^{K} E_k \tag{3.12}$$

Algorithm 4: Repair operator

Input : An individual p,
Output: The repaired individual p

```
leftContainerList ← null;
for each container c_i do
    j ← the VM index of c_i;
    containerList_j ← c_i;
end
for each VM j do
    for each container c_i in the containerList_j do
        if VM_j can host container c_i then
            allocate c_i to VM_j;
            Second entry of container i ← j;
        end
        else
            leftContainerList ← c_i;
        end
    end
end
for each container c_i in leftContainerList do
    allocate container c_i with FF&C/FF;
end
return the repaired individual p;
```

SGA

In previous sections, we introduce genetic operators and fitness function. They are the components used in Algorithm 5. The GA that we applied follows the standard GA procedure without a crossover operator. The crossover operator is difficult to preserve the information of the parents.

Considering the following two examples. In the first example, a con-

Algorithm 5: SGA for the *RAC* problem

Input : a set of containers, a set of VM types, a list of PM instances,

Output: an allocation of containers

```
1  population ← Initialization;
2  gen ← 0 ;
3  for gen does not reach the maximum generation do
4      fitness evaluation(population);
5      new population ← elitism(population);
6      while have not filled the new population do
7          children ← tournament selection(population);
8          mutation(children);
9          add children to the new population
10     end
11     gen ← gen + 1;
12 end
13 return an allocation of containers;
```

tainer is allocated to VM #2 with type 2 in parent 1. The same container is allocated to VM #2 with type 3. Although in both parents, the container is allocated to VM #2, the VM instances are not of the same type. In the second example, a container is allocated to VM #2 with type 2 in both parents. However, the container allocation in two VM #2 may be different. Hence, in the single-chromosome GA, we only applied the mutation operator. Hence, the *SGA* approach does not use a crossover operator.

3.5.2 Dual-Chromosome GA (DGA) Approach

This section introduces the design of the second vector-based approach, which includes the representation, genetic operators, and the fitness function. Since the overall procedure of *dual-chromosome GA (DGA)* is the same

with *SGA*, the procedure is not repeated.

Although the *SGA* solves the two-level *RAC* problem, it has disadvantages in its representation. The most critical flaw is that the single-chromosome representation heavily relies on the original sequence of containers provided by the input data. Additionally, the GA cannot use a crossover operator. Thus, this representation lacks flexibility and narrows the search space. The GA cannot provide a strong ability to search the solution space without a crossover operator. These disadvantages motivate us to develop another vector-based representation to solve the problem.

Representation

Unlike the representation of *SGA* which mixes the information of allocations from two-levels, the representation of the *DGA* consists of two vectors to separate the allocation decisions for two levels. An example of the representation is shown in Figure. 3.4. The first vector is designed for allocating containers to VMs which includes the *VM selection* and *VM creation* and the other vector is designed for allocating VMs to PMs that includes the *PM selection* and *PM creation*.

Both chromosomes are vectors of integer values. Specifically, in the vector of container allocation, each value represents the index of a container in the original input. The length of the chromosome is the total number of containers. In the chromosome of VM allocation, each entry represents the type of a VM with the value taken from the list of VM types. The length of the VM allocation vector equals the length of the container allocation vector. This is because we may use at most N VMs for allocating N containers (one-to-one mapping).

Similar to the *SGA*, to obtain a complete solution, a decoding process is needed. Bin packing heuristics can be used as the decoding process. Next-Fit (NF) heuristic is one of the most commonly used heuristics for the bin packing problem [51], and it can be used in both levels of allocation. The NF heuristic allocates items to bins sequentially until a bin cannot host the

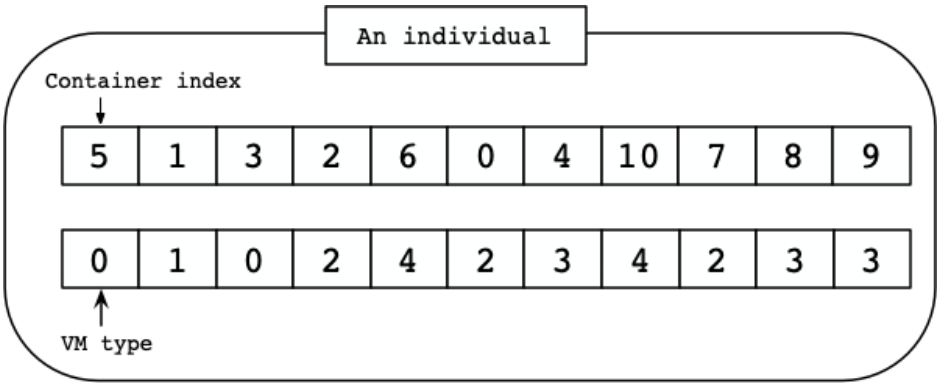


Figure 3.4: Representation

next item. Then, the bin is closed, and it will never be revisited. A new bin is used to allocate the next item. This process continues until all the items are allocated.

We show an example of applying NF on the individual in Figure 3.4. After container #5 is allocated to VM #0, if the following container #1 cannot fit into VM #0, we close VM #0 and open VM #1 to accommodate container #1. The closed VMs are never checked again later on. This decoding ends when all containers are allocated. Similarly, VMs are packed into PMs using the same rule. A decoded solution includes both levels of allocation as well as the types of VM.

Another bin-packing heuristic, First-Fit (FF) heuristic [51], can also be used as a decoding process. Similar to NF, FF also allocates items sequentially. The major difference between FF and NF is that FF never closes a bin. That is, for every item, FF always checks from the oldest bin to the latest. Although FF has a higher computational complexity than NF ($O(N)$ vs. $O(1)$), FF also obtains better performance than NF [51] because the bins are better utilized. In this work, we apply both NF and FF as the decoding procedures and, therefore, two variations of *DGA* are created. The variation with NF decoding is named *DGA-NF*, and the other variation is named *DGA-FF*. These two variations are the same for other parts of the algorithm, including the overall procedure and genetic operators.

Initialization

Two functions are designed for the initialization. The first function randomly shuffles the indexes of containers to generate a container allocation vector. The second function uniformly generates the types of VM. Both functions use the uniform distribution to generate a diverse set of solutions.

Crossover and Mutation

The design of crossover aims at retaining the "good" genes from the parents. For the vector of container allocation, the definition of "good" is the permutation, which leads to high utilization of VMs' resources. We apply the **order 1** crossover [170] to pass the useful permutation to the next generation. For the VM allocation vector, we apply the **single-point** crossover [15].

Order 1 crossover randomly selects a sequence of consecutive entries from one parent and copies them to the child. The remaining values in the child are placed with the same order in the other parent. For example, in Figure 3.5, containers 3 and 4 are selected and copied from parent to child 1. Then, the containers (3 and 4) which have been allocated are crossed out from parent 2. The rest values in the child are copied from parent 2, starting from the second cut point and rolling back to the head, e.g., containers 1, 5, and 2. The same rules are applied to generate a second child.

Single-point crossover first randomly cuts a vector into two parts. A child inherits one part from parent 1 and the other part from parent 2 (see Figure 3.6).

We also design two functions in the mutation operator. The **switch** function is used on container allocation vector. The mutation operator **Change VM type**, is used on VM allocation vector.

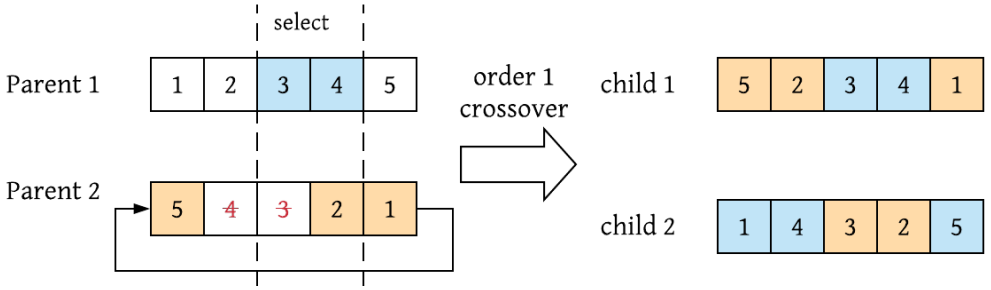


Figure 3.5: Order 1 crossover

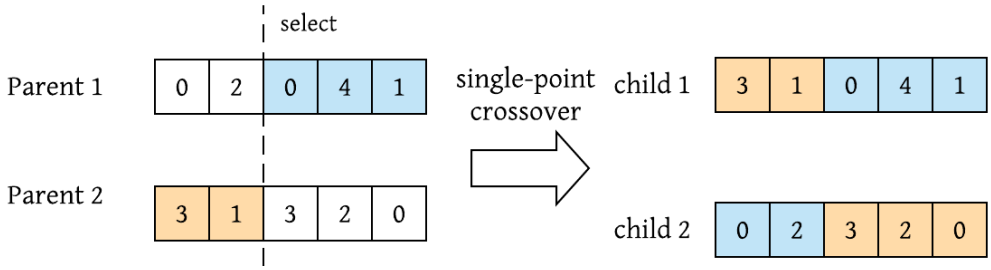


Figure 3.6: Single-point crossover

Switch mutation randomly selects two entries on container allocation vector and switch their values. This mutation changes the allocation of two containers.

Change VM type mutation loops through the VM allocation vector and changes the value uniformly from the VM type list by a probability. This mutation modifies the types of VM.

The overall *DGA* mutation process is shown in Algorithm 6. In the beginning, a random number of u is generated to determine whether applying the **switch** function on the container allocation vector. Then, for each entry on the VM allocation vector, a u is generated to decide whether to mutate the type of a VM.

Algorithm 6: *DGA* Mutation

Input : An individual, mutation rate β
Output: A mutated individual

```
1  u ← random();
2  if u < β then
3  |  Switch mutation() on container allocation vector;
4  end
5  for each entry on the VM allocation vector do
6  |  u ← random();
7  |  if u < β then
8  |  |  Change VM type mutation();
9  |  end
10 end
11 return the mutated individual;
```

DGA

The procedure of *DGA* is the standard GA (see Section 2.2.1). Hence, it is not repeated in this section.

This section also proposes two variations of *DGA*, e.g., *DGA-FF* and *DGA-NF*, which are based on two decoding methods.

3.6 Group-based GA for RAC

This section describes our *GGA-RAC* approach for the *RAC* problem which includes the overall procedure of GA, a group representation, and three problem-specific operators.

3.6.1 Overall Procedure

This section first introduces the overall procedure of our proposed *GGA-RAC* approach.

Algorithm 7: GGA-RAC for the *RAC* problem

Input : a set of containers, a set of VM types, a list of PM instances,
Output: an allocation of containers

```
1  population ← Initialization;
2  gen ← 0 ;
3  for gen does not reach the maximum generation do
4      fitness evaluation(population);
5      new population ← elitism(population);
6      while have not filled the new population do
7          parents ← tournament selection(population);
8          children ← gene-level crossover(parents);
9          unpack(children);
10         merge(children);
11         add children to the new population
12     end
13     gen ← gen + 1;
14 end
15 return an allocation of containers;
```

The algorithm (see Algorithm 7) starts with the initialization of a population. The individual is represented as a list of PMs. Then, the algorithm enters a loop of evolution where each loop is called a generation. In each generation, individuals are evaluated with a fitness function (Eq.(3.1)). Then, the best individuals are preserved and copied to the new population with Elitism [30]. Tournament selection [144] is used to direct the population to the high-fitness region. Then, we propose three problem-specific

operators, *gene-level crossover*, *unpack*, and *merge*. These operators modify the individuals so that they can perform an effective search in the solution space.

3.6.2 Representation

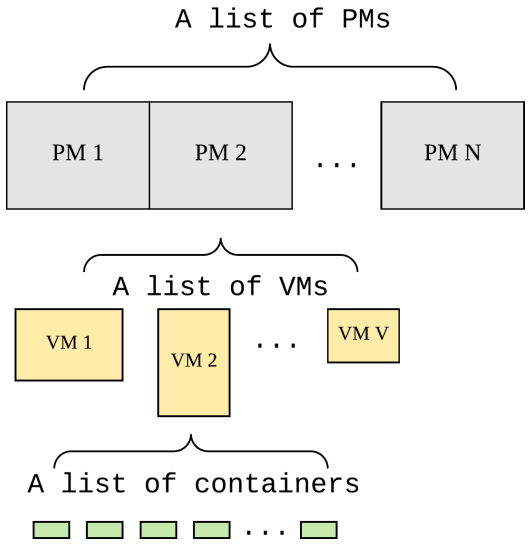


Figure 3.7: Representation

The representation of an individual (see Figure 3.7) is a complete solution for a *RAC* problem. The individual consists of a list of PM instances. Each PM consists of a list of VM instances, and each VM instance has a list of containers.

3.6.3 Initialization

The design of initialization aims at producing a diverse population of solutions. For each individual, we first randomly generate a permutation of containers. Then, we allocate containers to VM instances using a FF [49]. If there is no VM instance available, we create a VM instance with a random type. At last, a list of VM instances is allocated to PM instances with the

FF This representation ensures a diverse combination of containers and VM instances. It also locates the solutions in a high-quality region with FF instead of NF.

3.6.4 Gene-level Crossover

To inherit the useful parts from parents, one must define what is a "good gene". In the bin packing problem, a good gene at bins' level is when well-filled bins can lead to fewer bins [175]. Similarly, highly utilized PM instances could lead to fewer PM instances for the allocation problem. Therefore, we define a good gene as a PM instance with high utilization. In our case, we apply the *gene-level crossover* twice according to the utilization of CPU and memory respectively and generate two children.

The gene-level crossover preserves the highly utilized PM instances from both parents. In the beginning, we sort the PM instances in both parents according to PM instances' utilization of CPU or memory in descending order. Then, the crossover compares the PM instances from two parents in pairwise (see Figure 3.8). The winner's PM instance of the pair will be preserved. Preservation includes three steps. First, the crossover copies the VM instances combination inside the PM instance, including the types and number of VM instances. Second, the crossover checks whether a container from the original VM instance has been allocated in the previous PM instances. If the container has been allocated, then the container will not be allocated again. In the end, some containers may not be allocated to PM instances. They are called *free containers*. These free containers are reallocated with an operator called *rearrangement*, which will be introduced in the next section. After all the containers have been allocated, empty PM and VM instances are removed from an individual.

An example of crossover is shown in Figure 3.8. The left-hand side shows two parents and their allocation while the right-hand side shows a child and its allocation after completing the crossover and before the use

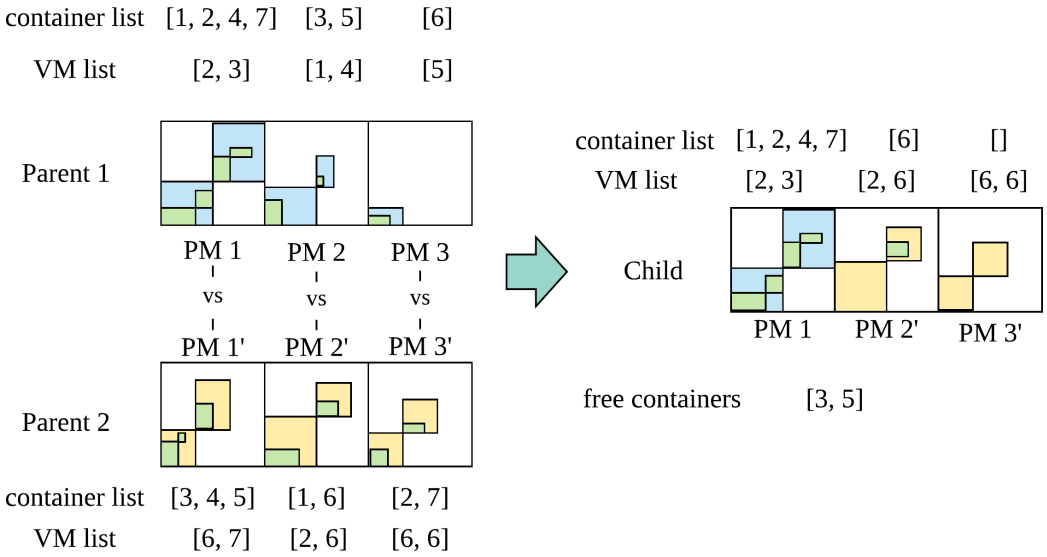


Figure 3.8: An example of gene-level crossover

of rearrangement. Six containers will be allocated to PM instances. Each container list (wrapped with brackets) includes the containers that have been allocated to the corresponding VM list. The allocation figure shows the detailed allocation of containers. It is easy to observe that the VM types and the number of VM instances in the two parents are entirely different. In both parents, PM instances have been sorted. Then, the pairwise comparisons start with PM 1 and PM 1′. The wining PM instance, PM 1, is copied to the child. Then, the second pair of PMs are compared, and PM 2′ wins. Instead of copying all its VM instances and containers to the child, a container 1 has been removed because it has been allocated in PM 1. Similarly, the PM 3′ from parent 2 is copied, but all its containers are removed. Finally, two free containers, 3 and 5, are allocated with a rearrangement operator which will be introduced in the next section.

3.6.5 Rearrangement

Rearrangement inserts free items to bins. In the beginning (see Algorithm 8), we sort the containers according to the product of their normalized resources R (see Eq.3.13) in ascending order.

$$R = \frac{\zeta^{cpu}(c_i)}{\Omega^{cpu}(p_k)} \cdot \frac{\zeta^{mem}(c_i)}{\Omega^{mem}(p_k)} \tag{3.13}$$

Then, we check whether in each VM instance, the smallest two containers can be replaced by the target container. If so, we replace the small containers with the target container. Otherwise, check the next VM instance. After replacing, we have two smaller containers that need to be allocated. At this point, we apply *First-Fit (FF)* & *Random Creation (RC) / First-Fit (FF)* heuristics to allocate them. If the target container cannot replace any of the two containers, we also applied *FF&RC/FF* to allocate it. The *FF&RC/FF* means, we first use FF to allocate containers to VMs. If no VM instance has enough space, we randomly create a new VM instance to allocate containers and use FF to allocate the new VM instance to PM instances.

Our rearrangement operator is inspired by [175] to avoid the drawback of FF and further improve the structure of a VM instance. In the bin-packing problem, FF-based approaches [61, 175] have been widely used. However, a simple FF-based approach cannot change the existing structure of a bin. Hence, the replacement heuristic is developed. The core idea of the replacement heuristic is that the smaller items are easier to allocate. Therefore, we replace smaller containers with a big one. Then, two smaller containers can be easily allocated to existing VM instances without creating a new VM instance. The heuristic does not consider more than two containers because of simplicity. The heuristic of replacing two containers is trying to avoid the computational complexity of searching for the replacement of multiple containers.

Algorithm 8: Rearrangement operator

Input : a target container, a list of PM instances,
Output: a list of PM instances

```
Sort the containers in all VM instances according to Eq.3.13 in
  ascending order;
for each VM instance do
    if the two smallest containers in each VM instance can be replaced by
      the target container then
        Replace two containers with the target VM instance;
        Allocate two containers with FF&RC/FF;
        return a list of PM instances;
    end
end
Allocate the target container with FF&RC/FF;
return a list of PM instances;
```

3.6.6 Unpack

Unpack operator eliminates low-utilized PM instances and reallocates their containers. This operator prevents premature convergence and introduces new gene component into the current population.

The operator has two steps. First, it calculates the probability of unpacking a PM instance, according to Eq.(3.14). The PM instance with high CPU utilization has a smaller chance of being unpacked. Second, it unpacks PM instances in a roulette wheel style. After unpacking, the free containers are reallocated with the *rearrangement* operator.

$$probability = \frac{1 - \Omega^{cpu}(p_k)}{\sum_{k=1}^{K} 1 - \Omega^{cpu}(p_k)} \tag{3.14}$$

The unpack operator is adaptive with the evolution process. In the beginning, the average utilization of PM instances is low. Therefore, more PM instances are unpacked. As the population evolved, high utilized

PMs move to the head of an individual and have a low chance of being unpacked. Therefore, the good genes are preserved, and new genes are introduced by the *rearrangement* operator.

3.6.7 Merge

The merge operator replaces small VM instances with a bigger one to reduce the free resources in PM instances. Free resources represent the resources that have not been allocated to any VM instances. The merge operator can improve the utilization of PM instances by reducing the free resources in PM instances as well as the overheads from VM instances.

The merge operator has two alternative functionalities, merge and enlarge. In the first one, it goes through all the PM instances and checks whether the two smallest VM instances can be replaced by a larger VM type. If it is possible, all the containers are migrated from these two small VM instances to the new larger VM instance, and the small VM instances are removed. If we cannot replace two VM instances with a larger one, we attempt to replace the smallest VM instance with a larger VM type. The large VM type is also selected randomly. The purpose of the replacement is to eliminate the unused the resources in PMs.

3.6.8 Time Complexity Analysis

Based on our implementation, we compare the computation complexity (the worst case) between *SGA*, *DGA* and *GGA-RAC* in Table 3.2. *DGA-NF* and *DGA-FF* are in the same column because they have the same worst case.

Where R is the number of free containers to be rearranged, N' is the max number of containers in a VM instance, and M is the number of VM instances used. N'' is the max number of containers in a PM instance. K is the number of PM instances used, M' is the max number of VM instances

Table 3.2: Time Complexity (worst case) Comparison between *SGA*, *DGA-NF* and *GGA-RAC*

	SGA	DGA-NF/FF	GGA-RAC
Initialization	$O(TN)$	$O(N)$	$O(N^2)$
Elitism	$O(N)$	$O(N)$	$O(N)$
Evaluation	$O(N^2)$	$O(N^2)$	$O(N)$
Crossover	-	$O(2N)$	$O(KM' + K\log K + R(MN'\log N' + M^2))$
Mutation	$O(N^2(T + N + R(MN' + M'K)))$	$O(2N)$	-
Unpack	-	-	$O(N''(MN'\log N' + M^2))$
Merge	-	-	$O(KTM'\log M')$
Rearrangement	-	-	$O(R(MN'\log N' + M^2))$
Overall	$O(N^5)$	$O(N^2)$	$O(N^4)$

in a PM instance, and T is the number of VM types. The upper bound of R, N', M, N'', and K is N.

Obviously, *DGA* has lowest complexity (overall $O(N^2)$) than *SGA* ($O(N^5)$) and *GGA-RAC* ($O(N^4)$) in each generation. For the *GGA-RAC* approach, the main reason for high complexity is the rearrangement operator, which has been used in both crossover and unpack operators. The heuristic in rearrangement includes more comparisons in order to swap out the smaller containers. Hence it is costly. Time complexity and performance are also the major trade-off between *GGA-RAC*, which embedded with heuristics versus *DGA*, which replied on stochastic search. The core difference between *DGA* and *GGA-RAC* is that *GGA-RAC* allows heuristics to be embedded into the search process, e.g., rearrangement, while *DGA* only relies on the stochastic process.

3.7 Experiments and Results

The overall goal of the experiment is to test the performance of our proposed algorithms, *SGA*, *DGA-NF*, *DGA-FF*, and *GGA-RAC* in terms of energy consumption. This section starts with the description of the dataset and test instances designed for the experiment. Then, a benchmark al-

gorithm, a rule-based approach *FF&BF/FF*, is presented. Next section explains the parameter settings of the algorithms. Experiment results are shown and analyzed to understand the performance of these approaches and explain the pros and cons of them.

3.7.1 Datasets and Test Instances

We design 8 test instances (see Table 3.3) which allocates an increasing number of containers (from 200 to 1500) in two sets of VM types. We use a real-world application trace (AuverGrid [189]) to get the test data about the resource requirements, e.g., CPU and memory, of containers. Figure 3.9 shows the distributions of CPU and memory requirements in the AuverGrid trace [189]. To generate the containers' resource requirements, we select the first 400,000 lines of the trace from the original datasets. Then we filtered the trace to exclude the containers that require more resources than the largest VM type. The last step randomly samples resource requirement and use them as the test data of containers.

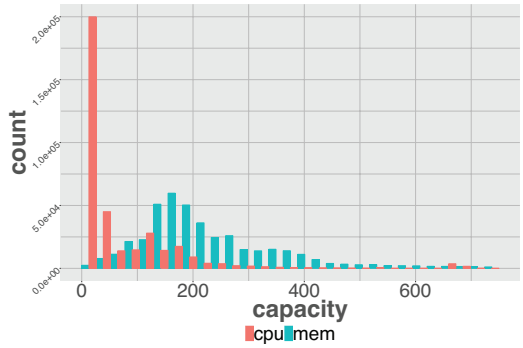


Figure 3.9: Resource usage frequency in the AuverGrid dataset [189]

For the settings of PMs and VMs, we assume homogeneous PMs which have 8 cores and the total capacity is [13200 MHz, 16000 MB]. The maximum energy consumption for the PM is set to 540 KWh. This setting has been used in [133]. We design two sets of VM types (see Table 3.4), real-world VM types (20 types from Amazon EC2), and a synthetic set of VM

types (10 types). The real-world VM types are proportional whereases the synthetic ones are random. The CPU and memory of synthetic VM types are sampled from [0, 3300 MHz] and [0, 4000 MB], representing the capacity of one core.

Table 3.3: Test instances

instance	VM types	number of containers
1	synthetic	200
2	synthetic	500
3	synthetic	1000
4	synthetic	1500
5	real-world	200
6	real-world	500
7	real-world	1000
8	real-world	1500

Table 3.4: VM types

real world VM types							
VM types	[CPU, Memory]	VM types	[CPU, Memory]	VM types	[CPU, Memory]	VM types	[CPU, Memory]
1	[206.25, 250]	6	[412.5, 1000]	11	[825, 2000]	16	[825, 1875]
2	[412.5, 500]	7	[825, 4000]	12	[1650, 250]	17	[1650, 3750]
3	[825, 1000]	8	[206.25, 500]	13	[1650, 500]	18	[412.5, 1312.5]
4	[1650, 2000]	9	[412.5, 2000]	14	[1650, 1000]	19	[825, 2625]
5	[412.5, 250]	10	[412.5, 4000]	15	[412.5, 937.5]	20	[2475, 2625]
synthetic VM types							
1	[719, 2005]	4	[1135, 3542]	7	[1363, 2634]	10	[2100, 3013]
2	[917, 951]	5	[1231, 1989]	8	[1648, 1538]		
3	[1032, 1009]	6	[1311, 3238]	9	[2047, 1181]		

3.7.2 Benchmark Algorithms

FF&BF/FF [133, 245] uses three heuristics to allocate containers. It uses *First-Fit* heuristics to allocate both containers and VMs and applies a

Best-Fit (BF) for selecting VM types. Whenever no VM instance is available to host a container, the *BF* selects a type of VM which has just enough resource to host the container. Explicitly, *BF* selects the VM type which has the minimum normalized free resources according to Eq.3.15.

$$Free\ resources = min\{\frac{\Omega^{cpu}(\tau_j) - \zeta^{cpu}(c_i) - \pi^{cpu}(\tau_j)}{\Omega^{cpu}(p_k)} \text{ and } \frac{\Omega^{mem}(\tau_j) - \zeta^{mem}(c_i) - \pi^{mem}(\tau_j)}{\Omega^{mem}(p_k)}\} \quad (3.15)$$

3.7.3 Parameter Settings

The parameter setting of *SGA*, *DGA-NF*, *DGA-FF*, and *GGA-RAC* are listed in Table 3.5. In addition to the operators that we proposed, we apply Elitism with size 5 and tournament selection with size 2 on all approaches. For all approaches, we set the population size as 100. *DGA* and *GGA-RAC* share the same crossover rate of 70% and mutation rate of 10%. *SGA* completely relies on the mutation to search. Hence, we set a high mutation rate of 80%.

These parameters are commonly used in other works. In addition, we have tested the parameters empiricially by trying different combination of values. The selected parameter values achieved the best results among the examined combinations

Table 3.5: Parameter Settings

	SGA	*DGA-NF/FF*	*GGA-RAC*
crossover rate	-	70%	70%
mutation rate	80%	10%	10%
elitism size	5	5	5
population	100	100	100
tournament size	2	2	2

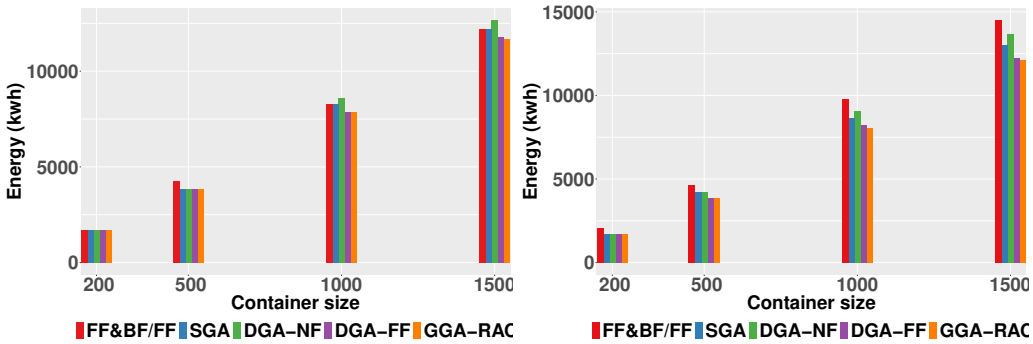
All algorithms were implemented in Java version 8, and the experiments were conducted on i7-4790 3.6 GHz with 8 GB of RAM running

Linux Arch 4.14.15. We ran each GA-based algorithm 30 times and applied the Wilcoxon rank-sum to test the statistic significance.

3.7.4 Results

This section illustrates the performance comparison among the five algorithms in terms of energy consumption. Then, we explain the drawbacks of the compared algorithms by comparing the convergence and the number of VM instances in the allocation.

The energy consumption of five algorithms are shown in Figure 3.10 and in Table 3.6. For each test instance, all algorithms are run with different numbers of generations. We show each algorithm that runs the same amount of time to ensure the comparison is fair. The benchmark algorithm *FF&BF/FF* has an overall poor performance in all test instances. *DGA-NF* also performs poorly and even worse than *FF&BF/FF* in test instances 3 and 4 of synthetic VM types. *SGA* has similar or better performance than *FF&BF/FF* and *DGA-NF*. *DGA-FF* and *GGA-RAC* are much better than other approaches. *DGA-FF* works well in small test instances (200 and 500 containers), while *GGA-RAC* shows its advantages in larger test instances (1000 and 1500 containers). In the following, we analyze the behaviors of these algorithms by showing detailed statistics.



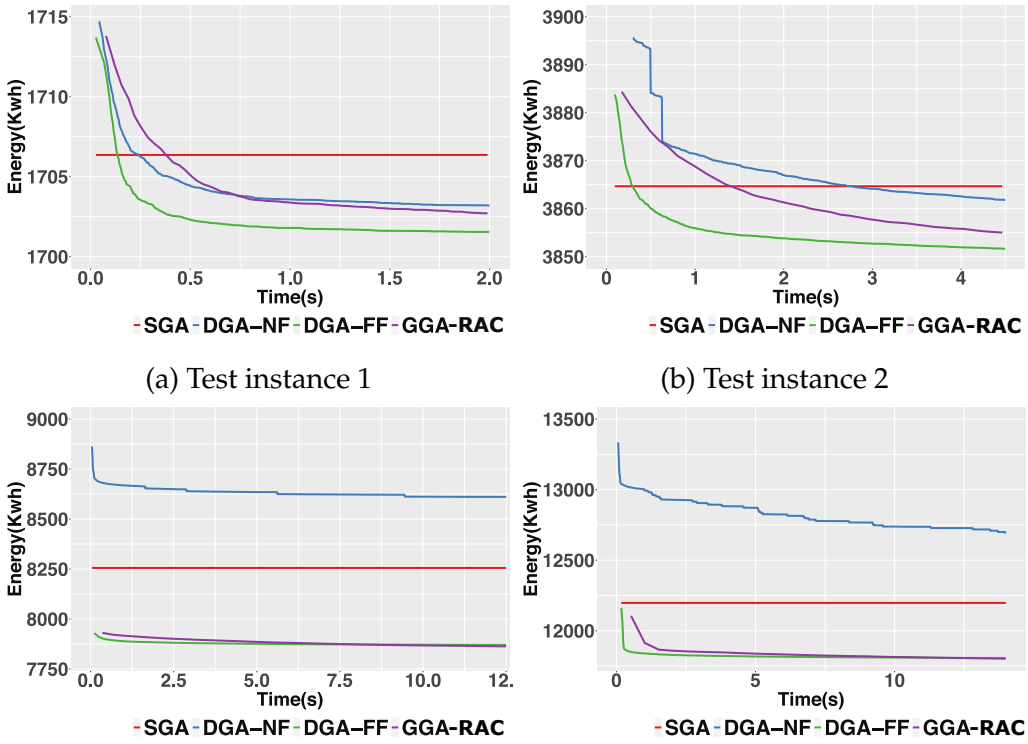
(a) Synthetic VM types (b) Real-world VM types

Figure 3.10: Comparison of the average energy consumption

Table 3.6: Energy consumption of all test instances obtained by five algorithms.

	synthetic VM types (test instances 1 to 4)			
container size	200	500	1000	1500
computation time	2.0s	4.5s	12.0s	18.0s
FF&BF/FF	1708.0 ± 0	4244.2 ± 0	8259.0 ± 0	12176 ± 0
SGA	1706.3 ± 0.6	3864.6 ± 1.4	8255.3 ± 2.6	12197.3 ± 58.4
DGA-NF	1703.1 ± 0.4	3861.1 ± 2.6	8610.8 ± 111.9	12662.2 ± 131.4
DGA-FF	1701.5 ± 0.2	3851.2 ± 1.2	7870.6 ± 3.7	11803.4 ± 4.2
GGA-RAC	1702.5 ± 0.5	3852.1 ± 1.1	7849.5 ± 3.3	11700.3 ± 136.4
	real-world VM types (test instances 5 to 8)			
container size	200	500	1000	1500
computation time	2.0s	5.0s	10.0s	12.0s
FF&;BF/FF	2093.2 ± 0	4635.0 ± 0	9808.2 ± 0	14500.4 ± 0
SGA	1711.3 ± 2.0	4221.2 ± 108.9	8669.5 ± 58.1	13024.5 ± 116.2
DGA-NF	1701.6 ± 0.9	4234.1 ± 2.4	9037.3 ± 84.7	13693.7 ± 124.0
DGA-FF	1699.1 ± 0.5	3848.8 ± 1.4	8246.9 ± 3.9	12220.4 ± 95.4
GGA-RAC	1699.2 ± 0.4	3845.4 ± 1.9	8064.2 ± 183.1	12127.8 ± 4.9

We show the convergence curves of proposed algorithms in terms of computation time in Figure 3.11 and Figure 3.12. The benchmark algorithm *FF&BF/FF* is not shown here because it performs much worse than the other algorithms. The first noticeable fact is that *SGA* (red) shows a flat trend, meaning the population hardly evolve during the search procedure. *DGA-FF* (green) converges in a short amount of time, and it performs well in small instances (1, 2, 5, 6). *DGA-NF* always starts at the worst solution with the highest energy consumption. However, the poor searchability of *DGA-NF* hinders it from finding good solutions. *GGA-RAC* has a slow convergence in all test instances. However, *GGA-RAC* performs the best in large instances (3, 4, 7, 8) and has similar performance in small instances as well. Overall, *GGA-RAC* shows strong searchability.



(a) Test instance 1

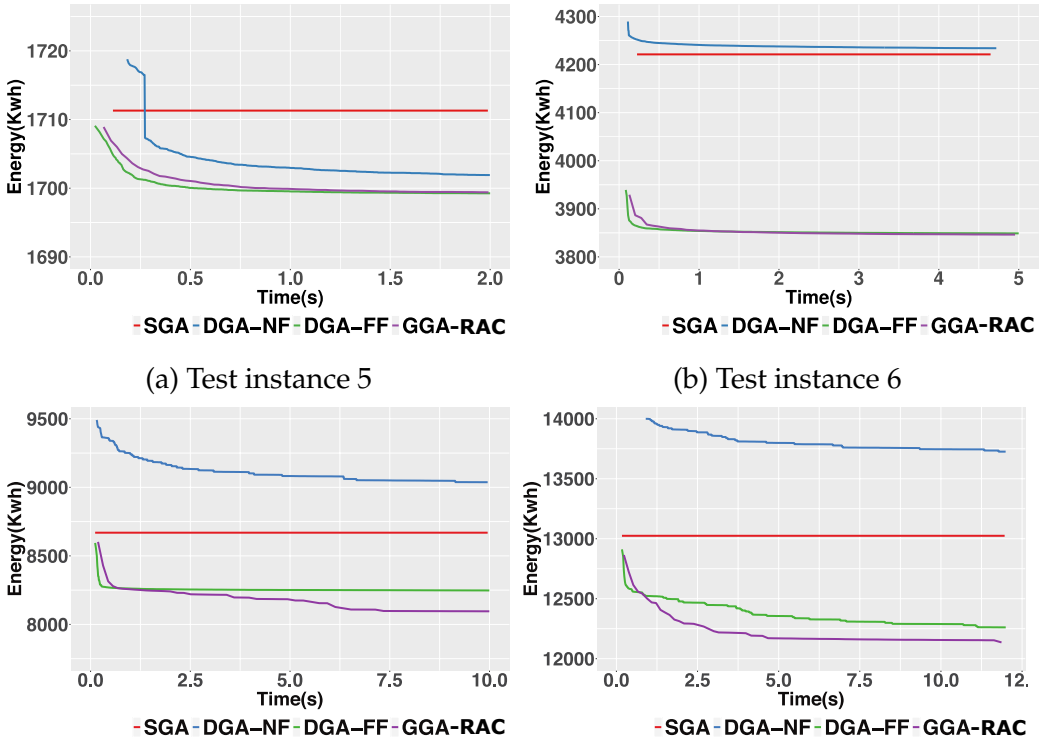
(b) Test instance 2

(c) Test instance 3

(d) Test instance 4

Figure 3.11: Comparison of the convergence among *SGA*, *DGA-NF*, *DGA-FF*, and *GGA-RAC* of test instance 1 to 4

The poor searchability of *SGA* is due to two reasons. The first reason is that *SGA* does not have a crossover operator. Crossover operator does not only combines the "good" parts from parent individuals but also generates a solution that "far away" from the original solution in the search space. Without crossover, the exploration of *SGA* is dramatically decreased. The second reason is that the *SGA* completely relies on the mutation operator to search. The mutation operator tends to allocate the container to existing VM instances or creates the first VM instance that can host the container. This heuristic is both greedy and highly dependent on the order of the predefined list of VM types. Since containers are generally small (see Figure 3.9), an individual of *SGA* is largely filled with the same type



(a) Test instance 5

(b) Test instance 6

(c) Test instance 7

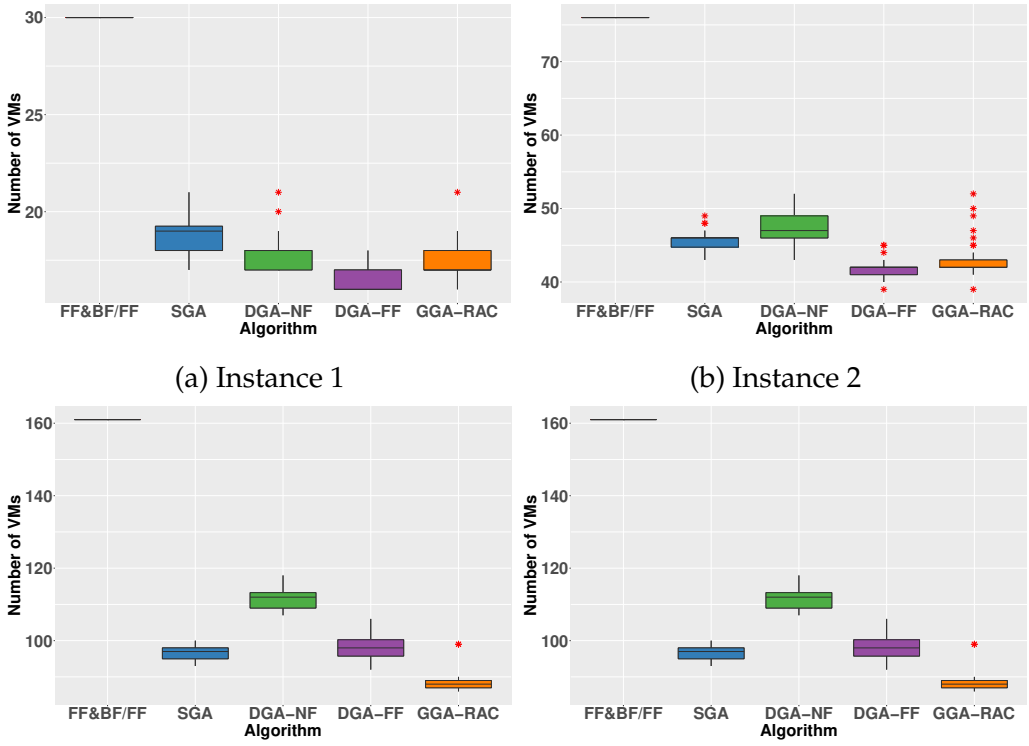
(d) Test instance 8

Figure 3.12: Comparison of the convergence among *SGA*, *DGA-NF*, *DGA-FF*, and *GGA-RAC* of test instance 5 to 8

of VM instances (type 1 from the Table 3.4). Hence, without introducing diverse VM types, switching containers from one VM instance to another can hardly improve the quality of a solution. This is because a group of the same type of VM instances cannot form different combinations of PM instances.

The major defect of *DGA-NF* is the decoding process. Compared to *FF*, *NF* closes a bin (such as VM and PM instance) whenever the current item (such as container and VM) cannot be allocated to it while *FF* never closes a bin so that the future items can be still put into the unfilled bins. It means that *NF* cannot guarantee a VM instance to be filled with containers. Consequently, we may observe that *DGA-NF* starts from a bad allocation

and takes a long time to converge. Replacing *NF* with *FF* immediately improves the performance. However, the *DGA-FF* is still inferior to the *GGA-RAC* approach in large test instances.



(a) Instance 1

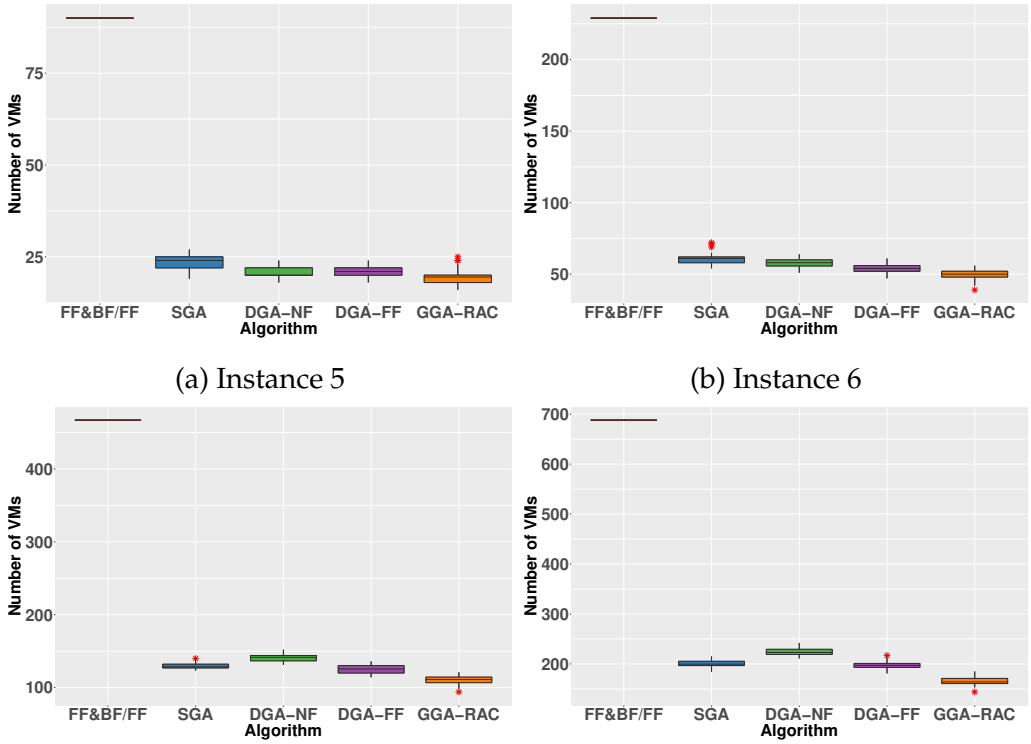
(b) Instance 2

(c) Instance 3

(d) Instance 4

Figure 3.13: Number of VM instances in test instances 1 to 4

The number of VM instances used by all five algorithms are compared in Figure 3.13 and Figure 3.14. The *FF&BF/FF* always uses the largest number of VM instances. This is because the algorithm uses *BF* to create new VM instances where *BF* always create a VM instance with the smallest VM type that can accommodate a container. Hence, *BF* creates a large number of small VM instances. These small VM instances increase the segmentation of PM instances and also increase the overheads. Hence, *FF&BF/FF* must use more PM instances. For the most test instances (except instance 2), *GGA-RAC* uses the fewest number of VM instances. That means *GGA-*



(a) Instance 5

(b) Instance 6

(c) Instance 7

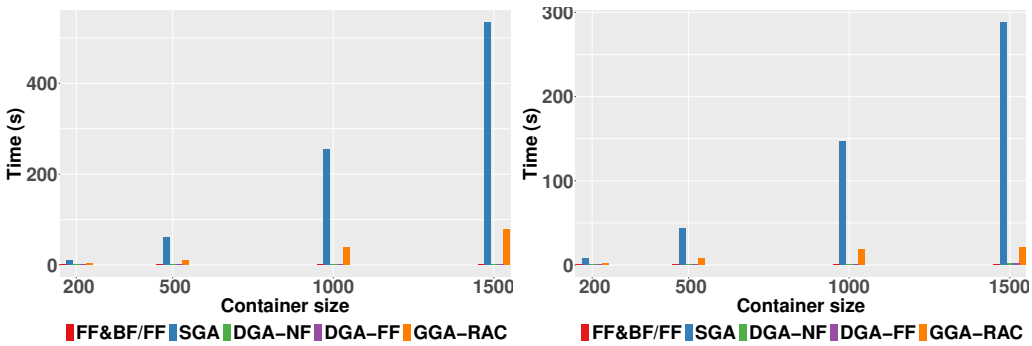
(d) Instance 8

Figure 3.14: Number of VM instances in test instances 5 to 8

RAC is good at selecting large VM instances and form good combinations of VM instances to fill PMs. With fewer VM instances, fewer resources overheads are generated, and segmentation of PM instances is also mitigated.

Lastly, we compare the computation time of all five algorithms with the same number of solution evaluations (e.g., 100 generations) in Figure 3.15. As we may clearly see that *SGA* uses much longer than other algorithms due to its high complexity of its mutation operator. The *GGA-RAC* uses much less time than *SGA* but *GGA-RAC* is still significantly slower than two *DGA* variations.

Overall, in the category of vector-based GA, *SGA* has poor searchability and long computation time compared with *DGA*. Based on the *SGA*,



(a) Test instances with ten VM types (b) Test instances with twenty VM types

Figure 3.15: Comparison of computation time of all algorithms.

we further develop two variations of *DGA*, *DGA-NF* and *DGA-FF*. These two *DGA* variations perform dramatically differently. *DGA-NF* performs poorly and even worse than *SGA*. The major reason is that its *NF* decoding process always closes VM instances without filling them. On the other hand, *DGA-FF* achieves the second-best performance and converges fast. To compare with vector-based GA, we further develop a GA with group-based representation, the *GGA-RAC* approach. *GGA-RAC* has the best overall performance. However, due to the high computation complexity in its operators *GGA-RAC* uses more time than *DGA* approaches.

3.8 Discussions on Representations

This section provides discussions on the representations that we developed in this chapter.

3.8.1 Single-chromosome Representation

The design of single-chromosome representation is a hybrid of direct and indirect representations. For the containers–VMs level, it is a direct representation because the chromosome specifies the allocation of containers,

including the VM indexes and types. For the VMs–PMs level, it is an indirect representation because of the NF decoding.

This hybrid representation has more disadvantages than advantages. The first disadvantage is that the indexes of VM instances need to be maintained after any mutation occurrence. It is important to keep the indexes of VM instances as a continuous sequence because it facilitates the mutation operator to reallocate containers to different VM instances. However, the cost of this maintenance is high in terms of computation complexity.

The second disadvantage is the NF decoding process. That is, VM instances are sequentially packed into PM instances. Although with NF, the decoded solutions are always feasible, meaning that a solution does not contain any violation, the decoding method is far from good. NF closes a PM instance once a VM instance cannot be allocated to the PM. It means that, although the PM instance could still accommodate other VM instances, it will never be revisited. Hence, the decoding method leads to low utilized PM instances.

The third disadvantage is that it is difficult to include a crossover operator for this representation. As mentioned in Section 20, the crossover operator is hard to preserve the information in the parents.

3.8.2 Dual-chromosome Representation

The dual-chromosome representation improves the single-chromosome representation from two aspects. Firstly, unlike the fixed sequence of containers, the dual-chromosome representation uses a permutation of container indexes. Thus, the containers can be arbitrarily combined when different permutation is generated. Hence, the dual-chromosome representation avoids the indexing of VM instances. Secondly, various crossover techniques can be used in this representation.

There are limitations of the dual-chromosome representation as well. The first one is the fixed length of the VM allocation vector. During the de-

coding process, after allocating all containers to VM instances, the unused VM instances in the VM allocation vector are ignored. Since the actual number of VM instances is much smaller than the number of containers, the majority of the VM allocation vector is not used (the ignored part of the vector). This causes a waste of computational resources. When a large number of individuals are generated, memories are wasted. Computing power is also wasted when applying genetic operators on the unused parts of VM allocation vectors. The second disadvantage is that the same solution can have different representations. For instance, Figure 3.16 shows two individuals with different representations. However, they can be decoded into the same solution with NF. The diversity of a population is decreased because of this disadvantage. The third disadvantage is that it is difficult to embed heuristics in the operators to accelerate the search. This is because the dual-chromosome representation needs a decoding process to interpret an individual to a solution. The decoding process follows a sequential order. Hence, any modification on an index leads to a major change in the solution.

3.8.3 Group Representation

Since the group representation can be directly evaluated without using any decoding process, the group representation mainly has two advantages over the vector-based representation. Firstly, the effectiveness of the decoding process can be neglected. Secondly, heuristics can be embedded in the genetic operators to improve the quality of the solution intentionally. Because *GGA-RAC* applies directed search instead of fully relying on stochastic search, *GGA-RAC* could find better solutions. For example, we switch two containers' allocation in different VM instances without changing the structure of the entire solution. Therefore, the disadvantage of indirect representation in *DGA* is avoided.

The *GGA-RAC* has a higher computational cost than the *DGA* due to

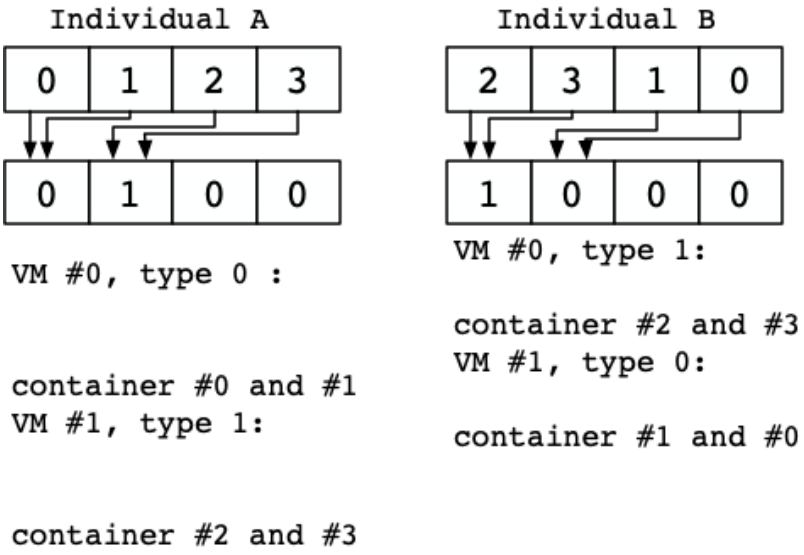


Figure 3.16: An example of two individuals with different representations being decoded to the same solution.

two reasons. The first reason is that the group representation is variable-length. It means that the number of PM instances is different for individuals. Hence, the representation is implemented with a linked list. On the contrary, the dual-chromosome representation has fixed-length vectors, which can be implemented as an array. As the computational cost for a linked list is inherently higher than an array, the *GGA* is also more computationally expensive than *DGA*. The second reason is that the complexities of the operators from *GGA-RAC* are higher than the operators from *DGA* (see Table 3.2).

Overall, both vector-based and group-based representations have pros and cons. Compared with *GGA-RAC*, *DGA* is faster. *DGA* and *SGA* can also easily apply existing genetic operators. *GGA-RAC* allows heuristics to be embedded into genetic operators to achieve better performance.

3.9 Chapter Summary

In summary, this chapter investigates two representations for applying EC algorithms to solve off-line *RAC* problem. In order to achieve this goal, we develop GA-based algorithms with two representations, vector-based and group-based representations.

To achieve this goal, four contributions were accomplished. (1) A formal model of *RAC* problem was proposed, two-level of allocation, variables, constraints are formally defined. (2) We develop two vector-based GAs, *SGA* and *DGA*. (3) We develop a group-based GA, *GGA-RAC*. The operators that we proposed address the two-level allocation problem. (4) Experiments were carried out to evaluate the performance of these approaches. The proposed algorithms were compared to a popular human-designed rule.

The analysis of these approaches shows that meta-heuristics with different representations have different performances. From the perspective of algorithm design, the vector-based GAs are simple and straightforward because existing search mechanics, e.g., various crossover operators, can be directly applied to the representation. However, it is difficult to embed domain knowledge and heuristics into the operators because of the decoding process. In terms of the performance of vector-based GAs, they highly depend on the decoding process. *DGA-FF* is much effective *DGA-NF*. In terms of computation time, vector-based GAs are relatively faster without repeatedly repairing process. Hence, *DGA* variations are much faster than *SGA*. The group-based representation, on the other hand, utilizes the domain knowledge well and shows good searchability, especially for large test instances. In terms of computation time, *GGA-RAC* is slower than *DGA*.

For algorithm designers and *cloud providers*, this chapter provides some important insights into two representations and GA-based algorithms that can be directly applied to solving the *RAC* problem. For algorithm design-

ers, vector-based algorithms can be designed quickly since the existing operators can be applied. The group-based representation can achieve better performance.

Next chapter will discuss the on-line allocation scenario of *RAC*. Different from the off-line scenario, the on-line scenario allocates containers on the fly with rule-based approaches. Hence, the quality of the allocation rules is critical in this problem.

Chapter 4

Genetic Programming for On-line Resource Allocation in Container-based Clouds (RAC)

4.1 Introduction

The purpose of the research presented in this chapter is to propose hyper-heuristic approaches for the on-line *resource allocation in container-based clouds (RAC)* to minimize accumulated energy consumption.

Current works on the on-line *RAC* problem mostly employ rule-based approaches [166] 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 that respond to arrivals of containers and the needs of container placement during the container migration process. 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 [85] are too slow for the on-line problem.

Current rule-based works have three major drawbacks. Firstly, most research works treat resource allocation in the container-based cloud as a

single-level problem [83] which only considers the allocation of containers directly to PM instances but neglect the container–VM level. Secondly, the current rules only consider simple features (e.g., residual resources of Physical Machines (PMs)) to make decisions. Since they do not consider the VM types or the features of workloads, these rules cannot adapt to various workload patterns of applications as well as different sets of Virtual Machine (VM) types. In constrast, we design a VM type selection mechanism and consider the features of workload pattern. Therefore, the evolved rules are adaptive to workload patterns and VM types. The workload patterns of applications have been proven a critical factor to the resource allocation problems [126]. Therefore, the rules that ignore the patterns and VM types will lead to poor performance. The third drawback of the AnyFit-based algorithms [51,98] is that it always starts from allocating containers to existing VM instances, while allocating to new VM instances with specific type may lead to lower overall energy consumption. Hence, it limits the decision and its performance.

Hence, this chapter addresses the above mentioned three drawbacks by proposing a hyper-heuristic approach for two-level *RAC* problem. The hyper-heuristic approach can automatically generate rules with complicated structures (e.g. non-linear relationship between the features and the target VMs shown in Section 4.8) using features from the cloud environment. Therefore, the generated rules that can adapt to the environment. In addition, hyper-heuristic approaches can also generate non-AnyFit-based rules such as reservation rules [51] that can create new VM instances for containers even when there are available VM instances.

Genetic Programming Hyper-Heuristics (GPHH) and Cooperative Coevolution GP (CCGP)(see Chapter 2.2.2) are both evolutionary computation hyper-heuristic algorithms which have 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 [152] the multi-task JSS [161], and

the JSS with machine breakdown [160]. The generated rules outperform neural network techniques. GPHH has been applied to evolve rules for variations of bin packing problem [17,34,35]. Furthermore, the automatic learning procedure greatly reduces the complexity of the heuristic-design process. CCGP has been used for generating multiple cooperative heuristics. In [239], CCGP generates sequencing and routing rules for Dynamic Flexible JSS (DFJSS) [243,244]. Similarly, Zhou et al. [250] 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 can adapt to the changing workload patterns.

Hence, in order to achieve the goal of proposing hyper-heuristic approaches for the on-line *RAC* problem, we need to accomplish four objectives. Firstly, we will model the on-line *RAC* problem. The off-line model that we proposed in the previous chapter (see Chapter 3.3) is not suitable for an on-line problem. As previous mentioned (see Chapter 2.3.1), the existing on-line models for *RAC* problem adopt the same model as the off-line problems [133]. The evaluation of the off-line problem is inappropriate to on-line algorithms. This is because the energy consumption of a data center is determined by the overall energy consumption of a given period [55]. Hence, the evaluation for the allocation algorithms is misleading.

Secondly, we will propose a *GPHH-RAC* that combines GPHH and human-designed rules for *RAC* problem. It is difficult to evolve both levels of rules in the first place. Therefore, we first evolve one-level allocation rules and combine them with human-designed rules on the other level. Then, we propose a *CCGP-RAC* method to solve the two-level problem. In this objective, we will use GPHH to evolve reservation-based rules for allocating containers to VM instances.

Thirdly, based on the *GPHH-RAC* approach, we will propose a *CCGP-RAC* approach that evolves two levels of allocation rules. Finally, we will

conduct experiments on these proposed approaches and compare them with the state-of-the-art approaches.

1. To propose an on-line model for the problem;
2. To propose a *GPHH-RAC* approach combining GPHH generated rules with human-designed rules for the two-level *RAC* problem; The *GPHH-RAC* approach follows a reservation framework instead of the Any-Fit framework;
3. To propose a CCGP approach for evolving the allocation rules for two levels *RAC* simultaneously;
4. To evaluate CCGP approach by comparing it with human-designed rules and the *GPHH-RAC* approach on benchmark datasets;

4.2 Chapter Organization

The remainder of this chapter is organized as follows. In Section 4.3, we present a formal model for the on-line *RAC* model. Section 4.4 illustrates the container allocation process and the assumptions that we considered. Section 4.5 describes our *GPHH-RAC* approach for the proposed on-line *RAC* problem. Section 4.6 introduces the proposed CCGP approach. Section 4.7 outlines the experimental design and discusses the experimental results for comparisons between different approaches. Section 4.8 provides a detailed analysis of both human-design rules and evolved rules. Section 4.9 summarizes this chapter.

4.3 On-line *RAC* Model

The on-line *RAC* problem allocates containers at the time when they arrive at a cloud, which is different from off-line *RAC* problem that allocates a batch of containers. The on-line *RAC* model is developed based on the

model proposed in Chapter 3.3. The major difference between the on-line *RAC* model and the off-line *RAC* models is the method of evaluation. In the on-line *RAC* problem, the evaluation calculates the accumulated energy consumption over a period of time [55], while in the off-line problem, the energy consumption is calculated at the time point of evaluation. We have discussed the reasons that we evaluate the problem with accumulated energy in Chapter 2.

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

$$E = \sum_{t=1}^{n} \sum_{k=1}^{K} E_{tk}, \tag{4.1}$$

where E_{tk} is the energy consumption of the kth PM instance (K is the number of PM used) at time t which is defined in Chapter 3.3. The variables, constraints have been described in Section 3.3; thus, they are not repeated.

4.4 The On-line *RAC* Process and Assumptions

This section describes the on-line *RAC* process (see the flowchart in Figure 4.1) and the assumptions that we considered in our problem. In the beginning, a data center is initialized with the initialization data. Then, it starts to allocate containers one by one. For each container, it first filters the incompatible VM instances out. The incompatible VM instances are the VM instances that install different OSs with the current container requires. The *vmSelectionCreation* is a function that makes two decisions, *VM selection* and *VM creation*, to either selects an existing VM instance or create a new VM instance. Later on, after allocating the container to the VM instance, if the VM instance is new, similar to *vmSelectionCreation*, *pmSelectionCreation* selects an existing PM instance or create a new PM instance. After each allocation of a container, the accumulated energy is calculated and the final output for the on-line *RAC* process.

The on-line *RAC* process (See Algorithm 9) describes the details about

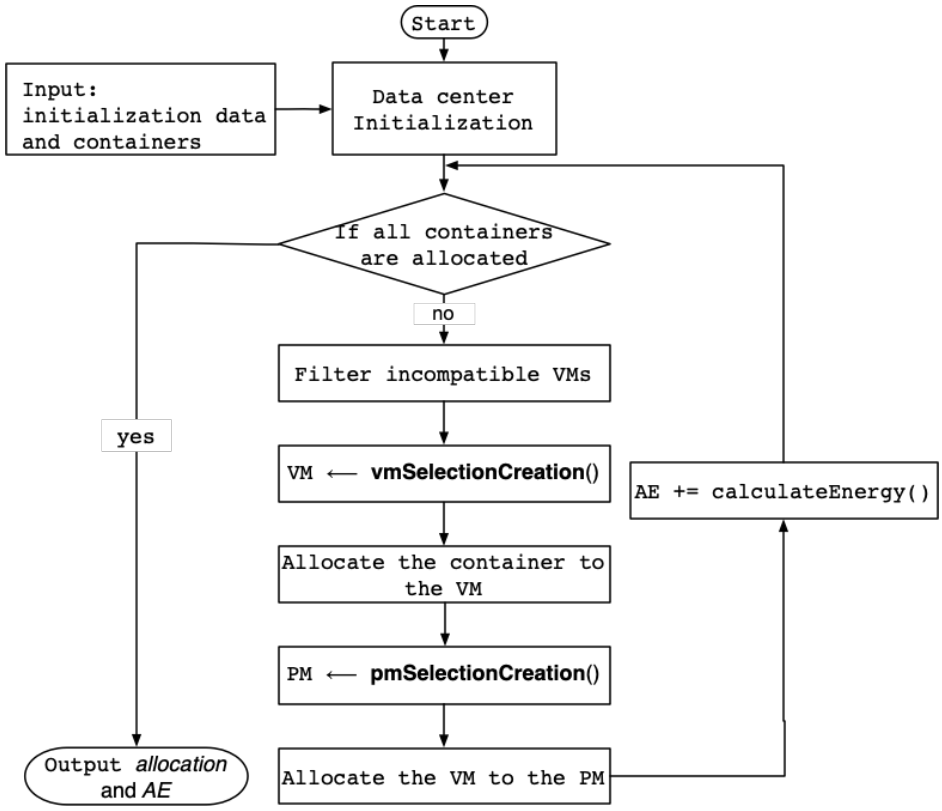


Figure 4.1: The flowchart of on-line *RAC* process.

the on-line container allocation process. The process focuses on the allocation of containers and VM instances. The input of the process includes five components, i.e. an initialized data center, a set of containers C which are unavailable at the beginning, *VM selection* rule sr, *VM creation* rule cr, and *PM selection* rule pr. The output is the allocation of these containers, and we use the accumulated energy consumption of the data center AE over a given period to evaluate the allocation.

The allocation process starts with an initialized data center. The initialized data center runs a number of PM instances, hosting a number of VM instances and containers. A set of containers C arrives (from *line* 2). The containers are allocated one by one. To allocate a container, first, all the VM instances that installed different OSs to the one required by the con-

Algorithm 9: The on-line *RAC* Process

Input : An initialized data center, A list of VM types τ, A set of containers C, VM selection rule sr, VM creation rule cr, PM selection rule pr

Output: Allocation of container, accumulated energy consumption of the data center AE

```
// The initialized data center includes a list of
   VM instances and a list of PM instances.
```

1 $AE = 0$;
2 **for** *each container in* C **do**
3 candidate VM instances ← Filter(a list of VM instances, OS(container));
4 **vm ← vmSelectionCreation(candidate VMs, container,** τ**,** sr**,** cr**)**;
5 allocate(container, vm);
6 **if** *vm is new* **then**
7 add(vm, the list of VM instances);
8 **pm ← pmSelection(a list of PMs, vm,** pr**)**;
9 **if** *pm is null* **then**
10 pm ← pmCreation();
11 add(pm, the list of PM instances);
12 **end**
13 allocate(vm, pm);
14 **end**
15 AE += calculateEnergy(the list of PM instances);
16 **end**
17 return allocation and AE;

tainer are filtered in *line 3*. The *VM selection* rule sr and *VM creation* rule cr are used to select a VM instance from the candidate VM instances or select a type of VM to create a new VM instance in *line 4*. Then, the container is

allocated to the VM instance in *line 5*. If a VM instance is newly created, the OS of the new VM instance is set to the same as the one required by the current container. Then, the *PM selection* rule pr is used to select a PM instance among the existing PM instances in *line 8*. If there is no available PM instance, the *PM creation* rule creates a new PM instance to host the new VM instance in *line 10*. The allocation process outputs the allocation and the accumulated energy consumption AE as introduced in Eq. (4.1).

Algorithm 10 shows the procedure of *VM selection* and *VM creation*. For the *VM selection*, we show a Best Fit algorithm which iterates all the VM instances and evaluates them with sr. The VM instance with the lowest value is selected. If there is no available VM instance, the *VM creation* rule selects a VM type from the VM type list τ. Finally, an existing VM instance or a new VM instance is returned. Notice that, since the procedure shows the Best-Fit algorithm for *VM selection*, the procedure is different from other selection algorithms such as reservation-based selection algorithms.

4.5 GPHH Approach for RAC (GPHH-RAC)

This section describes our *GPHH-RAC* approach to the on-line *RAC* problem. The purpose of developing *GPHH-RAC* is to investigate the effectiveness of applying hyper-heuristics on the *RAC* problem. Therefore, we first apply hyper-heuristic on one-level (containers to VMs) and use human-design rule on the other level (VMs to PMs). As our preliminary work [204] shows, GPHH can be used to generate rules for the one-level allocation problem. Therefore, we develop a *GPHH-RAC* approach that combines a learning algorithm GPHH and a human-designed rule. The GPHH is used to learn and generate the allocation rules for allocating containers to VM instances, and the human-designed rule is used to allocate VM instances to PM instances. In order to create sophisticated allocation rules, we first design a training framework for the GPHH to

Algorithm 10: Procedure of vmSelectionCreation

Input : A list of candidate VM instances, A list of VM types τ, A container, VM selection rule sr, VM creation rule cr,

Output: vm

```
bestScore ← null;
bestVM ← null;
for each vm_i in the candidate VM instances do
    score ← sr(vm_i, container);
    if bestScore ≥ score then
        bestScore ← score;
        bestVM ← i
    end
end
if bestScore is null then
    vm ← create a VM instance with the type of cr(τ, container);
end
else
    vm ← vm_bestVM
end
return vm;
```

generate reservation-based rules. Reservation-based rules can create a new VM instance even when there are available VM instances to provide more chances of optimizing overall energy consumption. That means reservation-based rules have a larger search space than the AnyFit-based rules. Therefore, reservation-based rules are more likely to find better solutions than AnyFit-based rules. Additionally, we design a set of novel problem-specific terminals for *VM creation* and *selection*. These terminals are used as components of the generated rules. In the following subsections, we first provide an overview of the GPHH framework. Then, we describe the representation and terminal set of GPHH, followed by the

fitness function and the GPHH algorithm.

4.5.1 The *GPHH-RAC* Approach Overview

We propose a *GPHH-RAC* approach that uses GPHH to automatically generate a rule that can handle both *VM selection* and *VM creation* simultaneously, and uses a human-designed rule for *PM selection*. In our approach, we combine the *VM selection* and *VM creation* decisions and use a single *VM selection and creation* rule to make both decisions. The *VM selection and creation* rule is based on a reservation framework [50].

To generate rules to solve the on-line *RAC* problem, we use GPHH to search for the rules which adapt to the training data that covers various application workload patterns and VM types. Although the training process may take a long computation time, automatic rule generation takes less effort than manually designing rules. It is efficient to apply the generated rules to solve the on-line *RAC* problem.

The overview of the training process of *GPHH-RAC* is shown in Figure 4.2. The *GPHH-RAC* initializes a population of *VM selection and creation* rules randomly. Then, the rules are evolved by the genetic (e.g., crossover and mutation) operators. A set of training instances is used for fitness evaluation. To evaluate a rule, first, the rule is combined with the prespecified human-designed rule. Then, the pair of rules are applied to each training instance to generate an allocation solution.

By iteratively evaluating and modifying the rules, the population gradually searches in the space of rules. The performance keeps improving because only the best rule is kept to the next generation. This process of evaluation and modification continues until a predefined number of iterations is reached. Finally, the *GPHH-RAC* outputs the rule with the best fitness value in the training process. Then, the generated *VM selection and creation* rule and the human-designed rule will be used to generate the allocation solution for any (unseen) *RAC* problem instance. Next section

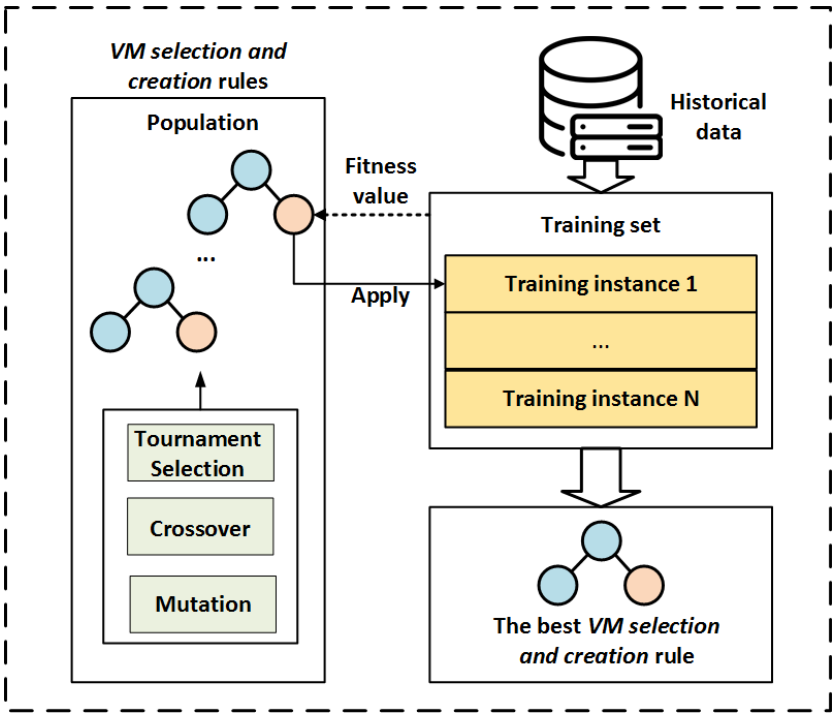


Figure 4.2: The overview of the training process of *GPHH-RAC*.

provides a detailed explanation of the training procedure.

The training procedure of *GPHH-RAC* is described in Algorithm 11. It starts with the initialization of a population of P_{vr}. The population contains N randomly generated *VM creation and selection* rules. We apply the *ramped Half-and-Half* [109] in constructing trees to ensure diversity in the population.

The iteration of evolution begins at *line 5* and ends at *line 18*. From *line 6* to *line 12*, each individual in the population is evaluated with a set of training instances. We have introduced the allocation process in Section 4.4, Algorithm 9. Notice that, we evolve a single *VM creation and selection* rule instead of two separate rules, i.e. *VM selection* and *VM creation* rules. Therefore, *line 4* in Algorithm 9 is changed from $vm \leftarrow vmSelectionCreation(candidateVMs, container, \tau, \mathbf{sr}, \mathbf{cr})$ to $vm \leftarrow vmSelectionCreation(container, \tau, \mathbf{p_i^{vr}})$. The allocation in *line 8* returns the

accumulated energy consumption of AE. Then, we calculate the fitness value (in *line 11*) by Eq.4.2 (see Section 4.5.3). A training instance s^{gen} contains the information of containers to be allocated and a data center with an initialization. We switch to a different training instance in each generation to ensure the generalization of the rules.

Algorithm 11: GPHH-RAC

Input : A set of training instance S, A terminal set and a function set, A human-designed First Fit rule FF,

Output: The best *VM selection and creation* rule p^{vr}

```
1  Initialize a population of rules P_vr;
2  P_vr ← {p_1^vr, p_2^vr, ··· , p_N^vr};
3  gen ← 0;
4  AE ← 0;
5  while maxGeneration is not reached do
6      for each rule p_i^vr in P_vr do
7          for a training instance s^gen in the training set do
8              AE ← apply p_i^vr and FF on the training instance s^gen
                (see Algorithm 9);
9          end
10         p_i^vr ← AE~ / N
11     end
12     TournamentSelection;
13     Crossover;
14     Mutation;
15     Reproduction;
16     gen ← gen + 1
17 end
18 return the best rule p^vr;
```

When all rules have been evaluated, we apply the tournament selection

and genetic operators on the population. The tournament selection [109] 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 to the selected rules. The reproduction copies a number of top individuals to the next generation. Then, the next iteration starts.

4.5.2 Rule Representation

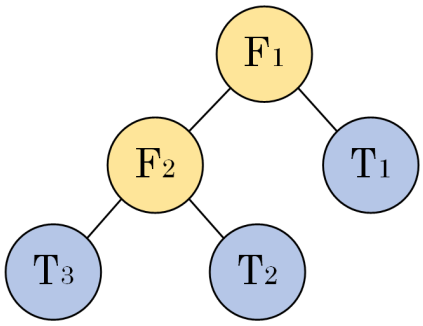


Figure 4.3: The tree-based representation of a rule.

We use trees to represent rules (see Figure. 4.3). One benefit of using trees is that they can be easily interpreted as a prefix notation [109]. 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 re-constructing. Therefore, we can easily evolve them to search in the rule space.

The nodes on Figure. 4.3 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 on-line *RAC* problem and use them as a terminal set of GP trees. Table 4.1 describes the terminal and function sets that we used in GPHH. The design of the terminals follows the research [34] on 2D bin packing.

Table 4.1: Terminal and Function sets

Symbol	Description
Attributes for Container Allocation	
leftVmMem	remaining memory of a VM instance
leftVmCpu	remaining CPU of a VM instance
coCpu	container CPU
coMem	container memory
vmMemOverhead	memory overhead of a VM instance
vmCpuOverhead	CPU overhead of a VM instance
Function set	+ , - , × , protected %

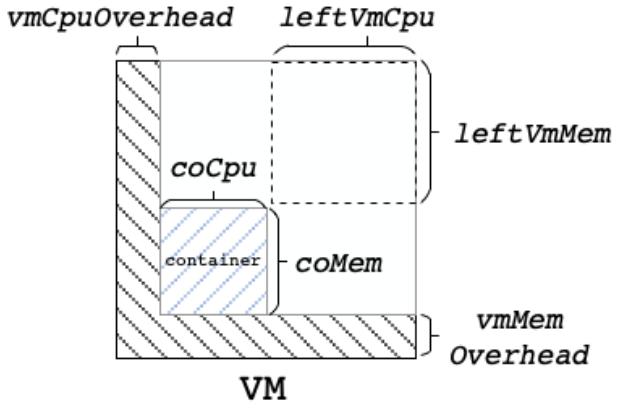


Figure 4.4: Illustration of the features used in the terminal set

We design six features for the terminal set, and they are shown in Figure 4.4. These features are inspired by Burke's work [35]. *leftVmMem* and *leftVMCpu* are the remaining resources of a VM instance after the container is allocated in. *coCpu* and *coMem* are the resource requirements of the container. This work introduces two extra features. *vmMemOverhead* and *vmCpuOverhead* are the amount of CPU and memory overheads when a new VM instance is created. If the evolved rule selects an existing VM instance, then the overhead is 0. For the function set, we use four basic arithmetic operators to construct rules (the protected division returns a

value of 1 when divided by 0).

4.5.3 Fitness Evaluation

As Section 4.4 discussed, when applying AnyFit-based algorithms and reservation-based algorithms, the algorithms for *VM selection* and *VM creation* are different.

Instead of using two rules, *VM selection* sr and *VM creation* cr, to allocate containers to VM instances, GPHH evolves reservation-based rules vr that have the above functionalities. Algorithm 12 shows the procedure of how to use reservation-based rules. In the beginning, lists of candidate VM instances and VM types are provided. The major difference is in *line 3*, where we temporarily append empty VM instances (one for each VM type) into the candidate VM instances. Consequently, the *VM selection and creation* rule vr evaluates the newly created VM instances as well as the existing VM instances. From *line 4* to *line 10*, the *VM creation and selection* rule evaluates all the candidate VM instances and assign them a score. *Line 12* removes the unused empty VM instances from the candidate VM instances. If the selected VM instance is a new VM instance, it is created and returned.

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

$$fitness = \frac{\widetilde{AE}}{N} \tag{4.2}$$

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.

$$\widetilde{AE} = \frac{AE}{AE_{sub\&Just-Fit/FF}} \tag{4.3}$$

We normalize the AE of a rule with the accumulated energy consumption of a benchmark rule (e.g. $AE_{sub\&Just-Fit/FF}$) using Eq. (4.3). The rea-

Algorithm 12: Procedure of vmSelectionCreation using reservation-based rules

Input : A list of candidate VM instances, A list of VM types τ, A container, A VM creation and selection rule vr,

Output: vm

```
1  bestScore ← null;
2  bestVM ← null;
3  Append the list of VM types to the candidate VM instances as
     empty VM instances;
4  for each vm_i in the candidate VM instances do
5  |   score ← vr(vm_i, container);
6  |   if bestScore ≥ score then
7  |   |   bestScore ← score;
8  |   |   bestVM ← i
9  |   end
10 end
11 Remove the unused empty VM instances from the candidate VM
     instances;
12 if vm_bestVM is a new VM instance then
13 |   vm ← create the vm_bestVM;
14 end
15 else
16 |   vm ← vm_bestVM;
17 end
18 return vm;
```

son 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. The use of normalized AE cannot affect the training process because the AE is normalized with a constant value. In our experiments, the normalization is based on the benchmark rule *sub*&*Just-Fit/FF*.

In summary, the *GPHH-RAC* approach automatically generates rules for *VM creation and selection* and uses a First Fit algorithm for *PM selection*. Next section describes the *CCGP-RAC* for the on-line *RAC* problem. Unlike the *GPHH-RAC* approach, *CCGP-RAC* generates rules for both *VM creation and selection* and *PM selection*.

4.6 Cooperative Coevolution GP Approach for RAC (CCGP-RAC)

This section describes the proposed Cooperative Coevolution Genetic Programming (CCGP) approach for the on-line *RAC* problem. The *CCGP-RAC* approach automatically generates rules for both allocating containers to VM instances and VM instances to PM instances. There are two reasons that we develop the *CCGP-RAC* to the problem. First, the on-line *RAC* problem has a two-level structure. Both levels of allocations need to be optimized to achieve the optimal solution. Second, two levels of allocations are interactive; hence, two levels of optimization must be done simultaneously. To develop a *CCGP-RAC* approach, as CCGP has not been applied to the *RAC* problem, we first design a training process that suits the CCGP framework. In addition to previously designed terminals for *VM creation and selection*, we designed a new terminal set for *PM selection*. In the following subsections, we first describe an overview of the *CCGP-RAC* framework. Then, we illustrate the representation and terminal sets of the problem.

4.6.1 The *CCGP-RAC* Approach Overview

The on-line *RAC* problem involves three decision processes, *VM selection*, *VM creation*, and *PM selection* (see Section 4.4). We propose a *CCGP-RAC* approach to automatically generate rules for all these processes. The *CCGP-RAC* approach adopts an idea from previous proposed *GPHH-RAC* approach (see Section 4.5) to generate reservation-based rules. That is, the *CCGP-RAC* also generates the rules to make the *VM selection* and *VM creation* decisions simultaneously. The other rule *PM selection* is generated by the *CCGP-RAC* together with the *VM selection* and *VM creation* rule.

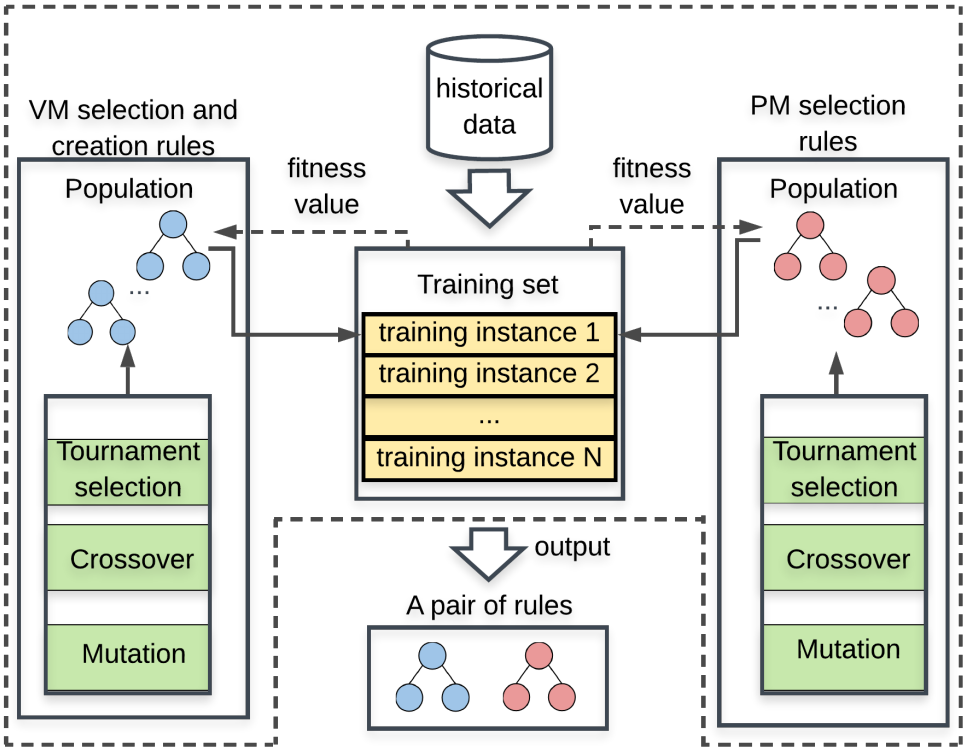


Figure 4.5: The overview of *CCGP-RAC*.

The overview of *CCGP-RAC* is shown in Figure 4.5. The *CCGP-RAC* initializes two populations of rules randomly. Then, the rules are evolved

cooperatively by the genetic operators (e.g., crossover and mutation). A set of training instances is 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-RAC* 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.

The proposed *CCGP-RAC* approach is described in Algorithm 13. 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* [109] 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 variable 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 9. The allocation returns the accumulated energy consumption of AE. The information on containers and PM instances are given by a train-

Algorithm 13: *CCGP-RAC*

Input : A set of training instance S, a terminal set for *VM selection and creation*, a terminal set for *PM selection*, a function set for both rules

Output: The best *VM selection and creation* rule, The best *PM selection* rule

```
1  Initialize each sub-population P_r with r = {vr, pr}
2  P_r ← {p_1^r, p_2^r, ··· , p_N^r};
3  gen ← 0
4  while maxGeneration is not reached do
5      for r = vr → pr do
6          for i = 1 → N do
7              AE ← apply p_i^r and p_rep^r' on the training instance s^gen
                 where r' ≠ r (see Algorithm 9);
8              p_i^r ← AE/N;
9          end
10     end
11     for r = vr → pr do
12         p_r^selected ← TournamentSelection(p_r);
13         p_r ← genetic_operators(p_r^selected);
14     end
15     gen ← gen + 1
16 end
```

ing instance s^{gen} 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 [109] guides the evolutionary process. The rules with higher fitness values have larger probabilities to be selected. Then, two genetic opera-

tors, crossover and mutation, are applied to the selected rules.

Crossover and mutation follow the standard GP algorithm (see Chapter 2.2.2).

4.6.2 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.

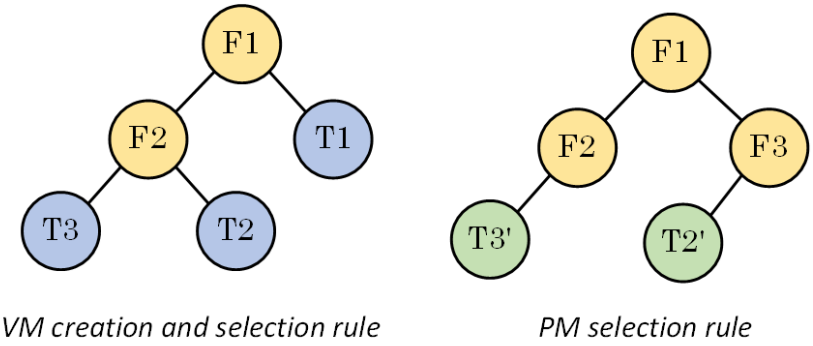


Figure 4.6: The representation of *CCGP-RAC*.

We use two trees to represent *VM creation and creation* and *PM selection* rules (see Figure 4.6). The terminal nodes (T and T' nodes from the Figure) from two rules are sampled from different terminal sets shown in Table 4.2. Two rules share the same function set.

An illustration of the terminals is shown in Figure. 4.7. The *leftVmMem* and *leftVmCpu* are the remaining resources of a VM instance. They are calculated as subtracting the configuration resources of the VM instance by the overhead of that VM and the resources used by the containers running on the VM instance. The *leftPmMem* and *leftPmCpu* are the remaining resources of a PM instance. They are calculated as subtracting the configuration resources of the PM instance by the configuration resources of

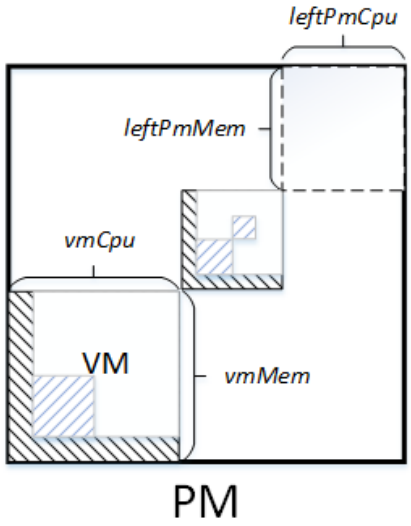


Figure 4.7: Illustration of the features used in the terminal set

the VM instances running on that PM instance. The protected % returns 1 when the denominator is 0.

4.6.3 Fitness Function

To evaluate a pair of rules, we need first to apply the pair of rules on a training instance. Then, we used a fitness function to evaluate its performance, a fitness value. The evaluation process and fitness function are the same as in *GPHH-RAC* (see Eq.4.2 in Section 4.5.3).

4.7 Experiments and Results

A set of experiments were carried out to evaluate the performance of the *GPHH-RAC*, and *CCGP-RAC* approaches. The objective of the experiments is to understand the strengths and weaknesses of the proposed approaches. This section first illustrates the experiment design, including datasets, parameters, and test instances, and then shows the results. Finally, the au-

Table 4.2: Terminal and Function sets of *CCGP-RAC*

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

tomatically generated rules are analyzed to show the insights of resource allocation in container-based clouds. For benchmark algorithm, we use a *sub&Just-Fit/FF* rule as introduced individually in Section 4.7.1.

4.7.1 Benchmark Algorithms

This section introduces the benchmark algorithms for the on-line *RAC* problem.

sub&Just-Fit/FF includes three sub-rules. *sub* rule [146] is represented as the absolute value of the difference between two resources, such as $|vmCPU - vmMem|$ in *VM selection* where $vmCPU$ and $vmMem$ are the remaining resources of a VM instance. *Just-Fit* rule creates the smallest type of VM instance that can satisfy the resource requirements of the container to be allocated. *First Fit* selection algorithm (FF) iterates from the oldest VM/PM instance to the latest one and allocates the container/VM to the first VM/PM instance that has enough resources.

4.7.2 Experiment Settings

This section discusses the simulation environment in both *GPHH-RAC*, and *CCGP-RAC* approaches. Then, the detailed settings of the training and testing scenarios are provided. 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.

Simulation

To reliably measure the effectiveness of the rules, a large number of simulations (e.g., 30 to 50) are usually needed [153]. 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 *VM creation* and *PM selection*, which are independent of training instances. This training method has been successfully applied in Job Shop Scheduling problems [90]. For testing, we use 30 simulations. The testing result shows an average performance on the simulations.

Datasets

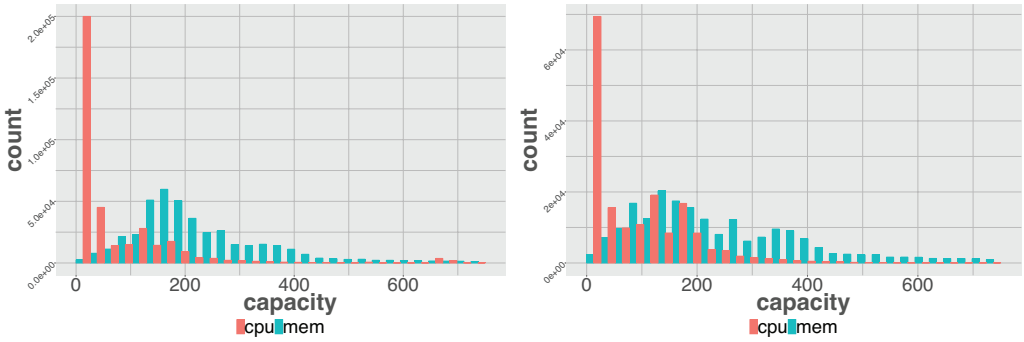
We design 12 scenarios for the experiments (see Table 4.3) which are divided into four groups. Each scenario has distinct numbers of OS from 3 to 5. For each scenario, we use 100 instances for training and 30 instances for testing. Each instance contains 2500 containers to be allocated. This number of containers is large enough for an algorithm to reach a stable status.

To generate the instances, we use two real-world workload datasets – AuverGrid trace and Bitbrains trace [189]. 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 filter the records to exclude the containers that

Table 4.3: 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

require more resources than the largest VM type (see Table 4.4 and Table 4.5). The last step is to randomly select the resource requirement of containers to construct an instance. Figure. 4.8 shows the distributions of CPU and memory requirements in two datasets.



(a) AuverGrid trace (b) Bitbrains trace

Figure 4.8: Resource usage frequency in the real-world 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 [133]. We assume each

core has eight threads. To simulate real-world VM configuration, we use the information of VM types offered by Amazon EC2 in Table 4.4. Additionally, to generalize the VM configuration, we randomly generate 10 VM types (see Table 4.5) where the values of CPU and memory are sampled from [0, 3300 MHz] and [0, 4000 MB] representing the capacity of one core because a quad-core PM has four independent units.

Table 4.4: 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]

Table 4.5: Synthetic VM types

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]

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 instance, which has the same OS installed. We simulate three scenarios where the number of OS increases from 3 to 5. The OS require-

ment of a container is generated from a distribution (see Table 4.6). We use this distribution to simulate a real-world market share of OS [11].

Table 4.6: 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

Algorithm 14: Pseudo code for the initialization of a data center

Input : A set of PM instances P,
// Each PM hosts a number of VM instances, Each VM hosts a number of containers
Output: an initialized data center
// An empty data center includes an empty $vmList$ and an empty $pmList$.

```
1 vmList ← null;
2 pmList ← null;
3 for each PM instance p in P do
4 |   pmList ← p;
5 |   for each VM instance v in the PM instance do
6 |   |   vmList ← v;
7 |   end
8 end
9 return the initialized data center;
```

A data center is initialized (see the procedure in Algorithm 14) before the allocation of containers. The initialization data includes a set of PM instances P. Each PM instance in P hosts a number of VM instances. Similarly, each VM instance hosts a number of containers. The initialization

set up the $vmList$ and the $pmList$ by adding VM instances and PM instances into the lists. These lists will be used in the container allocation process. The purpose of allocating containers to an initialized data center is to simulate a real-world scenario in which PM instances are running in different utilization levels. It also helps to train rules that are robust to a different initial state of data centers. We illustrate the method to generate initialization data in the experiment section (see Section 4.7.2).

Based on the allocation process, we designed and implemented a simulator. Below are the assumptions 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 as the threshold does not affect the behavior of allocation algorithm;
4. No weight or priority of containers, which means containers are equally important;
5. Homogeneous PM instances (all PM instances have the same initial resources);
6. Assume an infinite number of available VM/PM instances that can be used;

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

Parameter Settings

Table 4.7 shows the parameters that we used in all experiments. All the parameters follow the setting that has been commonly used in the literature (e.g. [140]). The *CCGP-RAC* and the *GPHH-RAC* were implemented by ECJ [43].

Table 4.7: 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)

4.7.3 Experiment Results

This section first shows the accumulated energy consumption comparison of the *sub&Just-Fit/FF* rule, the *GPHH-RAC* rules, and the *CCGP-RAC* 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.

Overall Results

The comparison of the accumulated energy consumption among the three methods are shown in Table 4.8. We can see that the *CCGP-RAC* rules have a major advantage over the *sub&Just-Fit/FF* rule and the *GPHH-RAC* rules in all scenarios. *GPHH-RAC* rules, although slightly worse than *CCGP-RAC* rules, also performed much better than the *sub&Just-Fit/FF* rule.

Detailed Results

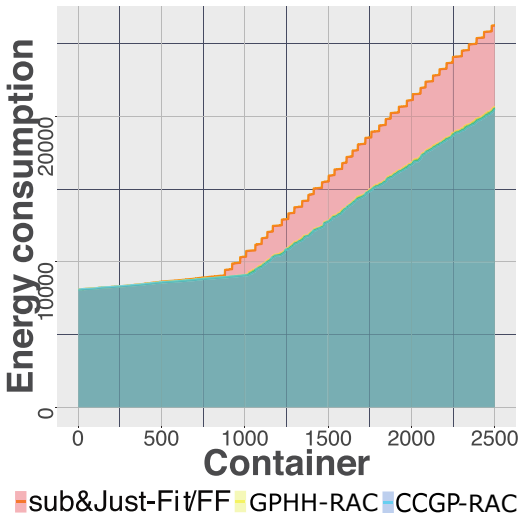
The *CCGP-RAC* rules achieve good performance in all scenarios and we showed scenario 3 and 12 (see Figure. 4.9) because others have similar trends. The allocation procedure shows the increment of energy consumption while allocating containers. The left-hand sides are the comparisons of three methods. The right-hand sides are the zoom-in comparisons be-

Table 4.8: Mean and standard deviation of the energy consumption (Kwh) of 30 instances for 12 scenarios among the *sub&Just-Fit/FF* rule, the *GPHH-RAC* rules, and the *CCGP-RAC* rules.

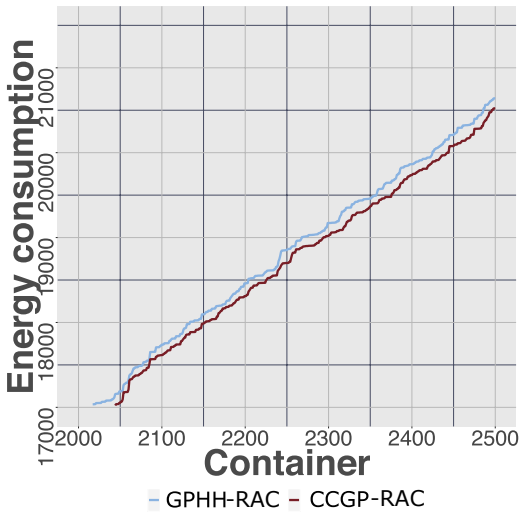
	sub&Just-Fit/FF	GPHH-RAC	CCGP-RAC
scenario 1	3.75E7 ± 1.6E6	3.16E7 ± 2.3E6	**3.12E7 ± 1.0E5**
scenario 2	3.75E7 ± 1.6E6	3.16E7 ± 2.3E6	**3.12E7 ± 1.5E5**
scenario 3	3.77E7 ± 1.6E6	3.18E7 ± 2.3E6	**3.15E7 ± 1.4E5**
scenario 4	4.11E7 ± 1.7E6	3.39E7 ± 2.3E6	**3.36E7 ± 1.8E5**
scenario 5	4.11E7 ± 1.7E6	3.41E7 ± 2.3E6	**3.37E7 ± 8.1E4**
scenario 6	4.12E7 ± 1.7E6	3.42E7 ± 2.3E6	**3.39E7 ± 8.3E4**
scenario 7	3.72E7 ± 1.7E6	3.17E7 ± 2.3E6	**3.10E7 ± 2.7E5**
scenario 8	3.74E7 ± 1.7E6	3.18E7 ± 2.3E6	**3.11E7 ± 3.4E5**
scenario 9	3.74E7 ± 1.7E6	3.19E7 ± 2.3E6	**3.14E7 ± 3.8E5**
scenario 10	3.86E7 ± 2.0E6	3.42E7 ± 2.4E6	**3.34E7 ± 3.4E5**
scenario 11	3.91E7 ± 1.9E6	3.42E7 ± 2.4E6	**3.37E7 ± 3.4E5**
scenario 12	3.91E7 ± 1.9E6	3.47E7 ± 2.3E6	**3.39E7 ± 3.6E5**

tween the *GPHH-RAC* rules and the *CCGP-RAC* 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 PM instances. 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 increment of energy consumption is the slowest using the *CCGP-RAC* rules. Another noticeable pattern is that the turning point of the *CCGP-RAC* rules is later than the *sub&Just-Fit/FF* rule. This means the *CCGP-RAC* rules allocate more containers into the existing PM instances than the *GPHH-RAC* and *CCGP-RAC*. Therefore, the *CCGP-RAC* rules use a smaller number of PM instances and the increment of energy consumption is slow.

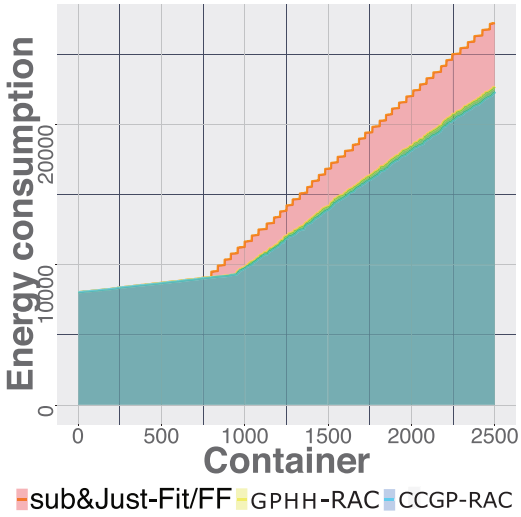
To understand why the *CCGP-RAC* rules have a slower increment of energy consumption compared to other rules, we show the CPU and memory utilization of four representative scenarios, i.e., 3, 6, 9, 12 in Figure. 4.10.



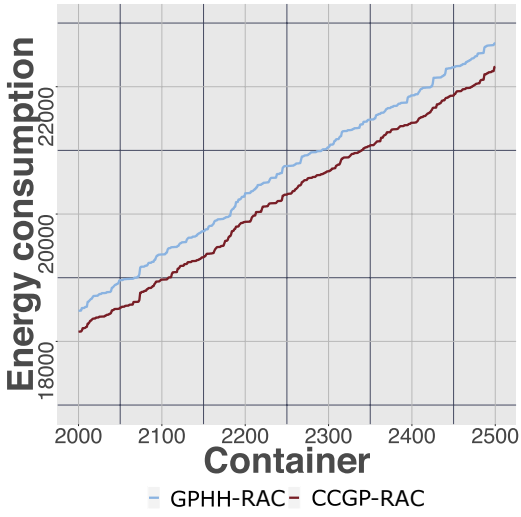
(a) Scenario 3



(b) Scenario 3



(c) Scenario 12

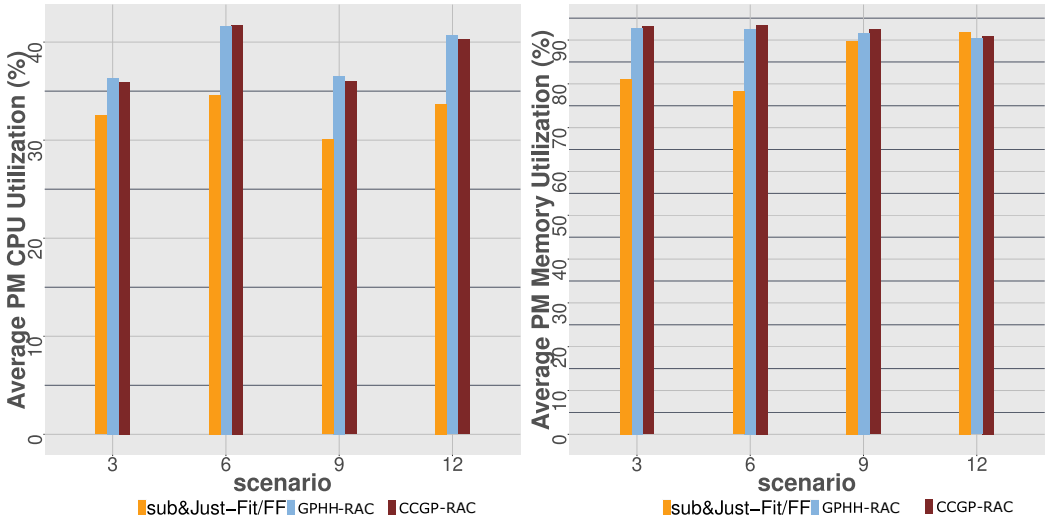


(d) Scenario 12

Figure 4.9: Allocation process of simulation 0 from scenarios 3 and 12

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 PM instances and more energy con-

sumption. To compare the *GPHH-RAC* rules and the *CCGP-RAC* rules, the *GPHH-RAC* rules have better CPU utilization, while the *CCGP-RAC* rules have better memory utilization in all scenarios. As shown in Section 4.7.2, memory resource is the bottleneck in both real-world datasets. It is now clear that the *CCGP-RAC* rules outperform *GPHH-RAC* rules on the critical resource, e.g., memory. The remaining question is that compared to the *GPHH-RAC* rules, why *CCGP-RAC* rules can obtain high utilization of memory?



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

Figure 4.10: PM resource utilization

To improve the utilization of resources, one can improve the utilization of VM instances, reduce the PM instances remaining resources, or both. Since the *GPHH-RAC* rules and the CCGP 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 PM instances, which are affected by two factors, i.e., the number of VM instances on the PM instance and the types of these VM instances. The only way to reduce the PM instance remaining resource is by constructing a combina-

tion of VM instances that uses all or majority resources in a PM instance. To construct such a combination, the *VM creation* and *PM selection* rules must be used together.

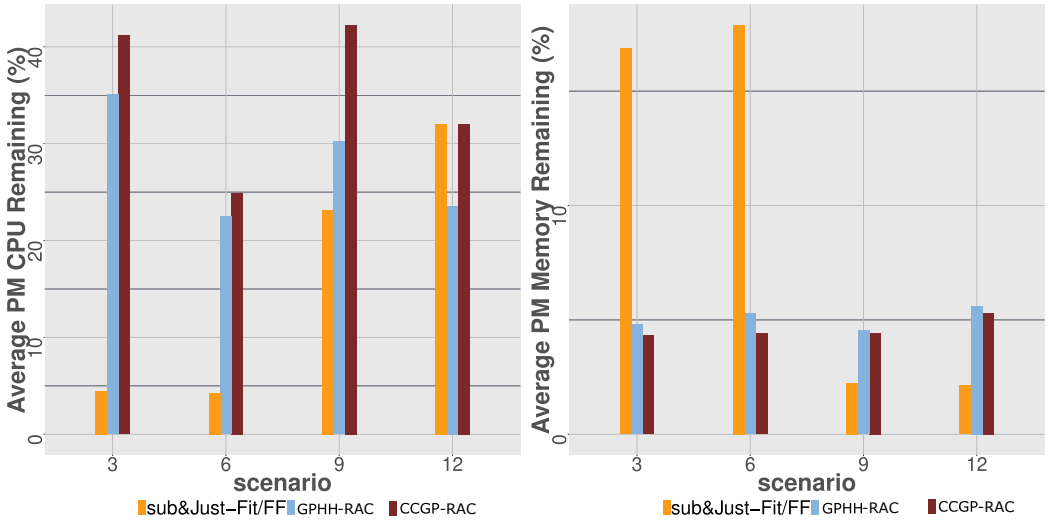
From Figure. 4.11, it can be observed that the *CCGP-RAC* rules have a higher remaining CPU and a lower remaining memory than the *GPHH-RAC* rules do. It means that the *CCGP-RAC* rules use the memory more effectively than the *GPHH-RAC* rules on PM instances. This is consistent with the PM utilization shown in Figure. 4.10. The *CCGP-RAC* rules achieve this performance because they are co-evolved while the *GPHH-RAC* rules consist of two rules generated independently, 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 explanation of why the *CCGP-RAC* rules achieve a better memory utilization is discussed in Section 4.8.

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

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

4.8 Rule Analysis

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



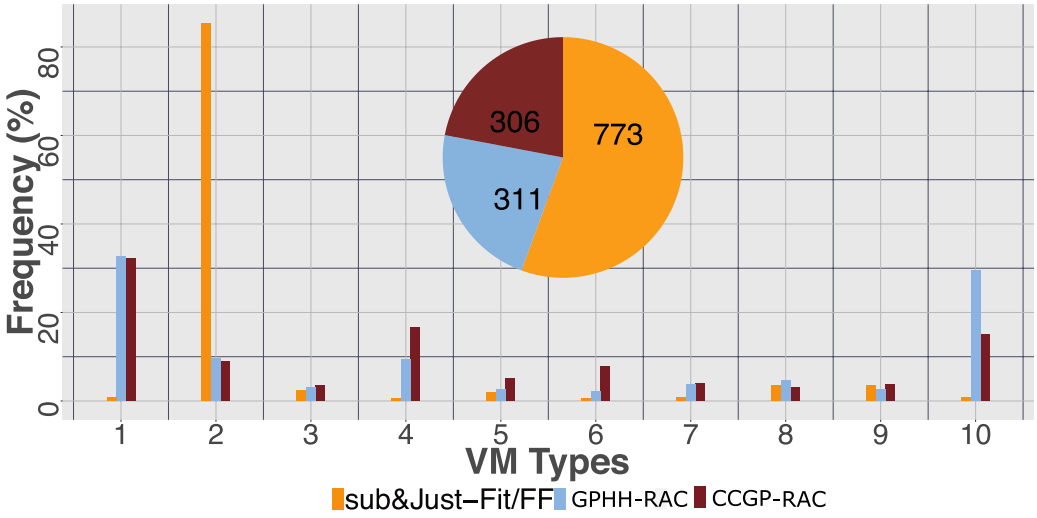
(a) PM remaining CPU (b) PM remaining memory

Figure 4.11: PM remaining resource

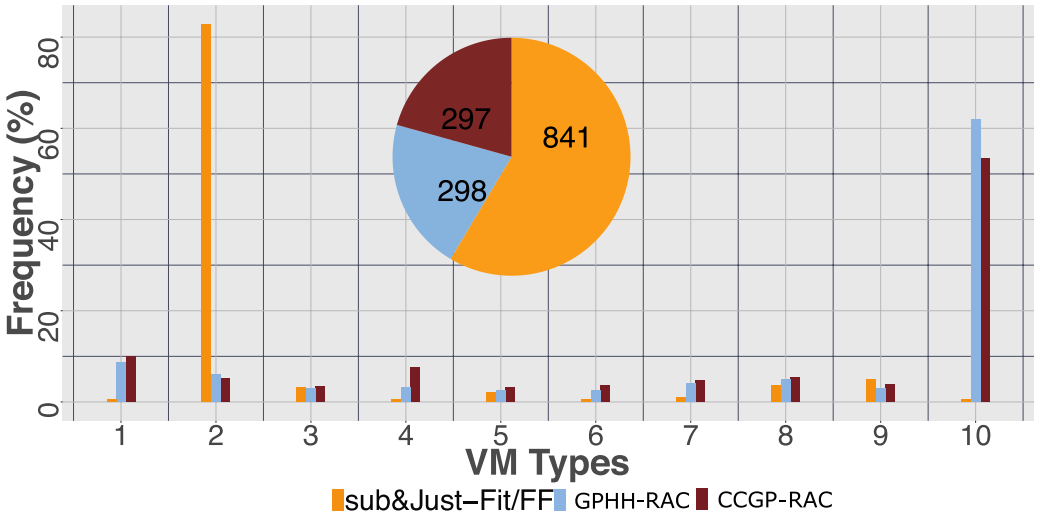
4.8.1 VM Creation Behavior

We analyze the ratio of VM types and the quantity of VM instances of three methods to show the patterns of VM types selection among three approaches. Figure. 4.13 illustrates the average ratio of VM types (bar chart) and the average quantity of VM instances (pie chart) used by three methods in scenarios 3, 6, 9, 12. We analyzed these patterns and found three facts. First, from the pie charts (see Figure. 4.13), we have seen that *sub&Just-Fit/FF* uses 2 to 4 times more VM instances 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 Figure. 4.10), 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 [186] 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, PM instances are filled with a large number of small VM instances and most of them are lowly utilized. Therefore, the average of PM



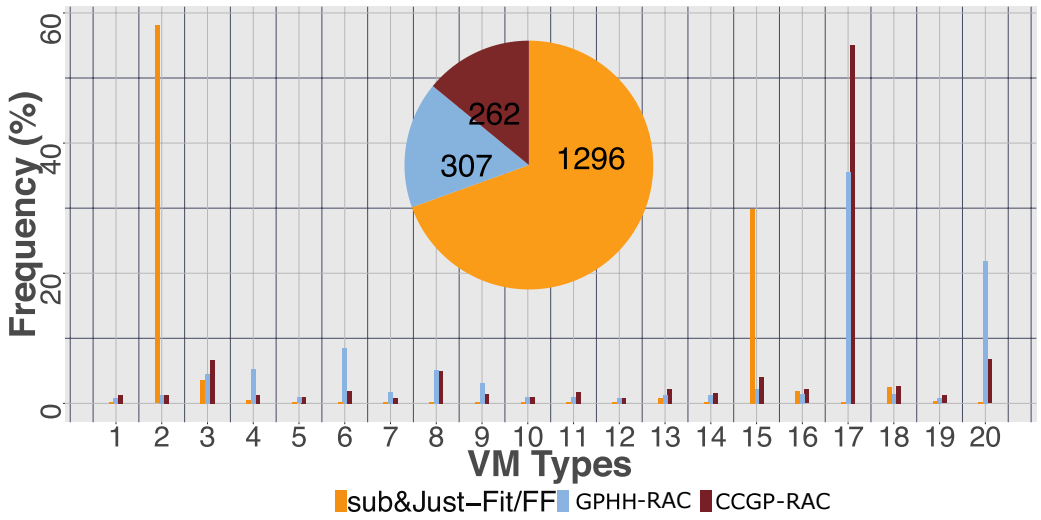
(a) scenario 3



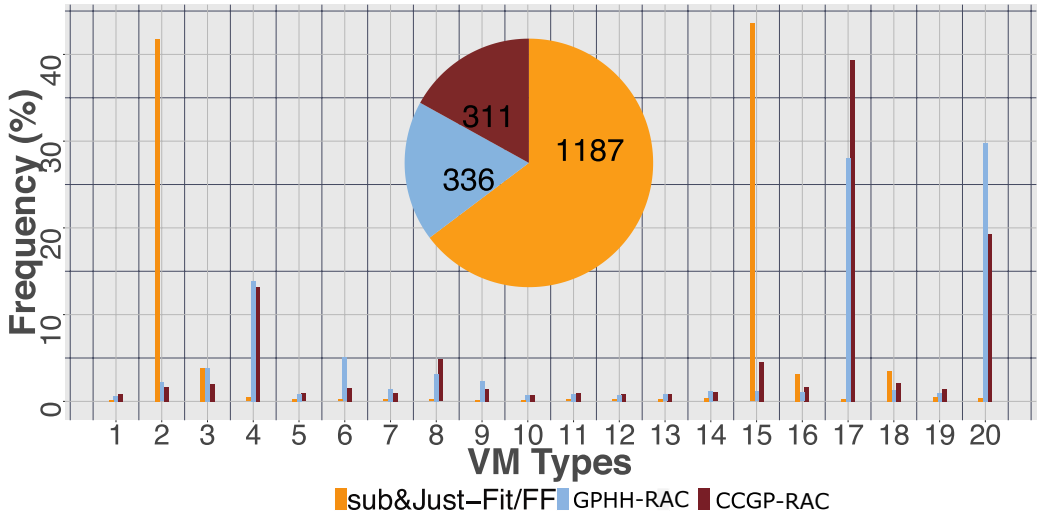
(b) scenario 6

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 VM instances 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 allo-



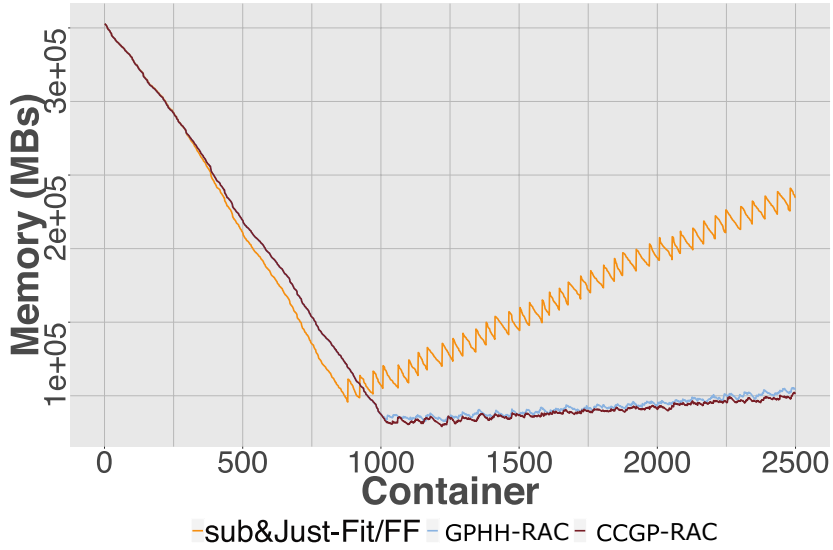
(a) scenario 9



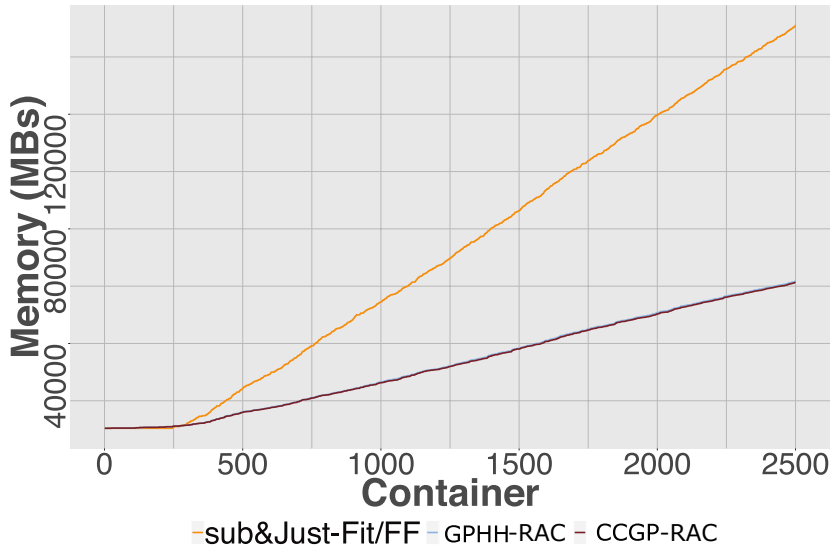
(b) scenario 12

Figure 4.13: The average frequency of VM types used by three algorithms

cation process in Figure. 4.14. 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 VM instances that will never be used. The fast accumulation of VM overheads



(a) Average Wasted Memory



(b) Average Memory Overhead

Figure 4.14: The average waste and overhead of memory in scenario 1, run #0.

and wastes are due to the vast number of VM instances. Therefore, the actual resources used by containers are low when VM sprawl occurs. How-

ever, 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 4.7.2). The *Just-Fit*, therefore, tends to create small VM instances, 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 Figure. 4.11) 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 that uses PM resources efficiently.

On the other hand, the evolved rules can select a good combination of VM instances with the given VM types to avoid VM sprawl. The evolved rules consider both the capacities of VM instances and the residual resources on PM instances. 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 PM instances. 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 *CCGP-RAC* rules create VM instances purposefully with the consideration of multiple factors, e.g., PM residual resources and VM types, so that they improve the utilization of PM instances and successfully avoid VM sprawl.

4.8.2 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 *CCGP-RAC* rules. We are 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 a *CCGP-RAC* rule from scenario 9, run 15, called ***Rule-15***, and illustrate its performance. The reason that we select this rule is that 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***.

$$score = ((leftVmCpu - (leftVmCpu \times (leftVmCpu \times leftVmCpu))) \times ((leftVmMem \div (leftVmCpu + leftVmCpu)) \times leftVmCpu)) \div normalizedVmMemOverhead \tag{4.4}$$

We start analyzing the *Rule-15v* rule. As previously introduced (see Section 4.5.3), the *VM selection and creation* rule *Rule-15v* has the functionalities of *VM selection* and VM type selection when creating a new VM instance. The *Rule-15v* rule achieves these two functionalities by evaluating both existing VM instances and a set of new VM instances. The major difference between selecting existing or creating new VM instances is the value of VM overhead at the evaluation stage.

Specifically, the original *Rule-15v* rule (see Eq 4.4) involves a variable *normalizedVmMemOverhead*. At the evaluation stage, the value of *normalizedVmMemOverhead* equals 0 when *Rule-15v* rule is applied on an existing VM instance. On the other hand, the value of *normalizedVmMemOverhead*

equals 0.0125 (200MB ÷ 16000MB to normalize) when *Rule-15v* rule is applied on a new VM instance.

Hence, the original *Rule-15v* can be simplified to Eq 4.5. For VM selection, the *vmMemOverhead* becomes 0. Therefore, the score of *Rule-15v* becomes a constant of 1 (as we applied the protected ÷). A constant means *Rule-15v* chooses the first VM instance that has enough resources for the container. In other words, the *VM selection* of *Rule-15v* acts like First-Fit. For VM creation, we replace *vmMemOverhead* with a constant of 0.0125 to simplified the rule. Since the simplified rule has two variables, e.g. *leftVmCpu* and *leftVmMem*, we plot the rule on a 3-D surface.

$$score = \begin{cases} 1, & VMselection \\ 10 \times (leftVmCpu - leftVmCpu^3) \times leftVmMem, & VMcreation \end{cases} \tag{4.5}$$

The 3-D surface plot of *Rule-15v* (see Figure 4.15) shows the behavior of *Rule-15v* when it creates new VM instances. The figure also shows why the rule favors **type 17** and **type 20**. The *leftVmMem* ranges in [0, 0.25] and *leftVmCpu* ranges in [0, 0.1875]. This is because the VM types with largest memory of 4000 MB, e.g. type 7 and 10, which is normalized to 0.25. The VM type, i.e. **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 Figure 4.13, scenario 9). For other types, such as **type 7** and **type 10**, although they have a large memory, their CPU capacities are too small.

From the previous analysis, we found that *Rule-15v* tends to create large VM instances. This is a major difference between the human-designed rule *sub&Just-Fit/FF* that favors small VM types, e.g., type 2 and type 15. Large VM instances, create less segmentation and VM overhead, can re-

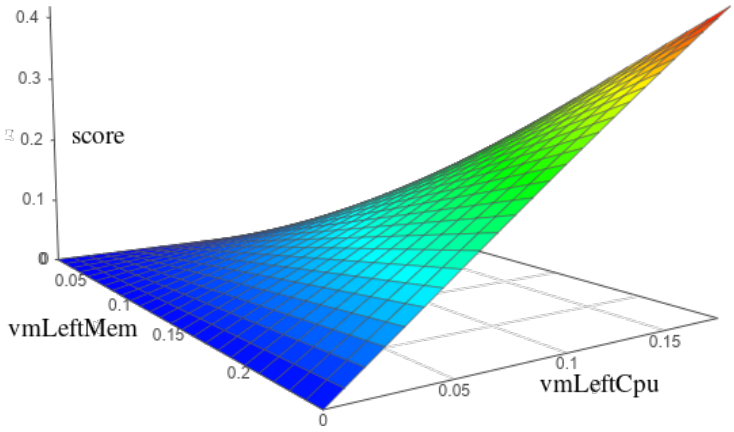


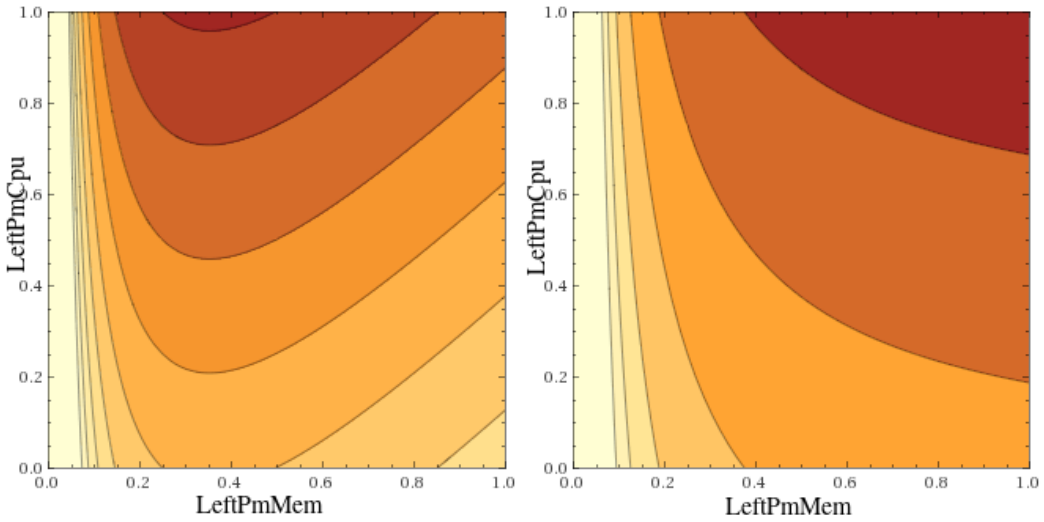
Figure 4.15: The 3D and contour plot of the GP tree: $f = 10 \times (leftVmCpu - leftVmCpu^3) \times leftVmMem$, where x-axis is the *leftVmMem* and y-axis is the *leftVmCpu*

duce the wasted resources in the PM instances.

$$score = leftPmMem - leftPmCpu + vmCpu/leftPmMem \tag{4.6}$$

Rule-15p (see Eq. 4.6) can achieve a good combination of VM instances. Here, we use the most frequently created VM types, i.e. **type 17** and **type 20**, as examples to show the behavior. If we allocate a **type 17** VM instance with *Rule-15p*, the *vmCpu* is replaced by 0.125. Figure 4.16a shows a contour plot of using *Rule-15p* to allocate a **type 17** VM instance. The lighter area represents the region of a higher score. It shows that the rule prefers two types of PM instances. The first type (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 PM instance can allocate more VM instances with large memory capacity and low CPU capacity. The other type of preferred PM instances (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 instance into a PM instance with the resources just enough for one **type 17** VM instance. Sim-

ilarly, if we allocate a **type 20** VM instance with *Rule-15p*, the $vmCpu$ is replaced by 0.1875. Figure 4.16b shows the preferred PM instances should have low residual memory regardless of residual CPU capacity. Notice that, the term $vmCpu/leftPmMem$ in the *Rule-15p* might counterintuition. However, the $vmCpu$ is a constant when *Rule-15p* evaluating PM instances. Hence, $vmCpu$ only provides a constant value in this case.



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

Figure 4.16: The contour map shows the high-score regions of *Rule-15p* when allocating VM of type 17 and 20.

To compare the generated rules and human-designed rules such as First-Fit or Best-Fit, from previous analysis, we observe that the generated rules can construct both linear and non-linear functions to decide the allocation. Therefore, the generated rules can easily mimic the behavior of First-Fit or Best-Fit. However, the non-linear relationship between different features is difficult to capture by human experts. Hence, the generated rules are much flexible compared to human-designed rules.

In summary, this section provides an analysis of a rule, generated by the *CCGP-RAC* 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 VM instances. Besides *Rule-15*, we have analyzed other rules and reach the same conclusion. Due to the page limit, we only described one rule.

4.9 Chapter Summary

The overall goal of this chapter was to propose novel hyper-heuristic approaches for the on-line *resource allocation in container-based clouds (RAC)* to minimize accumulated energy consumption. We achieve this goal by developing a *GPHH-RAC*, and *CCGP-RAC* approaches. Four novel contributions were accomplished to achieve this goal. (1) A novel model was proposed to model on-line *RAC* problem and to evaluate on-line *RAC* solutions. (2) A new set of terminals were proposed to generate rules for the *GPHH-RAC* and *CCGP-RAC* methods. (3) New training procedures were developed to generate reservation-based rules of *VM creation and selection* in both *GPHH-RAC* and *CCGP-RAC*. (4) Novel training procedures of *GPHH-RAC* and *CCGP-RAC* approach were developed as they are first applied to the *RAC* problem.

The analysis of the human-designed rule shows that it leads to VM sprawl by creating a large number of small VM instances. On the other hand, the evolved rules (from both *GPHH-RAC* and *CCGP-RAC*) successfully avoid VM sprawl. The evolved rules can select/create a good combination of VM instances by considering the information of workload patterns and the VM types. The information is reflected by the interaction of multiple cloud features, e.g., residual resources and VM overheads.

For *cloud providers*, our proposed *GPHH-RAC* and *CCGP-RAC* approaches provide several advantages for solving the on-line *RAC* problem. First of all, these algorithms automatically design 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 workload patterns and VM types. The insights can help cloud providers to develop more effective algorithms. Last but not least, our proposed *GPHH-RAC* approach can generate rules that suitable for heterogeneous workloads and VM types.

Next chapter will discuss the multi-objective scenario of *RAC* and the EC algorithm developed for the problem.

Chapter 5

Evolutionary Multi-objective Optimization for Resource Allocation in Container-based Clouds (RAC)

5.1 Introduction

The purpose of the research presented in this chapter is to propose a multi-objective algorithm for the multi-objective scenario of container-based clouds to reduce energy consumption and improve the availability of applications. In Chapter 3, we have studied the single-objective *RAC* problem, which allocates containers that are independent of each other. With the popularity of service-oriented computing (SOC), many applications are composed of micro-services, which are dependent on each other. This gives rise to the *RAC* problem that allocates containers that are components of micro-services. With dependencies among containers, more constraints and objectives need to be concerned in order to achieve the requirements from both *cloud providers* and *cloud users*. Since the new prob-

lem is multi-objective, the existing single-objective model and algorithms cannot be directly applied to the problem. Therefore, new models and algorithms are needed to be proposed. In particular, we will present a model that considers micro-service architecture because it is commonly used in the industry [7].

Micro-service architecture [148] gets extensive attention in recent years as it has the potential to develop large-scale web applications (e.g., Netflix, Spotify). Micro-service applications consist of a set of loosely coupled web services. That is, these web services are maintained independently, deployed distributed, and communicating through HTTP or messages. By deploying web services in container-based clouds, applications benefit not only from the seemingly infinite resources but also the fast deployment of micro-services and the low overheads of containers. In addition, container-based clouds take care of resource management and automatically add and remove resources to the micro-services. Hence, a container-based cloud is ideal for allocating large-scale applications due to its inherent advantages.

The allocation of micro-service can be described as following. Each application is composed of a set of micro-services that could have multiple replicas. The "replica" refers to the software entity of a micro-service. Multiple replicas of a micro-service can share the workload and avoid single-point failure. With each replica deploying in a container, these containers are allocated to VM instances and then to PM instances.

The nature of the micro-service allocation problem is a multi-objective *resource allocation problem in container-based clouds (RAC)*. Multiple objectives come from stakeholders where *cloud providers* have the primary concern of energy consumption, and Service Level Agreement (SLA)-related objectives, such as maximizing availability or minimizing the communication cost between containers. The problem can be considered as a vector bin packing problem because containers have multiple resources to be allocated. In addition, this chapter considers an off-line problem as the first

attempt to study the multi-objective *RAC*.

Existing works in the literature study the *RAC* problem with different focuses. The *RAC* problem is studied as a two-level allocation problem in [133,166,245]. They allocate independent containers to minimize the energy consumption of the used PM instances. However, these approaches only consider one objective, i.e., minimizing energy consumption, and neglect the performance of applications. Other research [85, 119, 185, 220] considers the micro-service allocation as a multi-objective problem and optimize objectives such as energy consumption, communication cost between containers, and availability of applications. However, these multi-objective approaches are applied in OS-level container architecture [187], where containers are allocated to PM instances directly. Hence, there is a need to develop a multi-objective approach for the multi-objective *RAC* problem.

Among numerous QoS optimization objectives of micro-services, e.g., communication cost, network distance, and availability, we consider availability as the additional objectives that need to be optimized. The Reliability, Availability, and Serviceability (RAS) are the most concerned QoS characteristics by both *cloud users* and *cloud providers* [163, 212] . Availability defines the time that the services are available. This characteristic fundamentally affected the users' experience. Hence, we focus on this objective. Besides, the availability and energy consumption are conflicting. Therefore, they cannot be optimized separately. This is because, from *cloud providers'* point of view, the replicas of a micro-service need to be allocated as spread across PM instances as possible in order to maximize the availability. However, the minimization of energy consumption requires to use as fewer PM instances as possible. Therefore, there is a conflict between maximizing availability and minimizing energy consumption.

Multi-objective evolutionary algorithms (MOEAs) are well suited for the *multi-objective RAC* problem. As previously mentioned, *multi-objective RAC* problem involves a two-level vector bin packing problem, which is

NP-hard [51]. Integer Linear Programming (ILP) or Mixed Linear Programming (MLP) approaches cannot be used in large-scale problems because of the high computation time [51]. Evolutionary algorithms (EAs) can find near-optimal solutions within a reasonable period. Compared to greedy-based heuristics, it has less chance to get stuck into local optima [225]. Also, MOEAs provide a set of trade-off solutions in a single run. Users of MOEAs can select one of the solutions based on their preferences. This is an effective way to provide multiple trade-off solutions.

Non-dominated Sorting Genetic Algorithm (NSGA)-II proposed by Deb et al. [57] is one of the most widely applied MOEAs. Due to its effectiveness of finding wide-spread solutions [13], NSGA-II has been successfully applied in many real-world multi-objective combinatorial problems such as web service allocation [205], service composition [217,218] and resource allocation in clouds [127, 203]. These problems have similar representations and problem structures with the multi-objective *RAC* problem.

Therefore, to address the multi-objective *RAC* problem as a multi-objective resource allocation problem, we propose an NSGA-II-based method, named *NS-GGA*, to minimize the energy consumption and maximize the availability. The *NS-GGA* is based on NSGA-II and our proposed *GGA-RAC* in Chapter 3. The major difference between *NS-GGA* and NSGA-II is that we propose problem-specific genetic operators to the *RAC* problem. *NS-GGA* and *GGA-RAC* are different both in genetic operators and the optimization procedure. The proposed *NS-GGA* provides a set of non-dominated solutions that allows *cloud providers* to choose from. Therefore, we propose three objectives to achieve the goal.

1. To propose a novel problem definition for multi-objective *RAC* problem;
2. To develop three novel operators for *NS-GGA* and;
3. To evaluate our proposed approach with three state-of-the-art algorithms on real-world datasets.

5.2 Chapter Organization

In this chapter, Section 5.3 presents the model of the problem. Section 5.4 illustrates the multi-objective *RAC* process. Then, Section 5.5 describes the proposed *NS-GGA*. Section 5.6 illustrates the experiment design, results, and analysis. Section 5.7 summarizes the contributions.

5.3 Multi-Objective *RAC* Problem Model

Table 5.1: Notation and description of the problem model

Notation	Description
a_s	an application of index s
ms_j	a micro-service of index j
c_i	a container of index i
τ_j	the VM type of a VM instance j
ψ_i	the OS type of a container i
p_k	a PM instance of index k
x_{il}	An indicator of whether the container i is allocated to the l VM instance
y_{lk}	An indicator of whether the l th VM instance is allocated to the kth PM instance
z_{jl}	An indicator of whether the l th VM instance 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 instance at time t
EP_k^{idle}, EP_k^{full}	The energy consumption when the kth PM instance 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
$\Upsilon(ms_j) = a_s$	The jth micro-service is a component of an application a_s
$\Phi(c_i) = ms_j$	The ith container is a replica of a micro-service ms_j
$OS(c_i)$	The operating system type of the ith container
$\mu_{tk}^{cpu}, \mu_{tk}^{mem}$	The CPU and memory utilization of a the kth PM instance at time t

The multi-objective *RAC* problem model is developed based on the single-objective *RAC* problem model introduced in Chapter 3.3 but considering two objectives. In the multi-objective *RAC* problem, a set of application $\mathcal{A} = \{a_1, \ldots, a_s\}$ arrive at the cloud to be allocated. Each application consists of a set of micro-services $\mathcal{MS} = \{ms_1, \ldots, ms_o\}$. $\Upsilon(ms_j) = a_s$

denotes that a micro-service ms_j is a component of the application a_s. Micro-services have multiple replicas with each mapping to a container $\mathcal{C} = \{c_1, \ldots, c_n\}$. $\Phi(c_i) = ms_j$ denotes that a container c_i is one of the containers of micro-service ms_j. Each container c_i has a CPU occupation $\zeta^{cpu}(c_i)$, a memory occupation $\zeta^{mem}(c_i)$. 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 $\Omega^{mem}(\tau_j)$. Also, it has a CPU overhead $\pi^{cpu}(\tau_j)$ and memory overhead $\pi^{mem}(\tau_j)$, indicating the CPU and memory occupation for creating a new VM instance of that type. There is an unlimited set of PM instances $\mathcal{P} = \{p_1, \ldots, \}$ for allocating the created VM instances. Each PM instance p_k has a CPU capacity $\Omega^{cpu}(p_k)$ and a memory capacity $\Omega^{mem}(p_k)$. Each PM instance also has a failure rate $\mathcal{F}(p_k)$ indicating that at any time point, a PM instance has a probability to crush and not available.

The multi-objective *RAC* problem has the same constraints as those in *single-objective RAC* problem:

1. Each container is allocated to one VM instance.
2. Each created VM instance is allocated to one PM instance.
3. For each created VM instance, the total CPU and memory occupations of the containers allocated to that VM instance does not exceed the corresponding VM capacity.
4. For each PM instance, the sum of the CPU and memory capacities of the VM instances allocated on the PM instance does not exceed the corresponding PM's capacity.
5. For each container, it must be allocated to a VM instance which has installed the same OS.

The energy consumption is calculated as follows:

$$E = \sum_{k=1}^{K} E_k, \tag{5.1}$$

where E_k is the energy consumption of the kth PM instance and K is the number of PM instance used.

E_k is calculated as follows:

$$E_k = E_k^{idle} + (E_k^{full} - E_k^{idle}) \cdot \mu_k^{cpu}, \tag{5.2}$$

where E_k^{idle} and E_k^{full} indicate the energy consumption of the kth PM instance per time unit if it is idle and fully loaded, respectively. μ_k^{cpu} indicates the CPU utilization level of the kth PM instance which sums up the utilization of the CPU of this PM instance. This energy model, which is proposed by Fan et al. [63], means that the energy consumption of a PM instance is ranging from E_k^{idle} to E_k^{full} depending on its CPU utilization level. μ_k^{cpu} is calculated as follows.

$$\mu_k^{cpu} = \frac{\sum_{l=1}^{L} \left(\sum_{j=1}^{m} \pi^{cpu}(\tau_j) \cdot z_{jl} + \sum_{i=1}^{n} \Omega^{cpu}(c_i) \cdot x_{il} \right) \cdot y_{lk}}{\Omega^{cpu}(p_k)}, \tag{5.3}$$

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

The availability of all the applications is calculated as follows:

$$Availability = \frac{\sum_{s=1}^{S} \Lambda(a_s)}{S} \tag{5.4}$$

where $\Lambda(a_s)$ is the availability of the application a_s. It is defined as the product of the availabilities of the application's micro-services.

$$\Lambda(a_s) = \lambda_{ms_1} \cdot \lambda_{ms_2} \cdot \ldots \lambda_{ms_o}, \forall \Upsilon(ms_j) = a_s \tag{5.5}$$

The availability of a micro-service is related to the PM instances that host its containers (see Eq.5.6). The micro-service ms_j is crushed if all its containers c_i are crushed (see Eq.5.7). Eq.5.7 means that if the PM instance p_k is crushed, then, all the containers in the PM instance are crushed. Since

these containers are not independent, the case statement returns the failure rate of the PM instance (denote as $F(p_k)$) once. Otherwise, it returns 1.

$$\lambda_{ms_j} = 1 - \prod_{k=1}^{K} crushPro(p_k) \tag{5.6}$$

$$crushPro(p_k) = \begin{cases} F(p_k), & if(\sum_{i=1}^{n}\sum_{l=1}^{L} x_{il} \cdot y_{lk}) > 1, \forall \Phi(c_i) = ms_j \\ 1, & else \end{cases} \tag{5.7}$$

The following example shows how to calculate the availability of an application. An application has two micro-services A and B. Micro-service A has two containers c_1 and c_2 which are both allocated to PM instance p_1. Micro-service B also has two container c_3 and c_4 which are allocated to PM instances p_2 and p_3. Assume the failure rate for all PM instances as 2%. Then, the availability of the application is calculated as following. Since containers c_1 and c_2 are allocated to the same PM instance, the availability of micro-service A is $\lambda(ms_A) = 1 - 2\% \cdot 1 = 98\%$. The availability of micro-service B is also $\lambda(ms_B) = 1 - 2\% \cdot 2\% = 99.96\%$. Then the availability of the application is $98\% \cdot 99.96\% = 97.9608\%$.

The *multi-objective RAC* problem is to find resource allocation with minimal overall energy consumption and minimal failure (1 - availability) as shown as follows.

$$\min \ \sum_{k=1}^{K} E_k, \tag{5.8}$$

$$\min \ 1 - \frac{\sum_{i=s}^{S} \Lambda(a_s)}{S}, \tag{5.9}$$

$$s.t. \ \sum_{l=1}^{L} x_{il} = 1, \ \forall \, i \in \{1, \ldots, n\}, \tag{5.10}$$

$$\sum_{k=1}^{K} y_{lk} = 1, \ \forall \, l \in \{1, \ldots, L\}, \tag{5.11}$$

$$\sum_{j=1}^{m} z_{jl} = 1, , \ \forall\, l \in \{1, \ldots, L\}, \tag{5.12}$$

$$\sum_{i=1}^{n} \zeta^{res}(c_i) x_{il} \leq \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl}, \quad \forall\, l \in \{1, \ldots, L\},\ res \in \{cpu, mem\}, \tag{5.13}$$

$$\sum_{l=1}^{L} \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl} \leq \Omega^{res}(p_k), \quad \forall\, k \in \{1, \ldots, K\},\ res \in \{cpu, mem\}, \tag{5.14}$$

$$OS(c_i) = OS(c_j),\ \forall l = 1, \cdots, L. \sum_{l=1}^{L} x_{il} x_{jl} = 1, \tag{5.15}$$

$$x_{il}, y_{lk}, z_{jl} \in \{0, 1\}, \tag{5.16}$$

where constraints (5.10) and (5.11) indicate that each container (or new created VM instance) is allocated to exactly one created VM instance (or PM instance). Constraint (5.12) indicates that each created VM instance must belong to a type. Constraint (5.13) implies that the total occupation of all the containers allocated to each created VM instance does not exceed its corresponding capacity. Constraint (5.14) indicates that the total capacity of the created VM instances allocated to each PM instance does not exceed its corresponding capacity. Constraint (5.15) means that the containers allocated to the same VM instance must have the same required operating system, which is the installed operating system on that VM instance. Constraint (5.16) defines the domain of the decision variables.

The major differences between the multi-objective and the single-objective models (introduced in Chapter 3) are in the following four aspects. First, the definition of applications and the organization of applications are different. In the single-objective model, we define that each container represents an application because an application is allocated to a container. In contrast, in the multi-objective model, we define that each application is comprised of a set of micro-services, and each micro-service could have

multiple replicas. With each replica allocated to a container, an application consists of many containers. In our model, we use $\Upsilon(ms_j) = a_s$ to define the relationship between micro-services and applications. We use $\Phi(c_i) = ms_j$ to define the relationship between replicas and micro-services.

Second, since the structure of applications is different in two models, the dependencies between containers are also different. In the single-objective model, each application is allocated to one container, and containers are independent of each other. In contrast, in the multi-objective model, each application is allocated to multiple containers. The containers could be dependent because they could host the replicas of the same micro-services. The calculation of the availability of an application is affected by the allocation correlation of the replicas. That is, if replicas of a micro-service are allocated to the same PM instance, then if the PM instance crushes, all replicas are crushed. Third, an additional characteristic of PM instances' failure rate is considered in the multi-objective model, which is not considered in the single-objective model. This new feature is used to calculate the availability of applications. Fourth, an additional objective, availability, is considered in the multi-objective model.

5.4 The Multi-Objective *RAC* Process and Assumptions

This section explains the allocation process of the multi-objective *RAC* process and our assumptions. The allocation process starts by grouping the applications by their OS requirements. We apply a preprocessing technique that groups the incoming applications into o groups where o is the total types of OSs (assuming all the containers of an application requires the same OS). Then, we allocate each group of applications into an empty set of PM instances using an off-line allocation algorithm. Finally, we evaluate the energy consumption of used PM instances and the average avail-

ability of applications.

The details of the multi-objective *RAC* process (is shown in Algorithm 15). The procedure is simplified from the real-world case to focus on the allocation of applications and VM instances. The input of the process includes three components, a set of applications A with each application includes a set of micro-services, each micro-service has multiple containers, a list of OS types, and a list of VM types. A preprocessing procedure is used to group applications into o categories. Then, it starts to allocate each group of applications to a set of empty PM instances using an allocation algorithm. The output is the allocation, total energy consumption E of the used PM instances, and the average availability of applications.

Algorithm 15: Multi-objective *RAC* Process

Input : A list of VM types τ, A list of OS types o, A set of applications A,

Output: allocation, energy consumption of the data center E

```
1  E = 0;
2  group(applications);
3  for each group of applications do
4  |   pmList ← null;
5  |   allocate applications to pmList;
6  |   E += calculateEnergy(pmList);
7  |   failure += calculateAvailability(A);
8  end
9  failure /= j;
10 return allocation, E and failure;
```

The preprocessing of grouping the applications according to their OS requirements is a commonly used procedure in the industry. The major reason is that traditional clouds allocate containers into a cluster of bare-metal PM instances [156]. It is straightforward to map groups of applications into clusters of PM instances. The grouping procedure of applica-

tions is similar to Chapter 3.4. Hence, they are not repeated.

We have the following assumptions in the experiments,

1. All containers are equally important;
2. PM instances are homogeneous. That is, all PM instances have the same initial resources;
3. There are infinite available VM/PM instances that can be used;

5.5 NS-GGA

This section introduces the design of our *NS-GGA*, which includes the representation, genetic operators, and the fitness function.

5.5.1 Algorithm

Our proposed algorithm follows the standard NSGA-II framework described in Algorithm.16. The algorithm starts with the initialization of a population of solutions in *line 1*. Solutions are represented as a group of PM instances hosting VM instances and containers (see Section 5.5.2). The main evolution is an iterative process consisting of a number of generations from *line 2* to *line 16*. In each generation, each individual is evaluated according to *objective functions* in *line 4*. In the subsequent loop from *line 6* to *line 10*, the binary tournament selects two parents and two children are generated by crossover and mutation operators. After a new population of U is generated, we evaluate U, and then sort and calculate the crowding distance of $P \cup U$ in *line 13, 14*. Finally, we select the top individuals to create a new population. This evolutionary procedure ends with a set of Pareto front solutions.

In the next a few sections, we introduce the detailed design of representation and operators.

Algorithm 16: *NS-GGA* for multi-objective *RAC*

Input : A set of VM types, A set of containers, A set of PM instances,

Output: The allocation of containers

```
Initialize a population P with individuals;
while Termination Condition is not meet do
    for Each individual do
        Evaluate the fitness values;
    end
    while children number is less than the population size do
        Apply binary tournament selection to select two parents;
        Apply crossover over the selected parents;
        Apply mutation on two children;
        Add the children into a new population U;
    end
    evaluate individuals from U;
    non-dominated sorting of {P ∪ U};
    calculate crowding distance of {P ∪ U};
    P ← select population size of individuals from {P ∪ U};
end
return the Pareto front of solutions;
```

5.5.2 Representation

The representation in *NS-GGA* has the same structure with the *GGA-RAC* approach proposed in Section 3.6, Chapter 3. A solution consists of a list of PM instances hosting VM instances. Each VM instance hosts a list of containers. The sizes of these lists vary according to the allocation.

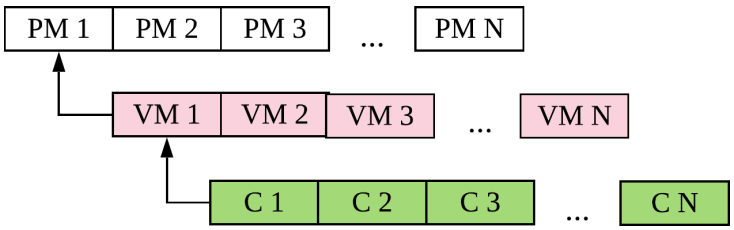


Figure 5.1: Group representation

Although both the *GGA-RAC* approach (in Section 3.6, Chapter 3) and the *NS-GGA* have the same representation, *GGA-RAC* focues only on the energy consumption while *NS-GGA* is a multi-objective algorithm that optimizes both objectives. Therefore, *NS-GGA* requires a different set of operators to achieve this goal.

5.5.3 Initialization

The initialization intends to create a diverse set of solutions. First, we randomly shuffle containers and use *First-Fit (FF)* heuristic to allocate them to a set of VM instances with random types (uniformly choose from a VM table). Then, the VM instances are allocated to a set of PM instances with *FF*. The use of *FF* guarantees valid solutions as well as a consolidated VM/PM allocation. The initialization is the only operator that is the same with *GGA-RAC* because randomly generated solutions are diverse in both objectives.

5.5.4 Crossover

We propose a gene-level crossover where PM instances on the chromosome are sorted, pair-wisely compared and preserved (see the flowchart in Figure.5.2). In the first step, the PM instances of a chromosome are sorted under a criterion, such as CPU utilization or duplication number (introduced later). Then, two PM instances p and p' from two chromosomes are compared pair-wisely (e.g., CPU utilization). The winning PM instance preserves all its content, including all VM instances and containers to the child. Before copying the containers, we need to check whether these containers have been allocated. Only the unallocated containers are copied so that the child solution does not validate the constraint on containers (Eq. 3.5). If one parent has more PM instances than the other, the extra PM instances are copied to the child as well. In the end, some containers may be unallocated and free to be allocated. These *free containers* are allocated with the *rearrangement operator*. After all containers have been allocated to the child, the empty PM and VM instances in the child are removed.

In order to optimize the two objectives, we apply the crossover twice to generate one child with each of the two sorting criteria. The first criterion focuses on optimizing energy consumption. It considers the CPU utilization of PM instances and prefers higher utilization. The heuristic considers that a good solution contains PM instances with higher CPU utilization. The second criterion intends to improve availability. Therefore, it favors PM instances with smaller duplication numbers. The duplication number is the total number of containers hosted by this PM instance that belong to the same micro-service. The PM instance with a high duplication number is undesirable because it leads to a high failure rate of applications.

The crossover procedure is similar to the crossover in Chapter 3.6.4. Hence, it is not repeated.

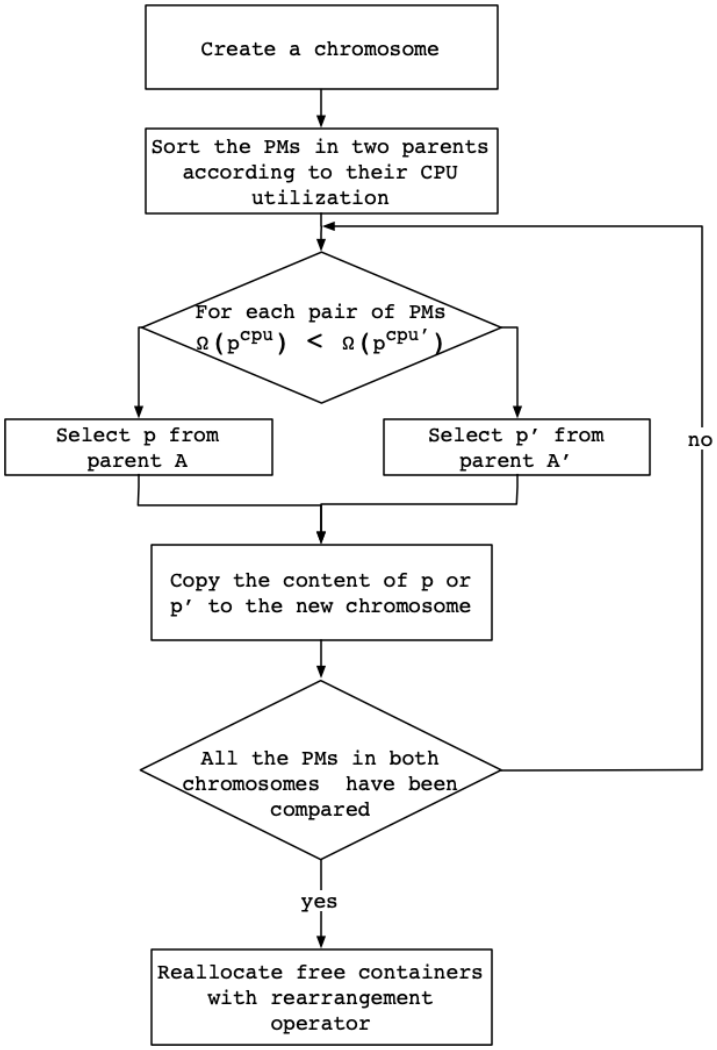


Figure 5.2: Flowchart of gene-level crossover for *NS-GGA*.

5.5.5 Rearrangement

To optimize both objectives, rearrangement (see Algorithm.17) inserts free containers into PM instances using two methods, i.e., an energy-aware method, and an availability-aware method. Rearrangement randomly selects (50% of chance) a method from the above two methods to insert containers (*line 2*). The energy-aware method (*line 3 to line 11*) attempts to

Algorithm 17: Rearrangement operator

Input : a target container, a list of PM instances,
Output: a list of PM instances

```
u ← Randomly selects from [0, 1];
if u > 0.5 then
    Sort the containers in all VM instances according to Eq.5.17 in
     ascending order;
    for each VM instance do
        if the two smallest containers in each VM instance can be
         replaced by the target container then
            Replace two containers with the target VM instance;
            Allocate two containers with FF&RC/FF;
        end
    end
    Allocate the target container with FF&RC/FF;
end
else
    for each PM instance do
        if the target container does not share a micro-service with any
         container within the PM instance then
            Replace a container that has duplicates;
            Allocate the container with FF&RC/FF;
        end
    end
end
return a list of PM instances;
```

replace two smaller containers with a larger free container and uses *FF* to allocate the smaller containers. We measure the size of a container using the product of a container's normalized utilization of resources (see

Eq. 5.17). The energy-aware method first sorts of containers in ascending order (*line 3*). This heuristic is based on a basic idea is that it is easy to allocate small items to bins. The availability-aware method (*line 12 to line 19*) intends to replace a duplicated container of a micro-service from a PM instance with a free container (defined in Section 5.5.4).

$$R = \frac{\zeta^{cpu}(c_i)}{\Omega^{cpu}(p_k)} \cdot \frac{\zeta^{mem}(c_i)}{\Omega^{mem}(p_k)} \tag{5.17}$$

5.5.6 Mutation

We design two functions in the mutation operator, *unpack* and *merge*. These two functions are executed in orders. The *unpack* function improves a solution by eliminating the inferior part of a solution. The function unpacks the inferior PM instances according to two criteria, i.e., the PM instances with low CPU utilization and the PM instances with high duplication numbers.

The *unpack* operator first sorts the PM instances according to CPU utilization (descending) or duplication number (ascending). The operator unpacks the PM instances in a roulette wheel style (see Algorithm 18). That is, the lower-ranking PM instances have a higher chance of being unpacked. In the beginning, each PM instance in the list is allocated with a probability (*line 3*). Then, a randomly generated number of u is compared with the probability. If u is smaller than the probability, the PM instance is unpacked, and all its containers are released and added to the free container list $cList$ (*line 11*).

$$probability = \frac{1 - \Omega^{cpu}(p_k)}{\sum_{k=1}^{K} 1 - \Omega^{cpu}(p_k)} \tag{5.18}$$

The second function of mutation is merged. *Merge* replaces small VM instances with a larger one. Hence PM instances could release more VM overheads. It also has two alternative ways; the first one merges two smallest VM instances in a PM with a large type of VM without violat-

Algorithm 18: Unpack procedure

Input : a list of PM instances $pmList$

Output: a list of PM instances $pmList$, a set of free containers $cList$

```
1  cList ← null;
2  for Each PM instance p in the pmList do
3  |  p ← probability(p) (see Eq.5.18);
4  end
5  for Each PM instance p do
6  |  u ← Random();
7  |  if u < p then
8  |  |  cList ← release(p);
9  |  end
10 end
11 return pmList and cList;
```

ing the resource constraint. An alternative way is to enlarge the smallest VM instance with a large type selected randomly.

5.5.7 Fitness Assignment

The fitness assignment includes three steps. The first step calculates the objective values. The two objective functions are introduced in the previous section (see Section 5.3). The energy consumption is calculated according to Eq. 5.19 and the availability is calculated according to Eq. 5.20 (see the equations below). The second step ranks the individuals into multiple fronts. Based on the fitness values of two objectives, the *NS-GGA* sorts the individuals with fast non-dominated sorting and ranked the individuals into fronts. In each front, the individuals are non-dominated to each other. The first front holds the best individuals. In the third step, crowding distance measures the density of individuals surrounding a particular individual.

$$Fitness\ 1 = \sum_{k=1}^{K} E_k, \tag{5.19}$$

$$Fitness\ 2 = 1 - \frac{\sum_{s=1}^{S} \Lambda(a_s)}{S} \tag{5.20}$$

With these steps, a selection operator prefers the individual with a lower rank. If two individuals are from the same front, the selection prefers the individual with larger crowding distance.

5.6 Experiments and Results

The goal of the experiment is to test the performance of the proposed algorithm in terms of two objective values: energy consumption and availability. We conduct experiments and compare our proposed algorithms with three benchmark algorithms, two rule-based *FF&BF/FF* [133,245] approach and a *Spread* [2] (a method in Kubernetes), and a multi-objective *Non-dominated Sorting dual-chromosome Genetic Algorithm (NS-DGA)* approach extended from the *DGA* approach proposed in Section 3.5.2, Chapter 3. Note that there is no existing multi-objective approach to the multi-objective *RAC* problem.

5.6.1 Benchmark Algorithms

FF&BF/FF [133, 245] uses three heuristics to allocate containers. It uses First Fit heuristics to allocate both containers and VM instances and applies a *Best Fit (BF)* for selecting VM types for creating new VM instances. Whenever no available VM instance can host a container, the *BF* selects a type of VM which has just enough resource to host the container.

Spread [2] is an approach provided by an open-source container management tool *Kubernetes*. The simple rule tries to allocate containers from

the micro-services to different PM instances so that it maximizes the availability of micro-services. *Spread* iteratively goes through PM instances and uses *FF* to select a VM instance to allocate the container. If no VM instance is available, it will create a VM instance with just enough resources. If no PM instance is available, a new PM instance is created. After allocating a container to a PM instance, it always avoids allocating the containers of the same micro-service to the same PM instance.

NS-DGA-FF combines the original NSGA-II and our proposed single-objective *Dual-chromosome GA* with First-Fit decoding (see Section 3.5.2. In particular, we apply the procedure from NSGA-II and the representation from *DGA-FF*. The genetic operators are also from *DGA-FF*. In *DGA-FF*, an individual requires a decoding process to construct a dual-chromosome into a solution. This approach uses a dual chromosome representation, which includes two vectors; one represents a permutation of containers, the other represents the selected VM types. In this case, we applied First-Fit (FF) decoding because the *DGA-FF* is competitive with *GGA-RAC* while *DGA-NF* is far worse than *GGA-RAC*. In previous Chapter 3, we have shown that the *DGA-FF* achieved a worse performance in energy consumption compared with *GGA-RAC* approach. However, However, because of the different selection criteria between single- and multi-objective GA algorithms, *DGA-FF* does not necessary performance badly in multi-objective problem. Hence, we still compare the *NS-GGA* with the multi-objective *NS-DGA-FF*.

5.6.2 Performance Metrics

To compare two multi-objective approaches, *NS-GGA* and *NS-DGA-FF*, we use hypervolume [252] and Inverted Generational Distance (IGD) values [214] as introduced in Chapter 2.2.3.

The calculation of hypervolume requires a reference point. In this case, we use (1, 1) because we are minimizing both objectives. We normalize

the fitness values for both objectives into the [0, 1] range with the linear normalization (Eq.5.21):

$$normalized\ x = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{5.21}$$

where x is the fitness value, x_{max} and x_{min} is the maximum and minimum value of x. In our experiments, we use the maximum and minimum values found in the experiments as x_{max} and x_{min}. This is because the optimum energy consumption is unknown.

Calculating the IGD value needs a true Pareto front. For our problem, the true Pareto front is unknown. Therefore, an approximated Pareto front is produced by combining all the solutions produced by the two compared algorithms and then applying a non-dominated sorting to obtain the final non-dominated set. The approximated Pareto front dominates all the other solutions we found.

5.6.3 Datasets and Test Instances

We design 8 test instances (see Table.5.2) with increasing number of applications from 50 to 200. For each application, we generate a maximum of 5 micro-services. Each micro-service has several replicas/containers selected from 2 to 5. We use a real-world application trace (AuverGrid trace [189]). We generate the containers using the same way as in Section 3.7.1, Chapter 3. We set a crush rate of 2.5% for PM instances.

For the settings of PM and VM types, we assume homogeneous PM instances which have eight cores and the total capacity is [13200 MHz, 16000 MB]. The maximum energy consumption for the PM is set to 540 KWh. This setting has been used in [133]. We design two sets of VM types (see Table 5.3), i.e., real-world VM types (20 types from Amazon EC2) and a synthetic set of VM types (10 types). The real-world VM types are proportional, while the synthetic ones are randomly generated. The CPU and memory of synthetic VM types are sampled from [0, 3300 MHz]

and [0, 4000 MB], representing the capacity of one core.

Table 5.2: test instances

instance	VM types	number of applications
1		50
2	synthetic	100
3	VM types	150
4		200
5		50
6	real-world	100
7	VM types	150
8		200

5.6.4 Parameter Settings

The parameter settings for *NS-DGA-FF* is listed in Table 5.4. In addition to our proposed operators, we apply the elitism [30] with size 5 and tournament selection [144] with size 2. These methods are standard and widely applied.

All algorithms were implemented in Java version 8, and the experiments were conducted on i7-4790 3.6 GHz with 8 GB of RAM running Linux Arch 4.14.15. We applied the Wilcoxon rank-sum test to test the statistical significance.

5.6.5 Experiment Results and Analysis

The overall performance of our proposed *NS-GGA* is much better than the other three algorithms. Figure.5.3 and 5.4 show the performance of solutions found in two single-objective algorithms and our proposed *NS-GGA*. Figure.5.6 and 5.7 compare the performance between two multi-objective

Table 5.3: VM types

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]
synthetic VM types			
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]

approaches, *NS-DGA-FF* and *NS-GGA*. Table 5.5 shows the detailed comparison of hypervolume and IGD values from two multi-objective approaches.

Our proposed *NS-GGA* is much better than the two baseline algorithms, *Spread* and *FF&BF/FF*. In Figure.5.3 and 5.4, we plot the best results of 30 runs from *NS-GGA* and solutions from rule-based approaches. As we minimize both energy consumption and the failure probability, better results are closer to the original point. These results show a similar pattern. Firstly, *Spread* always has the best availability throughout all test cases. In

Table 5.4: Parameter Settings

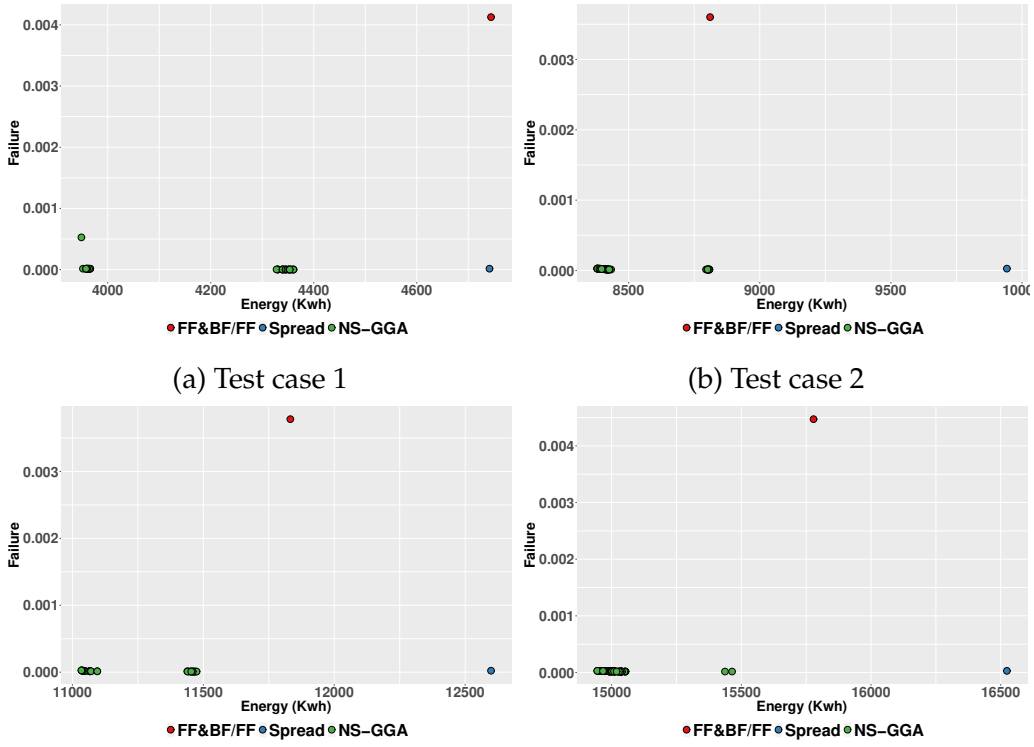
Parameter	Description
mutation rate	0.1
crossover rate	0.7
elitism	top 5 individuals
Number of generations	100
Population	100
Selection	tournament selection (size = 2)

the meanwhile, it also performs poorly in terms of energy consumption. Secondly, *FF&BF/FF* always gains the worst availability. Thirdly, our proposed *NS-GGA* achieves the best results in both objectives.

Spread is best in availability and performed poorly in energy consumption because it allocates containers as spread as possible. In the meanwhile, it does not consider the number of PM instances and uses as many as possible. Consequently, *Spread* uses much more PM instances than other algorithms.

FF&BF/FF has a high energy consumption and failure probability due to two disadvantages. Firstly, using *FF* to allocate containers according to the original sequence, applications by applications, cause most containers for the same micro-services are allocated in the same PM instance. Hence, the failure probability is high. Secondly, *BF* selects the smallest possible type of VM to allocate a container. This strategy creates many instances of small VM instances that cause a large amount of VM overheads and fragmented VM instance resources that cannot be used. The number of VM instances can be seen in Figure.5.5. *FF&BF/FF* creates the biggest number of VM instances and then followed by *Spread*. Both multi-objective approaches use much fewer VM instances.

Comparing with two multi-objective approaches, our proposed *NS-GGA* dominate *NS-DGA-FF*. Figure 5.6 and 5.7 show the results from a



(a) Test case 1

(b) Test case 2

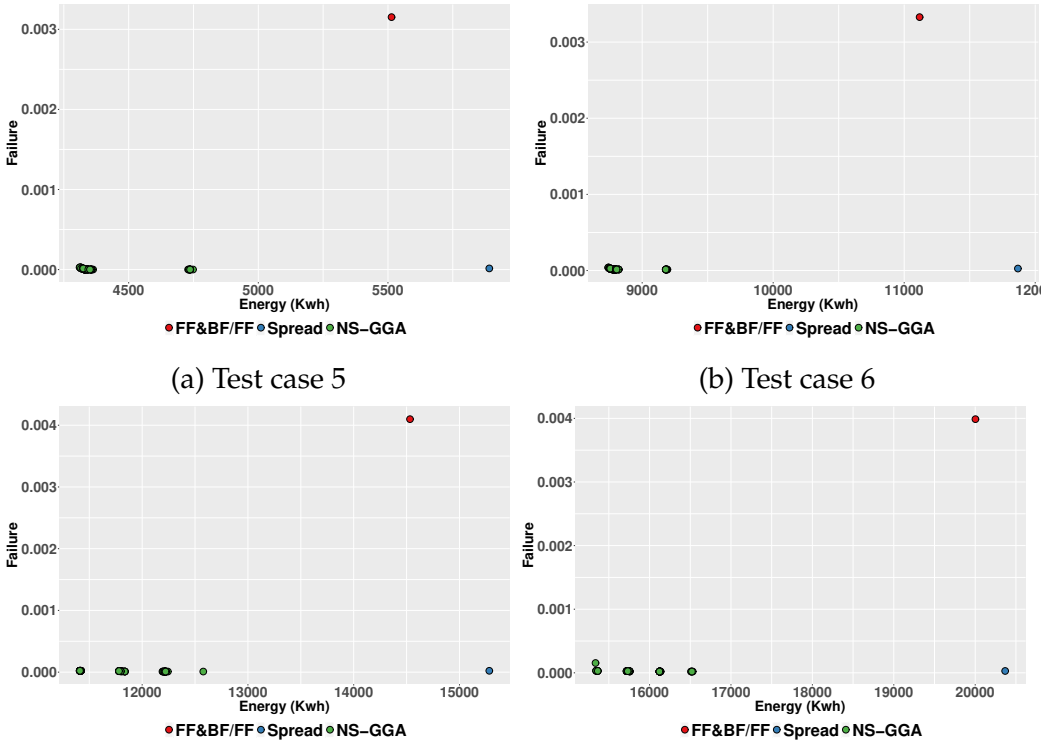
(c) Test case 3

(d) Test case 4

Figure 5.3: The best solutions found in three algorithms in test cases 1 to 4.

single run from two algorithms. The run includes the median hypervolume value among all 30 runs. *NS-GGA* clearly achieves better convergence than the *NS-DGA-FF* in all test cases. In Table 5.5, we can observe that *NS-GGA* performs better in both the hypervolume (bigger the better) and IGD value (smaller the better). The bigger hypervolume values indicate the better convergence of *NS-GGA*. The smaller IGD values refer to better coverage of the solutions from *NS-GGA* to the true Pareto front. The coverage means that the solutions from *NS-GGA* can cover more regions on the true Pareto front than *NS-DGA-FF*.

The main reason that causes the outperformance of *NS-GGA* over *NS-DGA-FF* is that *NS-GGA* uses a group representation. In *NS-DGA-FF*, because the vector-based representation needs to be decoded to evaluate,



(a) Test case 5

(b) Test case 6

(c) Test case 7

(d) Test case 8

Figure 5.4: The best solutions found in three algorithms in test cases 5 to 8.

the search and evaluation are separated in genotype space and phenotype space. Human-designed heuristics can hardly be used in the search process because of this separation. *NS-DGA-FF* can only rely on stochastic search without any domain knowledge. In contrast, our group representation does not require a decoding process. Hence, it is easy to embed heuristics in the operators to improve the performance, such as the switch of containers in the rearrangement operator.

It is easy to observe that during the evolution process both objectives are improving. Figure.5.8 shows the evolution of the Pareto front from test case 8. Different colors of circles (from red to orange) represent the solutions from generation 1 to 100. In the beginning (red circles), both the solutions are much worse in both objectives. Later generations of solutions

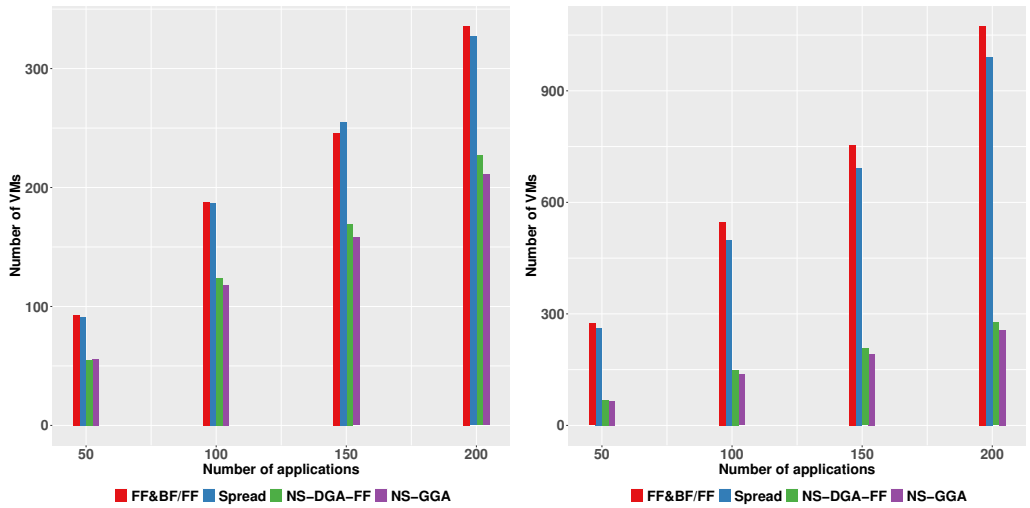


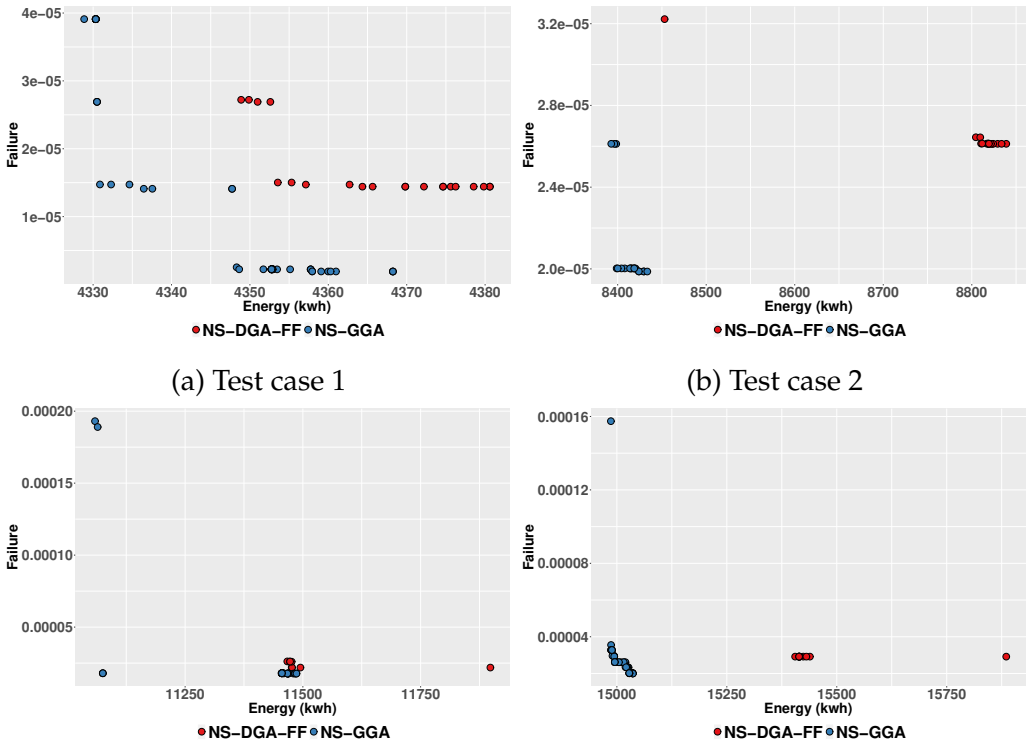
Figure 5.5: Number of VM instances that four algorithms used in the synthetic VM types (right) and real-world VM types (left).

are pushed towards the original point, and we may observe the solutions are converging with more and more solutions are overlapping.

5.7 Chapter Summary

The overall goal of this chapter was to propose a *NS-GGA* approach to solve the multi-objective *resource allocation in container-based clouds (RAC)* problem. Three objectives were accomplished to achieve this goal. (1) We proposed a multi-objective *RAC* problem model; (2) Our proposed *NS-GGA* adopts a group-based representation and embedded with bin-packing heuristics in the genetic operators. (3) We run experiments on real-world datasets with comparison with three state-of-the-art algorithms: *FF&BF/FF*, *Spread*, and a *NS-DGA*. The results show that our proposed *NS-GGA* approach outperforms all other approaches in both objectives. Also, our approach provides a set of solutions that has a trade-off between energy consumption and availability.

For *cloud providers*, our proposed *NS-GGA* approach offers an advan-



(a) Test case 1

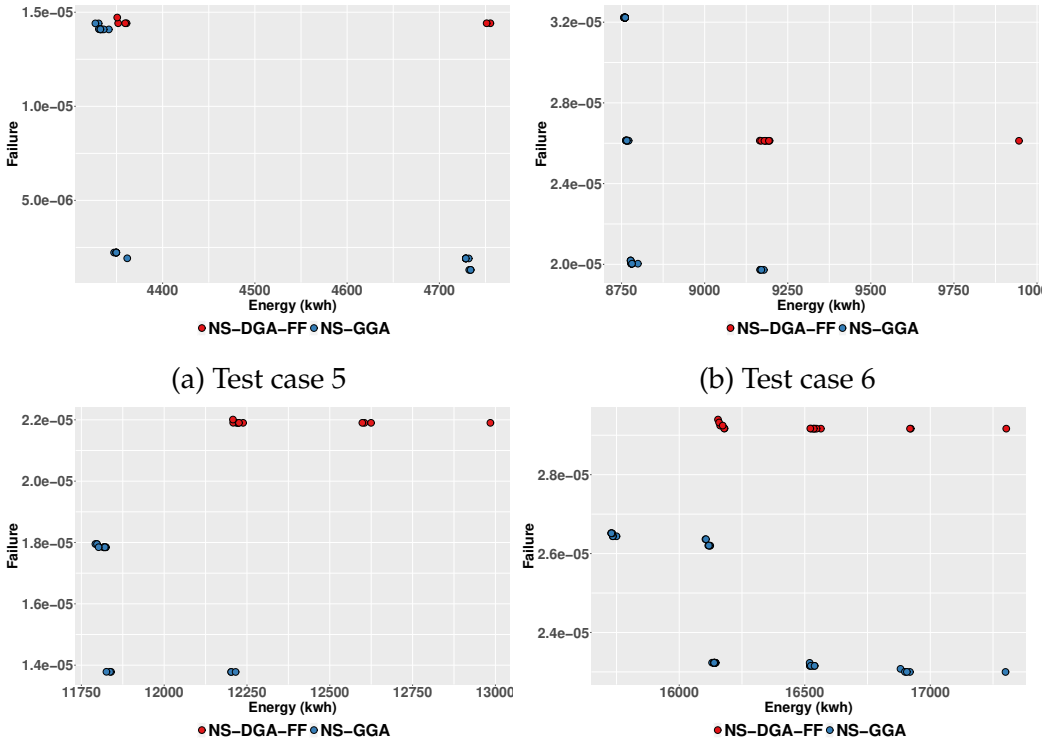
(b) Test case 2

(c) Test case 3

(d) Test case 4

Figure 5.6: The median solutions found in *NS-GGA* and *NS-DGA-FF* in test cases 1 to 4.

tage. It provides a set of solutions that has a trade-off between energy consumption and availability. *Cloud providers* can select a solution based on their preferences.



(a) Test case 5

(b) Test case 6

(c) Test case 7

(d) Test case 8

Figure 5.7: The median solutions found in *NS-GGA* and *NS-DGA-FF* in test cases 5 to 8.

Table 5.5: The comparison of hypervolume and IGD values between *NS-DGA-FF* and *NS-GGA*

instance	Method	HV (avg ± sd)	IGD (avg ± sd)	instance	Method	HV (avg ± sd)	IGD (avg ± sd)
instance 1	NS-GGA	**0.64 ± 0.19**	**0.15 ± 0.06**	instance 5	NS-GGA	**0.97 ± 0.00**	**0.02 ± 0.02**
	NS-DGA-FF	0.50 ± 0.00	0.2 ± 0.003		NS-DGA-FF	0.91 ± 0.06	0.04 ± 0.06
instance 2	NS-GGA	**0.96 ± 0.14**	**0.03 ± 0.03**	instance 6	NS-GGA	**0.97 ± 0.00**	**0.01 ± 0.005**
	NS-DGA-FF	0.69 ± 0.20	0.21 ± 0.16		NS-DGA-FF	0.81 ± 0.11	0.13 ± 0.09
instance 3	NS-GGA	**0.78 ± 0.20**	**0.12 ± 0.12**	instance 7	NS-GGA	**0.82 ± 0.05**	**0.04 ± 0.01**
	NS-DGA-FF	0.49 ± 0.00	0.31 ± 0.006		NS-DGA-FF	0.64 ± 0.08	0.15 ± 0.05
instance 4	NS-GGA	**0.95 ± 0.15**	**0.02 ± 0.01**	instance 8	NS-GGA	**0.82 ± 0.05**	**0.09 ± 0.02**
	NS-DGA-FF	0.50 ± 0.06	0.39 ± 0.05		NS-DGA-FF	0.66 ± 0.06	0.18 ± 0.04

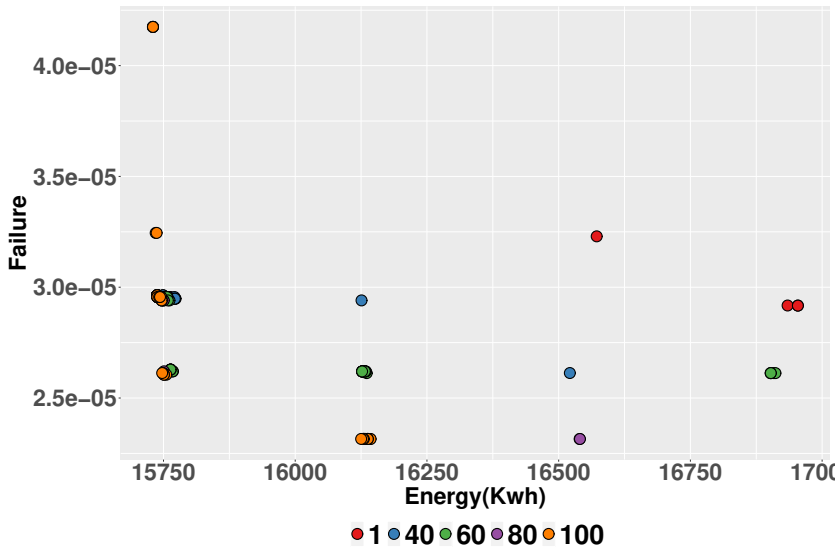


Figure 5.8: The evolution of Pareto front in *NS-GGA* from test case 8 run 27.

Chapter 6

Conclusions

The overall goal of this thesis was to improve the performance (e.g., energy efficiency and availability of applications) of the *resource allocation in container-based clouds (RAC)*. This goal was fulfilled by proposing evolutionary computation (EC) techniques combined with heuristics for different resource allocation problem scenarios, namely **off-line *RAC* problem**, **on-line *RAC* problem**, and **multi-objective *RAC* problem**. For each scenario, we first proposed a formal model that captures the key features of the studied problem. Then, we developed and analyzed appropriate EC-based algorithms based on the characteristics of these scenarios. Specifically, we investigated and proposed different representations, decoding strategies, and specialized genetic operators for solving these problems. These proposed algorithms were compared with state-of-the-art approaches using real-world datasets. Finally, we analyzed the advantages and disadvantages of each approach and provided insights to *cloud providers* and algorithm designers.

This remainder of this chapter is organized as follows. Section 6.1 outlines the objectives that were achieved in this thesis. Section 6.2 presents the main conclusions reached in this work. Section 6.3 explores possible future work directions.

6.1 Achieved Objectives

The following objectives were achieved in this thesis.

1. This research starts from the modeling of the off-line *RAC* problem in Chapter 3. This model formally defines the objective, variables, and constraints of the two-level *RAC* problem. Then, this chapter proposes three genetic algorithms (GA)-based approaches for solving the off-line *RAC* problem. These approaches are developed based on two types of representations, vector-based and group-based. Specifically, the vector-based representation encodes resource allocation with vectors of numbers. This representation requires a decoding process to translate an individual into a solution. We develop two approaches, the *single-chromosome GA (SGA)* and *dual-chromosome GA (DGA)*. Different representations and decoding processes were proposed and discussed. Then, we investigated the group-based representation and proposed a *group-based GA (GGA)* approach. One of the key contributions of the *GGA* is that the genetic operators, which are embedded with bin-packing heuristics, can modify the group structures both at the containers–VMs level and the VMs–PMs level. Experiments show that the *DGA* is fast and good at solving small-scale allocation problems (e.g., fewer than 500 containers). The *GGA* approach, although having a slower convergence than the *DGA*, can find better solutions for large-scale problems.

2. This thesis has proposed hyper-heuristics for the on-line *RAC* problem. To the best of our knowledge, it is the first time hyper-heuristic algorithms are used to automatically generate heuristics for resource allocation in clouds (Chapter 4). Instead of directly solving an on-line allocation problem, the proposed hyper-heuristic algorithm can generate a heuristic, that can generate a solution on-line. Hyper-heuristic approaches are particularly promising in solving on-line problems because they are trained off-line and can be used on-line.

Additionally, hyper-heuristics are able to learn the structure of good heuristics automatically. We propose two approaches, a genetic programming hyper-heuristic (GPHH) approach, namely *GPHH-RAC*, and a cooperative coevolution GP (CCGP) approach, namely *CCGP-RAC*. New terminal sets and training procedures were developed for these approaches. Experiments show that both *GPHH-RAC* and *CCGP-RAC* approaches can achieve better performance than human-design heuristics. In addition, *CCGP-RAC* achieves the best performance by evolving heuristics for resource allocation on both containers–VMs and VMs–PMs levels. The analysis shows that the inflexible design of human-designed on-line allocation heuristics can lead to low utilization of PMs (known as VM sprawl). On the contrary, the generated heuristics could discover non-linear functions to decide an allocation based on given features (e.g., remaining resources in VMs/PMs), which are difficult for human experts to manually design.

3. This thesis has proposed a multi-objective approach, *NS-GGA*, to solving the multi-objective *RAC* problem with the objective of minimizing energy consumption and maximizing availability of applications (Chapter 5). The multi-objective problem in two-level container-based clouds had never been studied previously. We propose domain-specific genetic operators that can balance these two objectives and achieve better performance than the existing algorithms. Experimental results show that the *NS-GGA* approach can successfully evolve trade-off solutions for *cloud providers* to choose based on their preferences, which are significantly better than the solutions obtained by other single-objective approaches such as rule-based heuristic, and another multi-objective vector-based GA.

6.2 Conclusions

This section outlines the main contributions presented in this thesis. The above contributions are interconnected to address the research goal in Figure 6.1.

The rest of the subsections summarize these contributions. Firstly, three problem models were proposed from Chapter 3 to Chapter 5. The models of on-line *RAC* and multi-objective *RAC* problems are extended from the model of off-line *RAC* problem. In all scenarios studied in this thesis, the allocation algorithms optimize the objective of energy consumption. Secondly, vector-based and group-based representations were developed. Multiple GA operators were designed for these representations. Thirdly, hyper-heuristic-based methods (e.g., CCGP and GPHH) were proposed for the on-line problem. Lastly, a multi-objective algorithm, namely *NSGGA*, is proposed to optimize two objectives, energy consumption and availability of applications. Moreover, the models and algortihms can be extended to solve other similar problems, e.g., allocating workloads with additional objectives and constraints, e.g., network latency.

6.2.1 Problem Models

This thesis proposes three novel problem models for three problem scenarios. Four decision-making procedures, i.e. *VM selection*, *VM creation*, *PM selection*, and *PM creation*, were identified in the two-level *RAC* problem. We proposed formal models of the *RAC* problem, including objectives, variables, and constraints. The new relationships between containers, VMs, and their overheads, various types of VM, and an affinity constraint, i.e., Operating Systems, are considered in the models. These models can be used to evaluate allocation algorithms.

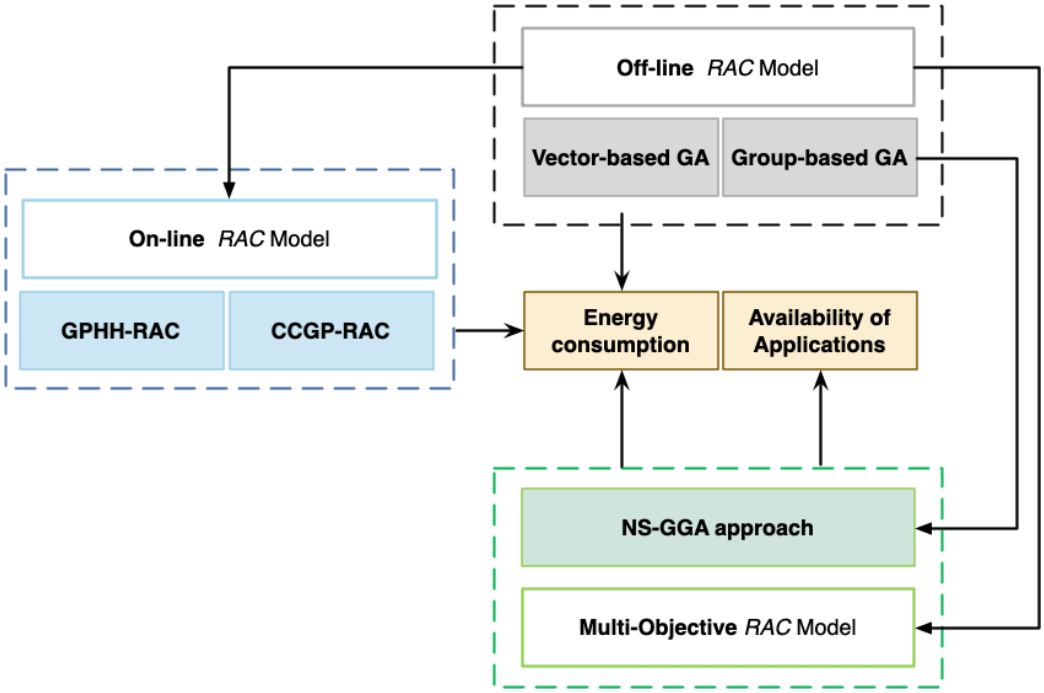


Figure 6.1: Illustration of thesis contributions.

6.2.2 Vector-based and Group-based Representations

This thesis proposes several novel schemes to represent the solutions of the off-line *RAC* problem so that they can be optimized using evolutionary computation algorithms. This includes a vector-based representation, where the allocation of containers and VMs are represented as vectors of integers. The vector-based representation must be decoded into a corresponding allocation in order to be evaluated. The major advantage of vector-based representation is its ability of using existing genetic vector-based genetic operators to search for solutions. The group-based representation is intuitive. Since it directly represents the allocation of containers, VM instances, and PM instances in groups, and does not require a decoding procedure. We have designed problem-specific genetic operators based on the group-based representation to evolve the solutions, meanwhile also ensuring the validity of the solutions. For algorithm design-

ers, the vector-based representation can be used to achieve fast allocation, since the existing operators can be applied. The group-based representation can achieve better performance if effective heuristics are used.

6.2.3 Complexity of Genetic Operations

This thesis also proposes novel problem-specific genetic operators to be used during the evolutionary processes. Specifically, in the *GGA-RAC* for the off-line *RAC* problem, novel bin-packing heuristics are designed to solve the two-level *RAC* problem. When compared with vector-based GAs that use generic search operators, our proposed *GGA-RAC* uses more effective problem-specific genetic operators to find solutions. Although the complexity of the proposed group-based operators is higher than the generic operators, the computation time is acceptable for off-line problems.

6.2.4 Hyper-Heuristic Framework

This thesis proposes two novel contributions in solving the on-line *RAC* problem. The first contribution is the *GPHH-RAC* approach using reservation-based heuristics. The novel approach combines two decision-making procedures, *VM selection* and *VM creation*, into a single procedure and uses the generated heuristics to decide whether a container is allocated to an existing VM instance or a new VM instance with selected type. The reservation-based heuristics are more flexible than AnyFit-based heuristics, where containers are always allocated to existing VMs first. The second contribution is the *CCGP-RAC* approach that can evolve two cooperative heuristics to solve the on-line *RAC* problem. Our on-line *RAC* approaches can automatically generate adaptive heuristics based on the historical workload data and the settings of available VMs without the domain knowledge or the involvement of domain experts.

6.2.5 Insights of Allocation Heuristics

This thesis analyzes both a manually designed rule and automatically generated heuristics to provide insights for *cloud providers* and algorithm designers. The analysis shows that human-designed heuristics can lead to the low utilization of Physical Machines (PMs) (known as VM sprawl). The reason is that these heuristics do not consider VM types and workload patterns. On the other hand, the generated heuristics contain non-linear functions of some problem-related features (e.g., remaining resources in VMs/PMs). It is impractical for human experts to manually identify these non-linear functions. This research also shows the interpretability of generated heuristics from *GPHH-RAC* and *CCGP-RAC*. The interpretability is important for *cloud providers* to understand why the heuristics are effective.

6.2.6 Independent Optimization Objectives

This thesis proposes the use of multi-objective evolutionary computing techniques for independently optimizing *RAC* with two objectives, minimizing energy consumption and maximizing availability of applications. Our novel approach combines the group-based representation and an NSGA-II framework. The proposed approach is compared at the industrial allocation algorithms and another multi-objective algorithm. Our proposed *NS-GGA* provides a set of solutions that has a better trade-off between energy consumption and availability. *Cloud providers* can select a solution based on their preferences.

6.3 Future work

Due to the scope of this research, there are still some areas for potential extensions and future work. This section briefly gives some directions to readers.

6.3.1 Hybrid Approach of Vector-based and Group-based Representation

Intuitively, a hybrid approach using both vector-based and group-based representations in an evolutionary algorithm may lead to better performance than applying them individually. As discussed in Chapter 3, vector-based, and group-based representations can be used to complement each other. Specifically, the vector-based representation can be used in exploration phases because it is easy and fast to search stochastically. The group-based representation, on the other hand, can be used in exploitation phases because domain knowledge and heuristics can be embedded into the local search operators (e.g., mutation). Hence, one way to employ both representations is to use a population that is initialized with vector-based represetation. During the local search phase, the local search operators were applied to the decoded population. Then, the population switches to vector-based representation again.

6.3.2 Time Sequence Analysis for Resource Requirement of Applications

Cloud environments are high dynamic, meaning that applications' resource requirements fluctuate over time. Our current work, including all three problem scenarios, makes the allocation decisions based on a single record of resource requirements, i.e., a pair of CPU and memory requirements of an application. For off-line scenarios, such as initial container allocation, it is a common approach because there is no monitoring data of an unallocated application. However, for on-line scenarios, the time sequence analysis can be useful because we need to avoid allocating applications with the same peak time. Furthermore, the time analysis can provide more evidence to develop strategies for proactive approaches. That is, we may handle the upcoming loads in advance. For developing hyper-heuristic approaches, time sequence-related features and novel training procedures

need to be introduced and developed.

6.3.3 Multi-Objective Hyper-Heuristic Approaches

On-line *RAC* may need to consider multiple optimization objectives. Currently, our proposed approaches in on-line scenarios focus on allocating independent containers. Similar to the multi-objective scenario in Chapter 5, the dependencies among containers could be considered. Hence, other QoS objectives, e.g., availability and the data transmission time, overall response time, need to be considered. Additionally, a multi-objective genetic programming based hyper-heuristic method can be developed to generate allocation heuristics automatically. In order to handle conflicting objectives, Had-MOEA [222] algorithm can be applied to explore the Pareto front of non-dominated allocation heuristics regarding the objectives mentioned above.

6.3.4 Lifelong Learning Hyper-heuristics for Allocation Heuristics

The resource requirement of applications changes over time. The automatically generated heuristics must also continuously evolve in order to remain effective. One way of doing this is to periodically use our proposed GPHH approaches on recently collected data to refine the rules. However, this also begs the question of how often *cloud providers* should refine the heuristics. Another approach of continuously evolving heuristics is developing an artificial immune system (AIS)-based approach [193]. AIS has been used to evolve allocation heuristics for 1D-bin packing problems, and it shows excellent performance [193]. AIS can continuously replace old heuristics with new ones when learning new tasks.

6.3.5 Location-aware Allocation in Container-based Clouds

Container-based clouds have become a popular choice for fog and edge computing [41, 42]. The container-based virtualization is lightweight and much flexible than VM-based clouds. This new combination of containers and Internet of Thing (IoT) brings new challenges. One of them is the additional consideration of geographical distances and network latencies between the correlated containers. New problem models will need to consider not only the resource allocation of containers but also the response time among locations. Also, the workflows of containers and replicas have a significant impact on the design of allocation algorithms.

Bibliography

[1] 2019 container adoption survey. https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf. Accessed: 2020-09-24.

[2] Advanced scheduling in kubernetes. https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes/. Accessed: 2019-12-12.

[3] Amazon ec2. https://aws.amazon.com/ec2/?nc1=h_ls. Accessed: 2020-04-27.

[4] Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/. Accessed: 2020-04-27.

[5] Amazon elastic container service. https://aws.amazon.com/ecs/. Accessed: 2020-1-22.

[6] Aws lambda, run code without thinking about servers. pay only for the compute time you consume. https://aws.amazon.com/lambda/. Accessed: 2020-1-22.

[7] Aws service level agreements (slas). https://aws.amazon.com/legal/service-level-agreements/?nc1=h_ls. Accessed: 2020-03-26.

[8] Function as a service. https://www.ibm.com/cloud/learn/faas. Accessed: 2020-03-26.

[9] Google app engine. https://cloud.google.com/appengine. Accessed: 2020-03-26.

[10] Openshift. https://www.openshift.com/. Accessed: 2020-1-22.

[11] 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.

[12] The state of container-based app development. https://www.ibm.com/downloads/cas/BBKLLK1L. Accessed: 2020-09-16.

[13] ABRAHAM, A., AND JAIN, L. *Evolutionary Multiobjective Optimization*. Springer, London, 2005, pp. 1–6.

[14] ADHIKARI, V. K., GUO, Y., HAO, F., VARVELLO, M., HILT, V., STEINER, M., AND ZHANG, Z. L. Unreeling netflix: Understanding and improving multi-CDN movie delivery. In *International Conference on Computer Communications (INFOCOM)* (2012), IEEE, pp. 1620–1628.

[15] AGRAWAL, R. B., DEB, K., AND AGRAWAL, R. Simulated binary crossover for continuous search space. *Complex systems 9*, 2 (1995), 115–148.

[16] AJIRO, Y., AND TANAKA, A. Improving packing algorithms for server consolidation. In *International Computer Measurement Group Conference (CMG)* (2007), vol. 253, Computer Measurement Group, pp. 399–406.

[17] ALLEN, S., BURKE, E. K., HYDE, M., AND KENDALL, G. Evolving Reusable 3D Packing Heuristics with Genetic Programming. In *Conference on Genetic and Evolutionary Computation (GECCO)* (2009), ACM, pp. 931–938.

[18] ARAKAKI, R. K., AND USBERTI, F. L. Hybrid genetic algorithm for the open capacitated arc routing problem. *Computers & Operations Research 90* (2018), 221–231.

[19] ASHLOCK, D., MCGUINNESS, C., AND ASHLOCK, W. *Representation in Evolutionary Computation*. Springer, Berlin, Heidelberg, 2012, p. 77–97.

[20] BÄCK, T., FOGEL, D. B., AND MICHALEWICZ, Z. *Handbook of evolutionary computation*. CRC Press, 1997.

[21] BALDINI, I., CASTRO, P., CHANG, K., CHENG, P., FINK, S., ISHAKIAN, V., MITCHELL, N., MUTHUSAMY, V., RABBAH, R., SLOMINSKI, A., ET AL. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[22] BANZHAF, W., NORDIN, P., KELLER, R. E., AND FRANCONE, F. D. *Genetic programming*. Springer, 1998.

[23] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Symposium on Operating systems principles (SOSP)* (2003), ACM, p. 164.

[24] BEAUMONT, O., EYRAUD-DUBOIS, L., AND LARCHEVÊQUE, H. Reliable service allocation in clouds. In *International Symposium on Parallel and Distributed Processing (IPDPS)* (2013), IEEE, pp. 55–66.

[25] BELOGLAZOV, A., ABAWAJY, J., AND BUYYA, R. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Computer Systems 28*, 5 (2011), 755–768.

[26] BELOGLAZOV, A., AND BUYYA, R. Adaptive Threshold-Based Approach for Energy-Efficient Consolidation of Virtual Machines in Cloud Data Centers. In *International Workshop on Middleware for Grids, Clouds and e-Science (MGC)* (2011), p. 6.

[27] BELOGLAZOV, A., AND BUYYA, R. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in Cloud data centers. *Concurrency Computation Practice and Experience 24*, 13 (2012), 1397–1420.

[28] BELOGLAZOV, A., AND BUYYA, R. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *IEEE Transactions on Parallel and Distributed Systems 24*, 7 (2013), 1366–1379.

[29] BERNSTEIN, D. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[30] BHANDARI, D., MURTHY, C., AND PAL, S. K. Genetic algorithm with elitist model and its convergence. *International journal of pattern recognition and artificial intelligence 10*, 06 (1996), 731–747.

[31] BONISSONE, P. P., SUBBU, R., EKLUND, N., AND KIEHL, T. R. Evolutionary algorithms + domain knowledge = real-world evolutionary computation. *IEEE Transactions on Evolutionary Computation 10*, 3 (2006), 256–280.

[32] BRAEKERS, K., RAMAEKERS, K., AND VAN NIEUWENHUYSE, I. The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering 99* (2016), 300–313.

[33] BRANKE, J., NGUYEN, S., PICKARDT, C. W., AND ZHANG, M. Automated Design of Production Scheduling Heuristics: A Review. *IEEE Transactions on Evolutionary Computation 20*, 1 (2016), 110–124.

[34] BURKE, E. K., HYDE, M., KENDALL, G., AND WOODWARD, J. A Genetic Programming Hyper-Heuristic Approach for Evolving 2-D Strip Packing Heuristics. *IEEE Transactions on Evolutionary Computation 14*, 6 (2010), 942–958.

[35] BURKE, E. K., HYDE, M. R., AND KENDALL, G. Evolving bin packing heuristics with genetic programming. In *Parallel Problem Solving from Nature (PPSN)* (2006), Springer, pp. 860—869.

[36] BURKE, E. K., HYDE, M. R., KENDALL, G., OCHOA, G., ÖZCAN, E., AND WOODWARD, J. R. *A Classification of Hyper-Heuristic Approaches: Revisited*. Springer, 2019, pp. 453–477.

[37] BURKE, E. K., HYDE, M. R., KENDALL, G., AND WOODWARD, J. Automating the packing heuristic design process with genetic programming. *Evolutionary computation 20*, 1 (2012), 63–89.

[38] BUYYA, R., GARG, S. K., AND CALHEIROS, R. N. Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In *International Conference on Cloud and Service computing* (2011), IEEE, pp. 1–10.

[39] CALZAROSSA, M. C., MASSARI, L., AND TESSERA, D. Workload characterization: A survey revisited. *ACM Computing Surveys 48*, 3 (2016), 1–43.

[40] CAO, J., JARVIS, S. A., SAINI, S., AND NUDD, G. R. Gridflow: Workflow management for grid computing. In *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)* (2003), IEEE, pp. 198–205.

[41] CELESTI, A., FAZIO, M., GIACOBBE, M., PULIAFITO, A., AND VILLARI, M. Characterizing cloud federation in IoT. In *International Conference on Advanced Information Networking and Applications Workshops (WAINA)* (2016), IEEE, pp. 93–98.

[42] CELESTI, A., MULFARI, D., FAZIO, M., VILLARI, M., AND PULIAFITO, A. Exploring container virtualization in IoT clouds. In *International Conference on Smart Computing (SMARTCOMP)* (2016), IEEE, pp. 1–6.

[43] CH., R. A java-based evolutionary computation research system. `https://cs.gmu.edu/~eclab/projects/ecj//`. Accessed: 2018-10-05.

[44] CHAWLA, Y., AND BHONSLE, M. Dynamically optimized cost based task scheduling in Cloud Computing. *International Journal of Emerging trends & technology in computer science 2*, 3 (2013), 38–42.

[45] CHEKURI, C., AND KHANNA, S. On multi-dimensional packing problems. In *Symposium on Discrete Algorithms (SODA)* (1999), ACM-SIAM, pp. 185–194.

[46] CHEN, J., CHIEW, K., YE, D., ZHU, L., AND CHEN, W. AAGA: Affinity-Aware Grouping for Allocation of Virtual Machines. In *International Conference on Advanced Information Networking and Applications (AINA)* (2013), IEEE, pp. 235–242.

[47] CHEN, J. C., WU, C.-C., CHEN, C.-W., AND CHEN, K.-H. Flexible job shop scheduling with parallel machines using Genetic Algo-

rithm and Grouping Genetic Algorithm. *Expert Systems with Applications 39*, 11 (2012), 10016–10021.

[48] CHOI, Y. H., AND KIM, J. H. Self-Adaptive Models for Water Distribution System Design Using Single-/Multi-Objective Optimization Approaches. *Water 11*, 6 (2019), 1293.

[49] CHRISTENSEN, H. I., KHAN, A., POKUTTA, S., AND TETALI, P. Approximation and online algorithms for multidimensional bin packing: A survey. *Computer Science Review 24* (2017), 63–79.

[50] COFFMAN, E. G., GAREY, M. R., AND JOHNSON, D. S. *Approximation Algorithms for Bin Packing: A Survey*. PWS Publishing Co., 1996, p. 46–93.

[51] COFFMAN JR., E. G., CSIRIK, J., GALAMBOS, G., MARTELLO, S., AND VIGO, D. *Bin Packing Approximation Algorithms: Survey and Classification*. Springer, 2013, pp. 455–531.

[52] D. SREENIVASAN, P. GAYATHRI, R. ANITHA, AND P. DHIVYA. Optimization of resource provisioning in cloud. *IEEE transactions on service computing 5*, 2 (2012), 14–16.

[53] DAWOUD, W., TAKOUNA, I., AND MEINEL, C. Elastic virtual machine for fine-grained cloud resource provisioning. In *International Conference on Computing and Communication Systems (I3CS)* (2011), Springer, pp. 11–25.

[54] DAYARATHNA, M., WEN, Y., AND FAN, R. Data center energy consumption modeling: A survey. *IEEE Communications Surveys Tutorials 18*, 1 (2016), 732–794.

[55] DE CAUWER, M., MEHTA, D., AND O'SULLIVAN, B. The Temporal Bin Packing Problem: An Application to Workload Management in

Data Centres. In *International Conference on Tools with Artificial Intelligence (ICTAI)* (2016), IEEE, pp. 157–164.

[56] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[57] DEB, K., PRATAP, A., AGARWAL, S., AND MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation 6*, 2 (2002), 182–197.

[58] DEELMAN, E., SINGH, G., SU, M.-H., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA, G., VAHI, K., BERRIMAN, G. B., GOOD, J., ET AL. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming 13*, 3 (2005), 219–237.

[59] DUA, R., RAJA, A. R., AND KAKADIA, D. Virtualization vs containerization to support PaaS. In *International Conference on Cloud Engineering (IC2E)* (2014), IEEE, pp. 610–614.

[60] EBERT, C., GALLARDO, G., HERNANTES, J., AND SERRANO, N. Devops. *Software 33*, 3 (2016), 94–100.

[61] FALKENAUER, E. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics 2*, 1 (1996), 5–30.

[62] FAN, C., WANG, Y., AND WEN, Z. Research on Improved 2D-BPSO-Based VM-Container Hybrid Hierarchical Cloud Resource Scheduling Mechanism. In *International Conference on Computer and Information Technology (ICCIT)* (2016), IEEE, pp. 754–759.

[63] FAN, X., WEBER, W.-D., AND BARROSO, L. A. Power provisioning for a warehouse-sized computer. *ACM SIGARCH Computer Architecture News 35*, June (2007), 13.

[64] FANG, D., LIU, X., LIU, L., AND YANG, H. TARGO: Transition and reallocation based green optimization for cloud VMs. In *International Conference on Green Computing and Communications (GreenCom)* (2013), IEEE, pp. 215–223.

[65] FARAHNAKIAN, F., LILJEBERG, P., AND PLOSILA, J. LiRCUP: Linear regression based CPU usage prediction algorithm for live migration of virtual machines in data centers. In *Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA)* (2013), IEEE, pp. 357–364.

[66] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An updated performance comparison of virtual machines and linux containers. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2015), IEEE, pp. 171–172.

[67] FERDAUS, H., AND MURSHED, M. R. N. C. R. B. Virtual machine consolidation in cloud data centers using ACO metaheuristic. In *European Conference on Parallel Processing (Euro-Par)* (2014), vol. 8632, pp. 306–317.

[68] FERRETO, T. C., NETTO, M. A., CALHEIROS, R. N., AND DE ROSE, C. A. Server consolidation with migration control for virtualized data centers. *Future Generation Computer Systems 27*, 8 (2011), 1027–1034.

[69] FOGEL, D. B. The Advantages of Evolutionary Computation. In *Biocomputing and Emergent Computation (BCEC)* (1997), World Scientific Press, p. 1–11.

[70] FORSMAN, M., GLAD, A., LUNDBERG, L., AND ILIE, D. Algorithms for automated live migration of virtual machines. *Journal of Systems and Software 101* (2015), 110–126.

[71] FREUND, R. F., GHERRITY, M., AMBROSIUS, S., CAMPBELL, M., HALDERMAN, M., HENSGEN, D., KEITH, E., KIDD, T., KUSSOW, M., LIMA, J. D., ET AL. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *Heterogeneous Computing Workshop (HCW)* (1998), IEEE, pp. 184–199.

[72] FREY, S., FITTKAU, F., AND HASSELBRING, W. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *International Conference on Software Engineering (ICSE)* (2013), pp. 512–521.

[73] FURMENTO, N., LEE, W., MAYER, A., NEWHOUSE, S., AND DARLINGTON, J. ICENI: an open grid service architecture implemented with Jini. In *ACM/IEEE Conference on Supercomputing (SC)* (2002), IEEE, pp. 37–37.

[74] GANESAN, R., SARKAR, S., AND NARAYAN, A. Analysis of SaaS business platform workloads for sizing and collocation. In *International Conference on Cloud Computing (CLOUD)* (2012), IEEE, pp. 868–875.

[75] GAO, C., WANG, H., ZHAI, L., GAO, Y., AND YI, S. An Energy-aware Ant Colony Algorithm for Network-aware Virtual Machine Placement in Cloud Computing. In *International Conference on Parallel and Distributed Systems (ICPADS)* (2016), IEEE, pp. 669–676.

[76] GAO, X., GU, Z., KAYAALP, M., PENDARAKIS, D., AND WANG, H. ContainerLeaks: Emerging security threats of information leakages in container clouds. In *International Conference on Dependable Systems and Networks (DSN)* (2017), IEEE, pp. 237–248.

[77] GAO, Y., GUAN, H., QI, Z., HOU, Y., AND LIU, L. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences 1*, 8 (2013), 1–13.

[78] GEROFI, B., VASS, Z., AND ISHIKAWA, Y. Utilizing memory content similarity for improving the performance of highly available virtual machines. *Future Generation Computer Systems 29*, 4 (2013), 1085–1095.

[79] GHOLIPOUR, N., ARIANYAN, E., AND BUYYA, R. A novel energy-aware resource management technique using joint vm and container consolidation approach for green computing in cloud data centers. *Simulation Modelling Practice and Theory 104* (2020), 102127.

[80] GIL, Y., DEELMAN, E., ELLISMAN, M., FAHRINGER, T., FOX, G., GANNON, D., GOBLE, C., LIVNY, M., MOREAU, L., AND MYERS, J. Examining the challenges of scientific workflows. *Computer 40*, 12 (2007), 24–32.

[81] GONG, Y.-J., CHEN, W.-N., ZHAN, Z.-H., ZHANG, J., LI, Y., ZHANG, Q., AND LI, J.-J. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing 34* (2015), 286–300.

[82] GU, J., HU, J., ZHAO, T., AND SUN, G. A new resource scheduling strategy based on genetic algorithm in cloud computing environment. *Journal of computers 7*, 1 (2012), 42–52.

[83] GUAN, X., WAN, X., CHOI, B., SONG, S., AND ZHU, J. Application oriented dynamic resource allocation for data centers using docker containers. *IEEE Communications Letters 21*, 3 (2017), 504–507.

[84] GUERRERO, C., LERA, I., AND JUIZ, C. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing 16*, 1 (2018), 113–135.

[85] GUERRERO, C., LERA, I., AND JUIZ, C. Resource optimization of container orchestration: a case study in multi-cloud microservices-

based applications. *The Journal of Supercomputing 74*, 7 (2018), 2956–
–2983.

[86] GUTIERREZ-GARCIA, J. O., AND SIM, K. M. A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling. *Future Generation Computer Systems 29*, 7 (2013), 1682–1699.

[87] GUZEK, M., BOUVRY, P., AND TALBI, E. G. A survey of evolutionary computation for resource management of processing in cloud computing. *IEEE Computational Intelligence Magazine 10*, 2 (2015), 53–67.

[88] HAMEED, A., KHOSHKBARFOROUSHHA, A., RANJAN, R., JAYARAMAN, P. P., KOLODZIEJ, J., BALAJI, P., ZEADALLY, S., MALLUHI, Q. M., TZIRITAS, N., VISHNU, A., ET AL. A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. *Computing 98*, 7 (2016), 751–774.

[89] HE, S., GUO, L., GUO, Y., WU, C., GHANEM, M., AND HAN, R. Elastic application container: A lightweight approach for cloud resource provisioning. In *International Conference on Advanced Information Networking and Applications (AINA)* (2012), IEEE, pp. 15–22.

[90] HILDEBRANDT, T., HEGER, J., AND SCHOLZ-REITER, B. Towards Improved Dispatching Rules for Complex Shop Floor Scenarios: A Genetic Programming Approach. In *Annual Conference on Genetic and Evolutionary Computation (GECCO)* (2010), ACM, pp. 257–264.

[91] HOLLAND, J. H. Outline for a Logical Theory of Adaptive Systems. *Journal of the ACM 9*, 3 (1962), 297–314.

[92] HOLT, C. C. Author's retrospective on 'forecasting seasonals and trends by exponentially weighted moving averages'. *International Journal of Forecasting 20*, 1 (2004), 11 – 13.

[93] HRUSCHKA, E. R., CAMPELLO, R. J., FREITAS, A. A., ET AL. A survey of evolutionary algorithms for clustering. *IEEE Transactions on Systems, Man, and Cybernetics 39*, 2 (2009), 133–155.

[94] HU, Y., DE LAAT, C., ZHAO, Z., ET AL. Multi-objective container deployment on heterogeneous clusters. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2019), pp. 592–599.

[95] ITURRIAGA, S., NESMACHNOW, S., DORRONSORO, B., TALBI, E. G., AND BOUVRY, P. A Parallel Hybrid Evolutionary Algorithm for the Optimization of Broker Virtual Machines Subletting in Cloud Systems. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)* (2013), pp. 594–599.

[96] JAYAMOHAN, M., AND RAJENDRAN, C. Development and analysis of cost-based dispatching rules for job shop scheduling. *European Journal of Operational Research 157*, 2 (2004), 307–321.

[97] JENNINGS, B., AND STADLER, R. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management 23*, 3 (2014), 567–619.

[98] JOHNSON, D. S. Fast algorithms for bin packing. *Journal of Computer and System Sciences 8*, 3 (1974), 272 – 314.

[99] KAAOUACHE, M. A., AND BOUAMAMA, S. Solving bin packing problem with a hybrid genetic algorithm for VM placement in cloud. *Procedia Computer Science 60* (2015), 1061–1069.

[100] KAEWKASI, C., AND CHUENMUNEEWONG, K. Improvement of container scheduling for docker using ant colony optimization. In *International Conference on Knowledge and Smart Technology (KST)* (2017), IEEE, pp. 254–259.

[101] KALRA, M., AND SINGH, S. A review of metaheuristic scheduling techniques in cloud computing. *Egyptian Informatics Journal 16*, 3 (2015), 275 – 295.

[102] KAUR, K., DHAND, T., KUMAR, N., AND ZEADALLY, S. Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers. *IEEE Wireless Communications 24*, 3 (2017), 48–56.

[103] KAUR, K., DHAND, T., KUMAR, N., AND ZEADALLY, S. Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers. *IEEE wireless communications 24*, 3 (2017), 48–56.

[104] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux Virtual Machine Monitor. In *The Ottawa Linux Symposium (OLS)* (2007).

[105] KLEIN, A., ISHIKAWA, F., AND HONIDEN, S. Sanga: A self-adaptive network-aware approach to service composition. *IEEE Transactions on Services Computing 7*, 3 (2013), 452–464.

[106] KOCH, S., AND WÄSCHER, G. A grouping genetic algorithm for the order batching problem in distribution warehouses. *Journal of Business Economics 86*, 1-2 (2016), 131–153.

[107] KOUSIOURIS, G., MENYCHTAS, A., KYRIAZIS, D., KONSTANTELI, K., GOGOUVITIS, S. V., KATSAROS, G., AND VARVARIGOU, T. A. Parametric design and performance analysis of a decoupled service-oriented prediction framework based on embedded numerical software. *IEEE Transactions on Services Computing 6*, 4 (2013), 511–524.

[108] KOZA, J. Genetic Programming: On the Programming of Computers by Means of Natural Selection. *Complex adaptive systems* (1992).

[109] KOZA, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing 4*, 2 (1994), 87–112.

[110] KOZHIRBAYEV, Z., AND SINNOTT, R. O. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems 68* (2017), 175–182.

[111] KRATZKE, N., AND QUINT, P.-C. Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software 126* (2017), 1–16.

[112] KUNKLE, D., AND SCHINDLER, J. A load balancing framework for clustered storage systems. In *High Performance Computing (HiPC)* (2008), Springer, pp. 57–72.

[113] LEE, Y. C., AND ZOMAYA, A. Y. Practical scheduling of bag-of-tasks applications on grids with dynamic resilience. *IEEE Transactions on Computers 56*, 6 (2007), 815–825.

[114] LEELIPUSHPAM, P. G. J., AND SHARMILA, J. Live vm migration techniques in cloud environment—a survey. In *Conference on Information & Communication Technologies (ICTD)* (2013), IEEE, pp. 408–413.

[115] LEGILLON, F., LIEFOOGHE, A., AND TALBI, E. G. CoBRA: A cooperative coevolutionary algorithm for bi-level optimization. *Congress on Evolutionary Computation (CEC)* (2012), 1–8.

[116] LIEN, C. H., BAI, Y. W., AND LIN, M. B. Estimation by software for the power consumption of streaming-media servers. *IEEE Transactions on Instrumentation and Measurement 56*, 5 (2007), 1859–1870.

[117] LIN, C., CHEN, J., LIU, P., AND WU, J. Energy-efficient core allocation and deployment for container-based virtualization. In *Interna-*

tional Conference on Parallel and Distributed Systems (ICPADS) (2018), IEEE, pp. 93–101.

[118] LIN, C.-C., LIU, P., AND WU, J.-J. Energy-efficient virtual machine provision algorithms for cloud systems. In *International Conference on Utility and Cloud Computing (UCC)* (2011), IEEE, pp. 81–88.

[119] LIN, M., XI, J., BAI, W., AND WU, J. Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud. *IEEE Access 7* (2019), 83088–83100.

[120] LIN, W., WANG, J. Z., LIANG, C., AND QI, D. A threshold-based dynamic resource allocation scheme for cloud computing. *Procedia Engineering 23* (2011), 695–703.

[121] LIU, B., LI, P., LIN, W., SHU, N., LI, Y., AND CHANG, V. A new container scheduling algorithm based on multi-objective optimization. *Soft Computing 22*, 23 (2018), 7741–7752.

[122] LIU, J., WANG, S., ZHOU, A., XU, J., AND YANG, F. Sla-driven container consolidation with usage prediction for green cloud computing. *Frontiers of Computer Science 14*, 1 (2020), 42–52.

[123] LIU, X.-F., ZHAN, Z.-H., DENG, J. D., LI, Y., GU, T., AND ZHANG, J. An energy efficient ant colony system for virtual machine placement in cloud computing. *IEEE Transactions on Evolutionary Computation 22*, 1 (2016), 113–128.

[124] LÓPEZ-CAMACHO, E., TERASHIMA-MARÍN, H., OCHOA, G., AND CONANT-PABLOS, S. E. Understanding the structure of bin packing problems through principal component analysis. *International Journal of Production Economics 145*, 2 (2013), 488–499.

[125] LORIDO-BOTRAN, T., MIGUEL-ALONSO, J., AND LOZANO, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing 12*, 4 (2014), 559–592.

[126] LU, C., YE, K., XU, G., XU, C., AND BAI, T. Imbalance in the cloud: An analysis on alibaba cluster trace. In *International Conference on Big Data (Big Data)* (2017), IEEE, pp. 2884–2892.

[127] MA, H., DA SILVA, A. S., AND KUANG, W. NSGA-II with local search for multi-objective application deployment in multi-cloud. In *Congress on Evolutionary Computation (CEC)* (2019), pp. 2800–2807.

[128] MAENHAUT, P.-J., VOLCKAERT, B., ONGENAE, V., AND DE TURCK, F. Resource management in a containerized cloud: Status and challenges. *Journal of Network and Systems Management* (2019), 1–50.

[129] MAHESWARAN, M., ALI, S., SIEGEL, H. J., HENSGEN, D., AND FREUND, R. F. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of parallel and distributed computing 59*, 2 (1999), 107–131.

[130] MAN, K.-F., TANG, K.-S., AND KWONG, S. *Genetic algorithms: concepts and designs*. Springer Science & Business Media, 2012.

[131] MANN, Z. Á. Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms. *ACM Computing Surveys 48*, 1 (2015), 1–34.

[132] MANN, Z. A. Approximability of VM allocation : Much harder than bin packing. *Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications (JHSDM)* (2015), 21–30.

[133] MANN, Z. Á. Interplay of virtual machine selection and virtual machine placement. In *Service-Oriented and Cloud Computing (ESOCC)* (2016), Springer, pp. 137–151.

[134] MANN, Z. Á. Resource optimization across the cloud stack. *IEEE Transactions on Parallel and Distributed Systems 29*, 1 (2018), 169–182.

[135] MANVI, S. S., AND SHYAM, G. K. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *Journal of network and computer applications 41* (2014), 424–440.

[136] MAO, Y., OAK, J., POMPILI, A., BEER, D., HAN, T., AND HU, P. Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *International Performance Computing and Communications Conference (IPCCC)* (2017), IEEE, pp. 1–8.

[137] MASHAYEKHY, L., NEJAD, M. M., GROSU, D., ZHANG, Q., AND SHI, W. Energy-aware scheduling of mapreduce jobs for big data applications. *IEEE transactions on Parallel and distributed systems 26*, 10 (2014), 2720–2733.

[138] MATTETTI, M., SHULMAN-PELEG, A., ALLOUCHE, Y., CORRADI, A., DOLEV, S., AND FOSCHINI, L. Securing the infrastructure and the workloads of linux containers. In *Conference on Communications and Network Security (CNS)* (2015), IEEE, pp. 559–567.

[139] MEI, Y., TANG, K., AND YAO, X. Decomposition-based memetic algorithm for multiobjective capacitated arc routing problem. *IEEE Transactions on Evolutionary Computation 15*, 2 (2011), 151–165.

[140] MEI, Y., ZHANG, M., AND NYUGEN, S. Feature Selection in Evolving Job Shop Dispatching Rules with Genetic Programming. In *Genetic and Evolutionary Computation Conference (GECCO)* (2016), ACM, pp. 365–372.

[141] MELL, P. M., AND GRANCE, T. The NIST definition of cloud computing. Tech. rep., National Institute of Standards and Technology, Gaithersburg, MD, Gaithersburg, MD, 2011.

[142] MENOUER, T., CÉRIN, C., AND LECLERCQ, É. New multi-objectives scheduling strategies in docker swarmkit. In *International Conference*

on Algorithms and Architectures for Parallel Processing (ICA3PP) (2018), Springer, pp. 103–117.

[143] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal 2014*, 239 (2014), 2.

[144] MILLER, B. L., GOLDBERG, D. E., ET AL. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems 9*, 3 (1995), 193–212.

[145] MISHRA, M., DAS, A., KULKARNI, P., AND SAHOO, A. Dynamic resource management using virtual machine migrations. *IEEE Communications Magazine 50*, 9 (2012), 34–40.

[146] MISHRA, M., AND SAHOO, A. On Theory of VM Placement: Anomalies in Existing Methodologies and Their Mitigation Using a Novel Vector Based Approach. In *International Conference on Cloud Computing (CLOUD)* (2011), IEEE, pp. 275–282.

[147] MORABITO, R., KJÄLLMAN, J., AND KOMU, M. Hypervisors vs. lightweight virtualization: a performance comparison. In *International Conference on Cloud Engineering (CLOUD)* (2015), IEEE, pp. 386–393.

[148] NADAREISHVILI, I., MITRA, R., MCLARTY, M., AND AMUNDSEN, M. *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, Inc., 2016.

[149] NAMIOT, D., AND SNEPS-SNEPPE, M. On micro-services architecture. *International Journal of Open Information Technologies 2*, 9 (2014), 24–27.

[150] NARDELLI, M., HOCHREINER, C., AND SCHULTE, S. Elastic Provisioning of Virtual Machines for Container Deployment. In *International Conference on Performance Engineering Companion (ICPE)* (2017), ACM, pp. 5–10.

[151] NARDELLI, M., HOCHREINER, C., AND SCHULTE, S. Elastic provisioning of virtual machines for container deployment (icpe). In *International Conference on Performance Engineering Companion* (2017), ACM, pp. 5–10.

[152] NGUYEN, S., ZHANG, M., JOHNSTON, M., AND TAN, K. C. Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming. *IEEE Transactions on Evolutionary Computation 18*, 2 (2014), 193–208.

[153] NGUYEN, S., ZHANG, M., AND TAN, K. C. Surrogate-assisted genetic programming with simplified models for automated design of dispatching rules. *IEEE transactions on cybernetics 47*, 9 (2017), 2951–2965.

[154] NIU, M., CHENG, B., FENG, Y., AND CHEN, J. Gmta: A geo-aware multi-agent task allocation approach for scientific workflows in container-based cloud. *IEEE Transactions on Network and Service Management 17*, 3 (2020), 1568–1581.

[155] ORZECHOWSKI, P., PROFICZ, J., KRAWCZYK, H., AND SZYMAŃSKI, J. Categorization of cloud workload types with clustering. In *International Conference on Signal, Networks, Computing, and Systems (ICSNCS)* (2017), Springer, pp. 303–313.

[156] PAHL, C., BOZEN-BOLZANO, L. U., AND PAHL, C. Containerisation and the PaaS Cloud Containerisation and the PaaS Cloud. *IEEE Cloud Computing 2*, September (2015), 24–31.

[157] PAHL, C., BROGI, A., SOLDANI, J., AND JAMSHIDI, P. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* (2017).

[158] PANDEY, S., WU, L., GURU, S. M., AND BUYYA, R. A particle swarm optimization-based heuristic for scheduling workflow appli-

cations in cloud computing environments. In *International conference on advanced information networking and applications (AINA)* (2010), IEEE, pp. 400–407.

[159] PANDIT, D., CHATTOPADHYAY, S., CHATTOPADHYAY, M., AND CHAKI, N. Resource allocation in cloud using simulated annealing. In *Applications and Innovations in Mobile Computing (AIMoC)* (2014), IEEE, pp. 21–27.

[160] PARK, J., MEI, Y., NGUYEN, S., CHEN, G., AND ZHANG, M. Investigating the generality of genetic programming based hyper-heuristic approach to dynamic job shop scheduling with machine breakdown. In *Artificial Life and Computational Intelligence (ACALCI)* (2017), Springer, pp. 301—313.

[161] PARK, J., MEI, Y., NGUYEN, S., CHEN, G., AND ZHANG, M. Evolutionary multitask optimisation for dynamic job shop scheduling using niched genetic programming. In *Advances in Artificial Intelligence (AI)* (2018), Springer, pp. 739—-751.

[162] PASCHKE, A., AND SCHNAPPINGER-GERULL, E. A Categorization Scheme for SLA Metrics. *Service Oriented Electronic Commerce 80*, 25-40 (2006), 25–40.

[163] PEREZ-BOTERO, D., SZEFER, J., AND LEE, R. B. Characterizing hypervisor vulnerabilities in cloud computing servers. In *International workshop on Security in cloud computing (SCC)* (2013), ACM, pp. 3–10.

[164] PHAN, D. H., SUZUKI, J., CARROLL, R., BALASUBRAMANIAM, S., DONNELLY, W., AND BOTVICH, D. Evolutionary multiobjective optimization for green clouds. In *The Genetic and Evolutionary Computation Conferences (GECCO)* (2012), ACM, pp. 19–26.

[165] PIRAGHAJ, S. F., CALHEIROS, R. N., CHAN, J., DASTJERDI, A. V., AND BUYYA, R. Virtual machine customization and task mapping

model for efficient allocation of cloud data center resources. *The Computer Journal 59*, 2 (2016), 208–224.

[166] PIRAGHAJ, S. F., DASTJERDI, A. V., CALHEIROS, R. N., AND BUYYA, R. A Framework and Algorithm for Energy Efficient Container Consolidation in Cloud Data Centers. In *International Conference on Green Computing and Communications (GreenCom)* (2015), IEEE, pp. 368–375.

[167] PIRAGHAJ, S. F., DASTJERDI, A. V., CALHEIROS, R. N., AND BUYYA, R. Efficient Virtual Machine Sizing for Hosting Containers as a Service, 2015.

[168] PIRAGHAJ, S. F., DASTJERDI, A. V., CALHEIROS, R. N., AND BUYYA, R. A survey and taxonomy of energy efficient resource management techniques in platform as a service cloud. In *Handbook of Research on End-to-End Cloud Computing Architecture Design*. IGI Global, 2017, pp. 410–454.

[169] POON, P. W., AND CARTER, J. N. Genetic algorithm crossover operators for ordering applications. *Computers & Operations Research 22*, 1 (1995), 135–147.

[170] POON, P. W., AND CARTER, J. N. Genetic algorithm crossover operators for ordering applications. *Computers & Operations Research 22*, 1 (1995), 135–147.

[171] PORTALURI, G., GIORDANO, S., KLIAZOVICH, D., AND DORRONSORO, B. A power efficient genetic algorithm for resource allocation in cloud computing data centers. In *International Conference on Cloud Networking (CloudNet)* (2014), IEEE, pp. 58–63.

[172] POTTS, C. N., AND STRUSEVICH, V. A. Fifty years of scheduling: a survey of milestones. *Journal of the Operational Research Society 60*, S1 (2009), S41–S68.

[173] PRAKASH, C., PRASHANTH, P., BELLUR, U., AND KULKARNI, P. Deterministic container resource management in derivative clouds. In *IEEE International Conference on Cloud Engineering (IC2E)* (2018), pp. 79–89.

[174] QIN, X., ZHANG, W., WANG, W., WEI, J., ZHAO, X., AND HUANG, T. Towards a cost-aware data migration approach for key-value stores. In *International Conference on Cluster Computing (CLUSTER)* (2012), pp. 551–556.

[175] QUIROZ-CASTELLANOS, M., CRUZ-REYES, L., TORRES-JIMENEZ, J., GÓMEZ, C., HUACUJA, H. J. F., AND ALVIM, A. C. A grouping genetic algorithm with controlled gene transmission for the bin packing problem. *Computers & Operations Research 55* (2015), 52–64.

[176] RADCLIFFE, N. Formal Analysis and Random Respectful Recombination. In *International Conference on Genetic Algorithms (CGA)* (1991), Morgan Kaufmann Publishers Inc., pp. 222–229.

[177] RAHIMI, M. R., VENKATASUBRAMANIAN, N., MEHROTRA, S., AND VASILAKOS, A. V. MAPCloud: Mobile applications on an elastic and scalable 2-tier cloud architecture. In *International Conference on Utility and Cloud Computing (UCC)* (2012), IEEE, pp. 83–90.

[178] RAJ, V. M., AND SHRIRAM, R. Power aware provisioning in cloud computing environment. In *International Conference on Computer, Communication and Electrical Technology (ICCCET)* (2011), IEEE, pp. 6–11.

[179] REZVANI, M., AKBARI, M. K., AND JAVADI, B. Resource allocation in cloud computing environments based on integer linear programming. *The Computer Journal 58*, 2 (2015), 300–314.

[180] RIQUELME, N., VON LÜCKEN, C., AND BARAN, B. Performance metrics in multi-objective optimization. In *Latin American Computing Conference (CLEI)* (2015), IEEE, pp. 1–11.

[181] RIQUELME, N., VON LUCKEN, C., AND BARAN, B. Performance metrics in multi-objective optimization. In *Latin American Computing Conference (CLEI)* (2015), IEEE, pp. 1–11.

[182] RONG, H., ZHANG, H., XIAO, S., LI, C., AND HU, C. Optimizing energy consumption for data centers. *Renewable and Sustainable Energy Reviews 58* (May 2016), 674–691.

[183] ROTHLAUF, F., AND GOLDBERG, D. E. *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, 2002.

[184] ŞAHIN, M., AND KELLEGÖZ, T. An efficient grouping genetic algorithm for u-shaped assembly line balancing problems with maximizing production rate. *Memetic Computing 9*, 3 (2017), 213–229.

[185] SAMPAIO, A. R., RUBIN, J., BESCHASTNIKH, I., AND ROSA, N. S. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications 10*, 1 (2019), 4.

[186] SARATHY, V., NARAYAN, P., AND MIKKILINENI, R. Next generation cloud computing architecture: Enabling real-time dynamism for shared distributed physical infrastructure. In *International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises* (2010), IEEE, pp. 48–53.

[187] SHARMA, P., CHAUFOURNIER, L., SHENOY, P., AND TAY, Y. Containers and virtual machines at scale: A comparative study. In *International Middleware Conference (Middleware)* (2016), ACM/IFIP, pp. 1–13.

[188] SHAW, S. B., AND SINGH, A. A survey on scheduling and load balancing techniques in cloud computing environment. In *International conference on computer and communication technology (ICCCT)* (2014), IEEE, pp. 87–95.

[189] SHEN, S., V. BEEK, V., AND IOSUP, A. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2015), IEEE/ACM, pp. 465–474.

[190] SHI, L., FURLONG, J., AND WANG, R. Empirical evaluation of vector bin packing algorithms for energy efficient data centers. In *Symposium on computers and communications (ISCC)* (2013), IEEE, pp. 9–15.

[191] SHI, T., MA, H., AND CHEN, G. Energy-aware container consolidation based on pso in cloud data centers. In *Congress on Evolutionary Computation (CEC)* (2018), IEEE, pp. 1–8.

[192] SIM, K., HART, E., AND PAECHTER, B. A hyper-heuristic classifier for one dimensional bin packing problems: Improving classification accuracy by attribute evolution. In *International Conference on Parallel Problem Solving from Nature (PPSN)* (2012), Springer, pp. 348–357.

[193] SIM, K., HART, E., AND PAECHTER, B. A lifelong learning hyper-heuristic method for bin packing. *Evolutionary computation 23*, 1 (2015), 37–67.

[194] SINGH, S., AND CHANA, I. Cloud resource provisioning: survey, status and future research directions. *Knowledge and Information Systems 49*, 3 (2016), 1005–1069.

[195] SINGH, S., AND CHANA, I. A survey on resource scheduling in cloud computing: Issues and challenges. *Journal of grid computing 14*, 2 (2016), 217–264.

[196] SOTELO-FIGUEROA, M. A., SOBERANES, H. J. P., CARPIO, J. M., FRAIRE HUACUJA, H. J., REYES, L. C., AND SORIA ALCARAZ, J. A. *Evolving Bin Packing Heuristic Using Micro-Differential Evolution with Indirect Representation*. Springer, 2013, pp. 349–359.

[197] SPEITKAMP, B., AND BICHLER, M. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE Transactions on services computing 3*, 4 (2010), 266–278.

[198] SPICUGLIA, S., CHEN, L. Y., BIRKE, R., AND BINDER, W. Optimizing capacity allocation for big data applications in cloud datacenters. In *International Symposium on Integrated Network Management (IM)* (2015), IEEE, pp. 511–517.

[199] SUGAVANAM, P., SIEGEL, H. J., MACIEJEWSKI, A. A., OLTIKAR, M., MEHTA, A., PICHEL, R., HORIUCHI, A., SHESTAK, V., AL-OTAIBI, M., KRISHNAMURTHY, Y., ET AL. Robust static allocation of resources for independent tasks under makespan and dollar cost constraints. *Journal of Parallel and Distributed Computing 67*, 4 (2007), 400–416.

[200] SUN, Y., LIN, F., AND XU, H. Multi-objective optimization of resource scheduling in fog computing using an improved nsga-ii. *Wireless Personal Communications 102*, 2 (2018), 1369–1385.

[201] SVÄRD, P., LI, W., WADBRO, E., TORDSSON, J., AND ELMROTH, E. Continuous datacenter consolidation. In *International Conference on Cloud Computing Technology and Science (CloudCom)* (2016), IEEE, pp. 387–396.

[202] TAN, B., HUANG, H., MA, H., AND ZHANG, M. Binary pso for web service location-allocation. In *Artificial Life and Computational Intelligence (AI)* (2017), Springer, pp. 366–377.

[203] TAN, B., MA, H., AND MEI, Y. A NSGA-II-based approach for service resource allocation in Cloud. In *Congress on Evolutionary Computation (CEC)* (2017), IEEE, pp. 2574–2581.

[204] TAN, B., MA, H., AND MEI, Y. A Genetic Programming Hyper-heuristic Approach for Online Resource Allocation in Container-Based Clouds. In *Australasian Joint Conference on Artificial Intelligence (AI)* (2018), Springer, pp. 146–152.

[205] TAN, B., MA, H., AND ZHANG, M. Optimization of location allocation of web services using a modified non-dominated sorting genetic algorithm. In *Australasian Conference on Artificial Life and Computational Intelligence (AI)* (2016), Springer, pp. 246–257.

[206] TAO, Y., WANG, X., XU, X., AND CHEN, Y. Dynamic resource allocation algorithm for container-based service computing. In *International Symposium on Autonomous Decentralized System (ISADS)* (2017), IEEE, pp. 61–67.

[207] TSAI, J.-T., FANG, J.-C., AND CHOU, J.-H. Optimized task scheduling and resource allocation on cloud computing environment using improved differential evolution algorithm. *Computers & Operations Research 40*, 12 (2013), 3045–3055.

[208] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KAGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[209] ULLMAN, J. D. Np-complete scheduling problems. *Journal of Computer and System sciences 10*, 3 (1975), 384–393.

[210] USMANI, Z., AND SINGH, S. A survey of virtual machine placement techniques in a cloud data center. *Procedia Computer Science 78* (2016), 491–498.

[211] VAN VELDHUIZEN, D. A. *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations*. PhD thesis, 1999.

[212] VAQUERO, L. M., RODERO-MERINO, L., AND BUYYA, R. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review 41*, 1 (2011), 45–52.

[213] VARASTEH, A., AND GOUDARZI, M. Server Consolidation Techniques in Virtualized Data Centers: A Survey. *IEEE Systems Journal 11*, 2 (2017), 772–783.

[214] VILLALOBOS-ARIAS, M. A., PULIDO, G. T., AND COELLO COELLO, C. A. A proposal to use stripes to maintain diversity in a multi-objective particle swarm optimizer. In *Swarm Intelligence Symposium (SIS)* (2005), IEEE, pp. 23–30.

[215] VOSE, M. D. Modeling simple genetic algorithms. In *Foundations of Genetic Algorithms*, L. D. WHITLEY, Ed., vol. 2 of *Foundations of Genetic Algorithms*. Elsevier, 1993, pp. 63 – 73.

[216] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (Dec 2002), 181.

[217] WANG, C., MA, H., AND CHEN, G. Using EDA-Based local search to improve the performance of nsga-ii for multiobjective semantic web service composition. In *Database and Expert Systems Applications* (2019), Springer, p. 434–451.

[218] WANG, C., MA, H., CHEN, G., AND HARTMANN, S. A Memetic NSGA-II with EDA-Based Local Search for Fully Automated Multi-objective Web Service Composition. In *The Genetic and Evolutionary Computation Conference Companion (GECCO)* (2019), Association for Computing Machinery, p. 421–422.

[219] WANG, J., YAO, Y., MAO, Y., SHENG, B., AND MI, N. Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters. In *International Conference on Cloud Computing (CLOUD)* (2014), IEEE, pp. 761–768.

[220] WANG, W., CHEN, H., AND CHEN, X. An availability-aware virtual machine placement approach for dynamic scaling of cloud applications. In *International Conference on Ubiquitous Intelligence and Computing (UIC)* (2012), IEEE, pp. 509–516.

[221] WANG, Y., AND XIA, Y. Energy optimal VM placement in the cloud. In *International Conference on Cloud Computing (CLOUD)* (2017), IEEE, pp. 84–91.

[222] WANG, Z., TANG, K., AND YAO, X. Multi-objective approaches to optimal testing resource allocation in modular software systems. *IEEE Transactions on Reliability 59*, 3 (2010), 563–575.

[223] WANG, Z., ZHANG, J., AND YANG, S. An improved particle swarm optimization algorithm for dynamic job shop scheduling problems with random job arrivals. *Swarm and Evolutionary Computation 51* (2019), 100594.

[224] WEINGÄRTNER, R., BRÄSCHER, G. B., AND WESTPHALL, C. B. Cloud resource management: A survey on forecasting and profiling models. *Journal of Network and Computer Applications 47* (2015), 99–106.

[225] WILCOX, D., MCNABB, A., AND SEPPI, K. Solving virtual machine packing with a Reordering Grouping Genetic Algorithm. IEEE, pp. 362–369.

[226] WOLKE, A., BICHLER, M., AND SETZER, T. Planning vs. dynamic control: Resource allocation in corporate clouds. *IEEE Transactions on Cloud Computing 4*, 3 (2016), 322–335.

[227] WOLKE, A., TSEND-AYUSH, B., PFEIFFER, C., AND BICHLER, M. More than bin packing: Dynamic resource allocation strategies in cloud data centers. *Information Systems 52* (2015), 83–95.

[228] WOOD, T., SHENOY, P. J., VENKATARAMANI, A., AND YOUSIF, M. S. Sandpiper - Black-box and gray-box resource management for virtual machines. *Computer Networks 53*, 17 (2009), 2923–2938.

[229] XAVIER, M. G., DE OLIVEIRA, I. C., ROSSI, F. D., DOS PASSOS, R. D., MATTEUSSI, K. J., AND DE ROSE, C. A. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)* (2015), IEEE, pp. 253–260.

[230] XAVIER, M. G., NEVES, M. V., ROSSI, F. D., FERRETO, T. C., LANGE, T., AND DE ROSE, C. A. F. Performance evaluation of container-based virtualization for high performance computing environments. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (2013), IEEE, pp. 233–240.

[231] XIAO, Z., JIANG, J., ZHU, Y., MING, Z., ZHONG, S., AND CAI, S. A solution of dynamic VMs placement problem for energy consumption optimization based on evolutionary game theory. *Journal of Systems & Software 101* (2015), 260–272.

[232] XING, B., AND GAO, W.-J. *Introduction to Computational Intelligence*. John Wiley & Sons, Ltd, 2014.

[233] XIONG, A.-P., AND XU, C.-X. Energy efficient multiresource allocation of virtual machine based on pso in cloud data center. *Mathematical Problems in Engineering 2014* (2014).

[234] XIONG, A.-P., AND XU, C.-X. Energy Efficient Multiresource Allocation of Virtual Machine Based on PSO in Cloud Data Center. *Mathematical Problems in Engineering 2014*, 6 (2014), 1–8.

[235] XU, J., AND FORTES, J. A. Multi-objective virtual machine placement in virtualized data center environments. In *International Conference on Green Computing and Communications (GreenCom)* (2010), ACM/IEEE, pp. 179–188.

[236] YAO, X., AND XU, Y. Recent advances in evolutionary computation. *Journal of Computer Science and Technology 21*, 1 (Jan 2006), 1–18.

[237] YOUNG, E. G., ZHU, P., CARAZA-HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The True Cost of Containing: A gVisor Case Study. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (July 2019), USENIX Association.

[238] YOUSAFZAI, A., GANI, A., NOOR, R. M., SOOKHAK, M., TALEBIAN, H., SHIRAZ, M., AND KHAN, M. K. Cloud resource allocation schemes: review, taxonomy, and opportunities. *Knowledge and Information Systems 50*, 2 (2017), 347–381.

[239] YSKA, D., MEI, Y., AND ZHANG, M. Genetic programming hyper-heuristic with cooperative coevolution for dynamic flexible job shop scheduling. In *European Conference on Genetic Programming (EuroGP)* (2018), Springer, p. 306–321.

[240] YU, J., BUYYA, R., AND RAMAMOHANARAO, K. *Workflow Scheduling Algorithms for Grid Computing*. Springer, 2008, pp. 173–214.

[241] ZAHARIA, M., BORTHAKUR, D., SARMA, J. S., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Job scheduling for multi-user mapreduce clusters. Tech. rep., University of California, 2009.

[242] ZHANG, D., YAN, B., FENG, Z., ZHANG, C., AND WANG, Y. Container oriented job scheduling using linear programming model. In *International Conference on Information Management (ICIM)* (2017), Association for Information Systems, pp. 174–180.

[243] ZHANG, F., MEI, Y., AND ZHANG, M. Surrogate-assisted genetic programming for dynamic flexible job shop scheduling. In *Advances in Artificial Intelligence (AI)* (2018), Springer, pp. 766–772.

[244] ZHANG, F., MEI, Y., AND ZHANG, M. 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 (EvoCop)* (2019), Springer, pp. 33–49.

[245] ZHANG, R., ZHONG, A.-M., DONG, B., TIAN, F., AND LI, R. Container-VM-PM architecture: A novel architecture for docker container placement. In *International Conference on Cloud Computing (CLOUD)* (2018), Springer, pp. 128–140.

[246] ZHANG, X., WU, T., CHEN, M., WEI, T., ZHOU, J., HU, S., AND BUYYA, R. Energy-aware virtual machine allocation for cloud with resource reservation. *Journal of Systems and Software 147* (2019), 147–161.

[247] ZHANG, Y., AND ANSARI, N. Heterogeneity aware dominant resource assistant heuristics for virtual machine consolidation. In *Global Communications Conference (GLOBECOM)* (2013), IEEE, pp. 1297–1302.

[248] ZHENG, H., FENG, Y., AND TAN, J. A hybrid energy-aware resource allocation approach in cloud manufacturing environment. *IEEE Access 5* (2017), 12648–12656.

[249] ZHOU, R., LI, Z., AND WU, C. Scheduling frameworks for cloud container services. *IEEE/ACM Transactions on Networking 26*, 1 (2018), 436–450.

[250] ZHOU, Y., YANG, J., AND ZHENG, L. Hyper-heuristic coevolution of machine assignment and job sequencing rules for multi-objective dynamic flexible job shop scheduling. *IEEE Access 7* (2019), 68–88.

[251] ZIKOPOULOS, P., EATON, C., AND IBM. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*, 1st ed. McGraw-Hill Osborne Media, 2011.

[252] ZITZLER, E., AND THIELE, L. Multiobjective optimization using evolutionary algorithms — A comparative case study. In *Parallel Problem Solving from Nature (PPSN)* (1998), Springer, pp. 292–301.

[253] ZOMAYA, A. H. K. R. P. J. K. B. Z. M. M. T. V. U. K. A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. *Computing NA*, 7 (2014), 751–774.