A NSGA-II-based Approach for Multi-objective Micro-service Allocation in Container-based Clouds

Boxiong Tan, Hui Ma, Yi Mei
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
Email: {Boxiong.Tan, Hui.Ma, Yi.Mei}@ecs.vuw.ac.nz

Abstract—Micro-services is a widely adopted architecture to develop large scale web applications. To provide a scalable and low-overhead resource service to micro-service applications, the new container-based clouds are proposed. The new clouds use both containers and VMs to manage resources to achieve a low-overhead, high-utilization data center. However, existing resource allocation approaches either do not consider the dependencies between containers or can only be applied in OS-level container clouds which allocate containers directly to physical machines. To address the multi-objective optimization problem, this work proposes a multi-objective NSGA-II to optimize the availability of applications and the energy consumption requirement of container-based clouds. Our goal is to provide solutions with different tradeoffs between two objectives for cloud providers to choose from. We evaluate the algorithm with a wide range of scenarios by simulation and compare with state-of-the-art algorithms. The results show that our approach significantly outperforms other approaches.

Index Terms—micro-service, container, server consolidation, cloud resource allocation, genetic algorithm, evolutionary computation

I. INTRODUCTION

Micro-service architecture [1] 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 clouds, applications benefit from the seemingly infinite resources and can scale up and down according to workload fluctuation. To adapt to the new software architecture, cloud providers propose a new type of cloud, namely a container-based cloud, to provide the low-overhead running environment for large-scale applications.

Container-based clouds [2], [3] are a new variant of Platform as a Service (PaaS) clouds [4] that are designed for managing large scale web applications with containers. Unlike VM-based clouds where Virtual Machines (VMs) are used as a resource unit and allocated to Physical Machines (PMs), container-based clouds [3] use both containers and VMs. As containers are allocated to various sizes of VMs and VMs are allocated to PMs, thus container-based clouds have a two-level allocation structure. On one hand, this structure increases the flexibility of allocating resources, e.g. containers from different providers can be co-located to increase the utilization of VMs, meanwhile, VMs provide high-security isolation for containers. On the other hand, such a structure increases the difficulty for resource allocation.

As a key task in container-based clouds, the Micro-service Allocation in Container-based clouds (MAC) allocates a set of applications into a data center. Each application is composed of a set of micro-services where each micro-service could have multiple replicas. With each replica wrapped with a container, these containers are allocated to VMs and then to PMs. The nature of the MAC problem is a multi-objective, two-level vector bin packing problem [5]. Multiple objectives come from stakeholders where the cloud providers have the primary concern of energy consumption and application providers consider Service Level Agreement (SLA)-related objectives, such as maximizing availability or minimizing the communication cost between containers. The MAC problem is a vector bin packing problem because containers have multiple resources to be allocated.

Literature has discussed several related problems. Much research discusses the resource allocation in container-based clouds [3], [5]–[7]. They allocate independent containers to minimize the energy consumption of the used PMs. However, these approaches only consider one objective, i.e. minimizing energy consumption, and neglect the performance of applications. Other research [8]–[11] consider 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 could only be applied in OS-level container architecture where containers are allocated to PMs directly. Hence, there is a need to develop a multi-objective approach for the micro-services allocation in container-based clouds.

Multi-objective evolutionary algorithms (MOEAs) are well suited for the MAC problem. As previously mentioned, MAC problem involves a two-level vector bin packing problem which is NP-hard. Integer Linear Programming (ILP) or Mixed Linear Programming (MLP) approaches cannot be used in large-scale problems because of the high computation time. Evolutionary algorithms (EAs) search for near-optimal solutions within a reasonable period. Compared to heuristics, EAs search with a population of solutions. Thus, it has less chance to get stuck into local optima. Also, MOEAs provide a set of
trade-off solutions in a single run. This is an effective way to find solutions.

Non-dominated Sorting Genetic Algorithm (NSGA-II) is one of the most widely applied MOEAs proposed by Deb et al. [12]. Due to its powerfulness of finding wide-spread solutions and implementation simplicity [13], NSGA-II has been successfully applied in many real-world multi-objective combinatorial problems such as web service allocation [14], service composition [15], [16] and resource allocation in clouds [17], [18]. These problems have similar representations and problem structures with the MAC problem.

Therefore, to address the MAC as a multi-objective resource allocation problem, we propose a NSGA-II-based approach to minimize the energy consumption and maximize the availability. The proposed NSGA-II approach provides a set of non-dominated solutions that allows cloud providers to choose from. More specifically, we have the following objectives,

1) To propose a novel problem definition for MAC problem;
2) To develop three novel operators in our NSGA-II-based approach;
3) To evaluate our proposed approach with three state-of-the-art algorithms on real-world datasets;

The paper is organized as follows. Section II gives a background of our methodology and discusses related studies of the MAC problem. Section III presents the model of the problem. Then, section IV describes the proposed NSGA-II approach. Section V illustrates the experiment design, results, and analysis. Section VI summarizes the contributions and discusses the future work.

II. LITERATURE AND BACKGROUNDS

A. Related Work

Existing research mostly focuses on two related problems, i.e. resource allocation in container-based clouds and micro-service allocation in OS-container clouds. The first one mostly focuses on allocating independent containers to two-level container-based clouds to optimize energy consumption. Pirghaj et al. [3], Mann [6], [19] and Zhang et al. [5] develop AnyFit-based [20] algorithms. These simple heuristics use human-designed scoring rules to select existing VMs/PMs or types of VMs. For example, if no existing VM is available for a container, both [19] and [5] create the smallest VM to host the container. Tan et al. [21], [22] propose automatically evolved rules with Genetic Programming that adapt to the dynamically changing workloads. These approaches are designed for dynamic scenarios where containers are needed to be allocated immediately.

Other works allocate a set of containers together and treat it as a static problem. Guan et al. [23] define one type of VM and each PM is filled with ten VMs. Then, they propose an Integer Linear Programming (ILP) to allocate containers. Tan et al. [7] propose a dual-chromosome Genetic Algorithm (GA) approach that can be used in container-based clouds with heterogeneous VMs and homogeneous PMs. The above approaches treat containers independently without considering the correlation among them. They focus on satisfying the resource requirement of containers and optimizing energy consumption. For MAC problem usually has multi-objective, hence these approaches perform poorly on other objectives.

The research on micro-service allocation consider a wide range of objectives and mostly apply MOEAs. Most of these research consider multiple objectives such as network transmission time [11], [24]–[26], load-balancing or energy consumption [11], [24]–[28], availability [11], [28] and other affinity requirements [26]. Most of the works apply MOEAs such as Ant colony algorithm (ACO) [11], NSGA-II [8], and Particle swarm optimization [27] to solve the problem. Since these works can only be applied in one-level clouds, they are not suitable for the two-level MAC problem. Therefore, we intend to develop a novel multi-objective algorithm to address the two-level MAC problem.

Among all the objectives, we consider two objectives, energy consumption and availability of applications. The reason is that these two objectives are both crucial to the cloud providers and conflicting in nature. Reducing energy consumption or improving resource utilization is often the priority for cloud providers as it minimizes the energy cost. The availability of an application is a key SLA component that guarantees the time of an application can be accessed. Maximizing availability is conflicting with minimizing energy consumption because it requires the replicas of micro-services to spread across PMs to avoid single point failure.

B. Multi-Objective Optimization

Multi-objective problems consist of multiple conflicting objectives. Hence, in these problems, not a single solution achieves the optimal solution among all the objectives. The goal of multi-objective problems is to find a set of Pareto optimal solutions. If all the objectives of a solution A are equal or better than another solution B and at least one objective are better than B, we say A dominates B. If two solutions can not dominate each other, then they are non-dominated. A solution is a Pareto optimal solution if it is a non-dominated solution.

III. PROBLEM MODEL

In the MAC problem, a set of application $\mathcal{A} = \{a_1, \ldots, a_s\}$ arrive to the cloud to be allocated. Each application consists of a set of micro-services $\mathcal{MS} = \{ms_1, \ldots, ms_s\}$. $\Upsilon(ms_j) = a_i$ denotes that a micro-service $ms_j$ is a component of the application $a_i$. Micro-services have multiple replicas with each mapping to a container $\mathcal{C} = \{c_1, \ldots, c_n\}$. Similarly, $\Upsilon(c_i) = ms_j$ 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 with $\tau_j$ 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 of that type. There is an unlimited set of PMs $\mathcal{P} = \{p_1, \ldots\}$ for allocating the created VMs. Each PM $p_k$ has a CPU capacity $\Omega^{cpu}(p_k)$ and a memory capacity $\Omega^{mem}(p_k)$.
\( \Omega \) is calculated as follows.

where \( \mu \) is the number of PM used.

where \( \Omega \) is the amount of PM used.

4) For each PM, the sum of the CPU and memory capacities of the VMs allocated on the PM does not exceed the corresponding PM’s capacity.

The energy consumption is calculated as follows:

\[
E = \sum_{k=1}^{K} E_k,
\]

where \( E_k \) is the energy consumption of the \( k \)th PM and \( K \) is the number of PM used.

\( E_k \) is calculated as follows:

\[
E_k = E_k^{idle} + (E_k^{full} - E_k^{idle}) \cdot \mu_k^{cpu},
\]

where \( E_k^{idle} \) and \( E_k^{full} \) indicate the energy consumption of the \( k \)th PM per time unit if it is idle and fully loaded, respectively. \( \mu_k^{cpu} \) indicates the CPU utilization level of the \( k \)th PM. \( \mu_k^{cpu} \) is calculated as follows.

\[
\mu_k^{cpu} = \frac{\sum_{l=1}^{\Omega} (\sum_{j=1}^{J} \pi^{cpu}(\tau_j) \cdot z_{jl} + \sum_{l=1}^{\Omega} \Omega^{cpu}(c_i) \cdot \alpha_{il}) \cdot y_{lk}}{\Omega^{cpu}(p_k)}
\]

where \( x_{il}, y_{lk} \) and \( z_{jl} \) are binary decision variables, and \( J \) is the number of created VMs. \( x_{il} \) takes 1 if \( c_i \) is allocated to the \( l \)th created VM, and 0 otherwise. \( y_{lk} \) takes 1 if the \( k \)th created VM is allocated to the \( l \)th PM, and 0 otherwise. \( z_{jl} \) takes 1 if the \( j \)th created VM is of type \( j \), and 0 otherwise.

The availability is calculated as follows:

\[
\text{Availability} = \frac{\sum_{i=1}^{S} \Lambda(a_i)}{S}
\]

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

\[
\Lambda(a_i) = \lambda_{ms_1} \cdot \lambda_{ms_2} \cdot \ldots \lambda_{ms_n}, \forall \Theta(ms_j) = a_i
\]

The availability of a micro-service is related to the PMs that host its containers (see Eq.6). The micro-service \( ms_j \) is crushed if all its containers \( c_i \) are crushed (see Eq.7). Eq.7 means that if the PM \( p_k \) is crushed, then, all the containers in the PM are crushed. Since these containers are not independent, the case statement returns the failure rate of the PM (denote as \( F(p_k) \)) once. Otherwise, it returns 1.

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 \( p_1 \). Micro-service \( B \) also has two container \( c_3 \) and \( c_4 \) which are allocated to PMs \( p_2 \) and \( p_3 \). We set failure rate for all PMs 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, 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 \( 98\% \cdot 99.96\% = 97.9608\% \).

\[
\lambda_{ms_j} = 1 - \prod_{k=1}^{K} \text{crushPro}(p_k)
\]

\[
\text{crushPro}(p_k) = \begin{cases} 
F(p_k), & \text{if } \left( \sum_{i=1}^{\Omega} \sum_{l=1}^{\Omega} x_{il} \cdot y_{lk} \right) > 0, \forall \Theta(c_i) = ms_j \\
1, & \text{else}
\end{cases}
\]

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

\[
\min \quad \sum_{k=1}^{K} E_k,
\]

\[
\min \quad 1 - \frac{\sum_{i=1}^{S} \Lambda(a_i)}{S},
\]

s.t.

\[
\sum_{l=1}^{L} x_{il} = 1, \forall i = 1, \ldots, n,
\]

\[
\sum_{k=1}^{K} y_{lk} = 1, \forall l = 1, \ldots, L,
\]

\[
\sum_{j=1}^{M} z_{jl} = 1, \forall l = 1, \ldots, L,
\]

\[
\sum_{i=1}^{n} \zeta^{res}(c_i) x_{il} \leq \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl},
\]

\[
\forall l = 1, \ldots, L, \quad \zeta^{res}(c_i) x_{il} \leq \Omega^{res}(\tau_j) z_{jl},
\]

\[
\forall k = 1, \ldots, K, \quad \zeta^{res}(c_i) x_{il} \leq \Omega^{res}(\tau_j) z_{jl},
\]

IV. THE PROPOSED NSGA-II BASED APPROACH

This section introduces the design of our NSGA-II approach, which includes the representation, genetic operators, and the fitness function.
A. Algorithm

Our proposed algorithm follows the standard NSGA-II framework described in Algorithm 1. The algorithm starts with the initialization of a population of solutions. Solutions are represented as a group of PMs hosting VMs and containers (see next section). The main evolution is an iterative process consisting of some generations. In each generation, each individual is evaluated according to objective functions. Then, we sort and calculate the crowding distance of \( P \). In the subsequent loop, a population of children is generated by crossover and mutation operators. After a new population of \( U \) is generated, we evaluate the combined \( P \cup U \), then select the top individuals to create a new population. This evolutionary procedure ends when a predefined generation number or a satisfactory level of the fitness value has been reached.

**Algorithm 1:** NSGA-II-based Approach for MAC

\[
\begin{align*}
\text{Input} & : \text{A set of VM types, A set of containers,} \\
\text{Output} & : \text{The allocation of containers} \\
1 & \text{Initialize a population } P \text{ with individuals;} \\
2 & \text{while } \text{Termination Condition is not meet} \\
3 & \quad \text{for Each individual} \ 
4 & \quad \quad \text{Evaluate the fitness values;} \\
5 & \quad \text{end} \\
6 & \quad \text{while } \text{children number is less than the population size} \\
7 & \quad \quad \text{Apply binary tournament selection to select two parents;} \\
8 & \quad \quad \text{Apply crossover over the selected parents;} \\
9 & \quad \quad \text{Apply mutation on two children;} \\
10 & \quad \quad \text{Add the children into a new population } U; \\
11 & \quad \text{end} \\
12 & \quad \text{evaluate individuals from } U; \\
13 & \quad \text{non-dominated sorting of } \{P \cup U\}; \\
14 & \quad \text{calculate crowding distance of } \{P \cup U\}; \\
15 & \quad P \leftarrow \text{select population size of individuals from } \{P \cup U\}; \\
16 & \text{end} \\
17 & \text{return the Pareto front of solutions;} \\
\end{align*}
\]

B. Representation

The representation of a solution consists of a list of PMs hosting VMs. Each VM hosts a list of containers. The sizes of these lists vary according to the allocation.

C. 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 VMs with random types (uniformly choose from a VM table). Then, the VMs are allocated to a set of PMs with FF. The use of FF guarantees valid solutions as well as a consolidated VM/PM allocation.

D. Crossover

We propose a gene-level crossover [29] where PMs on the chromosome are sorted, pair-wisely compared and preserved (see Fig. 2). In the first step, the PMs of a chromosome are sorted under a criterion, such as CPU utilization or duplication number (introduced later). Then, two parents are compared pair-wisely on PMs also under this criterion. The winning PM preserves its structure by copying all VMs’ types and containers to the child. Before copying the containers, we check whether these containers have been allocated in this child solution. Only the unallocated containers are copied so that the child solution does not validate the constraint on containers (Eq. 10). If one parent has more PMs than the other, the exceeding PMs are copied to the child as well. In the end, some containers may be unallocated. These free containers are allocated with the rearrangement operator. After all, containers have been allocated to the child, the empty PMs and VMs in the child are removed.

![Fig. 2: Gene-level crossover](image)

E. Rearrangement

Rearrangement (see Algorithm 2) inserts free containers into PMs. Rearrangement randomly selects (50% of chance) a method from the following methods, i.e. an energy-aware method, and an availability-aware method. The energy-aware method (line 3 to line 11) attempts to 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. 16). The energy-aware method first sorts the containers in ascending order (line 3). The basic idea for this heuristic 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 with a free container.

### Algorithm 2: Rearrangement operator

**Input**: a target container, a list of PMs,

**Output**: a list of PMs

1. $u \leftarrow$ Randomly selects from $[0, 1]$;
2. if $u > 0.5$ then
   3. Sort the containers in all VMs according to Eq.16 in ascending order;
   4. for each VM do
      5. if the two smallest containers in each VM can be replaced by the target container then
         6. Replace two containers with the target VM;
         7. Allocate two containers with FF&RC/FF;
      end
   end
   8. Allocate the target container with FF&RC/FF;
3. end
4. else
   5. for each PM do
      6. if the target container does not belong to the same micro-service with any containers in this PM then
         7. Replace a container that has duplicates;
         8. Allocate the container with FF&RC/FF;
      end
   end
5. end
6. return a list of PMs;

$$R = \frac{\zeta_{cpu}(c_i)}{\Omega_{cpu}(p_k)} \cdot \frac{\zeta_{mem}(c_i)}{\Omega_{mem}(p_k)}$$

(16)

### F. Mutation

We design two functions in the mutation operator, unpack and merge. The unpack function intends to improve the solution by eliminating the PMs with low CPU utilization or high duplication number. The selection of objectives is also controlled by a random number. Then, the unpack operator sorts the PMs according to CPU utilization (descending) or duplication number (ascending). The operator unpacks the PMs in a roulette wheel style. That is, the lower-ranking PMs have a higher chance to be unpacked.

The second function of mutation is merged. Merge replaces small VMs with a larger one, hence PMs could release more VM overheads and reduce the degree of segmentation. It also has two alternative ways, first one merge two smallest VMs in a PM with a large type of VM without violating the resource constraint. The alternative is to enlarge the smallest VM with a larger one. The large type is also selected randomly.

### G. Fitness Assignment

The two objective functions are introduced in the previous section. The energy consumption is calculated according to Eq. 1 and the availability is calculated according to Eq. 4.

### V. EXPERIMENT

The goal of the experiment is to test the performance of algorithms in two conflicting objectives: energy consumption and availability. We conduct experiments and compare our proposed algorithms with three benchmark algorithms, two rule-based FF&BF/FF [6], [30] approach and a Spread [31] (a method in Kubernetes), and a single-objective dual-chromosome Genetic Algorithm (dual-chromosome GA) approach [32].

#### A. Dataset and Test Instance

We design 8 test instances (see Table.I) 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 [33]). We generate the containers using the same way as in [7]. We set a crush rate of 2.5% for PMs.

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 [6]. We design two sets of VM types (see Table II), a real-world VMs (20 types from Amazon EC2) and a synthetic set of VMs (10 types). The real-world VM types are proportional whereas 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 I: test instances

<table>
<thead>
<tr>
<th>instance</th>
<th>VM types</th>
<th>number of applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>synthetic</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>VM types</td>
<td>150</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>real-world</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>VM types</td>
<td>150</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>

#### B. Benchmark Algorithms

FF&BF/FF [6], [30] 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 available VM can host a container, the BF selects a type of VM which has just enough resource to host the container.

Spread [31] is an approach provided by an open-source container management tool Kubernetes. The simple rule tries
TABLE II: VM types

<table>
<thead>
<tr>
<th>real world VM types</th>
<th>VM types [CPU, Memory]</th>
<th>synthetic VM types</th>
<th>VM types [CPU, Memory]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[206.25, 250]</td>
<td>6</td>
<td>[719, 2005]</td>
</tr>
<tr>
<td>2</td>
<td>[412.5, 500]</td>
<td>7</td>
<td>[917, 951]</td>
</tr>
<tr>
<td>3</td>
<td>[825, 1000]</td>
<td>8</td>
<td>[1032, 1009]</td>
</tr>
<tr>
<td>4</td>
<td>[1650, 2000]</td>
<td>9</td>
<td>[1135, 3542]</td>
</tr>
<tr>
<td>5</td>
<td>[412.5, 250]</td>
<td>10</td>
<td>[1231, 1989]</td>
</tr>
<tr>
<td>6</td>
<td>[412.5, 1000]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[825, 4000]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[206.25, 500]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>[412.5, 2000]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>[412.5, 4000]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C. Parameter Settings

The parameter settings for dual-chromosome GA is listed in Table III. In addition to our proposed operators, we apply the elitism [34] with size 5 and tournament selection [35] 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 to test the statistical significance.

D. Experiment Results and Analysis

The performance of our proposed NSGA-II is far better than the other three algorithms. Fig.3a shows the solutions from 4 algorithms of test case 8 and Fig.3b shows the zone-in comparison of two GAs. We only show one out of eight results because their patterns are similar. We plot the combined results of 30 runs from two GAs and solutions from rule-based approaches. As we minimizing both energy consumption and the failure probability, better results are closer to the origin. As the results showed, the result from FF&BF/FF is far worse than other approaches at both objectives. Spread has the best failure probability but it also has the largest energy consumption. Compared to two GAs, the results from dual-chromosome GA are dominated by our proposed NSGA-II approach.

Fig. 3: The best solutions found in four algorithms in test case 8.
same PM. Hence, the failure probability is high. Secondly, BF selects the smallest VM to allocate a container. This strategy creates many small VMs that causing a large amount of VM overheads and fragmented resources inside VM that cannot be used. The number of VM can be seen in Fig.4. FF&BF/FF creates the most number of VMs and then followed by Spread. Both GAs use much fewer VMs.

![Fig. 4: Number of VMs that four algorithms used in the test cases with real-world VMs](image)

Later generations of solutions are pushed towards the original and we may observe the solutions are converging as more and more solutions are overlapping. This means they can hardly find better solutions for a long time.

To compare two GAs, our proposed NSGA-II has two advantages. The first advantage is that NSGA-II uses a group representation inspired by group GA. In dual-chromosome GA, because the vector-based representation needs to be decoded to evaluate, 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. Dual-chromosome GA 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. The second advantage is that our NSGA-II provides a set of solutions that has a trade-off between energy consumption and availability. Also, NSGA-II provides a set of solutions that has a trade-off between energy consumption and availability.

Fig. 5: The evolution of Pareto front in NSGAII from test case 8 run 27.

VI. Conclusion

This work proposes a NSGA-II approach to solving the micro-service allocation problem in container-based clouds. Our approach considers the dependencies between micro-services of applications, and two conflict objectives, minimizing energy consumption and maximizing application availability. Our proposed NSGA-II adopts a group-based representation and embedded with bin-packing heuristics in the genetic operators. We run experiments on real-world datasets with comparison with three state-of-the-art algorithms: FF&BF/FF, Spread, and a dual-chromosome GA. The results show that our proposed NSGA-II outperforms all other approaches in both objectives. Also, NSGA-II provides a set of solutions that has a trade-off between energy consumption and availability.

REFERENCES


