2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Composite SaaS Scaling in Cloud Computing using a Hybrid Genetic Algorithm

Zeratul Izzah Mohd Yusoh
Faculty of Information and Communication Technology
Universiti Teknikal Malaysia Melaka
Melaka, Malaysia
zeratul@utem.edu.my

Maolin Tang, *Senior Member IEEE*
Science and Engineering Faculty
Queensland University of Technology
Brisbane, Australia
m.tang@qut.edu.au

***Abstract*—A Software-as-a-Service or SaaS can be delivered in a composite form, consisting of a set of application and data components that work together to deliver higher-level functional software. Components in a composite SaaS may need to be scaled – replicated or deleted, to accommodate the user's load. It may not be necessary to replicate all components of the SaaS, as some components can be shared by other instances. On the other hand, when the load is low, some of the instances may need to be deleted to avoid resource underutilisation. Thus, it is important to determine which components are to be scaled such that the performance of the SaaS is still maintained. Extensive research on the SaaS resource management in Cloud has not yet addressed the challenges of scaling process for composite SaaS. Therefore, a hybrid genetic algorithm is proposed in which it utilises the problem's knowledge and explores the best combination of scaling plan for the components. Experimental results demonstrate that the proposed algorithm outperforms existing heuristic-based solutions.**

***Index Terms*—Cloud Computing, Composite SaaS, Clustering, Grouping Genetic Algorithm.**

I. Introduction

Cloud computing [1] offer users off-premises high performance IT facilities, including applications, data and computation resources. Its services can be categorised into three main categories: 1) Infrastructure as a Service (IaaS), [1], 2) Platform as a Service (PaaS), and 3) Software as a Service (SaaS) [2].

One of the essential characteristics of the Cloud services is to provide access to a large pool of computation resources in which the resources can be dynamically provisioned based on the demand. The automated provision mechanism in a Cloud is responsibles for adding or removing the resources for a particular Cloud service, such that the usage of the resources is minimised while its performance is maintained. The process of adding/removing the resources based on the demand represents the scalability of the service [3]. It can be classified into two categories – vertical scaling and horizontal scaling. Vertical scaling, is where more computation resources are added to the service while in horizontal scaling, more similar services are created and users' demands will be directed to the appropriate service. Both categories of scaling are triggered by scalability metrics including the number of current users, the quality of solution of the service and the number of incoming requests [4], [5], [6]. In a Cloud data centre, the scaling approaches are implemented at two different levels: 1) at the infrastructure level to scale the servers, and 2) at the application level to scale the application components [3], [4], [5].

This research focuses on SaaS as the problem domain as SaaS is receiving substantial attention today from both software providers and users. Gartner forecast that the SaaS revenue would reach $22 billion by 2015 [8]. This raises new challenges for SaaS providers managing the SaaS, especially in large-scale data centres. A SaaS can be delivered as a composite application in which the software is composed from a group of loosely coupled individual applications that communicate with each other in order to form a higher-level functional system or application [9]. Delivering the SaaS in such an approach allows flexibility of the SaaS functionalities, where components can be combined and recombined as needed. In addition, SaaS providers can gain a number of benefits including reduced delivery cost, flexible offers of the SaaS functions and decreased cost of subscription for users. However, this type of delivery also raises several new challenges concerning the scaling process of the SaaS in a Cloud data centre. The new challenges are: 1) The performance of a composite SaaS is measured by a set of interacting application components instead of by a single component. 2) A particular component in a composite SaaS may be requested more by users than other components in the same SaaS. Thus, it is an additional challenge to determine which component is more suitable to scale, and how many replicas are needed for that particular component. 3) The placement of a new replica of a composite SaaS has to consider its communication with other components. It is assumed that more than one application component can be placed in a virtual machine (VM). As such, to some extent, the replication process for a composite SaaS has to consider both the application and infrastructure level scalability metrics.

A large number of literature has been published on SaaS scalability in a Cloud. However, most works proposed solutions catering for an atomic SaaS, thus ignoring the new challenges noted above. In addition, the solutions are mainly developed at the infrastructure level for IaaS management, whilst this paper focuses on solutions at the application level to handle the composite SaaS. To address these new challenges, a new scaling algorithm for a composite SaaS that uses

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

information at both the application and infrastructure layers in making the decisions is proposed.

The remainder of the paper is organised as follows. Section II reviews the relevant literature. Section III formulates the problem. Section IV presents a hybrid genetic algorithm for the problem. Section V evaluates the algorithm using simulations. Finally, Section VI summarises and concludes the paper.

II. Related Work

In this section, existing research on scalability is reviewed based on the scalability levels: infrastructure and application. The discussion will focus on replication and deletion as the mechanism of scalability.

At the infrastructure level, there are a couple of commercial Cloud providers that offering add-on features that aim to scale the IaaS that users are renting. For instance, Amazon[1] offers Amazon Cloud Watch and Amazon Auto-scale for its EC2 service. Amazon EC2 service provides resizable computation capacity for users. A unit of EC2 can be considered as a VM that has its own memory, processing capacity and storage. Amazon provides Cloud Watch to monitor the performance of each EC2, including its CPU utilisation, disks read and write and network traffic. This information is then used in Amazon Auto-scale, in which users determine their scalability rules to replicate or destroy EC2 in order to cope with their demand or to minimise their costs. The scalability solutions offered by Cloud providers focus on servers' resources in respect of the scalability metrics. While they work well for handling VM scalability, finer granularity is needed to handle application scalability, particularly for applications like composite SaaS.

One of the most recent research at the application level is that of Rodero-Merino et al. [4]. In this research, the authors differentiated the roles of Service Providers (SPs) between the one owning the service and Cloud provider, the one owning the infrastructure. Another abstraction layer, named Claudia, is proposed and implemented to offer a high level interface for SPs in managing their services. Application components under Claudia management will be replicated or deleted based on the selected scalability rules. Claudia provides no solution concerning the placement of the replica, as this is under the authority of the Cloud provider. Lee and Kim [7] also proposed an application-oriented solution for a scalable service in the Cloud. In their work, the replication of an application is triggered if the application exceeds the threshold for the response time. All the studies described above have provided a convenient way to manage application scalability using application-oriented metrics, as opposed to infrastructure-oriented metrics. However, the replication and deletion process is still done at the VM level instead of the application level. As such, there might be some applications that are forced to be replicated due to the replication of their host.

Bonvin et al. [10] proposed geographically diverse application component replication in a Cloud infrastructure. In their solution, each component acts as an agent that rents resources from its host server. The agent migrates, deletes and replicates itself based on its economic fitness, where the fitness represents the value of the utility of the component as compared to its cost. The placement of the replica is based on the geographical location, where the least loaded and closest server from its communication component will be chosen. Wu et al. [6] investigated the scalability of composite web services in a Cloud based on the productivity of the service, which is measured based on the bandwidth consumed by a particular composite service. They designed a GA to search the best replication and placement plan for the web service, such that the productivity of the scaled service is maximised. However, none of these studies considers the overall performance of the application as its performance indicator. Although to some extent the dependency of one application on another application is considered, the evaluation of the action taken is still independent. This differs for a composite SaaS that has an overall target performance to meet.

The prior research discussed above has motivated the direction of this research towards studying the replication problem for a composite SaaS in the Cloud data centre. This is due to the challenges resulting from a composite SaaS replication problem, which may be more suitably addressed to both the application and infrastructure layers than to one layer only. In addition, to utilise the resources and to avoid unnecessary replication, the granularity of the replication must be at the application level. The overall performance of the SaaS must also be considered in evaluating the replication decisions. To address such issues, a hybrid genetic algorithm for a composite SaaS replication problem is proposed. The following sections will describe the algorithm in detail.

III. Problem Formulation

Given a set of computation servers, storage servers, the Cloud communication network, the composite SaaS, and the set of tenants, the objective is to determine which SaaS component should be replicated/deleted, how many replicas to create/delete and where to place the new replica in the Cloud data centre such that the performance of the SaaS complies with its constraints while minimising the Cloud running cost. These inputs, constraints and output can be formulated as below:

Input:

1) A Cloud data centre, $D = \langle CS \cup SS, E \rangle$, where
   - $CS = \{cs_1, cs_2, ..., cs_n\}$ denotes the set of all computation servers in D, and n is the number of computation servers. The resource capacities and virtual machines of each computation server are represented in a tuple $\langle pc_i, mem_i, disk_i, VMC_i \rangle$, $1 \leq i \leq n$, where pc_i is the processing capacity, mem_i is the memory, $disk_i$ is the disk storage capacity, and VMC_i is the set of VMs for cs_i. $VMC_i \subseteq VM$ where VM is the set of all virtual machines. Each VM is defined as

[1] http://aws.amazon.com

$vm_{ij} = \langle pc_{ij}, mem_{ij}, disk_{ij}, price_{ij} \rangle$, $j \in \mathbb{N}$, where $price_{ij}$ is the cost of the VM.
- $SS = \{ss_1, ss_2, ..., ss_m\}$ denotes the set of all storage servers in D, and m is the number of storage servers.
- E is the set of undirected edges connecting the vertices, if and only if there exists a communication link transmitting information from v_i to v_j, where $v_i, v_j \in CS \cup SS$. $B_{v_i,v_j} : E \rightarrow \mathbb{R}^+$and $L_{v_i,v_j} : E \rightarrow \mathbb{R}^+$ are the bandwidth and latency functions of the link from v_i to v_j respectively.

2) A set of composite SaaS, $S = \langle AC \cup DC, DAC, DDC \rangle$, where
- $AC = \{ac_1, ac_2, ..., ac_x\}$ denotes the set of all application components in S, and x is the number of application components. The resource requirements for each component represented in a tuple $\langle pcReq_i, memReq_i, size_i, amountRW_i, maxUser_i \rangle$, $1 \leq i \leq x$, where $pcReq_i$ is the requirement for processing capacity, $memReq_i$ is the memory requirement, $size_i$ is the size of the component for disk storage requirement, $amountRW_i$ is the amount of read/write to other communication components, and $maxUser_i$ is the maximum user for ac_i.
- $DC = \{dc_1, dc_2, ..., dc_y\}$ denotes the set of data components for S, and y is the number of data components.
- S modelled by directed acyclic graphs (DAGs), DAC is the the set of dependencies between application component where $DAC = AC \times AC$ and DDC is the set of dependencies between application and data components where $DDC = AC \times DC$.

3) The current placement of S and its placement constraint: a) A current placement configuration, P, of application components, AC, onto VM, $P : AC \rightarrow VM$ where $ac_i \mapsto P(ac_i) = vm_k$, and $1 \leq i \leq x$, $1 \leq k \leq j$. b) A current location, L, of the data components, DC, at storage servers, SS, $L : DC \rightarrow SS$ where $dc_x \mapsto L(dc_x) = ss_z$, and $1 \leq x \leq y$, $1 \leq z \leq m$.

4) A response time for the composite SaaS, rt_s.

5) A set of tenant, $T = \{T_1, T_2, T_i, ..., T_v\}$, and $1 \leq i \leq v$. Each tenant may have one or more users denoted as $\langle u_i \rangle$.

Constraints:
- Resource constraint: The placement of application components onto VM are subject to the VM capacities.
- Placement constraint: There are two types of placement constraint: a) An anti-location constraint that determines the list of virtual machines that should not be considered for hosting a specific component, $ac_{i,j}$. The list is defined as $AL = \{(ac_{i,j}, vm_{x,y})_z, ...\}$ where $z \in \mathbb{N}$, b) An anti-colocation constraint that determines the list of application components that cannot be placed in the same virtual machine. The list is defined as $ACL = \{(ac_{i,j}, ac_{s,t})_w, ...\}$where $w \in \mathbb{N}$. The solution must comply with the anti-location and anti-colocation constraint defined in the lists.
- Response time constraint: This constraint enforces the total response time of the SaaS, TET, to be bounded by rt, $TET \leq rt_s$. Calculation of the TET is similar to that defined in in our previous work [11], with some modification in which the changes are made to include the affect of having more than one replica as well as to include the number of tenants.

Output: A scaling plan for application components including its replication/deletion and the placements.

Objectives: The objectives of the problem are:

1) to minimise the number of computation servers used,$min \sum_{cs_i \in CS} d_{cs_i}$where

$$d_{cs_i} = \begin{cases} 1, & \forall_{j \in AC} \exists_{i \in CS} \mid P(ac_j) = cs_i \\ 0 & otherwise \end{cases}$$

2) to minimise the cost of VM used, $min \sum_{vm_i \in VM} cost_{vm_i}$where

$$cost_{vm_i} = \begin{cases} price_{vm_i}, & \forall_{j \in AC} \exists_{i \in VM} \mid P(ac_j) = vm_i \\ 0 & otherwise \end{cases}$$

3) to minimise the number of replica created, $min \sum_{ac_i \in AC} k_{ac_i}$, while satisfying all the constraints.

IV. A HYBRID GA

It has been proven in the existing literature that deciding how many replicas for an object and where to place them is an NP-hard problem [12]. Therefore, a hybrid GA (HGA) is developed to address the problem formulated in Section III. A composite SaaS consists of two types of components – application and data. For this scaling problem, only the application component will be considered for replication; however, communication between these two types of components is formulated as one of the deciding factors in finding the solution.

The proposed algorithm will find a near-optimal scaling plan for a composite SaaS such that it can minimise the total resource usage without violating its constraints. There are two main processes in producing the scaling plan: 1) selection of components to replicate/delete and the number of replicas, and 2) the placement of the replica. The HGA also utilises two types of domain knowledge in order to improve its performance, as outlined below.

Knowledge at the application level: Each component has a number of maximum users it can serve at a time. It is assumed that this knowledge is given by the SaaS developers based on their own experience and knowledge of the SaaS. This knowledge will then be used for calculating the total execution time of the SaaS, given the current users at a particular time. The response time is formulated as one of the constraints in this replication problem.

Knowledge at the infrastructure level : At the infrastructure level, the algorithm will use the knowledge of the location and the utilisation rate of VMs in order to decide the placement of a replica.

Algorithm 1 describes the HGA. In the HGA, Steps 1 and 2 are to initialise the best solution and the best replication plan. Step 3 is responsible for generating the initial population using domain knowledge, instead of the traditional random generation of the initial population. The initial population generation procedure is described in a later section. Each individual is checked for resource requirement constraints in Steps 6 and 7. Steps 8 and 9 are responsible for evaluating the fitness of all individuals in the population. The best replication plan and best fitness will be stored. Steps 12 to 15 are the evolving processes in which selection, crossover, mutation and fitness evaluation are performed. The following sections explained the main parts of the algorithm in detail.

Algorithm 1: Hybrid Genetic Algorithm

1 $bestFitness = 0$
2 $bestPlan = 0$
3 create an initial population $Population$ of $PopSize$ individuals, based on the initialisation procedure in Section 1.4.2
4 **while** *termination condition is not true* **do**
5 **for** $X \in Population$ **do**
6 **if** X *violates SaaS resource requirements constraint* **then**
7 Repair(X)
8 **end**
9 Calculate X fitness value, $F(X)$, penalised if X violates SaaS placement constraint and response time constraint
10 **if** $F(X) > bestFitness$ **then**
11 Replace $bestFitness$
12 $bestPlan = X$
13 **end**
14 **end**
15 Select individuals from the $Population$ based on roulette wheel selection
16 Probabilistically apply the crossover operator to generate new individual
17 Probabilistically select individuals for mutation
18 Use the new individuals to replace the old individuals in the $Population$
19 **end**
20 output $bestFitness$
21 output $bestPlan$

A. Genetic Encoding

The chromosome is encoded by three one-dimensional parallel integer arrays of n genes, where n is the number of components in the composite SaaS. The first array represents the component, the second represents the scaling flag, and the third array stores the VM for the replica(s). The scaling flag is encoded by a signed integer where a positive value means the number of component's instances created, a negative value means the number of component's instances deleted, and 0 means the current number of instances remains. Figure 1 illustrates the representation of one chromosome with n application components.

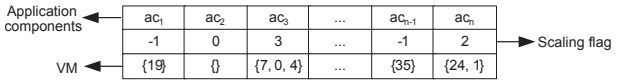

Figure 1. An example of a chromosome representation

B. Initial Population Generation

To ensure that the population starts off the exploration with good seeds, HGA uses a procedure that utilises the domain knowledge. There are two tasks involved in generating the initial population.

The first task is to determine the number of replicas for each component. The value will be generated from a range, with a lower bound of 1 (no replication at all). The upper boundary is determined using the results produced by a random replication algorithm. The algorithm will randomly replicate the component. If the replication can shorten the SaaS total execution time, the replica will be kept. The process will continue until the time constraint is satisfied or until there is no improvement for 100 iterations. The result of each component will be stored as the upper boundary for the component.

The second task in generating the initial population is to find the placement for the new replica or to select a replica to delete. For the deletion process, a replica in a least-loaded VM will be selected. For the placement, the least utilised VM and closest to its communicating component is chosen. The utilisation of a VM is calculated based on the SaaS demands for the computation resources, including processing capacity, memory, secondary storage and bandwidth. The composite SaaS is served to a number of tenants in which each tenant has a different number of users. It is assumed that the resource demands of the SaaS components are linearly proportional to the number of tenants and the total number of users for all tenants [13].

C. Genetic Operators

The crossover and mutation operators are responsible for exploring the search space for the HGA. A single-point crossover is implemented where in this crossover, two chromosomes are selected based on the roulette wheel selection scheme. A randomly selected point in each of the chromosomes is chosen and the part after the point is exchanged and merged together to produce two child chromosomes. The chromosomes will be checked for resource requirement constraints and will be repaired accordingly. The best two chromosomes from the parents and child chromosomes will be copied to the next generation.

The chromosomes are randomly selected for mutation operation. A new value for the replication flag of a component

is generated based on the component's replica range. Based on the new value, the replication or destroy process will be carried out.

D. Fitness Function

As formulated in Section III, the problem has three different objectives that need to be optimised and constraints that need to be complied with. The HGA uses a weighted aggregation approach in which each objective and constraint is given a weight value depending on its importance in the context of the problem. The sum of all the weight values is one. Then a composite objective function is formed by summing the weighted objectives and converting them into a single objective optimisation.

Equations 1 to 3 provide the definition of the objective functions of the problem, and Equations 4 and 5 give the value of the constraint violation for the placement and response time constraint:

1) $F_{obj_1} = 1 - \left(\frac{R_{max} - \sum_{cs_i \in CS} d_{cs_i}}{R_{max}}\right)$ where R_{max} is the maximum number of possible replication for all components, defined as $R_{max} = UB \times |AC|$, and d_{cs_i} is the number of computation server used by the SaaS.
2) $F_{obj_2} = 1 - \left(\frac{R_{maxCost} - \sum_{vm_i \in VM} cost_{vm_i}}{R_{maxCost}}\right)$ where $R_{maxCost}$ is the maximum cost for R_{max}, defined as $R_{maxCost} = R_{max} \times max(cost_{vm})$ and $cost_{vm_i}$ is the VM cost for the SaaS.
3) $F_{obj_3} = 1 - \left(\frac{R_{max} - \sum_{ac_i \in AC} k_{ac_i}}{R_{max}}\right)$ where k_{ac_i} is the number of replica created for ac_i.
4) $CV_1 = \frac{\sum_{ac_{i,k} \in R_i} PC}{\sum_{ac_i \in AC} k_{ac_i}}$ where $PC = 1$ if the placement of component ac_i violated its constraint and k_{ac_i} is the number of replica created for ac_i.
5) $CV_2 = \begin{cases} 1, & TET \geq rt_s \\ 0, & otherwise \end{cases}$ where TET is the total execution time of the SaaS, and rt_s is the response time constraint.

A penalty function will be used to determine the degree of infeasibility raised by violation of these constraints. The fitness function is designed so that an infeasible solution has a lower fitness value than any feasible solution, and an infeasible individual that violates more constraints will be penalised more than the solution that has lower constraint violations.

Equation 6 defines the fitness function of the problem.

$$F(X) = \begin{cases} \sum_{j \in 3} F_{obj_j}(X) \times w_{obj_j} & \\ +0.3, & \sum_{i \in 2} CV_i = 0 \\ \sum_{j \in 3} F_{obj_j}(X) \times w_{obj_j} & \\ \times \left(1 - \frac{\sum_{i \in 2} CV_i}{|CV|}\right), & otherwise \end{cases} \quad (6)$$

where $\sum_{j \in 3} w_{obj_j} + 0.3 = 1$.

In Equation 6, the fitness of an individual is determined by the sum of objective functions multiplied by its weightage. If X is a feasible solution, its fitness will be added by 0.3, where this value represents the reward for feasible solutions. This value is obtained through a number of trial experiments in which the value that produced the best solutions is used. Otherwise, its fitness will be multiplied by the expression $\left(1 - \frac{\sum_{i \in 2} CV_i}{|CV|}\right)$, which guarantees that the more constraints that an infeasible individual violates, the lower is its fitness.

V. Experiments

The scalability and quality of the HGA were tested on a number of problem instances with different sizes and complexities. The size of the problem is determined by three dimensions: 1) the number of Cloud servers in the problem, 2) the number of tenants and their corresponding users, and 3) the number of applications and data components. Three types of problem representing each of the dimensions are created to evaluate how these dimensions affect both the quality of the solution, produced by HGA, and its scalability. The quality of solution is evaluated based on the objective function as a whole as well as on the three objectives separately. For scalability evaluation, the computation time taken to produce the solution is measured. Below is the elaboration of these three test problems.

Test problems with different numbers of Cloud servers: Five test problems were constructed with different numbers of Cloud computation servers and storage servers. For computation servers, the number ranges from 100 to 500, while for storage servers the number ranges from 60 to 300. The attributes of the Cloud servers were randomly generated using the models presented in the Hewlett-Packard and IBM websites [14]. The communication between the servers was also generated randomly. For all test problems, the number of application components is set to 10, data components to 5 and the number of tenants to 200.

Test problems with different numbers of tenants : Five test problems were created with a fixed number of computation servers (400), storage servers (240), application components (10), and data components (5), and with varying number of tenants ranging from 300 to 500, with an increment of 50. For each tenant, the minimum user is set to 1, the maximum number of users is 200 and the value will be generated randomly.

Test problems with different numbers of application components : Four test problems were designed with composite SaaS with different numbers of application and data components and a fixed number of Cloud servers and tenants. The number of application components ranges from 5 to 20, while the number of data components is from 2 to 10. The number of computation servers is set to 500, storage servers is set to 300, and the number of tenants, to 200.

A. A Greedy Algorithm

Although comparing the HGA to the optimal solution obtained by an exhaustive search is the best way to illustrate the performance of the algorithm, such a search is able to provide an optimum solution for small size problems only. Since this is not the case, a simple greedy algorithm is developed for comparison. Basically, the greedy algorithm will replicate

a component if the replication brings benefit to the SaaS performance in terms of its total execution time. It is assumed that at the start of the algorithm, all components have only one instance. Algorithm 2 shows the greedy replication algorithm.

Algorithm 2: The Greedy Algorithm

```
count = 0
while (TET_s > rt_s and count < 100) do
    for (ac_i ∈ AC) do
        Create a new replica for ac_i
        Find ac_i placement
        Calculate the new newTET_s
        if (newTET_s < TET_s) then
            Accept the replica and its placement
            count = 0
        end
        else
            count = count + 1
        end
    end
end
```

Table I
SIMULATION PARAMETERS FOR THE EXPERIMENTS

Parameter	Value
Population size	100
Crossover rate	90%
Mutation rate	10%
Termination condition (# of generation without improvements)	25
w_{obj_1}	0.25
w_{obj_2}	0.25
w_{obj_3}	0.2

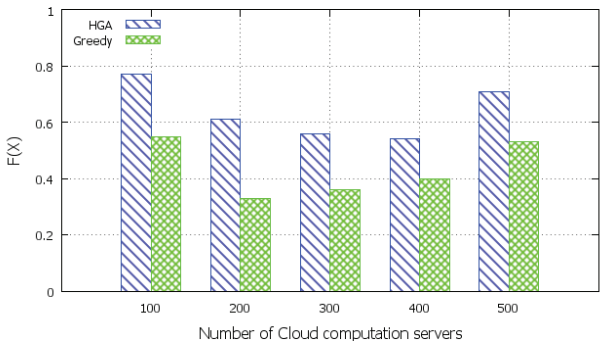

Figure 2. Comparison of the overall objective function, $F(X)$, of the HGA and the greedy algorithm for different numbers of Cloud computation servers

B. Evaluation

A number of experiments were conducted to evaluate the scalability and effectiveness of the proposed algorithm. In order to conduct the evaluation, the HGA and the greedy algorithm was implemented in Microsoft Visual Studio C++ 2010. All the experiments were conducted using a desktop computer with 3 GHz Intel Core 2 Duo CPU and 4GB RAM. The parameter setting for HGA is illustrated in Table I. These parameters were obtained through trials on randomly generated test problems. Parameters that led to the best performance in the trials were selected as the settings of the algorithms for the experiments below.

1) Experiments on the Number of Servers : Considering the stochastic nature of HGA, and that the random element exists in the greedy algorithm, the experiments were executed 30 times for all problem instances. The objective of the problem is to minimise the replication plan's cost while satisfying the SaaS constraints with the minimum number of replicas possible. The cost here refers to the resource cost that is used for the components, which includes the number of computation servers (CS) used and the price of VM.

To evaluate the overall performance of the algorithms, an overall objective function, $F(X)$, is formulated based on the fitness function in Equation 6. The overall objective function is defined in Equation 7. The $F(X)$ value is based on the sum of scores for each category multiplied by its weightage. High values of $F(X)$ indicate good solutions. The weightage for CS used and VM cost is set to 0.4, while the number of replicas is set to 0.2. Figure 2 illustrate the value of $F(X)$ for both algorithms for the five test problems. Based on the figure, the HGA has higher values of $F(X)$ in all test problems, which indicates that it produced a better replication plan for the problem.

$$F(X) = \sum_{j \in 3} F_{obj_j}(X) \times w_{obj_j} \quad (7)$$

The growth trend of the computation time of both algorithms is shown in Figure 3. From the figure, it can be seen that there is no correlation between the number of Cloud servers and the time taken by both algorithms. Although the greedy algorithm is fast in computation time, the quality of results produced are not convincing for the use of this algorithm for solving the replication problems. For all test problems, the best result achieved by the greedy algorithm for the number of replicas created, as well as for the CS and VM cost, is still higher than the average result of the HGA.

2) Experiments on the Number of Tenants : This experiment evaluated how the number of tenants affects the quality of solutions as well as the computation times of both the proposed algorithm and the comparison algorithm. Both algorithms were executed 30 times for each test problem.

The overall performance of both algorithms based on the objective function defined in Equation 7 is shown in Figure 4. It can be seen that both algorithms can always find feasible solutions for each test problem. It also shows that the HGA can always find a better solution for all the problem objectives in the five test problems, compared to the greedy algorithm.

For this experiment, it should be highlighted that, compared to the HGA, the greedy algorithm tends to generate more replicas as the number of tenants increased. Figure 5 shows

Figure 3. Comparison of the computation time of the HGA and the greedy algorithm for different numbers of Cloud computation servers

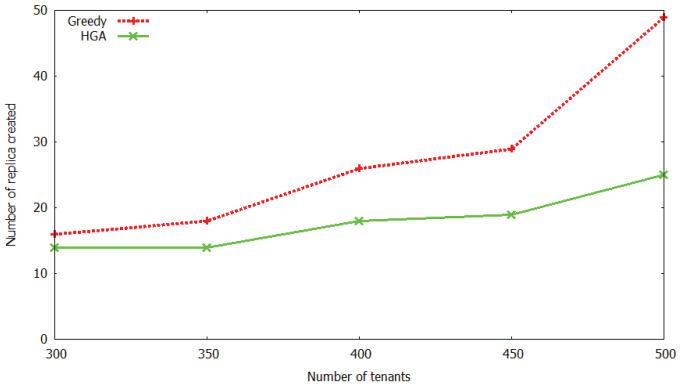

Figure 5. Comparison of the number of replica created of the HGA and the greedy algorithm for different numbers of tenant

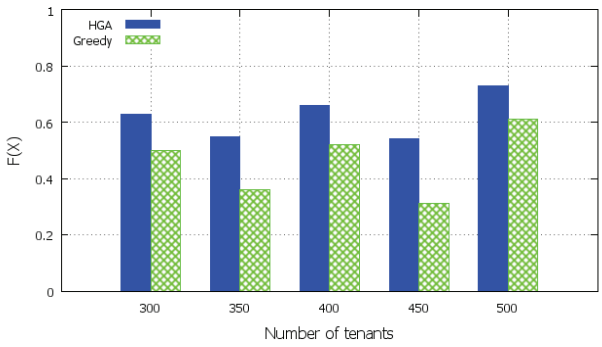

Figure 4. Comparison of the overall objective function, $F(X)$, of the HGA and the greedy algorithm for different numbers of tenants

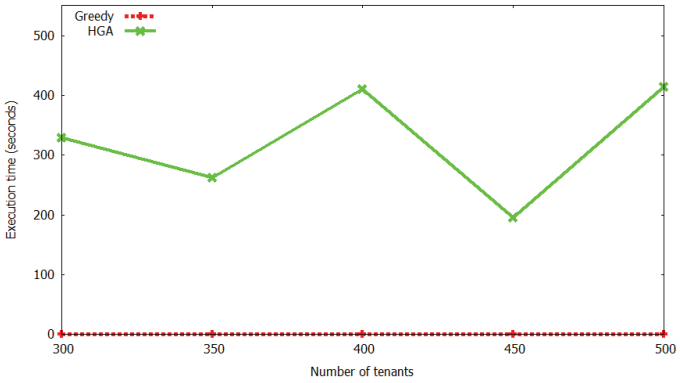

Figure 6. Comparison of the execution time of the HGA and the greedy algorithm for different numbers of tenant

the pattern for the number of replicas generated between these two algorithms. It can be seen that the gap between the two algorithms is increasing as the number of tenants grows. With new tenants added, the HGA manages to retain a minimal degree of replications, resulting in a linear increment. This is due to the non-deterministic nature of the HGA in deciding the replications. The greedy algorithm created far more replicas for a larger number of tenants.

Unfortunately, the better performance and the large cost savings of the HGA are achieved at the expense of a large execution time. As shown in Figure 6, the HGA computation time is much higher compared to that of the greedy algorithm. In addition, there is no correlation at all between the number of tenants and the computation time taken.

3) Experiments on the Number of SaaS Components: This experiment was an evaluation of how the number of SaaS application components affects the quality of solution and computation time of the HGA compared with the greedy algorithm. In each test problem, both algorithms were executed 30 times.

Figure 7 presents the overall performance based on the objective function in Equation 7. It can be seen that the HGA achieved better results than the greedy algorithm. The difference in the score for $F(X)$ for the test problem with 5 components is only about 5%, while in the test problem with 20 components the difference is 40%. This overall result is also consistent with the results for each criterion, and it is particularly obvious in terms of the number of replicas created. Figure 8 shows the replication trend for both algorithms; the gap between the two results increases as the number of SaaS application component increases. This observation should be attributed to the fact that, by having more components in the SaaS, the algorithms have more options of components to be replicated, together with additional dependency complexities caused by the communication between the components. The HGA tackles these effects more successfully than the greedy algorithm does, through its better exploration of which of the components are worth replicating.

No trend was observed for the computation time for the HGA as the number of application components increases. The longest time taken is around 6 minutes. On the other hand, the greedy algorithm took less than one second for all test problems conducted.

In the above experiments, HGA achieves more cost savings in its replication plan as well as the replica placement plan in all variable dimensions. Compared to the greedy algorithm, HGA can save up to 46% in overall performance when

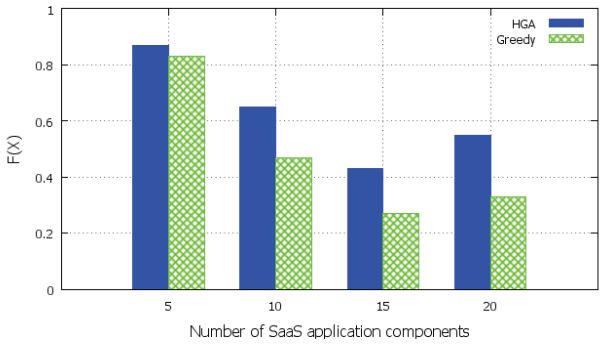

Figure 7. Comparison of the overall objective function, $F(X)$, of the HGA and the greedy algorithm for different numbers of SaaS application components

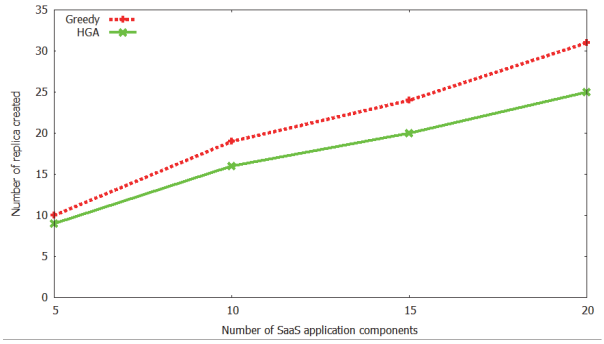

Figure 8. Comparison of the number of replica created of the HGA and the greedy algorithm for different numbers of SaaS application components

evaluated with a different number of Cloud servers, and up to 43% and 40% when evaluated with various numbers of tenants and SaaS components. It has also been shown that the proposed algorithm achieves far better results than the greedy method when the number of tenants and the number of application components are both large. This shows that the algorithm can respond well to changes of demands and to the increment of SaaS components. It also demonstrates the exploring ability of the algorithm in a large and complex search space with constraints. However, the performance of the HGA is a trade-off with its computation time. The HGA did not exhibit any trends of computation time taken in all the experiments conducted. The longest time it took is about eight minutes, while the greedy method, on average, took less than one second for all the experiments.

VI. Conclusion

This paper has studied a new composite SaaS scalability problem in Cloud. The problem has been formulated as a combination of the components' scaling and placement problems, with the main objective being to minimise the cost of resources with the minimum number of replicas possible while satisfying the problem's constraints. A hybrid genetic algorithm has been proposed. The hybrid genetic algorithm makes good use of the problem's domain knowledge at the application and infrastructure levels to make its search more efficient and effective. The simulation results have shown that the proposed algorithm constantly outperforms a heuristic algorithm in terms of the quality of solution. The proposed algorithm gives an impressive performance by achieving low cost replication and placement plans compared to the performance of the greedy algorithm. Although the results show that the computation times for HGA are long, it is still acceptable as the algorithm is meant to be executed during the offline maintenance phase of the SaaS.

This problem has multiple objectives but it was transformed into a single objective problem. Therefore, a possible future work is to implement a multi-objective evolutionary algorithm. However, further works need to be carried out, as the involvement of human administrators is not available in an automated SaaS resource management

Acknowledgments

This research was carried out as part of the activities of, and funded by the Smart Services Cooperative Research Centre (CRC) through the Australian Government's CRC Programme (Department of Innovation, Industry, Science and Research). The research was sponsored by Universiti Teknikal Malaysia Melaka, Malaysia.

References

[1] I. Foster, Z. Yong, I. Raicu, S. Lu, Cloud computing and grid computing 360-degree compared, Proceedings of the Grid Computing Environments Workshop, IEEE, 2008, pp. 1–10.

[2] L. M. Vaquero, L. Rodero-Merino, J. Caceres, M. Lindner, A break in the clouds: Towards a cloud definition, SIGCOMM Computer Communication Review 39 (1), 2009 pp. 50–55.

[3] J. Caceres, L. M. Vaquero, L. Rodero-Merino, A. Polo, J. J. Hierro, Service scalability over the cloud, Handbook of Cloud Computing, 2010 pp. 357–377.

[4] L. Rodero-Merino, L. M. Vaquero, V. Gil, F. Galan, J. Fontan, R. S. Mon- tero, I. M. Llorente, From infrastructure delivery to service management in clouds, FGCS 26 (8), 2010 pp. 1226–1240.

[5] L. M. Vaquero, L. Rodero-Merino, R. Buyya, Dynamically scaling applications in the cloud, ACM SIGCOMM Computer Communication Review 41 (1), 2011 pp. 45–52.

[6] J. Wu, Q. Liang, E. Bertino, Improving scalability of software cloud for composite web services, Proceedings of the IEEE International Conference on Cloud Computing, IEEE, 2009, pp. 143–146.

[7] J. Y. Lee, S. D. Kim, Software approaches to assuring high scalability in cloud computing, Proceedings of the IEEE 7th International Conference on e-Business Engineering, IEEE, 2010, pp. 300–306.

[8] G. Inc, Gartner says worldwide software-as-a-service revenue to reach $14.5 billion in 2012, 2012. URL http://www.gartner.com/it

[9] C. S. Inc, Cisco service-oriented network architecture: Support and optimize SoA and web 2.0 applications, [Online; accessed 9-March-2010] 2008. URL http://www.cisco.com/

[10] N. Bonvin, T. G. Papaioannou, K. Aberer, An economic approach for scalable and highly-available distributed applications, Proceedings of the IEEE 3rd Int. Conf. on Cloud Computing, IEEE, 2010, pp. 498–505.

[11] Z. Mohd Yusoh, M. Tang, A penalty-based GA for the composite saas placement problem in the cloud, Proceeding of IEEE Congress on Evolutionary Computation (CEC), IEEE, 2010, pp. 600–607.

[12] M. Karlsson, C. Karamanolis, Bounds on the replication cost for QoS, Tech. rep., Citeseer 2003.

[13] T. Kwok, A. Mohindra, Resource calculations with constraints, and placement of tenants and instances for multi-tenant SaaS applications, Proceedings of the 6th Int. Conf. on SoC, Springer, 2008, pp. 633–648.

[14] IBM, System & servers, [Online; accessed 9-November-2011] (2009). URL http://www-07.ibm.com/storage/au/