A Multi-Objective ACS Algorithm to Optimize Cost, Performance, and Reliability in the Cloud

Ashraf, Adnan; Byholm, Benjamin; Porres Paltor, Ivan

Published in:
8th IEEE/ACM International Conference on Utility and Cloud Computing

DOI:
10.1109/UCC.2015.54

Published: 01/01/2015

A Multi-Objective ACS Algorithm to Optimize Cost, Performance, and Reliability in the Cloud

Adnan Ashraf, Benjamin Byholm, Ivan Porres
Faculty of Natural Sciences and Technology, Åbo Akademi University, Finland
aashraf@abo.fi, bbyholm@abo.fi, iporres@abo.fi

***Abstract*—In this paper, we present a novel Multi-Objective Ant Colony System algorithm to optimize Cost, Performance, and Reliability (MOACS-CoPeR) in the cloud. The proposed algorithm provides a metaheuristic-based approach for the multi-objective cloud-based software component deployment problem. MOACS-CoPeR explores the search-space of architecture design alternatives with respect to several architectural degrees of freedom and produces a set of Pareto-optimal deployment configurations. We also present a Java-based implementation of our proposed algorithm and compare its results with the Non-dominated Sorting Genetic Algorithm II (NSGA-II). We evaluate the two algorithms against a cloud-based storage service, which is loosely based on a real system.**

I. Introduction

Cloud computing is a relatively new computing paradigm. It leverages several existing concepts and technologies, such as data centers and hardware virtualization, and gives them a new perspective [1]. With its pay-per-use business model for the customers, cloud computing shifts the capital investment risk for under or over provisioning to the cloud providers [2]. Public cloud providers such as Amazon, Google, and Microsoft operate large-scale cloud data centers around the world and strive to provide a multitude of cloud resources at competitive prices by exploiting economies of scale. For instance, Infrastructure as a Service (IaaS) clouds provide different types of virtual machines (VMs) that can be used to deploy software applications and services [3]. The different types of VMs often vary in terms of cost, performance, and reliability. Therefore, renting the right amounts and types of VMs is essential for ensuring the desired levels of Quality of Service (QoS).

A large number of contemporary software systems are built from software components that can be deployed on one or more VMs. A typical software system comprises a number of software components, where each component often requires certain levels of performance and reliability. Therefore, when deploying a component-based software system in an IaaS cloud, performance and reliability requirements of individual software components should be taken into account. In practice, it is often possible to provide high QoS levels by over-provisioning of resources [4]. However, over-provisioning of cloud resources results in an increased operational cost. Therefore, performance and reliability of software systems can not be optimized in isolation from the cost of cloud resources. Thus, a software deployment configuration should be simultaneously optimized in terms of cost, performance, and reliability.

The cloud-based software component deployment problem is a special case of the generic software architecture optimization problem [5], in which the search-space of architecture design alternatives is explored with respect to one or more objectives. The component-based software development paradigm provides various generic architectural Degrees of Freedom (DoFs) that can be exploited to create different functionally-equivalent alternatives of an architectural design [4], [6]. An architectural DoF refers to a way an architecture model can be modified and improved in terms of certain quality properties without affecting the functionality of the system [5], [6]. For instance, component allocation DoF allows to change the allocation of software components to VMs in order to optimize a software architecture model with respect to certain objectives. Thus, architectural DoFs define the search-space for optimization in which all solutions provide the same functionality, but with different levels of quality properties.

In this paper, we formulate the cloud-based software component deployment problem as a multi-objective optimization problem with three antagonistic objectives: cost, performance, and reliability. Manually exploring the search-space of deployment configurations with respect to three antagonistic objectives is time-consuming, error-prone, and may lead to suboptimal solutions [4]. Moreover, since the multi-objective cloud-based software component deployment problem is an NP-hard combinatorial optimization problem [5], it should be approached in a systematic way by using efficient optimization techniques. Furthermore, since the problem involves multiple objectives, the traditional single objective optimization techniques are not appropriate for it [7]. Therefore, it should be approached with a multi-objective optimization technique that produces a set of Pareto-optimal configurations [8].

The existing works on software architecture optimization can be classified into several different categories such as rule-based approaches, metaheuristic-based approaches, and hybrid approaches [5]. These approaches tend to use a particular modeling language and often require an initial architecture configuration. However, in many cases, an initial architecture configuration may not exist or the system under study may have been modeled in a different modeling language. Moreover, a disadvantage of using an initial architecture configuration is that it may restrict the search to a subset of the search-space which is reachable from the initial architecture

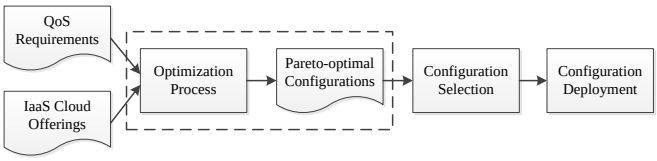

Fig. 1. Overview of the proposed approach

configuration. Furthermore, some of the existing approaches do not take into account the performance and reliability requirements of individual software components during the search process. Instead, the search process is followed by an additional evaluation step which checks the feasibility and desirability of the solutions and eliminates infeasible and undesired solutions.

In this paper, we present a metaheuristic-based approach for the multi-objective cloud-based software component deployment problem. The proposed approach uses ant colony optimization (ACO) [9], [10] metaheuristic and is based on a multi-objective ant colony system (ACS) algorithm by Barán and Schaerer [11]. The proposed **M**ulti-**O**bjective **A**nt **C**olony **S**ystem algorithm to optimize **Co**st, **Pe**rformance, and **R**eliability (MOACS-CoPeR) finds a set of Pareto-optimal solutions for the multi-objective cloud-based software component deployment problem. We consider three generic architectural DoFs: component allocation, VM selection, and number of VMs [4], [5], [6], [12]. We also present a Java-based implementation of our proposed algorithm and compare its results with a highly efficient genetic algorithm called Non-dominated Sorting Genetic Algorithm II (NSGA-II) [13]. We evaluate the two algorithms against a cloud-based storage service, which is loosely based on a real system. The results show that MOACS-CoPeR outperforms NSGA-II in terms of number and quality of Pareto-optimal configurations found.

Figure 1 presents an overview of the proposed approach. Based on the available IaaS cloud offerings and the QoS requirements of individual software components, the proposed approach runs an optimization process which yields a set of Pareto-optimal deployment configurations. Afterwards, the set of Pareto-optimal configurations is analyzed further, either manually by a software architect or automatically based on an aggregate objective function [14] and the remaining trade-offs, in order to select a final deployment configuration. Finally, the selected configuration is deployed, resulting in a new allocation of software components to VMs. As depicted in Figure 1, our main focus in this paper is on the optimization process to produce a set of Pareto-optimal deployment configurations. Two salient features of our proposed approach are that it is not dependent on a particular modeling language and it does not require an initial architecture configuration. Moreover, it eliminates undesired and infeasible configurations at an early stage by using performance and reliability requirements of individual software components as heuristic information to guide the search process.

II. Multi-Objective ACS Algorithm

In this section, we present our proposed **M**ulti-**O**bjective **A**nt **C**olony **S**ystem algorithm to optimize **Co**st, **Pe**rformance, and **R**eliability (MOACS-CoPeR) of software deployment configurations in the cloud. In the context of cloud-based software component deployment, each VM $v \in V$ hosts one or more software components $c \in C$. Both VMs and software components are characterized by their performance and reliability. The main objective is to allocate software components to VMs in such a way that the cost of deployment infrastructure is minimized while satisfying the performance and reliability requirements of individual software components. For simplicity, we assume that the most important deployment infrastructure cost is the cost of provisioning VMs from an IaaS cloud. The problem is similar to the multidimensional vector bin packing problem (MDVPP) [15], where the VMs are the bins and the software components are the objects to be packed into the bins. Moreover, the performance and reliability requirements can be modeled as the different dimensions in the MDVPP. The problem of finding a packing that uses a minimum number of bins is known to be NP-hard [15]. Therefore, it is difficult to find an optimal solution in the multidimensional cloud-based software component deployment problem with a large number of VMs and software components.

We formulate the multidimensional cloud-based software component deployment problem as a multi-objective combinatorial optimization problem with three objectives and three generic architectural DoFs. For the sake of clarity, important concepts and notations used in the following sections are tabulated in Table I. The three generic architectural DoFs in our approach are component allocation, VM selection, and number of VMs. Component allocation DoF allows to change the allocation of components to VMs in order to optimize a deployment configuration for cost, performance, and reliability. Similarly, VM selection DoF allows to select VMs with different levels of cost, performance, and reliability. Moreover, with number of VMs DoF, it is possible to add or remove VMs to optimize the cloud-based deployment configuration. Adding more VMs may provide better performance and higher reliability, but it may also result in an increased operational cost. The optimization process explores different architecture alternatives with respect to these three DoFs without affecting system functionality. It uses the generic architectural DoFs to identify a set of system-specific DoFs. An example of a system-specific DoF with respect to component allocation is allocating a particular software component $c \in C$ to a particular VM $v \in V$. Therefore, c may be allocated to a different VM in the set of VMs V in order to optimize cost, performance, and reliability. Thus, the search-space of architecture alternatives can be viewed as the Cartesian product of the design options of all system-specific DoFs [4]. In the next step, our proposed multi-objective ACS algorithm searches the search-space for non-dominated deployment configurations with respect to cost, performance, and reliability, resulting in a set of Pareto-optimal configurations.

TABLE I
SUMMARY OF CONCEPTS AND THEIR NOTATIONS

C	set of software components
P	set of Pareto-optimal configurations
T	set of tuples as defined in (1)
T_k	set of tuples not yet traversed by ant k
V	set of VMs
V_{Ψ^P}	set of VMs in a non-dominated configuration Ψ^P
R_c	required level of availability of software component c
R_v	availability of VM v
I_c	performance requirement of software component c
I_v	processing rate of VM v in MIPS
I_{v_A}	amount of allocated processing rate of VM v
q	a uniformly distributed random variable
S	a random variable selected according to (3)
η	heuristic value
τ	amount of pheromone
τ_0	initial pheromone level
Ψ	a software component deployment configuration
Ψ^P	a Pareto-optimal configuration in P
Ψ_k	ant-specific configuration of ant k
$\Delta^P_{\tau_s}$	additional pheromone amount given to the tuples in a Ψ^P
q_0	parameter to determine relative importance of exploitation
α	pheromone decay parameter in the global updating rule
β	parameter to determine relative importance of η
λ	parameter to determine relative importance of η_1 and η_2
ρ	pheromone decay parameter in the local updating rule
nA	number of ants that concurrently build their solutions
nI	number of iterations of the main loop

Since each software components $c \in C$ is deployed on a VM $v \in V$, the proposed MOACS-CoPeR algorithm makes a set of tuples T, where each tuple $t \in T$ consists of two elements: software component c and VM v

$$t := (c, v) \tag{1}$$

The output of the MOACS-CoPeR algorithm is a set of Pareto-optimal software component deployment configurations P, in which each configuration Ψ^P simultaneously optimizes the three objectives. In addition, each configuration $\Psi^P \in P$ should satisfy the performance and reliability requirements of individual software components. Therefore, MOACS-CoPeR seeks to find software component deployment configurations that maximize performance and reliability and minimize cost subject to the performance and reliability requirements of individual software components.

In ACS, each ant builds a complete solution. In our proposed approach, a software component deployment configuration Ψ comprises a set of tuples, which are stochastically chosen from the set of tuples T. Since there is no notion of path in the cloud-based software component deployment problem, ants deposit pheromone on the tuples defined in (1). Each of the nA ants uses a stochastic state transition rule to choose the next tuple to traverse. The state transition rule in ACS called the pseudo-random-proportional-rule [10] determines the next decision in solution construction. According to this rule, an ant $k \in nA$ chooses a tuple s to traverse next by applying

$$s := \begin{cases} \arg\max_{u \in T_k} \{[\tau_u] \cdot [\eta_{1_u}]^{\lambda\beta} \cdot [\eta_{2_u}]^{(1-\lambda)\beta}\}, & \text{if } q \leq q_0 \\ S, & \text{otherwise} \end{cases} \tag{2}$$

where τ denotes the amount of pheromone and η_1 and η_2 represent the heuristic values associated with a particular tuple. η_1 denotes the optimization objective concerning the performance requirement of a software component c on a VM v. Similarly, η_2 represents the objective concerning the reliability requirement of c on v. The parameter $\lambda = k/nA$ is used to determine the relative importance of η_1 and η_2. Therefore, each ant weighs the relative importance of these two optimization objectives differently when choosing a tuple s to traverse [16]. Similarly, β is a parameter to determine the relative importance of the heuristic values η_1 and η_2 with respect to the pheromone value τ. The expression *arg max* returns the tuple for which $[\tau_u] \cdot [\eta_{1_u}]^{\lambda\beta} \cdot [\eta_{2_u}]^{(1-\lambda)\beta}$ attains its maximum value. $T_k \subset T$ is the set of tuples that remain to be traversed by ant k. $q \in [0, 1]$ is a uniformly distributed random variable and $q_0 \in [0, 1]$ is a parameter. S is a random variable selected according to the probability distribution given in (3), where the probability $prob_{k_s}$ of an ant k to choose tuple s to traverse next is defined as

$$prob_{k_s} := \begin{cases} \frac{[\tau_s] \cdot [\eta_{1_s}]^{\lambda\beta} \cdot [\eta_{2_s}]^{(1-\lambda)\beta}}{\sum_{u \in T_k} [\tau_u] \cdot [\eta_{1_u}]^{\lambda\beta} \cdot [\eta_{2_u}]^{(1-\lambda)\beta}}, & \text{if } s \in T_k \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

The heuristic value concerning performance η_{1_s} for a tuple s is defined as

$$\eta_{1_s} := \begin{cases} \frac{I_{v_A} + I_c}{I_v}, & \text{if } I_{v_A} + I_c \leq I_v \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where I_v is the processing rate of the VM v in millions of instructions per second (MIPS), I_{v_A} is the amount of processing rate of v in MIPS that has already been allocated to some software components, and likewise I_c is the performance requirement of the software component c in tuple s in terms of MIPS. In an open workload system [4], the required amount of MIPS for a software component can be derived from the performance requirement of the software component in millions of instructions (MI), the expected user request arrival rate in the system, and the probability of the component to be invoked in a user request. The heuristic value concerning performance η_1 is based on the ratio of $(I_{v_A} + I_c)$ to I_v. Therefore, VMs with the minimum unallocated processing rate receive the highest amount of heuristic value. Moreover, the constraint $I_{v_A} + I_c \leq I_v$ prevents deployments that would violate the required performance level of the software component in tuple s.

For the reliability requirements, we use the availability metric. Therefore, the heuristic value concerning reliability η_{2_s} for a tuple s is defined as

$$\eta_{2_s} := \begin{cases} 1 - (R_v - R_c), & \text{if } R_c \leq R_v \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

where R_v is the availability of the VM v measured as the percentage of time when v is operational and R_c is the required level of availability of the software component c in tuple s. The heuristic value concerning reliability η_2 is

based on the difference between 1 and $(R_v - R_c)$. It favors component deployments where the VM availability level R_v closely matches the availability requirement of the software component R_c. Moreover, the constraint $R_c \leq R_v$ prevents deployments that would violate the availability requirement of component c in tuple s.

The stochastic state transition rule in (2) and (3) prefers tuples with a higher pheromone concentration and which result in fewer or smaller VMs while satisfying the performance and reliability requirements of the software components. The first case in (2) where $q \leq q_0$ is called exploitation [10], which chooses the best tuple that attains the maximum value of $[\tau_u] \cdot [\eta_{1_u}]^{\lambda\beta} \cdot [\eta_{2_u}]^{(1-\lambda)\beta}$. The second case, called biased exploration, selects a tuple according to (3). The exploitation helps the ants to quickly converge to a high quality solution, while at the same time, the biased exploration helps them to avoid stagnation by allowing a wider exploration of the search-space [17], [18]. In addition to the stochastic state transition rule, ACS also uses a global and a local pheromone trail evaporation rule. The global pheromone trail evaporation rule is applied towards the end of an iteration after all ants complete their solutions. In MOACS-CoPeR, the pheromone trail is updated with each non-dominated configuration Ψ^P in the current Pareto set P [11]. The global pheromone trail evaporation rule is defined as

$$\tau_s := (1 - \alpha) \cdot \tau_s + \alpha \cdot \Delta^P_{\tau_s} \tag{6}$$

where $\Delta^P_{\tau_s}$ is the additional pheromone amount that is given to only those tuples that belong to a non-dominated configuration Ψ^P in order to reward them. Therefore, if a tuple belongs to multiple non-dominated configurations, it is more likely to receive a higher amount of pheromone. The additional pheromone amount $\Delta^P_{\tau_s}$ is defined as

$$\Delta^P_{\tau_s} := \begin{cases} (|V_{\Psi^P}|)^{-1}, & \text{if } s \in \Psi^P \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

$\alpha \in (0, 1]$ is the pheromone decay parameter, Ψ^P is a non-dominated configuration in the Pareto set P, and $|V_{\Psi^P}|$ is the number of VMs in the non-dominated configuration Ψ^P. The objective here is to favor the configurations that use fewer VMs.

The local pheromone trail update rule is applied on a tuple when an ant traverses the tuple while making its solution. It is defined as

$$\tau_s := (1 - \rho) \cdot \tau_s + \rho \cdot \tau_0 \tag{8}$$

where $\rho \in (0, 1]$ is similar to α and τ_0 is the initial pheromone level, which is computed as the multiplicative inverse of the product of the number of software components $|C|$ and the number of VMs $|V|$

$$\tau_0 := (|C| \cdot |V|)^{-1} \tag{9}$$

The pseudocode of the proposed MOACS-CoPeR algorithm is given as Algorithm 1. It outputs the set of Pareto-optimal deployment configurations P (line 30), which is initially empty

Algorithm 1 MOACS-CoPeR

```
1:  P := ∅ {Set of Pareto-optimal configurations}
2:  ∀t ∈ T|τ_t := τ_0 {Initial pheromone level}
3:  for i ∈ [1, nI] do
4:    for k ∈ [1, nA] do
5:      Ψ_k := ∅ {Ant-specific configuration of the k-th ant}
6:      while all software components in C are allocated do
7:        compute η_{1_s} and η_{2_s} ∀s ∈ T using (4) and (5)
8:        generate q ∈ [0, 1] with a uniform distribution
9:        if q > q_0 then
10:         compute probability prob_{k_s} ∀s ∈ T using (3)
11:       end if
12:       choose a tuple t ∈ T to traverse using (2)
13:       apply local update rule in (8) on t
14:       if ant k has not allocated component c in t then
15:         if deployment of c does not overload v in t then
16:           if v meets reliability requirement of c then
17:             update allocated processing rate I_{v_A} of v
18:             Ψ_k := Ψ_k ∪ {t}
19:           end if
20:         end if
21:       end if
22:     end while
23:     if P is empty or Ψ_k is non-dominated then
24:       P := P ∪ {Ψ_k}
25:       remove dominated configurations from P
26:     end if
27:   end for
28:   apply global update rule in (6) on all s ∈ T
29: end for
30: return P
```

(line 1). The algorithm makes a set of tuples T by using (1) and sets the pheromone value of each tuple to the initial pheromone level τ_0 by using (9) (line 2). It iterates over nI iterations, where each iteration uses a new generation of ants (line 3). The total number of iterations nI may depend on a stopping criterion. For instance, when a certain amount of clock time has elapsed or when no further improvements are achieved in multiple consecutive iterations [19]. In each iteration of the main loop, nA ants concurrently build their solutions (lines 4–27). Each ant builds a complete solution by allocating the software components in C to the VMs in V. Therefore, an ant continues to build its solution until it allocates all components in C (lines 6–22). It computes heuristic value concerning performance η_{1_s} and heuristic value concerning reliability η_{2_s} $\forall s \in T$ by using (4) and (5) (line 7). Then, it generates a uniformly distributed random value for $q \in [0, 1]$ (line 8) and if $q > q_0$ (line 9), it computes the probabilities for choosing the next tuple to traverse $\forall s \in T$ by using (3) (line 10). Afterwards, based on the computed probabilities and the stochastic state transition rule in (2), each ant k chooses a tuple t to traverse next (line 12). Then, the local pheromone trail update rule in (8) and (9) is applied on the selected tuple t (line 13). If ant k has not already allocated component c

in tuple t (line 14), the deployment of component c does not overload VM v in tuple t (line 15), and VM v satisfies the reliability requirement of component c (line 16), the amount of allocated processing rate I_{v_A} of VM v in t is updated to reflect the impact of the component deployment (line 17) and the tuple t is added to the ant-specific configuration Ψ_k (line 18). Afterwards, when all ants complete their solutions, each ant-specific configuration Ψ_k is compared to the current Pareto set P to see if it is non-dominated (line 23). Then, each new non-dominated configuration is added to the current Pareto set P (line 24) and the dominated configurations in P are removed (line 25). Finally, the global pheromone trail update rule in (6) and (7) is applied on all tuples (line 28).

III. Implementation and Experimental Evaluation

We have implemented our proposed algorithm as a Java program called MOACS-CoPeR solver. It uses as inputs a set of software components C representing the system under study and a set of VMs V on which the components are deployed. Previous work, such as [4], used the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [13]. We have developed a Java program using the same algorithm, here referred to as the NSGA-II solver. The NSGA-II solver implementation is based on jMetal [20], which is a Java-based framework for multi-objective optimization. The two solvers are evaluated against a cloud-based storage service (CBSS), which is loosely based on the F-Secure Input Output (FSIO) platform of the F-Secure Corporation[1].

A. Experimental Design and Setup

The FSIO platform provides a foundation for content-centric applications that store, share, and synchronize different types of digital content in the cloud. Table II presents a list of software components from CBSS. It comprises 13 software components, numbered from c1 to c13, along with their performance and availability requirements. The performance requirement of each software component is given in millions of instructions (MI) along with the probability of the component to be invoked in a user request. We assumed an open workload system with a user request arrival rate of 10 requests per second. Therefore, the required amount of millions of instructions per second (MIPS) for each software component was computed as the product of the component performance requirement in MI, the component usage probability, and the request arrival rate 10. Table III presents a list of VMs, numbered from 1 to 10, along with their performance, availability, and hourly cost. Thus, the search-space comprised 10 VMs and 13 components, giving 10^{13} possible configurations.

Given the set of software components C along with their performance and availability requirements and the set of VMs V with their cost, performance, and availability levels, the MOACS-CoPeR solver explored different architecture alternatives with respect to the three generic architectural DoFs described in Section II. Each deployment configuration was analyzed in terms of cost, performance, and availability. The cost of a configuration was computed by aggregating the cost of individual VMs in the configuration. The performance of a configuration was computed in a similar fashion by aggregating the performance of individual VMs in the configuration. Moreover, for availability of a configuration, we computed the product of the availability levels of individual VMs in the configuration. Finally, the MOACS-CoPeR solver produced a set of Pareto-optimal deployment configurations P. The ACS parameters used in the MOACS-CoPeR solver are given as Table IV. These parameter values were obtained in a series of preliminary experiments. The NSGA-II solver used the following parameters: single point crossover (probability 0.9), bit flip mutation adjusted for integer representation (probability $1.0/$number of variables), binary tournament selection, population size 100, and number of generations $\in [10, 1000]$. The chromosome structure in the NSGA-II solver was the same as in [4]. The two solvers were run on an Intel Core i7-4790 processor with 16 gigabytes of memory.

[1]www.f-secure.com

TABLE II
System under study: cloud-based storage service

Component	Performance	Probability	Availability
c1: API Frontend	490 MI	0.1020	0.99
c2: Data Frontend	367 MI	0.0816	0.98
c3: Data Backend	367 MI	0.0816	0.98
c4: Database Frontend	840 MI	0.0952	0.99
c5: Database Backend	630 MI	0.0952	0.999
c6: Cache	176 MI	0.3401	0.99
c7: Gaffer	420 MI	0.0476	0.98
c8: Antivirus Scanner	2940 MI	0.0204	0.98
c9: Thumbnailer	588 MI	0.0340	0.98
c10: Metadater	490 MI	0.0204	0.98
c11: File Typer	367 MI	0.0272	0.98
c12: Transcoder	7350 MI	0.0136	0.98
c13: Garbage Collector	245 MI	0.0408	0.98

TABLE III
Set of VMs

VM	Performance	Availability	Cost ($)
1	1000 MIPS	0.98	0.08
2	1000 MIPS	0.99	0.10
3	1200 MIPS	0.98	0.10
4	1200 MIPS	0.999	0.16
5	1500 MIPS	0.99	0.15
6	1500 MIPS	0.999	0.20
7	2000 MIPS	0.99	0.20
8	2000 MIPS	0.999	0.25
9	2500 MIPS	0.99	0.25
10	2500 MIPS	0.98	0.20

TABLE IV
ACS parameters in the proposed approach

α	β	ρ	q_0	nA	nI
0.1	2.0	0.1	0.9	10	$[100, 10000]$

B. Results and Analysis

For a comprehensive comparison of the MOACS-CoPeR and NSGA-II solvers, we report results with 1000, 10000, and 100000 objective function evaluations. The number of

objective function evaluations in MOACS-CoPeR is computed as the product of the number of iterations nI and the number of ants nA. Similarly, in NSGA-II, it is computed as the product of the population size and the number of generations. The comparison of the results is based on the following metrics [11].

- Overall true non-dominated vector generation (OTNVG) count: counts the number of configurations in the calculated Pareto set P that are also in the *true* Pareto set P_{true}. Since P_{true} is not known in theory, we computed an approximation of P_{true} by calculating a Pareto set from all individual calculated Pareto sets P of all runs of the two solvers. The approximated P_{true} comprises 922 non-dominated deployment configurations and is presented in Figure 2.
- OTNVG percentage: percentage of the number of configurations in the calculated Pareto set P that are also in P_{true}. It is computed as $\frac{\text{OTNVG count}}{|P_{true}|} \cdot 100$.
- Time: execution time of a solver in seconds.
- Combined Pareto set size: size of the combined Pareto set of a solver. The combined Pareto set of a solver is calculated from all individual calculated Pareto sets P of all runs of the solver.
- Aggregated OTNVG count: sum of all individual OTNVG counts of a solver from all runs.
- Aggregated OTNVG percentage: sum of all individual OTNVG percentages of a solver from all runs.
- Aggregated OTNVG count to combined Pareto set size ratio: computed as $\frac{\text{Aggregated OTNVG count}}{\text{Combined Pareto set size}}$.
- Error ratio: proportion of configurations in the combined Pareto set of a solver that is not found in P_{true}. It is computed as $\frac{\text{Combined Pareto set size} - \text{Aggregated OTNVG count}}{\text{Combined Pareto set size}}$.

Table V presents three example configurations along with their hourly cost, performance, and availability. The configurations in Table V are represented as a vector of 13 values, in which each value is a VM number and the position of a VM number in the vector corresponds with the component number. For instance, the first configuration in Table V used a total of four VMs and deployed five components c1, c2, c3, c5, c7 on VM 8, three components c4, c8, c13 on VM 5, four components c6, c9, c10, c11 on VM 2, and one component c12 on VM 1.

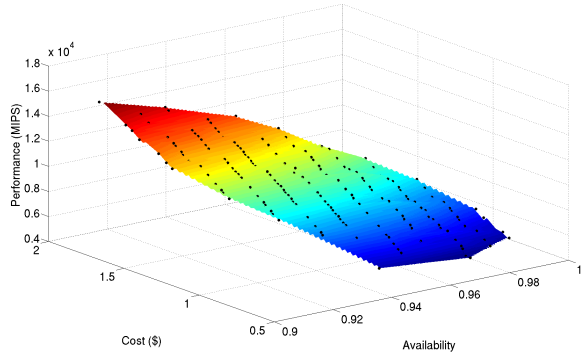

Fig. 2. Approximation of the true Pareto set P_{true}

TABLE V
PARTIAL PARETO SET (MOACS-COPER)

Cost ($)	Performance	Availability	Configuration
0.58	5500 MIPS	0.9595	8 8 8 5 8 2 8 5 2 2 2 1 5
1.69	16400 MIPS	0.9014	5 8 9 4 8 6 7 3 1 10 5 2 9
0.7	6000 MIPS	0.988	9 8 9 9 6 6 9 8 9 9 8 8 9

TABLE VI
SUMMARY OF RESULTS

	MOACS-CoPeR	NSGA-II
Combined Pareto set size	698	674
Aggregated OTNVG count	612	310
Aggregated OTNVG percentage	66.38%	33.62%
Aggregated OTNVG count to combined Pareto set size ratio	0.88	0.46
Error ratio	0.12	0.54

Table VII presents OTNVG count, OTNVG percentage, and execution time results of the MOACS-CoPeR and NSGA-II solvers. The results comprise three runs of each solver with 1000, 10000, and 100000 objective function evaluations. Moreover, we also report average results with respect to 1000, 10000, and 100000 evaluations. The results show that in eight out of nine runs, the MOACS-CoPeR solver outperformed the NSGA-II solver in terms of OTNVG count and OTNVG percentage. Moreover, in run 1 of 1000 evaluations, the two solvers produced Pareto sets with the same OTNVG count. Therefore, the results show that in almost all runs, the MOACS-CoPeR solver found more configurations in the approximated P_{true}. The execution time results in Table VII show that the NSGA-II solver executes faster than the MOACS-CoPeR solver. This is because the MOACS-CoPeR solver is currently not optimized in terms of execution time. We plan to consider this enhancement to the MOACS-CoPeR solver in our future work.

Table VI provides a summary of the results. It comprises five metrics: combined Pareto set size, aggregated OTNVG count, aggregated OTNVG percentage, aggregated OTNVG count to combined Pareto set size ratio, and error ratio. The results show that the MOACS-CoPeR solver outperformed the NSGA-II solver with respect to all five metrics.

IV. CONCLUSION

In this paper, we presented a novel **M**ulti-**O**bjective **A**nt **C**olony **S**ystem algorithm to optimize **Co**st, **Pe**rformance, and **R**eliability (MOACS-CoPeR) in the cloud. The proposed algorithm provides a metaheuristic-based approach for the multi-objective cloud-based software component deployment problem. It is based on a multi-objective ant colony system (ACS) algorithm that simultaneously optimizes multiple antagonistic objectives. MOACS-CoPeR explores the search-space of architecture design alternatives with respect to several generic architectural Degrees of Freedom (DoFs) and produces a set of Pareto-optimal deployment configurations. The currently supported architectural DoFs in our proposed approach

TABLE VII
OTNVG COUNT, OTNVG PERCENTAGE, AND EXECUTION TIME OF MOACS-COPER AND NSGA-II SOLVERS

		OTNVG Count		OTNVG Percentage		Time (seconds)	
Evaluations		MOACS-CoPeR	NSGA-II	MOACS-CoPeR	NSGA-II	MOACS-CoPeR	NSGA-II
1000	Run 1	29	29	3.15%	3.15%	1.201	0.165
	Run 2	35	23	3.80%	2.49%	1.217	0.034
	Run 3	49	30	5.31%	3.25%	1.201	0.029
	Average	**37.67**	**27.33**	**4.09%**	**2.96%**	**1.206**	**0.076**
10000	Run 1	93	35	10.09%	3.80%	15.078	0.170
	Run 2	100	42	10.85%	4.56%	10.296	0.104
	Run 3	96	29	10.41%	3.15%	10.506	0.089
	Average	**96.33**	**35.33**	**10.45%**	**3.84%**	**11.96**	**0.121**
100000	Run 1	53	49	5.75%	5.31%	122.093	0.777
	Run 2	100	36	10.85%	3.90%	109.512	0.768
	Run 3	57	37	6.18%	4.01%	112.857	0.674
	Average	**70**	**40.67**	**7.59%**	**4.41%**	**114.821**	**0.740**

are component allocation, virtual machine (VM) selection, and number of VMs. In contrast to the existing software architecture optimization approaches, our proposed approach is not dependent on a particular modeling language and it does not require an initial architecture configuration. Moreover, it takes into account the performance and reliability requirements of individual software components during solution construction and uses them as heuristic information to guide the search process, resulting in the elimination of undesired and infeasible configurations at an early stage.

We also presented a Java-based implementation of our proposed approach and compared its results with the Non-dominated Sorting Genetic Algorithm II (NSGA-II). The experimental evaluation involved 13 software components and 10 VMs. The system under study was a cloud-based storage service, which is loosely based on a real system. The results showed that MOACS-CoPeR outperformed NSGA-II in terms of number and quality of Pareto-optimal configurations found.

ACKNOWLEDGEMENTS

This work was supported by the Need for Speed (N4S) Program (http://www.n4s.fi). Benjamin Byholm also received financial support from the Foundation of Nokia Corporation.

REFERENCES

[1] F. Farahnakian, A. Ashraf, T. Pahikkala, P. Liljeberg, J. Plosila, I. Porres, and H. Tenhunen, "Using ant colony system to consolidate VMs for green cloud computing," *Services Computing, IEEE Transactions on*, vol. 8, no. 2, pp. 187–198, March 2015.

[2] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, Dec. 2008.

[3] A. Ashraf, "Cost-efficient virtual machine management: Provisioning, admission control, and consolidation," Ph.D. dissertation, Turku Centre for Computer Science (TUCS) Dissertations Number 183, October 2014.

[4] A. Martens, H. Koziolek, S. Becker, and R. Reussner, "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," in *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*. ACM, 2010, pp. 105–116.

[5] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *Software Engineering, IEEE Transactions on*, vol. 39, no. 5, pp. 658–683, May 2013.

[6] R. Etemaadi and M. R. Chaudron, "New degrees of freedom in metaheuristic optimization of component-based systems architecture: Architecture topology and load balancing," *Science of Computer Programming*, vol. 97, Part 3, no. 0, pp. 366 – 380, 2015.

[7] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "ArcheOpterix: An extendable tool for architecture optimization of AADL models," in *Model-Based Methodologies for Pervasive and Embedded Software, 2009. ICSE Workshop on*, May 2009, pp. 61–71.

[8] M. Ehrgott, *Multicriteria Optimization*, ser. Lecture notes in economics and mathematical systems. Springer, 2005.

[9] M. Dorigo, G. Di Caro, and L. M. Gambardella, "Ant algorithms for discrete optimization," *Artif. Life*, vol. 5, no. 2, pp. 137–172, Apr. 1999.

[10] M. Dorigo and L. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 53–66, 1997.

[11] B. Barán and M. Schaerer, "A multiobjective ant colony system for vehicle routing problem with time windows." in *Proceedings of the 21st IASTED International Conference on Applied Informatics*, February 2003, pp. 97–102.

[12] A. Koziolek, D. Ardagna, and R. Mirandola, "Hybrid multi-attribute QoS optimization in component based software systems," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2542 – 2558, 2013.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, Apr 2002.

[14] X.-B. Hu, M. Wang, and E. Di Paolo, "Calculating complete and exact Pareto front for multiobjective optimization: A new deterministic approach for discrete problems," *Cybernetics, IEEE Transactions on*, vol. 43, no. 3, pp. 1088–1101, June 2013.

[15] R. M. Karp, M. Luby, and A. Marchetti-Spaccamela, "A probabilistic analysis of multidimensional bin packing problems," in *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '84. ACM, 1984, pp. 289–298.

[16] S. Iredi, D. Merkle, and M. Middendorf, "Bi-criterion optimization with multi colony ant algorithms," in *Evolutionary Multi-Criterion Optimization*, ser. Lecture Notes in Computer Science, E. Zitzler, L. Thiele, K. Deb, C. Coello Coello, and D. Corne, Eds. Springer Berlin Heidelberg, 2001, vol. 1993, pp. 359–372.

[17] A. Ashraf and I. Porres, "Using ant colony system to consolidate multiple web applications in a cloud environment," in *22nd Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2014, pp. 482–489.

[18] F. Farahnakian, A. Ashraf, P. Liljeberg, T. Pahikkala, J. Plosila, I. Porres, and H. Tenhunen, "Energy-aware dynamic VM consolidation in cloud data centers using ant colony system," in *7th IEEE International Conference on Cloud Computing (CLOUD)*, 2014.

[19] L. M. Gambardella, É. Taillard, and G. Agazzi, "MACS-VRPTW: A multiple ant colony system for vehicle routing problems with time windows," in *New Ideas in Optimization*, D. Corne, M. Dorigo, and F. Glover, Eds. McGraw-Hill, London, UK, 1999, pp. 63–76.

[20] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760 – 771, 2011.