Immune Network Algorithm applied to the Optimization of Composite SaaS in Cloud Computing

Simone A. Ludwig and Kevin Bauer
North Dakota State University
Fargo, ND, USA
simone.ludwig@ndsu.edu

***Abstract*—In order to serve the different application needs of the different Cloud users efficiently and effectively, a possible solution is the decomposition of the software or so-called composite SaaS (Software as a Service). A composite SaaS constitutes a group of loosely-coupled applications that communicate with each other to form higher-level functionality. The benefits to the SaaS providers are reduced delivery cost and flexible SaaS functions, and the benefit for the users is the decreased cost of subscription. For this to be achieved effectively, the optimization of the process is required in order to manage the SaaS resources in the data center efficiently. In this paper, the optimization task of composite SaaS is investigated using an Immune network optimization approach. The approach makes use of activation and suppression that are mimicked by the natural immune system triggering an immune response not only when antibodies interact with antigens but also when they interact with other antibodies. Experiments are conducted with a series of SaaS configurations and the proposed immune network algorithm is compared with a formerly proposed grouping genetic algorithm. The results show that the immune network algorithm outperforms the grouping genetic algorithm.**

I. Introduction

Cloud computing is the provisioning of computer processing, networks, data, and applications to a consumer over the Internet [1]. This is becoming the preferred way for businesses to provision resources so they may outsource part of their workload to reduce costs in labor and in maintaining hardware [2], [3]. These resources are delivered on demand and are scalable in a pay-as-you-go approach.

Cloud computing can also be comprised of a large quantity of physical machines with heterogeneous computing resources distributed around different geographic locations [3]. The consumer is unaware of the infrastructure or platform details of the system, but expects that the service is always operational and that the service performs as if it always has the optimal amount of resources available on-demand.

The scalability of cloud computing must address the scaling up or the scaling down of resources. In addition, the balance between proactively scaling resources and reactively scaling resources has to be achieved [3]. For example, when a consumer no longer requires a service, the resources are quickly relinquished. Or, when a consumer requires a service, resources are allocated quickly. These operations must be implemented in a manner that is non-disruptive.

Although cloud computing by definition is still evolving [4], [5], cloud computing usually falls into one or a hybrid of three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

IaaS is the provisioning of computer processing, networks, and storage to the consumer so that their operating systems and software applications may utilize them. An example of IaaS is Amazon's EC2 (Elastic Cloud Computing) [6]. However, consumers must still manage the services that run on IaaS. For example, operating systems must be updated, databases must be administered, and, if consumers are administering a web service traffic must be managed by them.

PaaS hides infrastructure details from the consumer that are visible in IaaS: database administration, load balancing, and server configuration. The PaaS provider manages infrastructures for customers. The consumer may then deploy on top of this service applications, libraries, and other tools [6]. A well-known example of PaaS is Google's App Engine [7].

SaaS, the service that is the focus of this paper, is applications that are available to a consumer via a client. Clients that connect to SaaS are usually browsers [6]. Examples of SaaS are Microsoft's Office 365 and Google Drive.

Optimally provisioning resources for cloud computing services is a difficult problem, because the physical machines that deploy the resources can be in different geographic locations yet together must sufficiently deliver services to the consumer. In addition, Virtual Machines (VMs) that run atop these physical machines appear to be uniform, but usually they are not [8]. The machines that comprise the infrastructure are heterogeneous with storage disks that read/write at different rates or the VMs are sharing physical machine resources with other VMs servicing other consumers. Load balancing is also a difficult problem in cloud computing. Migrating VMs to different physical machines without negatively affecting Quality of Service (QoS) is problematic [9].

Applications that demand resources should be provisioned enough to fulfill the Service Licensing Agreement (SLA), yet no more then is required. Then, service providers can distribute resources economically. There are many methods for determining the best placement for an application among a selection of VMs.

In this paper, the optimization task of composite SaaS

is investigated by searching for the optimal distribution of resources in a cloud computing environment. In particular, SaaS application components and the associate data components are to be placed onto computing servers and storage servers optimally. An immune network optimization approach is applied that is inspired by natural immune system using the idea of activation and suppression when antibodies interact with antigens.

The organization of this paper is as follows: In Section II, related work in the area of optimization applied to cloud computing is given. Section IV introduces and describes the proposed approach as well as the comparison algorithm. In Section V, the experiments conducted and results obtained are outlined. Section VI describes the conclusions reached from this study.

II. Related Work

Related work in the area of optimization in a Cloud environment include the following. For large scale cloud-computing, service deployment architectures may aid in efficiently provisioning resources. In [10], the cloud-computing environment is engineered to be SLA aware. Content Delivery Networks (CDN), like Akamai, contain edge servers that are physically close to clients and deliver content to them. The problem with this architecture is that all content is considered equal and unexpected bottlenecks can occur at these edge servers. For example, data in cloud environments can be active or passive. Active and passive data have different read/write requirements. Thus, [10] suggests providing different tiers of servers for different types of content and the resources that they require. First, the CDN is a tree of service resource providers. The leaf nodes are the edge servers that deliver content to customers. Next, each Named Node Server (NNS) and its corresponding Block Servers (BS), together they act as the edge servers, have multiple Resource Monitors (RM) and Resource Allocators (RA), the inner nodes of the tree. The RMs monitor the rates of data moving uplink and downlink from the BSs. As needed, the delta of the rate of data transfer that is bottlenecking and overloading the constraints set for the tiered BS can be sent to parent RAs in the CDN, ameliorating the breach in the SLA. The MiniMax algorithm strictly regulates the rates of resource transfer sent from children to parent nodes and from parent to children nodes. These methods coupled with h tiers of service in the BS servers also mitigate overloading the BSs, because each tier can be allotted its required resources that are usually expected for the type of content that they are holding.

The Hadoop Yarn system is commonly used for administering resources for scheduled tasks [11]. Resource managers assign application managers to worker nodes that monitor how many resources the worker nodes are using. The application managers also negotiate with the resource managers for resources to complete their scheduled tasks. Then, the resource managers will deliver the requirements according to the set policies. For example, one simple policy is queue based (FIFO). The tasks are scheduled for resources and Worker Nodes in the order of their arrival. Fair policies evaluate all the tasks and the resource managers attempt to distribute equal amounts of resources to all of the tasks in the job queue or the average equal share of the dominant resource requirements. The last common policy is the capacity policy. For each task the resource manager attempts to schedule an equal share of the resources for each task in the queue. The leftover resources that are still free are given to tasks that are using more than what was scheduled for them. And, some tasks within the queue are prioritized and provisioned resources accordingly. However, according to [11] these policies are not optimal.

Another approach uses the genetic algorithm for distributing resources by focusing on obtaining near optimal quality of service from infrastructures rather than finding optimal placement for VMs [12]. SaaSs are modeled as a cell. The cell is composed of application requirements: processing, memory, storage, network latency, and read/write times. These constraints must be fulfilled. VMs that satisfy the requirements are stochastically paired with the application components. After the pairings, the roulette wheel selection of the genetic algorithm and the probabilistic application of the crossover operation and the mutation operation are applied. Once these operations finish, if new cells were generated they are added to the population. Then, each cell's fitness is determined. The fitness calculation is the distance from constraints to constraint satisfaction, so the smaller the difference the better the fitness. Thus, the fitness function is to be minimized. The cell with the best fitness is employed and should have near optimal quality of service for its application components.

Another approach is based on an auction system. Markets are considered efficient distributers of resources. Therefore, modeled after market resource allocation, VMs are distributed via an auction system. Traditionally, cloud providers bundle together homogenous VMs that remain static, and then they sell these to the highest bidder. However, this is not always the most efficient use of the infrastructure [13]. This is because customer requirements and the VMs that are required are not homogeneous. Consequently, a method that takes into account the heterogeneous requirements of customers was proposed in [14].

For example, for each customer the cloud provider builds a number of VMs from a set of combinations of attributes. Customers bid on these VMs and the top bidder wins. Yet, this method is susceptible to shill bidding. Shill bidding occurs when a customer impersonates multiple customers to lower the price of VMs. In addition, customers may conspire to bid at lower prices to bring down the overall cost. These problems lead to low profits for the service provider. To make VM auction systems shill bidding proof and to maximize cloud provider revenue, a variation of the previously proposed method was proposed in [14], [15].

Another novel method for efficiently distributing resources in a cloud-computing environment is via the ant colony algorithm [16]. The cloud is modeled as an undirected graph. The vertexes are clusters of VMs, and the edges connect these resources in a cloud-computing environment. The clusters of VMs have boundary conditions on the load that they can

accept for processing, memory, storage, and bandwidth. If these boundaries are broken k times out of n, k and n are set arbitrarily, that vertex or rather that cluster of nodes is identified as a hotspot, meaning it is overloaded. To find available resources to offset the overloaded hotspot, the ant colony algorithm is employed to find a node in the undirected graph representing the cloud that can provision more resources.

For example, in the hotspot, VMs belonging to the cluster are searched for idle resources. Idle resources are the minimum amount of resources that are available from all VMs in that cluster. This constraint is enforced so other VMs are not overloaded. If none are found, then neighboring clusters are searched on a path from the hotspot. As neighboring clusters of VMs are searched for idle resources the following operations are performed: the pheromone density is calculated for each node visited on the path from the hotspot, the nodes are added to an avoidance list, and the pheromone evaporation rates are calculated for previously visited nodes. Eventually, a path to a node with idle resources obeying the constraints converges. Else, there are no nodes that satisfy the constraints to alleviate overloaded hotspots in the cloud-computing environment. One benefit for using this method is that the search process for finding idle resources in node clusters may utilize parallel processing.

III. Problem Formulation

The composite SaaS optimization problem can be described as follows. The SaaS application components and the associate data components are placed onto computing servers and storage servers, whereby the application components (ACs) are deployed within the virtual machine (VM) for execution. It is necessary for the VM hosting the ACs to have sufficient resources in order to guarantee users' service level agreement. Since the workload of Cloud data centers varies with time, the initial deployment may need to be modified. Therefore, scheduled reconfiguration of the VM is initialized at certain time intervals in order to maintain the performance of the application components and to minimize resource usage. This reconfiguration triggers the optimization of the composite SaaS. One way to achieve this is by combining two or more ACs into one VM. The placement reconfiguration needs to consider the communication or dependencies between the application components.

The problem is graphically displayed in Figure 1. The top row indicates the ACs with their corresponding number, and the bottom row indicates the corresponding VMs. What we essentially have is an assignment problem where different ACs are deployed within different VMs.

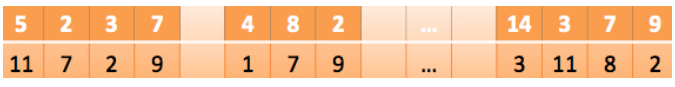

Fig. 1. Comparison of one run of GGA and IN

The optimization task is to assign the ACs to the VMs so that a certain fitness criterion is minimized. The fitness function of the composite SaaS consists of two parts: total cost and migration cost and is calculated as follows:

$$F = w_1 \times F(TC) + w_2 \times F(MC) \tag{1}$$

where w_1 and w_2 are weights for the different components, $F(TC)$ is the fitness of the total cost, and $F(MC)$ is the fitness of the migration cost. The fitness of the total cost is calculated as:

$$F(TC) = \begin{cases} 0, & \text{if } TC > C_{init}, \\ \frac{C_{init} - TC}{C_{init}}, & \text{otherwise}, \end{cases} \tag{2}$$

whereby C_{init} is a predefined value, and the total cost, TC, is calculated as follows:

$$TC = \sum_{vm \in VM} C_{vm} \tag{3}$$

where C_{vm} is the cost of running a particular virtual machine vm from the pool of all available virtual machines VM, and is calculated as:

$$C_{vm} = \begin{cases} C_{vm}, & \text{if } vm \text{ is part of the SaaS}, \\ 0, & \text{otherwise}. \end{cases} \tag{4}$$

The fitness of the migration cost $F(MC)$ is normalized by:

$$F(MC) = 1 - \frac{MC}{N(AC)} \tag{5}$$

where $N(AC)$ is the number of ACs, and the migration cost MC is calculated as follows:

$$MC = \sum_{ac \in AC} \frac{S_{ac}}{max(S_{AC}) \times 2} + \frac{M_{ac}}{max(M_{AC}) \times 2} \tag{6}$$

where S_{ac} is the size of the component, and M_{ac} is its memory requirement.

IV. Proposed Approach

Artificial Immune Systems (AIS) [17], [18], [19] are inspired by the natural immune system and the principles and processes of natural immune systems are exploited. The computational intelligent algorithms or AIS use the ideas of the natural immune system's characteristics of learning and memory to solve particular problems in computer science and engineering.

Computationally intelligent systems are inspired by the principles and processes of the vertebrate immune system. The algorithms typically exploit the immune system's characteristics of learning and memory to solve a problem.

Several different algorithms have been developed in the past. These include Danger Theory Algorithm [20], Negative Selection Algorithm [21], Clonal Selection Algorithm [22], and Immune Network Algorithm [23].

The Danger Theory algorithm works as follows. The immune system does not differentiate self from non-self, but does differentiate between what is harmful and not harmful to the body. According to the theory, an alarm signal called a 'danger signal' is activated when harmful invaders enter the body and therefore an adaptive immune response is triggered.

One of the first AIS algorithms was proposed in [21] inspired by the negative selection occurring in the thymus on T-cells during the T-cell maturation process.

The Clonal Selection algorithm is inspired by the B-cell activation process that results in the generation of plasma cells and antibodies that are released into the bloodstream in order to capture similar antigens. The main components of this algorithm are antibodies, cloning and hypermutation, and affinity measure and selection.

The Immune Network algorithm is based on the fact that antibodies trigger an immune response not only when they interact with antigens but also with other antibodies. Antibodies either respond positively (leads to cell activation and differentiation) or negatively (leads to tolerance or suppression) to a recognition signal. More details are provided in the following subsection.

The different application domains in which the AIS algorithms have been applied to include computer security [24], [25], clustering/classification [26], [22], [27], [28], [29], optimization [22], [30], and robotics [31], [32], [33], [34]. For a detailed review of application areas the reader is referred to [18].

A. Immune Network Algorithm

Jerne [23] suggested that antibodies trigger an immune response not only when they interact with antigens but also when they interact with other antibodies. This immune response is known as the idiotype network, in which antibodies may respond either positively or negatively to a recognition signal. A positive response leads to cell activation, whereas a negative response leads to suppression. This theory of natural immune systems is used in the algorithm and known as the immune network algorithm. The main idea is that the components of an AIS not only interact with antigens but also with each other to form a stable network.

Algorithm 1 Immune Network Algorithm

```
Input: populationSize p, noOfClones n, noRandom r,
       affinityThreshold a
Output: bestCell b
population ⇐ InitializePopulation(p)
while (stoppingCriterion not met) do
    EvaluatePopulation(population)
    b ⇐ GetBestSolution(population)
    progeny ⇐ ∅
    population ⇐ AverageAffinityAndRemoval(population)
    for (cell ∈ population) do
        clones ⇐ CreateClones(cell,n)
        for (clone ∈ clones) do
            clone ⇐ Mutate(clone,cell)
        end for
        EvaluatePopulation(clones)
        progeny ⇐ GetBestSolutions(clones)
    end for
    progeny ⇐ SuppressLowAffinityCells(progeny,a)
    progeny ⇐ CreateRandomCells(r)
    population ⇐ progeny
end while
return b
```

The immune network algorithm description is given in Algorithm 1. The algorithm starts by randomly initializing the population of cells. The iterative steps are as follows: First, the affinity or fitness of each cell is evaluated, and a clone set is created that is mutated. Afterwards, cells with an affinity value below a certain threshold are removed. Then, a certain number of clones with the highest affinity replace the old population. Afterwards, the so-called network interactions are performed thereby removing the lowest affinity cells. In addition, a certain number of random cells are added to the population. The above steps are repeated until the termination condition is reached and the fittest cell is returned.

B. Immune Network Algorithm Implementation: IN Algorithm

In the SaaS composition problem, the selections of SaaSs are grouped together to model a cell. The SaaSs are composed of many application components (ACs), each with their own processing-, memory-, storage-, and read/write requirements. The sum of these requirements is the SLA for the SaaS. VMs rest atop distributed physical machines that may reside in different physical locations. These VMs can be paired with any of the application components. VMs may also be the sole resource provider for an application component or the resource provider for many application components. An arbitrary amount of cells are created with the same SaaSs, ACs, and VMs, each with different pairings and placements.

A fitness evaluation is ascribed to each cell. The lower the fitness the better, thus the fitness function is to be minimized. The costs for VMs are determined by the amount of resources that are being used. The more resources that are used, the more the fitness degrades since then a VM has a larger workload. Therefore, it is better for an AC to be paired with a VM that has a lower workload. Thus, it is more likely the SLA for the AC is adhered to. The fitness is also determined by the storage and memory requirements of ACs. The reason is that ACs with larger memory and storage requirements are more costly to move to different VMs over a network. Therefore, the larger the memory and storage requirements are, the more fitness degrades in the cell.

The IN implementation closely follows the algorithm description given in Algorithm 1. The first step is the random initialization of the population, i.e., the cells are encoded as in Figure 1. Afterwards, the fitness value for each cell is calculated using Equations 1-6 as well as the average fitness of the population is determined. Next, the cloning of the cells and the mutation steps occur. Again, the population (or new mutated clones) are evaluated and the best solutions are retained. Once this is done, during the suppression step, the low affinity cells (the cells with the lowest fitness value) are deleted and newly created cells are added to the population. These steps repeat until a certain number of FEs are reached, and the fittest cell is returned as the result.

V. Experiments and Results

A. Comparison Algorithm: Grouping Genetic Algorithm - GGA

The algorithm implemented for comparison is the Grouping Genetic Algorithm (GGA) [1]. The GGA is modified from standard Genetic Algorithm (GA) for solving grouping optimization problems such as the composite SaaS optimization. The difference between GA and GGA mainly lies in the way the chromosomes are evaluated. GGA divides the chromosomes based on the relevant groupings and is evaluated accordingly.

The GGA works as follows. First, all the cells are evaluated and those cells that do not adhere to the SLAs are repaired, whereby the costs for the cells that are repaired are recalculated. Then cells are selected from the population based on the genetic algorithm's roulette wheel selection. After this, a crossover operation and a mutation operation are probabilistically applied to the selected cells. New cells are added to the population via the genetic algorithm's crossover operation. The cell with the best fitness will contain VMs with a minimized load and the ACs will be placed in such a manner that the storage and memory requirements are not overloading the VMs. A more detailed description can be found in [1].

B. Experimental Setup

Table I shows the different configurations used for the experiments. Configuration 1 has a variable setup of ACs but fixed setup of the VMs, whereas Configurations 2-6 have a fixed setup of ACs and VMs. Configuration 7 has a fixed setup of ACs, but a variable setup of VMs, and Configuration 8 has both variable setups for ACs and VMs. Essentially, all different scenarios are optimized in order to perform thorough experimentation. The characteristics of the AC and VM instances in terms of CPU (number of cores; 2.6-2.8GHz), memory (measured in GiB), and storage (measured in GB) are obtained from the Amazon website [35]. The problem setup was configured as follows: w_1 and w_2 in Equation (1) were set to 0.5; the number of components was set to 100 for both ACs and VMs, and the number of SaaSs was 20.

C. Results

Two different experiments are conducted. The first experiment investigates the affinity threshold of the IN algorithm, and the second experiment compares the IN algorithm with the GGA algorithm.

The parameter setup used for the GGA algorithms were:

- No. of chromosomes = 30
- Max. no. of iterations = 50
- Crossover rate = 0.7
- Mutation rate = 0.1

The parameter setup used for the IN algorithms were:

- No. of cells = 30
- Max. no. of iterations dependent on FEs of GGA
- Affinity threshold factor: alpha = 0.1-0.9 (Exp. 1); alpha = 0.6 (Exp. 2)
- Affinity threshold = alpha * abs(bestFit-averageFit)
- Number of clones (per iteration) = 30

The first experiment investigates the effect of the affinity threshold on the optimization of the IN algorithm. The different threshold factors that were investigated were 0.1 to 0.9 in increments of 0.1. The affinity threshold is calculated as the difference between the best and average fitness of the population multiplied by the threshold factor. The SaaS configuration used was Configuration 7 as given in Table I.

Table II shows the effect on the fitness value for the different threshold factors used as well as the number of new cells added.

TABLE II
Effect of affinity threshold factor on fitness

Threshold factor	Fitness	Cells added
0.1	0.26871699148207795±0.011390817079472893	428
0.2	0.25844564540259120±0.011604727234169044	351
0.3	0.24730482221067915±0.016548811886150625	282
0.4	0.23810788834736082±0.016465887044548210	203
0.5	0.23625232418390960±0.017846618521530000	145
0.6	0.22752460309513292±0.017833955125571930	91
0.7	0.22319969403289720±0.016287563069805173	54
0.8	0.22264649358441121±0.022019014355932197	32
0.9	0.22517136793958110±0.016659456416619235	24

The table shows that the fitness values are significantly improving until the threshold factor reaches a value of 0.6. All the higher values are similar also when looking at the associated standard deviation values. Therefore, for the following experiments the affinity threshold factor was set to 0.6.

The second experiment compares the IN algorithm with the GGA algorithm. In order to guarantee that both algorithms can be evaluated fairly, the number of Function Evaluations (FEs) for the IN algorithm was fixed to be equal to the GGA algorithm. For example, given a chromosome size of 30 and the number of iterations to be executed is 50, the number of FEs of the GGA algorithm was 1,500, and therefore, the IN algorithm was stopped once 1,500 FEs had been reached.

The experiments were run 30 times and average results are reported. The results of the optimization are shown in Table III. The results show the best fitness achieved and the standard deviation calculated after 1,500 FEs have elapsed. It can be seen that for all 8 configurations, the proposed IN algorithm achieves better results outperforming the GGA approach.

In order to show the convergence of both algorithms, the following experiment was run with Configuration 1 as shown in Table I. However, the parameters set for this experiment were:

- Population size = number of cells = 50
- Maximum number of iterations of GGA = 200
- Number of FEs of IN = 10,000
- Number of components was set to 50 for both ACs and VMs
- Number of SaaSs was 10

Figure 2 shows one optimization run for GGA and IN. The figure clearly shows that the IN algorithm converges to the

TABLE I
AC AND VM SPECIFICATIONS FOR DIFFERENT CONFIGURATIONS

Configuration	AC			VM		
	CPU	Memory	Storage	CPU	Memory	Storage
1	1-8 (incr. of 1)	1-15 (incr. of 1)	100-1000 (incr. of 100)	8	15	1000
2	1	2	125	8	15	1000
3	2	4	250	8	15	1000
4	3	6	375	8	15	1000
5	4	8	500	8	15	1000
6	5	10	625	8	15	1000
7	16	30	640	1,4,8,16,32	2,15,30,60	160,320,640
8	1-16 (incr. of 1)	1-30 (incr. of 1)	100-640 (incr. of 120)	1,4,8,16,32	2,15,30,60	160,320,640

TABLE III
RESULTS OF 8 OPTIMIZATION RUNS

Configuration	GGA	IN
1	0.29014569512910265±0.013766813558770480	**0.26003501623467570**±0.012698010015395455
2	0.29886779550208880±0.009525306680397881	**0.26954338776377020**±0.016747854056233430
3	0.26748592671216950±0.007601405885665940	**0.26322945732398020**±0.007284422954970293
4	0.21272085813931474±0.025360151355255750	**0.14164378917416415**±0.024851595058252040
5	0.20130537934903858±0.014431808584132006	**0.14681527573377423**±0.016670084298508320
6	0.34453974206241195±0.005122261819739397	**0.33375416538522560**±0.004465340093833100
7	0.28934530643446342±0.011623490453343462	**0.27542572213946932**±0.011834258781234767
8	0.25890645006913254±0.010094919455751951	**0.23111801781516520**±0.007740423912255129

optimal solution after 51 iterations, whereas it takes the GGA algorithm 99 iterations to converge.

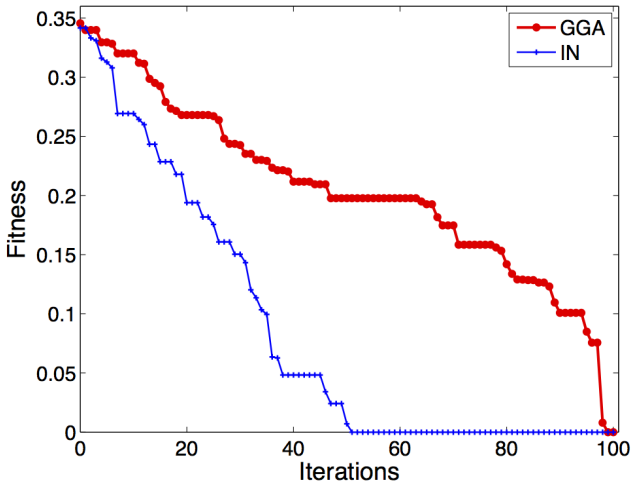

Fig. 2. Comparison of one run of GGA and IN

VI. CONCLUSION

This paper investigated the applicability of an immune network algorithm applied to the SaaS configuration optimization in Cloud data centers. The SaaS application components and data components are placed onto computing servers and storage servers, whereby the ACs are deployed within the virtual machine (VM) for execution. A necessary condition for the VM hosting the ACs is to have sufficient resources in order to guarantee users' service level agreement. Therefore, this optimization problem is essentially an assignment problem where different ACs are deployed within different VMs. The fitness function of the composite SaaS includes the total cost and migration cost.

In order to conduct a fair comparison, a comparison algorithm GGA was implemented beside the proposed IN algorithm and 8 different configurations were optimized with varying parameters. The experiments revealed that the IN algorithm converges much faster to the optimal solution than the GGA algorithm as evident from the run conducted as shown in Figure 2 as well as the results tabulated in Table III. Thus, the preferred choice of algorithm for the composite SaaS optimization is the IN algorithm.

As for future work, since the optimization takes quite a bit of running time, and for even larger SaaS optimization configurations, the parallelization of the IN algorithm becomes a necessity. A possible solution would be to use the MapReduce paradigm to speed up the optimization task.

REFERENCES

[1] Z. I. M. Yusoh, M. Tang, "Clustering composite SaaS components in Cloud computing using a Grouping Genetic Algorithm," 2012 IEEE Congress on Evolutionary Computation (CEC), June 2012.

[2] T. Grandison, E. M. Maximilien, S. Thorpe, A. Alba, "Towards a Formal Definition of a Computing Cloud," 2010 6th World Congress on Services (SERVICES), July 2010.

[3] C. A. Ardagna, E. Damiani, F. Frati, G. Montalbano, D. Rebeccani, M. Ughetti, "A Competitive Scalability Approach for Cloud Architectures," 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), July 2014.

[4] P. Mell, T. Grance, "The NIST Definition of Cloud Computing, September 2011.

[5] L. M. Vaquero, J. C. Luis Rodero-Merino, M. Lindner, "A break in the clouds: towards a cloud definition." ACM SIGCOMM Computer Communication Review 39, no. 1, pp. 50-55, 2008.

[6] I. Foster, Y. Zhao, I. Raicu, I. S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," Grid Computing Environments Workshop, 2008. GCE'08, Nov. 2008.

[7] Google App Engine, “Google App Engine: Platform as a Service”, last retrieved in March 2015, from: https://cloud.google.com/appengine/docs.

[8] M. Unuvar, Y. Doganata, M. Steinder, A. Tantawi, S. Tosi, “A Predictive Method for Identifying Optimum Cloud Availability Zones,” 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), July 2014.

[9] K. Tsakalozos, V. Verroios, Vasilis, M. Roussopoulos, A. Delis, “Time-Constrained Live VM Migration in Share-Nothing IaaS-Clouds,” 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), July 2014.

[10] D. F. Kassa, K. Nahrstedt, “SCDA: SLA-Aware Cloud Datacenter Architecture for Efficient Content Storage and Retrieval,” 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), July 2014.

[11] Y. Yao, J. Wang, B. Sheng, J. Lin, N. Mi, “HaSTE: Hadoop YARN Scheduling Based on Task-Dependency and Resource-Demand,” 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), July 2014.

[12] G. F. Anastasi, E. Carlini, M. Coppola, P. Dazzi, “QBROKAGE: A Genetic Approach for QoS Cloud Brokering,” 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), July 2014.

[13] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, X. Zhu. “Vmware distributed resource management: Design, implementation, and lessons learned.” VMware Technical Journal 1, no. 1, pp. 45-64, 2012.

[14] H. Fu, Z. Li, C. Wu, X. Chu, “Core-Selecting Auctions for Dynamically Allocating Heterogeneous VMs in Cloud Computing,” 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), July 2014.

[15] T. Groves, “Incentives in teams.” Econometrica: Journal of the Econometric Society, 617-631, 1973.

[16] X. Lu, Z. Gu, “A load-adapative cloud resource scheduling model based on ant colony algorithm,” 2011 IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS), Sept. 2011.

[17] D. Dasgupta and F. Gonzalez, “Artificial immune system (AIS) research in the last five years,” Proceedings of the Congress on Evolutionary Computation, pp. 123-130, 2003.

[18] E. Hart and J. Timmis, “Application area of AIS: The Past, The Present and the Future,” Applied Soft Computing, Elsevier Science, Amsterdam, vol. 8, 2008.

[19] J. E. Hunt and D. E. Cook, “Learning using an artificial immune system,” Journal of Network and Computer Applications, vol. 19, pp. 189-212, 1996.

[20] P. Matzinger, “Tolerance, danger and the extended family,” Annual Review of Immunology, vol. 12, pp. 991-1045, 1994.

[21] S. Forrest, A. Perelson, L. Allen, and R. Cherukuri, “Self-nonself discrimination in a computer,” Proceedings - IEEE Computer Society Symposium on Research in Security and Privacy, pp. 202-212, 1994.

[22] L. N. d. Castro and J. Zuben, “The Clonal Selection Algorithm with Engineering Applications,” Workshop Proceedings of GECCO, Workshop on Artificial Immune Systems and Their Applications, Las Vegas, pp. 36-37, 2000.

[23] N. K. Jerne, “The Generative Grammar of the Immune System,” Nobel Lecture, 8 December 1984, 1984.

[24] J. Kim, J. Greensmith, J. Twycross, and U. Aickelin, “Malicious Code Execution Detection and Response Immune System inspired by the Danger Theory,” Proceedings of Adaptive and Resilient Computing Security Workshop (ARCS-05), 2005.

[25] P. K. Harmer, P. D. Williams, G. H. Gunsch, and G. B. Lamont, “An artificial immune system architecture for computer security applications,” IEEE Transactions on Evolutionary Computation, vol. 6, pp. 252-280, 2002.

[26] L. N. d. Castro and J. Timmis, “Artificial immune system: a new computational intelligence approach,” Springer, 2002.

[27] A. Watkins, J. Timmis, and L. Boggess, “Artificial Immune Recognition System (AIRS): An Immune-Inspired Supervised Learning Algorithm,” Journal of Genetic Programming and Evolvable Machines, vol. 5, pp. 291-317, 2004.

[28] J. Timmis, M. Neal, and J. Hunt, “An artificial immune system for data analysis”, Biosystems, vol. 55, pp. 143-150, 2000.

[29] M. Ayara, J. Timmis, R. d. Lemos, and S. Forrest, “Immunising Automated Teller Machines,” Jacob et. al (Eds.): ICARIS 2005, LNCS 3627, pp. 404-417, 2005.

[30] A. Khaled, H. M. Abdul-Kader, and N. A. Ismail, “Artificial Immune Clonal Selection Algorithm: A Comparative Study of CLONALG, opt-IA and BCA with Numerical Optimization Problems,” International Journal of Computer Science and Network Security, vol. 10, pp. 24-30, 2010.

[31] A. M. Whitbrook, U. Aickelin, and J. M. Garibaldi, “Idiotypic Immune Networks in Mobile Robot Control,” IEEE Transactions on Systems, Man, and Cybernetics - Part B: CYBERNETICS, vol. 37, pp. 1581-1598, 2007.

[32] A. M. Whitbrook, U. Aickelin, and J. M. Garibaldi, “The transfer of evolved artificial immune system behaviours between small and large scale robotic platforms,” Proceedings of the 9th international conference on Artificial evolution (EA’09), 2009.

[33] A. M. Whitbrook, U. Aickelin, and J. M. Garibaldi, “An Idiotypic Immune Network as a Short-Term Learning Architecture for Mobile Robots,” 7th International Conference, ICARIS, LNCS: 5132, pp. 266-278, 2008.

[34] H. Lau, I. Bate, and J. Timmis, “An Immuno-engineering Approach for Anomaly Detection in Swarm Robotics,” 8th International Conference, ICARIS 2009, LNCS: 5666, pp. 136-150, 2009.

[35] Amazon Compute and Storage Instances, “Amazon EC2 Instance Details”, last retrieved in March 2015, from: http://aws.amazon.com/ec2/instance-types/.