A Dual-chromosome Representation Genetic Algorithm for Resource Allocation in Container-based Clouds

***Abstract*—Containerization does not only support fast development and deployment of web applications, but also provides the potential to improve the energy efficiency in cloud data centers. In container-based clouds, containers are allocated to virtual machines and VMs are allocated to physical machines. This new architecture requires consolidation algorithms to select heterogeneous VMs to host containers and consolidate VMs to PMs simultaneously. Existing server consolidation techniques in VM-based clouds can hardly be applied because of the two-level architecture. This paper proposes a novel dual-chromosome genetic algorithm (GA) to solve the static container allocation problem. The experiments show that the proposed GA achieves significantly higher energy efficiency than the state-of-the-art algorithms on a wide range of test problems.**

***Index Terms*—container, server consolidation, cloud resource allocation, genetic algorithm, evolutionary computation**

I. Introduction

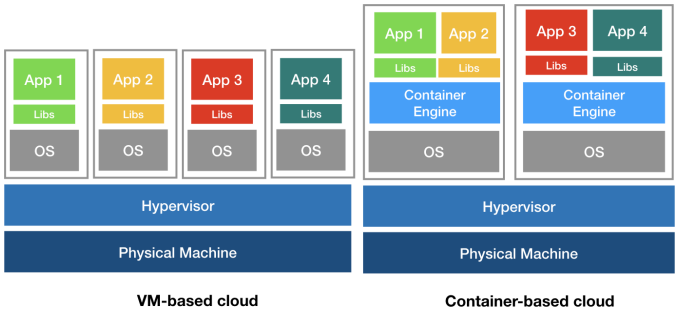
Container-based clouds [1] have gradually become the pillar of the modern software industry with the arise of micro-services and serve-less applications. With containers, application providers can pack, migrate, and deploy web applications in cloud environment. Internet companies such as Google and Microsoft run their applications with billions of containers in their data centers.

In addition to their advantages for application providers, container-based clouds also provide flexibility of resource allocation for cloud providers. Containers can be used to further improve the energy efficiency on top of the virtual machine (VM)-based clouds [2]. Traditionally, VMs are used to run multiple applications to improve the utilization in a single physical machine. Containers are able to further boost the utilization of VMs since they are sharing operating system kernels [2]. Therefore, the energy efficiency of data centers can be further improved.

However, the major gap to improve the energy efficiency in container-based clouds is the lack of server consolidation algorithms. Server consolidation means to allocate applications in fewer PMs so that the overall energy is reduced. However, since the architecture (see Fig. 1) in container-based clouds is two-level: containers are deployed on VMs and VMs are deployed on PMs, the extensive server consolidation algorithms in VM-based clouds [3] can hardly be reused.

Existing algorithms in both industry (e.g Kubernetes, Mesos) and academia mostly simplify the container allocation problem by fixing the types of VMs and use heuristics to allocate containers and VMs. In this sense, they can reuse the VM-based algorithms such as AnyFit-based algorithms [4] to solve the problem. However, the energy efficiency is highly constrained by the fixed type of VMs or greedy allocation heuristics.

Fig. 1: A comparison of architectures of VM-based clouds and container-based clouds [6]



To solve the server consolidation problem in two-level resource allocation in container-based clouds, two major difficulties have puzzled researchers. The first difficulty is the interaction between container allocation and VM allocation. Since the VMs are normally predefined with certain resource capacities labeled as VM types, allocation algorithms should create VMs with suitable types to host containers. However, the number of VMs does not have a linear relationship with the number of PMs, e.g. the least number of VMs does not necessarily lead to the least number of PMs (see the analysis in Section V-F). Hence, the allocation of the two levels must be conducted simultaneously. Secondly, both levels of allocation are vector bin packing problems [5] which are NP-hard.

This research aims at developing a novel consolidation technique for two-level container-based clouds to minimize the energy consumption of data centers. To address the VM selection problem and the interaction between the two levels, this work proposes a Genetic Algorithm (GA) approach.

GA [7] has been successfully applied to various combinatorial optimization problem over its fifty-years of history [8]. 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 difficulty of applying GA to solve the problem is the design of representation of solutions and genetic operators that can evolve solutions [9]. This is because a good representation narrows the search space and good operators are able to

accelerate the searching for near-optimal solutions.

Therefore, the overall goal for this work is to propose a GA-based approach for the container allocation problem. More specifically, we have the following objectives:

1) To propose a new dual-chromosome representation for GA on container allocation problem, which can improve the search ability over the single-chromosome GA [10].
2) To develop genetic operators, more specifically, the crossover and mutation operators for the newly proposed representation.
3) To evaluate our proposed approach by comparing it with the state-of-the-art algorithms: BestFit descending and a single-chromosome GA.

II. Related works

In this section, we first review the related works of the energy-aware container allocation problem. We then provide a brief background of genetic algorithms.

A. Existing Approaches

Currently, most container allocation approaches treat the problem as a dynamic problem [3], which means they allocate one container at a time when the request arrives. Piraghaj [11] and Kaur [12] approach the problem by considering a set of predefined VM types in a cloud and First Fit heuristic to allocate VMs to PMs. At the migration stage, containers are allocated according to rules such as Least Full Host Selection Algorithm (LFHS) [11] or First Fit Host Selection (FFHS) [11]. These rules are mostly greedy-based heuristics which are designed for fast resource allocations.

The major drawbacks of the above methods are two. First, as Wolke et al [3] conclude, dynamic approaches are worse than static approaches in terms of violations and energy consumption because the extra network overhead and the halt on migrated containers. Secondly, greedy-based heuristics often provide local optimal solutions. Wolke et al [3] also suggest the static approaches are more suitable for initial allocation of containers because the initial allocation normally has a longer time tolerance. Therefore, we aim at allocating a batch of containers with a number of new VMs and PMs. Although the static approach might have longer computational time, it generally provides much better results than greedy approaches.

As the container-based cloud is a new technology, few static approaches are proposed in the literature. Although extensive research have been made in the field of VM allocation, because of the two levels of allocation, they are hardly applied in the container allocation. Guan et al [13] define one type of VM and each PM are filled with ten VMs. Then, they propose an Integer linear programming (ILP) to allocate containers. With a predefined type of VM, the data center cannot host applications that larger than the predefined VM. Furthermore, allocating more VMs lead to more overhead. In the perspective of their approach, it is known that the computational time of ILP-based approaches grow exponentially with the problem size increases. Therefore, it is infeasible to apply an ILP approach on large allocation problems.

A multi-objective genetic algorithm approach [10] has been proposed to solve the two-level container allocation problem. This approach proposes a novel chromosome representation. However, their approach might be limited with the design of genetic operators. Their GA has no crossover operator and relies completely on mutation for local search. This motivates us to design a new representation and operators to over come the drawbacks. In section V, we reuse their representation in GA and compare with our proposed approach.

B. Genetic Algorithms

Genetic algorithms (GAs) are artificial intelligent algorithms that are in spired by biological mechanisms of evolution. They are famous for strong search ability because they have several distinguished characteristics such as the use of a population-based search and the ability of avoiding local optimum.

GAs have been applied successfully to solve a variety of real-world combinatorial optimization problems problems such the assembly line balancing problem [14], scheduling in Grid computing [15] and resource allocation problem in clouds [16].

III. Problem Model

Given N containers $n = 1, \ldots, N$, the overall objective is to allocate containers to V Virtual Machines (VMs) $v = 1, \ldots, V$, then allocate VMs to D Physical Machines (PMs) $d = 1, \ldots, D$, so that the energy consumption of PMs E is minimized. The following equations calculate the overall energy consumption E and the energy of individual PMs P_d.

$$E = \sum_{d=1}^{D} P_d \cdot [u_{cpu}(d) > 0] \tag{1}$$

$$P_d = P_d^{idle} + (P_d^{max} - P_d^{idle}) \cdot u_{cpu}(d) \tag{2}$$

E (Eq. (1)) aggregates all PMs' energy consumption P_d if a PM d is activated. The term $[u_{cpu}(d) > 0]$ returns 1 when the CPU utilization of a PM d is greater than 0, and 0 otherwise. The energy model of a PM P_d (Eq. (2)) is a widely used model proposed by Fan [17]. In their energy model, P^{idle} and P^{max} are the energy consumption when a PM is idle and fully used.

In our model, containers, VMs, and PMs are associated with two types of resources, CPU and Memory. Containers' CPU and memory requirements are denoted as C_n and M_n; VMs' CPU and memory capacities VC_v and VM_v; PMs' CPU and memory capacities DC_d and DM_d. The amount of resources is defined by a domain (i.e CPU capacity of an entity) of $[1, \ldots, R]$. For example, the CPU required by a container is defined by a value between 0 and 100. In addition, each VM has an overhead of CPU and memory denoted as $OC(v)$ and $OM(v)$. The overheads are also represented as resources. For example, the memory overhead of a VM is $OM(v) = 200Mb$.

This work considers a data center with homogeneous PMs which means all the PMs have the same CPU and memory capacities. We consider a number of types of VMs which constrain the combination of VC and VM into $Type_v$. For containers, unlike Piraghaj's approach [11] which uses three types of containers for all applications, we consider one-on-one mapping between applications and containers. That is, we

define the domain of containers' resource requirement between 1 to the capacity of the largest VM type. Therefore, our model is much flexible and more realistic (real-world containers, e.g Jelastic, use much granular measurement (MHz and Mb) for applications [18]).

The two-level allocation model mainly reflects in modeling the resource utilizations. At the VMs–PMs level, the CPU and memory utilization of a PM is computed with Eq.(3) and Eq.(4). The utilization of a PM aggregates the used resources by the VMs which are deployed on the PM. Then, it is divided by the capacity of the PM to calculate the utilization. The binary variable y_{vd} denotes if a VM v is allocated on a PM d.

$$u_{cpu}(d) = \frac{\sum_{v=1}^{V}(OC(v) + \mu_{cpu}(v) \cdot VC_v) \cdot y_{vd}}{DC_d} \quad (3)$$

$$u_{mem}(d) = \frac{\sum_{v=1}^{V}(OM(v) + \mu_{mem}(v) \cdot VM_v) \cdot y_{vd}}{DM_d} \quad (4)$$

At the containers–VMs level, the CPU and memory utilization of a VM are computed with Eq.5 and Eq.6. The binary variable x_{nv} indicates whether a container n is deployed on a VM v.

$$\mu_{cpu}(v) = \frac{\sum_{n=1}^{N} C_n \cdot x_{nv}}{VC_v} \quad (5) \qquad \mu_{mem}(v) = \frac{\sum_{n=1}^{N} M_n \cdot x_{nv}}{VM_v} \quad (6)$$

We consider two types of constraints in the model. First, similar to other models [11], the total resource requirement of containers cannot exceed the capacity of the target VM v (see Eq. (7)). The aggregate resource requirement of VMs cannot exceed the capacity of the target PM d (see Eq. (8)). Second, the container n can only be deployed once (see Eq. (9)).

$$\sum_{n=1}^{N} C_n \cdot x_{nv} \leq VC_v \qquad \sum_{n=1}^{N} M_n \cdot x_{nv} \leq VM_v \quad (7)$$

$$\sum_{v=1}^{V} VC_v \cdot y_{vd} \leq DC_d \qquad \sum_{v=1}^{V} VM_v \cdot y_{vd} \leq DM_d \quad (8)$$

$$\sum_{n=1}^{N} x_{nv} = 1 \quad (9)$$

IV. The Proposed Dual-Chromosome GA

This section introduces the design of our GA approach, which includes the representation, genetic operators, and the fitness function.

A. Algorithm

GA is a population-based search algorithm [9]. In GA, a solution to a problem is represented as a vector of numbers called chromosome or individual. GA searches for the best solution by iteratively changing the values in the chromosome by genetic operators such as crossover and mutation.

Our proposed algorithm is described in Algorithm 1. In the beginning, a population of individual is initialized with the *Initialization process*. The individual is represented as a *dual-chromosome representation*.

The evolution is an iterative process consisting of a number of generation. In each generation, each individual is evaluated according to a defined *fitness function*. Then, we apply *Elitism* to copy the top ten individuals sorted by their fitness values to the next generation of population U. Then, we apply *tournament selection* to select two parents. These two parents *crossover* by a probability called crossover rate. The generated children *mutate* according to a mutation rate. In the end, these children are added to the new population U. As a search mechanism, these two operators move solutions to explore the search space.

This evolutionary procedure ends when a predefined generation number or a satisfactory level of the fitness value has been reached. The parameters of GA such as crossover rate, mutation rate, and generations are defined in Table II.

Algorithm 1: GA framework

```
Input : A set of VM types, A set of containers,
Output: The allocation of containers
Initialize a population P with dual-chromosome individuals;
while Termination Condition is not meet do
    for Each individual do
        Evaluate the fitness value by a fitness function;
    end
    Apply elitism to copy ten top individuals to the new population U;
    while children number is less than the population size do
        Apply tournament selection to select two parents;
        Apply crossover on two parents and obtain two children;
        Apply mutation on two children;
        Add the children into a new population U;
    end
end
return the best individual;
```

B. Representation

Fig. 2 shows the design of chromosome, the input, and the output of a solution. The representation of an individual consists of two separate chromosomes: one for container allocation and the other for VM allocation. A complete allocation solution needs to be decoded from these two chromosomes.

Both the chromosomes of container allocation and VM allocation are vectors of integer value. Specifically, in the chromosome of container allocation, each value represents the indexes of the containers in the original input. The length of the chromosome is the total number of containers.

In the chromosome of VM allocation, each entry represents a VM. The value denotes the type of the VM. The length of the VM allocation chromosome is the maximum number of VMs reserved for the containers. Since no container can

Fig. 2: Representation

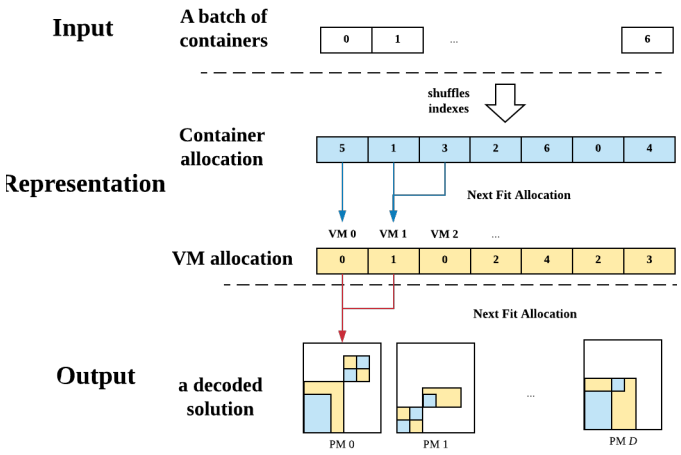


Fig. 3: Order 1 crossover

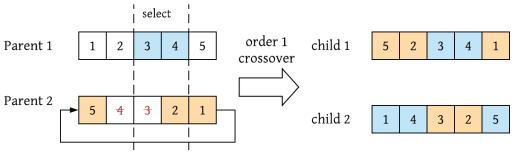
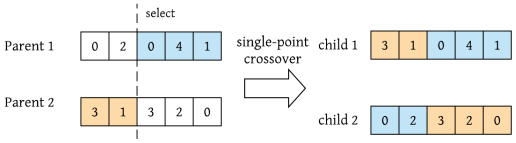


Fig. 4: Single-point crossover



require more resources than the capacity of the largest VM, we may infer that the maximum number of VMs is used when all containers are allocated to different VMs. Therefore, the maximum number of VMs equals the number of the input containers. The reserved VMs are always more than the actual number needed by containers.

To decode an individual, we apply a bin packing heuristic – Next Fit algorithm [19] in both levels. At the container–VM level, containers are packed sequentially into VMs. For example, In Fig. 2, after *container 5* was packed into *VM 0*, the following *container 1* cannot fit into *VM 0*. Therefore, we close *VM 0* and open the *VM 1* to accommodate *container 1*. The closed VMs are never check 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 VMs.

On one hand, this representation guarantees the validity of solutions by applying Next Fit heuristic to decode the solution. Therefore, we do not need extra constraint handling methods. On the other hand, this representation is able to cover the entire solution space so that we may find the optimal solution.

C. Initialization

The design of initialization aims at producing a set of diverse solutions. Based on this principle, for each individual, we randomly shuffle the indexes of containers to generate container allocation chromosomes. For initializing VM allocation chromosomes, we uniformly generate the types of VMs.

D. Crossover

The design of crossover aims at retaining the "good" genes from the parents. For the chromosome of container allocation, the definition of "good" is the permutation which leads to a high utilization of VMs' resources. We apply the **order 1** crossover [20] to pass the useful permutation to the next generation. For the chromosome of VM allocation, we apply the **single-point** crossover [21].

Order 1 crossover randomly selects a sequence of consecutive entries from one parent. The remaining values are placed in the child with the same order in the other parent. For example, in Fig 3, container 3 and 4 are selected and copied from parent to child 1. Then, the same containers (3 and 4) are crossed out from parent 2. The rest values are copied from parent 2 starts from the second cut point and roll back to the head, e.g. container 1, 5, and 2. The same rules are applied to the second child.

Single-point crossover first randomly cuts a chromosome into two parts. A child inherits one part from parent 1 and the other part from parent 2 (see Fig. 4).

E. Mutation

Mutation operator provides a local search mechanism that aims at exploring the neighbors of the current individual. We design two types of mutation to achieve this goal. The mutation can be see in Algorithm 2.

Algorithm 2: Mutation

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

```
u = random();
if u < β then
    Switch_mutation() on container allocation
      chromosome;
end
for each entry on the VM allocation chromosome do
    u = random();
    if u < β then
        Change_type_mutation();
    end
end
return the mutated individual;
```

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

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

F. Fitness Function

The fitness is determined by the decoded sequence of PMs. The energy consumption is calculated according to Eq. (1) where all PMs' energy consumption is aggregated.

V. Experiment

This section first introduces the purpose of the experiment. Then we illustrate test instances, experiment settings, and briefly introduce the compared algorithms: the modified single-chromosome GA and BestFit descending algorithm. The reason that we compare with these two algorithms is because the single-chromosome GA solves the same problem with ours. The BestFit descending is widely used in the cloud resource allocation research. Finally, we illustrate and analyze the results.

A. Design of Experiments

The overall goal of the experiment is to test the performance of our proposed GA algorithm in terms of energy consumption. To achieve this goal, we conduct experiments on real-world datasets with our proposed GA and two benchmark algorithms (single-chromosome GA and BestFit descending). Their effectiveness are compared according to the energy consumption. Then, we will analyze the efficiency of the representation and the search mechanisms by comparing the convergence between our proposed GA and the single-chromosome GA. Last but not least, we will also compare the execution time of these algorithms.

Additionally, to benefit other researchers, we share two insights of two-level container allocation problem. The first insight shows the number of VM does not proportional to the total energy consumption. In other words, least number of VMs does not always lead to the least energy consumption. Second insight shows that providing more VM types may have a negative influence on the algorithm's performance because redundant VM types unnecessarily enlarges the search space.

B. Dataset and Test instance

We use a real-world dataset where applications' resource requirement are recorded in a data center (AuverGrid trace [22]). We assume homogeneous PMs with the resource capacity of [3300 MHz, 4000 MB]. The energy consumption for the fully utilized PM is set to 135W.

To test the algorithms' performance on a variety of VM configurations, we design three sets of VM type with increasing numbers of VM types (see Table III). The first set (yellow area) contains five VM types. The second set includes seven VM types (yellow and green areas). The third set includes all ten VM types. We designed four test instances shown in Table I, listed with increasing sizes and difficulties. We test algorithms on these instances for each VM type settings, which includes total 12 experiments.

TABLE I: Instance Settings

Instances	1	2	3	4
Number of containers	100	200	500	1000

C. Benchmark Algorithms

Single-chromosome GA is proposed in [10] to solve service allocation problem which is a similar problem with container allocation problem.

The two differences prevent us from directly reuse their algorithm. First, their algorithm is a multi-objective version of GA called NSGA-II. Second, one of the major assumption in their approach is that they allow *VM overbooking* [3] which tolerates a certain degree of utilization overhead. In our assumption, VM overbooking is not allowed. Therefore, in order to make a comparison, we only reuse their representation and re-design the initialization (see Algorithm 3) and mutation operator. The mutation operator randomly switches two containers.

Algorithm 3: Initialization for single-chromosome GA

```
Input  : A set of containers with CPU,memory
         requirement, VM types
Output: A population of individuals
while number of individual i < population_size do
    for each container in an individual do
        Apply First Fit to find an existing VM to
         allocate;
        if no existing VM is available then
            candidate VM Type = Iterate the VM type
             and find the first VM that has the capacity
             to host this container;
            Generate a VM type which equal or
             stronger than the candidate VM Type;
        end
    end
    Add the individual to population;
    i++;
end
```

BestFit Descending algorithm is static bin packing heuristic that can only select existing VMs for containers.

In this work, we implement a BestFit Descending with sum rule [4] which is used to measure the residual resources from multiple dimensions (e.g CPU and memory). The sum rule assigns each candidate VM with a score. The score is calculated by aggregating the each dimension of remaining resources on the VMs after the container is allocated. The BestFit algorithm always selects the VM with the minimum score which indicates the minimum residual resources.

Since BestFit Descending does not have the functionality to select VM types, in order to create new VMs, we always select the largest VM type (Type five in Table III).

TABLE II: Parameter Settings

Parameter	Description
crossover	80%
mutation rate for dual-chromosome GA	10%
mutation rate for single-chromosome GA	80%
elitism	top 10 individuals
Number of generations	1000
Population	100
Selection	tournament selection (size = 7)

TABLE III: Configuration of VMs

VM types	CPU (MHz)	Memory (MB)	VM types	CPU (MHz)	Memory (MB)
1	660	800	6	660	2400
2	1320	1600	7	1320	2800
3	1320	2800	8	660	2800
4	1980	2400	9	1980	1200
5	2310	2800	10	2310	1600

D. Parameter Settings

The parameter settings for both single- and dual-chromosome GAs are listed in Table II. In addition to our proposed operators, we apply the elitism [23] with size 10 and tournament selection [24] with size 7. These methods are standard and widely applied. For single-chromosome GA, we set its mutation rate to 0.8 because it completely relies on mutation to search.

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

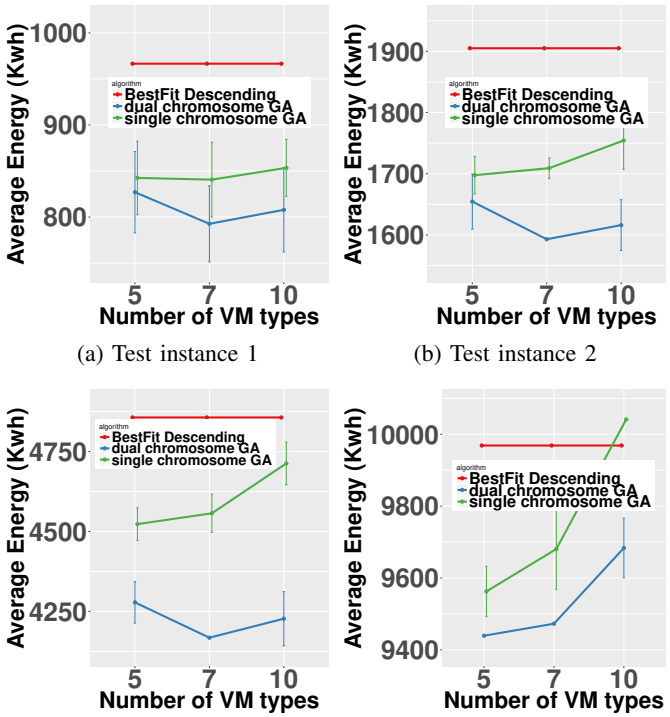
E. Experiment Results

This section first illustrates the performance comparison among three algorithms in terms of energy consumption. Then, we show the efficiency of the representation and search mechanisms by comparing the convergence curve between our GA and the single-chromosome GA [10]. Lastly, we compare the time complexity of three algorithms.

Fig. 5 shows the comparison of the average energy consumption among three algorithms for test cases with three VM types. BestFit descending algorithm (Red line) uses the most energy consumption. Single-chromosome GA (Green line) shows better performance than BestFit Descending in most of the cases except one. Our proposed dual-chromosome GA has shown the best performance in all test instances. The Wilcoxon signed rank test show that it is significantly different between our GA and others with a confidence interval of 95%.

Fig. 6 shows the average number of PMs for the test cases with 5 VM types and 10 VM types. We omit the figure of 7 VM types because it shows a similar pattern with 5 VM types. In the test instances with 5 and 7 VM types, BestFit Descending (Green line) uses the largest number of PMs. Single-chromosome GA (Blue line) uses more PMs than our dual-chromosome GA. In the test instances with 10 VM types, single-chromosome uses the most PMs in the fourth instances (allocating 1000 containers).

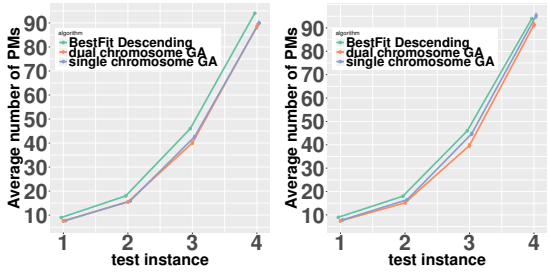
Fig. 5: Comparison of the average energy consumption among three algorithms for three VM type configurations



(a) Test instance 1 (b) Test instance 2

(c) Test instance 3 (d) Test instance 4

Fig. 6: Comparison of the average number of PMs



(a) 5 VM types (b) 10 VM types

The energy consumption of the solution generated by the BestFit Descending is not affected by the number of VM types because it always use the largest VM type (see Section V-C). Additionally, Best Fit descending is a deterministic algorithm, which means with the same input set of containers, the output energy will always be the same.

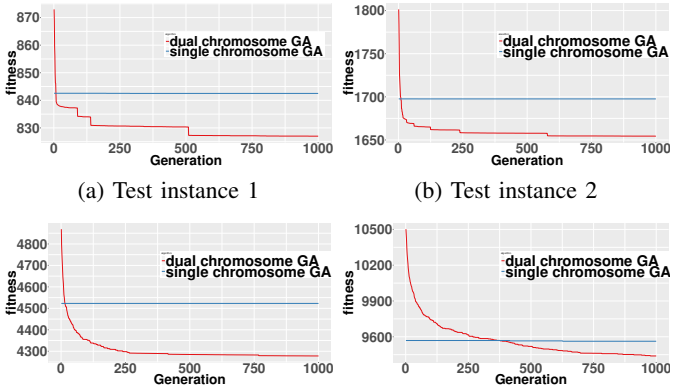
Fig. 7 compares the convergence curve of two GAs. Convergence curve is depicted by connecting the average fitness values from two GAs of 30 runs during 1000 generations. Dual-GA's (Red line) starts with worse fitness values than single-chromosome GA. Only within a few generations, the fitness values from dual-chromosome GA quickly decrease (less than 400 generations in the largest instance) and surpass the single-chromosome GA. In contrast, the convergence curve

of single-chromosome GA (Blue line) remain stagnant during the entire evolution.

We can explain the reason that the single-chromosome GA has better initial fitness values with the detailed comparison of two GAs' initialization methods. In the single-chromosome GA, shown as Algorithm 3, we apply First Fit to allocate containers in *line 3*. Since First Fit always scan the VMs from the oldest and never close a VM, it guarantees the existing VMs are filled before creating a new VM. In contrast, in dual-chromosome GA, containers are allocated to VMs with Next Fit. Next Fit closes a VM whenever the current container cannot be allocated even though other containers can still be allocated into the VM. Therefore, Next Fit does not guarantee the closed VMs are filled. Therefore, at the initial stage, the utilization in PMs of single-chromosome GA are much higher than dual-chromosome GA.

The stagnant of single-chromosome GA can be explained by its search operators. The search operator in single-chromosome GA is very limited as it completely relies on mutation operator and does not have a crossover operator. This design has two drawbacks. Firstly, without crossover, there is no interaction among individuals. The algorithm lacks the ability of combining good parts from multiple individuals. Therefore, it leads to a lower convergence. Secondly, the mutation operator in single-chromosome GA can only be conducted when two host VMs have enough resources for the switching, otherwise, the individual will remain unchanged. Hence, this mutation also limits the search ability.

Fig. 7: Convergence curve



(a) Test instance 1 (b) Test instance 2

(c) Test instance 3 (d) Test instance 4

Table. IV shows the mean and standard deviation of calculation time of three algorithms. BestFit descending is the fastest algorithm among all. Dual-GA uses much less time than single-chromosome GA, especially when the test instance gets larger. Specifically, in test instance 3 and 4, single-chromosome GA uses more than 20 times more time than dual-chromosome GA.

We may explain the fast execution of BestFit descending with two reasons. First, unlike GAs, BestFit does not calculate the fitness values for a population of solutions. Second, the VM type is predefined, hence, BestFit does not spend time on optimizing the VM types and their locations.

TABLE IV: Average execution time (s) for three algorithms on four test instances with five VM types

Instances	Time (s)		
	BestFit Descending	dual-chromosome GA	single-chromosome GA
Test instance 1	0.11 ± 0.0	0.92 ± 0.17	5.20 ± 0.28
Test instance 2	0.16 ± 0.0	1.45 ± 0.17	12.04 ± 0.43
Test instance 3	0.48 ± 0.0	3.07 ± 0.28	60.79 ± 2.38
Test instance 4	1.01 ± 0.0	7.12 ± 0.26	181.06 ± 7.25

The reason that the single-chromosome GA is much slower than dual-chromosome GA due to the extensively use of mutation operator. With 0.8 mutation rate, the mutation is almost conducted on every container. This is very time consuming operation because the algorithm must guarantee the host VMs have enough resource for switching. Furthermore, the computation time is proportional to the number of containers.

Our proposed dual-chromosome GA, on the other hand, has acceptable time complexity and produces the best performance among all three algorithms.

F. Result Analysis

These experiments provide two insights for the container allocation problem. The first insight shows the number of VM does not proportional to the total energy consumption. In other words, the least number of VMs does not necessary lead to the least energy consumption. Fig. 8 shows the average number of VMs used in all test instances. As we may seen, BestFit descending (Red bar) always uses the least number of VMs because it chooses the largest VM type in default. However, the largest VM type (Type five in Table III) is account for 70% of total resources in a PM. That means, the maximum resources that can be used by BestFit is only 70% of the total resources in a PM because a PM can only accommodate one largest VM.

In contrast, both single- and dual-chromosome GAs use more VMs than BestFit. When we examine the VM types that GAs used, we found that in most of the cases, GAs found the complementary VMs. For example, in Table III, the combination of types (2, 4), (3, 9), (2, 2, 1) is able to use 100% of PM's resources. We observed that GAs can always find these complementary VM types to maximize the utilization. The best combination of VM types may not use the least number of VMs. As in our test instances, dual-chromosome GA always generates better performance with more VMs than the single-chromosome GA does.

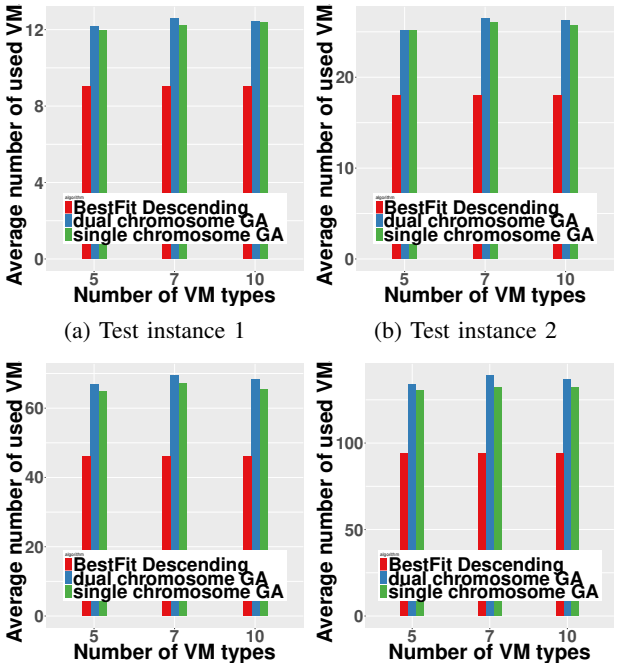
Second insight shows that providing more VM types may have a negative influence on algorithms because of redundant VM types extends the search space. In Fig.5, with test instance 1 to 3, a clear pattern shows the dual-chromosome GA achieves the best performance applying on seven VM types and the performance drops on ten VM types. As mentioned above, GA searches for complementary VM types to fully utilized PMs. The performance increased when providing seven VM types because the additional VM types are useful to construct better combination of types. However, when

providing ten types, the additional types (compared to 7 VM types) are redundant, therefore, it only lead to longer searching time and worse performance.

In instance 4, since the size of containers are much more than the previous instances, the search space are also much larger. The performance of both GAs decreases.

These two insights provide a better understanding of the container allocation problem, which may be useful for developing new allocation algorithms. In addition, the second insight may inspire the cloud providers to provide a minimum set of complementary VM types which may accelerate the optimization process.

Fig. 8: Average number of used VMs



(a) Test instance 1 (b) Test instance 2

(c) Test instance 3 (d) Test instance 4

VI. Conclusion

Container-based clouds have become increasing popular in recent years. However, the existing algorithms for server consolidation in traditional VM-based clouds can hardly be used in container-based cloud because two-level allocations are needed: containers to VMs and VMs to PMs. The traditional algorithms did not consider the selection of heterogeneous VMs and VM allocation at the same time.

This work proposes a dual-chromosome representation GA to solve the two-level container allocation problem. The experiments ran on real-world datasets with the comparison of a single-chromosome GA and a BestFit descending algorithm. The results show that our proposed GA performs much better than the compared algorithms in all test instances with a much shorter time. This work also provide two insights for the container allocation problem: The number of VMs does not proportional to the total energy consumption. Second insight shows redundant VM types have an negative influence on the allocation algorithms.

References

[1] B. Familiar, *Microservices, IoT and Azure: leveraging DevOps and Microservice architecture to deliver SaaS solutions*. Apress, 2015.

[2] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[3] A. Wolke, M. Bichler, and T. Setzer, "Planning vs. dynamic control: Resource allocation in corporate clouds," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 322–335, 2016. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TCC.2014.2360399

[4] K. Maruyama, S. K. Chang, and D. T. Tang, "A general packing algorithm for multidimensional resource requirements," *International Journal of Computer & Information Sciences*, vol. 6, no. 2, pp. 131–149, Jun 1977.

[5] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," *research. microsoft. com*, 2011.

[6] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[7] M. D. Vose and A. Hall, "Modeling Simple Genetic Algorithms," in *Evolutionary Computation*. Elsevier, 1996, vol. 3, no. 4, pp. 453–472.

[8] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, and J.-J. Li, "Distributed evolutionary algorithms and their models: A survey of the state-of-the-art," *Applied Soft Computing*, vol. 34, pp. 286–300, 2015.

[9] K.-F. Man, K.-S. Tang, and S. Kwong, *Genetic algorithms: concepts and designs*. Springer Science & Business Media, 2012.

[10] B. Tan, H. Ma, and Y. Mei, "A nsga-ii-based approach for service resource allocation in cloud," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, June 2017, pp. 2574–2581.

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

[12] K. Kaur, T. Dhand, N. Kumar, and S. Zeadally, "Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers," *IEEE Wireless Communications*, vol. 24, no. 3, pp. 48–56, June 2017.

[13] X. Guan, X. Wan, B. Choi, S. Song, and J. Zhu, "Application oriented dynamic resource allocation for data centers using docker containers," *IEEE Communications Letters*, vol. 21, no. 3, pp. 504–507, March 2017.

[14] E. Anderson and M. Ferris, "Genetic algorithms for combinatorial optimization: The assemble line balancing problem," *ORSA Journal on Computing*, vol. 6, no. 2, pp. 1–23, may 1994.

[15] A. Y. Zomaya and K. Crnomarkovic, "Genetic Algorithms for Scheduling in Grid Computing Environments." *Handbook of Bioinspired Algorithms and Applications*, vol. 20053845, pp. 13–208, 2005.

[16] D. Wilcox, A. McNabb, and K. Seppi, "Solving virtual machine packing with a Reordering Grouping Genetic Algorithm," *2011 IEEE Congress of Evolutionary Computation, CEC 2011*, pp. 362–369, 2011.

[17] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," *ACM SIGARCH Computer Architecture News*, vol. 35, no. June, p. 13, 2007.

[18] "Automatic vertical scaling," https://docs.jelastic.com/automatic-vertical-scaling/, accessed: 2018-10-05.

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

[20] P. Poon and J. Carter, "Genetic algorithm crossover operators for ordering applications," *Computers and Operations Research*, vol. 22, no. 1, pp. 135 – 147, 1995, genetic Algorithms.

[21] R. B. Agrawal, K. Deb, and R. Agrawal, "Simulated binary crossover for continuous search space," *Complex systems*, vol. 9, no. 2, pp. 115–148, 1995.

[22] S. Shen, V. v. Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 465–474.

[23] D. Bhandari, C. Murthy, and S. K. Pal, "Genetic algorithm with elitist model and its convergence," *International journal of pattern recognition and artificial intelligence*, vol. 10, no. 06, pp. 731–747, 1996.

[24] B. L. Miller, D. E. Goldberg *et al.*, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.