Using Ant Colony System to Consolidate Multiple Web Applications in a Cloud Environment

Adnan Ashraf*†‡, Ivan Porres*†
* Department of Information Technologies, Åbo Akademi University, Turku, Finland.
Email: aashraf@abo.fi, iporres@abo.fi
† Turku Centre for Computer Science (TUCS), Turku, Finland.
‡ Department of Software Engineering, International Islamic University, Islamabad, Pakistan.

***Abstract*—Infrastructure as a Service (IaaS) clouds provide virtual machines (VMs) under a pay-per-use business model, which can be used to create a dynamically scalable cluster of servers to deploy one or more web applications. In contrast to the traditional dedicated hosting of web applications where each VM is used exclusively for one particular web application, the shared hosting of web applications allows improved VM utilization by sharing VM resources among multiple concurrent web applications. However, in a shared hosting environment, dynamic scaling alone does not minimize over-provisioning of VMs. In this paper, we present a novel approach to consolidate multiple web applications in a cloud-based shared hosting environment. The proposed approach uses Ant Colony Optimization (ACO) to build a web application migration plan, which is then used to minimize over-provisioning of VMs by consolidating web applications on under-utilized VMs. The proposed approach is demonstrated in discrete-event simulations and is evaluated in a series of experiments involving synthetic as well as realistic load patterns.**

***Keywords*-Web applications; consolidation; metaheuristic; ant colony optimization; cloud computing; shared hosting**

I. Introduction

Web applications are often deployed in a three-tier computer architecture, where the application server tier usually uses a computer cluster to process a large number of concurrent user requests. Traditionally, these clusters are composed of a fixed number of computers and are dimensioned to serve a predetermined maximum number of concurrent users. However, Infrastructure as a Service (IaaS) clouds, such as Amazon Elastic Compute Cloud (EC2)[1], currently offer computing resources such as network bandwidth, storage, and virtual machines (VMs) under a pay-per-use business model. Thus, the IaaS clouds enable the creation of a dynamically scalable server tier, where VMs can be added and removed to dynamically provide a good trade-off between cost and performance.

Determining the number of VMs to provision for a dynamically scalable cluster is an important problem. The exact number of VMs needed at a specific time depends upon the user load and the Quality of Service (QoS) requirements. Under-provisioning leads to subpar service, while over-provisioning results in increased operation costs. There are several research works [1], [2], [3], [4], [5], [6], [7], [8], [9], [10] as well as some vendor-specific commercial solutions, such as AWS Elastic Beanstalk[2], which provide dynamic VM provisioning. However, a common characteristic of these solutions is that they use dedicated hosting [11], where each VM is exclusively used for one particular application. This is a reasonable approach if an application has enough user load to keep at least one VM sufficiently well-utilized. However, in many cases, dedicated hosting may introduce unnecessary overhead and cost due to under-utilization of servers.

The under-utilization of VMs becomes even more pertinent when a Software as a Service (SaaS) or a Platform as a Service (PaaS) provider wants to leverage an IaaS cloud to cost-efficiently deploy a large number of web applications of varying resource needs. The solution to this problem is to create a dynamically scalable application server tier that manages multiple applications simultaneously, while using shared hosting [11] to deploy multiple applications on a VM [12], [13]. A similar fine-grained resource sharing approach was used in Mesos [14]. However, it provides a platform for multiple cluster computing frameworks rather than web applications. In our previous works, we presented a reactive [12] and a proactive [13] VM provisioning approach for the application server tier. These approaches provide dynamic scaling of multiple web applications on a given IaaS cloud in a shared hosting environment. However, dynamic scaling alone does not guarantee cost-efficient deployment of multiple web applications. Thus, it is difficult to provide a good trade-off between cost and performance, which minimizes over-provisioning of VMs.

In this paper, we present a novel approach to consolidate [15], [16], [17], [18] multiple web applications in a cloud-based shared hosting environment. We propose an application consolidation algorithm that uses a metaheuristic [19], [20] approach called Ant Colony Optimization (ACO) [21], [22] to build a web application migration plan, which is then used to minimize over-provisioning of VMs by consolidating web applications on under-utilized VMs. The proposed approach is demonstrated in discrete-event simulations and is evaluated in a series of experiments involving synthetic as well as realistic load patterns.

We proceed as follows. Section II provides background and discusses important related works. Section III presents

[1]http://aws.amazon.com/ec2

[2]http://aws.amazon.com/elasticbeanstalk/

the system architecture of the cloud-based shared hosting environment and sets the context for the proposed application consolidation algorithm. The proposed algorithm is presented in Section IV. In Section V, we describe experimental design and setup. The results of the experimental evaluation are presented in Section VI. Finally, we present our conclusions in Section VII.

II. Background and Related Work

Most of the existing works on VM provisioning for web-based systems can be classified into two main categories: Plan-based approaches and control theoretic approaches [8], [9], [10]. Plan-based approaches can be further classified into workload prediction approaches [1], [2] and performance dynamics model approaches [3], [4], [5], [6], [7]. For web application hosting in a cloud-based environment, the existing works tend to use dedicated hosting on the VM level. It provides the desired level of isolation to safely host multiple third-party applications, which should not interfere with each other. However, the main drawback of dedicated hosting is that it prohibits sharing of VM resources among multiple concurrent applications. This is especially important when deploying a large number of web applications of varying resource needs, where most of the applications may have very few users at a given time, while a few applications may have many users. Therefore, in order to use shared hosting of multiple third-party applications, there is a need for a way to prevent applications from interfering with one other.

In our previous works [12], [13], [23], we presented a cloud-based shared hosting approach where each virtualized application server runs multiple Java Servlet-based web applications in the same Java Virtual Machine (JVM). However, since Java lacks some important features needed to safely run multiple third-party web applications in one JVM, we extended and used a widely adapted OSGi specification [24], which partly addresses this problem [23]. Thus, in this way, shared hosting enables safe deployment of multiple concurrent third-party web applications on each virtualized application server. However, it is difficult to provide cost-efficient deployment of multiple web applications in a cloud-based shared hosting environment without consolidation of web applications on under-utilized VMs.

The existing server consolidation approaches, such as [15], [16], [17], [18] are used in data centers to minimize under-utilization of physical machines and to optimize their power-efficiency. The main idea in these approaches is to use live VM migration to periodically consolidate VMs so that some of the under-utilized physical machines could be released for termination. In this paper, we propose to use a similar technique to cost-efficiently consolidate multiple concurrent third-party web applications in a cloud-based shared hosting environment. Therefore, a key difference in our proposed approach is that we consolidate applications on VMs, rather than consolidating VMs on physical machines. Thus, our prime concern is to release some of the under-utilized VMs for termination so that the total number of provisioned VMs can be reduced without compromising the overall performance. Although the application consolidation problem has certain similarities with the server consolidation problem, an important difference is that the application consolidation problem is intrinsically more dynamic. This is because, based on the user load, web applications keep on changing their resource demands. On the other hand, the existing server consolidation approaches assume that the VMs are static in nature [16], that is, they do not change their resource demands. Thus, one of the challenges in the application consolidation problem is to reduce the computation time of the consolidation algorithm so that the dynamic nature of web applications and their changing resource demands can be accommodated.

Since cost-efficient application consolidation is a combinatorial optimization problem, we apply a highly adaptive online optimization [20] approach called Ant Colony Optimization (ACO) [21], [22] to find a near-optimal solution. ACO is a multi-agent approach to difficult combinatorial optimization problems, such as, travelling salesman problem (TSP) and network routing [21]. It is inspired by the foraging behavior of real ant colonies. While moving from their nest to the food source and back, ants deposit a chemical substance on their path called pheromone. Other ants can smell pheromone and they tend to prefer paths with a higher pheromone concentration. Thus, ants behave as agents who use a simple form of indirect communication called *stigmergy* to find better paths between their nest and the food source. It has been shown experimentally that this simple pheromone trail following behavior of ants can give rise to the emergence of the shortest paths [21]. It is important to note here that although each ant is capable of finding a complete solution, high quality solutions emerge only from the global cooperation among the members of the colony who concurrently build different solutions. Moreover, to find a high quality solution, it is imperative to avoid *stagnation*, which is a premature convergence to a suboptimal solution or a situation where all ants end up finding the same solution without sufficient exploration of the search space [21]. In ACO metaheuristic, stagnation is avoided mainly by using pheromone evaporation and stochastic state transitions.

There are a number of ant algorithms, such as, Ant System (AS), Max-Min AS (MMAS), and Ant Colony System (ACS) [21], [22]. ACS [21] was introduced to improve the performance of AS and it is currently one of the best performing ant algorithms. Therefore, in this paper, we apply ACS to the web application consolidation problem.

One of the earlier works on applying ACO to the general resource allocation problem include [25]. The authors in [25] applied ACO to the nonlinear resource allocation problem, which seeks to find an optimal allocation of a limited amount of resources to a number of tasks to optimize their nonlinear objective function. A more recent work by Feller et al. [16] applied MMAS to the server consolidation problem in the context of cloud computing. However, to the best of our knowledge, currently there are no existing works on using ACO metaheuristic to consolidate multiple web applications in a cloud-based shared hosting environment.

III. System Architecture

The proposed application consolidation algorithm is part of a larger project that aims at providing an open-source PaaS integrated solution for web application development, deployment, and dynamic scaling in a shared hosting environment [23]. In this context, it provides a cost-efficient scaling down algorithm for the PaaS clouds. The system architecture of the cloud-based shared hosting environment consists of the following components, as shown in Figure 1: *global controller*, *admission controller*, *application server*, *local controller*, *load predictor*, *application repository*, *cloud provisioner*, *HTTP load balancer*, and *entertainment server*. In this paper, our primary focus is on the *global controller*, which implements VM provisioning and application consolidation algorithms.

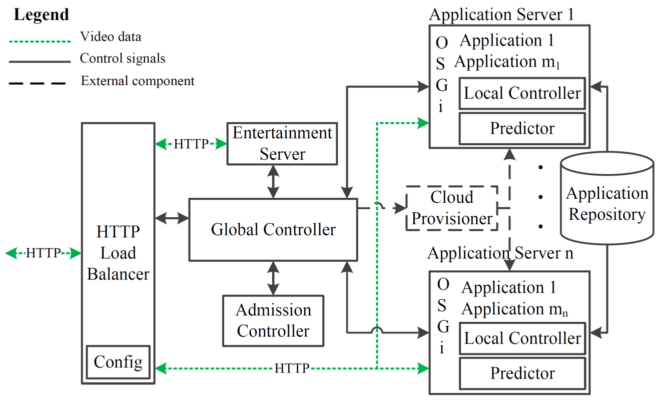

Fig. 1. System architecture of the cloud-based shared hosting environment

An application server instance runs on a dynamically provisioned VM. Each application server runs multiple web applications in an OSGi [24] environment. In such an environment, each application runs as an OSGi component called a bundle. OSGi also introduces the dynamic component model, which allows dynamic loading and unloading of bundles. In addition to the web applications, each application server also runs a local controller. The local controller monitors and logs server resource utilizations. It also controls the OSGi environment for loading and unloading of web applications. Web applications are stored in an application repository, from where they are loaded onto application servers. For applications under the OSGi environment, we use Apache Felix OSGi Bundle Repository (OBR)[3]. The global controller implements VM provisioning and application consolidation algorithms along with the session-to-server allocation and application-to-server allocation policies [12]. These policies and our VM provisioning algorithms are described in detail in [12], [13].

The cloud provisioner refers to the cloud provisioner in an external IaaS cloud, such as the provisioner in Amazon EC2. While the VM provisioning and termination decisions are made by the global controller, the actual lower level tasks of starting and terminating VMs are done by the cloud provisioner. The HTTP requests are routed through a high performance HTTP load balancer and proxy. For this, we use HAProxy[4], which balances the load of requests for new user sessions among the application servers. For its functions, HAProxy maintains a configuration file containing information about application servers and application deployments on each server. As a result of the VM provisioning and termination operations, the configuration file is frequently updated with new information.

When a new session request arrives, the admission controller [26] obtains measured resource utilizations of individual servers from the global controller and likewise the predicted resource utilizations from the load predictor. Based on the measured and predicted resource utilizations, it updates the server states. At any given time, an application server can be in one of three states: *open*, *closed*, or *overloaded*. The open state implies that the server is open for new sessions. Similarly, the closed state means that the server does not accept new sessions. The overloaded is an undesirable state, which is characterized by very high utilization of the bottleneck server resource that may result in deteriorated performance. The admission controller uses these server states to make decisions for new session requests. If the admission controller finds at least one open server, the new session is admitted. If it cannot find an open server, the session is deferred onto the entertainment server. However, if the entertainment server also becomes closed or overloaded, the new session request is rejected. All deferred sessions are automatically redirected to an application server in a FIFO (First In, First Out) order as soon as a new server is provisioned or a closed server becomes open. When admitting a mix of deferred and new sessions, deferred sessions are given priority over new sessions. Our admission control and load prediction algorithms are described in detail in [26].

IV. ACO-Based Consolidation of Web Applications

The pseudocode of the proposed web application consolidation algorithm is given as Algorithm 1. For the sake of clarity, the concepts used in the algorithm and their notations are tabulated in Table I.

In the cloud-based shared hosting environment presented in Section III, each virtualized application server hosts one or more web applications from the set of applications A. Moreover, since frequently used applications are often concurrently run on multiple VMs, if a particular application $a_1 \in A$ is concurrently hosted by two VMs ($v_1 \in V$ and $v_2 \in V$), then each deployment of a_1 is considered a separate *application instance*. An application instance not only contains a web application $a \in A$, but also the user sessions that belong to it. Furthermore, for the sake of application migration, each VM is a potential *source VM*. Both the source VM and the application instance are characterized by their resource utilizations, such as CPU load average and

[3]http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html

[4]http://haproxy.1wt.eu/

TABLE I
SUMMARY OF CONCEPTS AND THEIR NOTATIONS

A	set of web applications
A_v	set of applications running on a VM v
MS	a set of migration plans
T	a set of tuples
T_k	a set of tuples not yet traversed by ant k
V	set of VMs
V_R	a set of VMs that are released when M is enforced
a	application instance in a tuple
$C_{V_{de}}$	total capacity vector of the destination VM V_{de}
M	a migration plan
M^+	the global best migration plan
M_k	ant-specific migration plan of ant k
M_k^m	ant-specific temporary migration plan of ant k
N	a neighborhood of VMs
q	a uniformly distributed random variable
RT_v	remaining time of a VM v with respect to the renting hour
S	a random variable selected according to (6)
Scr_k	thus far best score of ant k
U_a	used capacity vector of the application instance a
$U_{V_{de}}$	used capacity vector of the destination VM V_{de}
$U_{V_{so}}$	used capacity vector of the source VM V_{so}
V_{de}	destination VM in a tuple
V_{so}	source VM in a tuple
η	heuristic value
τ	amount of pheromone
τ_0	initial pheromone level
$\Delta_{\tau_s}^+$	additional pheromone amount given to the tuples in M^+
q_0	parameter to determine relative importance of exploitation
α	pheromone decay parameter in the global updating rule
β	parameter to determine the relative importance of η
γ	parameter to determine the relative importance of $\lvert V_R \rvert$
ρ	pheromone decay parameter in the local updating rule
nA	number of ants that concurrently build their migration plans
nI	number of iterations of the main loop in the algorithm
RT_L	remaining time lower threshold
RT_U	remaining time upper threshold

memory utilization. Likewise, an application instance can be migrated to any other VM located in any neighborhood N. The neighborhoods of VMs are mutually exclusive subsets of V. Therefore, every other VM within as well as outside the neighborhood of the source VM is a potential *destination VM*, which is also characterized by its resource utilizations. Thus, the proposed ACO-based algorithm makes a set of tuples T, where each tuple $t \in T$ consists of three elements: source VM V_{so}, application instance a, and destination VM V_{de}

$$t := (V_{so}, a, V_{de}) \tag{1}$$

Therefore, the VMs in the web application consolidation problem are analogous to the cities in the TSP, while the tuples are analogous to the edges that connect the cities. As noted in Section II, the application consolidation problem is intrinsically more dynamic than the server consolidation problem. Therefore, it is imperative to reduce the computation time of the consolidation algorithm, which is primarily based on the number of tuples $|T|$. Thus, when making the set of tuples T, the algorithm applies two constraints, which result in a reduced set of tuples by removing some least important and unwanted tuples. The first constraint ensures that only under-utilized, close to the completion of renting hour VMs are used as the source and destination VMs. In other words, migrations from and to well-utilized VMs are excluded. The rationale is that a well-utilized VM should not become part of the consolidation process because migrating to a well-utilized VM may result in its overloading. Similarly, migrating from a well-utilized VM is less likely to result in the termination of the source VM and thus it would not reduce the total number of required VMs. Moreover, since some contemporary IaaS providers, such as Amazon EC2, charge on hourly basis, only those VMs participate in the consolidation process which are close to the completion of their renting hour [27]. The second constraint further restricts the size of the set of tuples $|T|$ by preventing inter-neighborhood migrations. Therefore, an application instance can only be migrated to another VM within the neighborhood of its source VM. By applying these two simple constraints in a series of preliminary experiments, we observed that the computation time of the algorithm was significantly reduced without compromising the quality of the solutions.

The output of the application consolidation algorithm is a migration plan, which, when enforced, would result in a minimal set of VMs needed to host all web applications without compromising their performance. Thus, the objective function for the proposed algorithm is

$$max\ f(M) := |V_R|^\gamma + |M| \tag{2}$$

where M is the migration plan and V_R is the set of VMs that will be released when M is enforced. The parameter γ determines the relative importance of $|V_R|$ with respect to $|M|$. Since the ultimate objective in the cloud-based application consolidation algorithm is to minimize VM provisioning cost, which is a function of the number of provisioned VMs and time, the objective function is defined in terms of number of released VMs $|V_R|$. Moreover, it prefers larger migration plans because with a large set of web applications A in a shared hosting environment, each VM $v \in V$ typically hosts a number of applications $A_v \in A$, which makes it less likely to find a feasible solution with a smaller migration plan. Later on, when a migration plan is enforced, we apply a constraint which reduces the number of actual migrations by restricting migrations to only those VMs that are not included in the set of released VMs V_R, that is

$$\forall\, V_{de} \in V \mid V_{de} \notin V_R \tag{3}$$

In our approach, a VM can only be considered released when all the application instances are migrated from it, that is, when the VM no longer hosts any applications. Moreover, only those VMs should be terminated which are close to the completion of their renting hour. Thus, the set of released VMs V_R is defined as

$$V_R := \{\forall v \in V | A_v = \emptyset \wedge RT_L < RT_v < RT_U\} \tag{4}$$

where A_v is the set of applications running on a VM v, RT_v is the remaining time of a VM v from the completion of its renting hour, RT_L is the remaining time lower threshold, and similarly RT_U is the remaining time upper threshold. Thus, a

VM can only be included in the set of released VMs V_R when it no longer hosts any applications and its remaining time $RT_v \in (RT_L, RT_U)$. The rationale is to terminate only those VMs that are close to the completion of their renting hour, while excluding any VMs that are extremely close and therefore it is difficult to terminate them before the start of the next renting hour [27].

Unlike the TSP, there is no notion of a path in the application consolidation problem. Therefore, 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 is called pseudo-random-proportional-rule [22]. According to this rule, an ant k chooses a tuple s to traverse next by applying

$$s := \begin{cases} \text{arg max}_{u \,\in\, T_k}\{[\tau_u] \cdot [\eta_u]^\beta\}, & \text{if } q \le q_0 \\ S, & \text{otherwise} \end{cases} \tag{5}$$

where τ denotes the amount of pheromone and η represents the heuristic value associated with a particular tuple. β is a parameter to determine the relative importance of the heuristic value with respect to the pheromone value. The expression *arg max* returns the tuple for which $[\tau] \cdot [\eta]^\beta$ 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 (6), where the probability p_s of an ant k to choose tuple s to traverse next is defined as

$$p_s := \begin{cases} \frac{[\tau_s] \cdot [\eta_s]^\beta}{\sum\limits_{u \,\in\, T_k} [\tau_u] \cdot [\eta_u]^\beta}, & \text{if } s \in T_k \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

The heuristic value η_s of a tuple s is defined in a similar fashion as in [16] as

$$\eta_s := \begin{cases} (|C_{V_{de}} - (U_{V_{de}} + U_a)|_1)^{-1}, & \text{if } U_{V_{de}} + U_a \le C_{V_{de}} \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

where $C_{V_{de}}$ is the total capacity vector of the destination VM V_{de}, $U_{V_{de}}$ is the used capacity vector of V_{de}, and likewise U_a is the used capacity vector of the application instance a in tuple s. The heuristic value η is based on the multiplicative inverse of the scalar-valued difference between $C_{V_{de}}$ and $U_{V_{de}} + U_a$. It favors application migrations that result in a reduced under-utilization of VMs. Moreover, the constraint $U_{V_{de}} + U_a \le C_{V_{de}}$ prevents application migrations that would result in the overloading of the destination VM V_{de}. In the proposed algorithm, we assumed two resource dimensions, which represent CPU load average and memory utilization. However, if necessary, it is possible to add more dimensions in the total and used capacity vectors.

The stochastic state transition rule in (5) and (6) prefers tuples with a higher pheromone concentration and which result in a higher number of released VMs. The first case in (5) where $q \le q_0$ is called exploitation [22], which chooses the best tuple that attains the maximum value of $[\tau] \cdot [\eta]^\beta$. The second case, called biased exploration, selects a tuple according to (6). 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. 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 migration plans. It is defined as

$$\tau_s := (1 - \alpha) \cdot \tau_s + \alpha \cdot \Delta^+_{\tau_s} \tag{8}$$

where $\Delta^+_{\tau_s}$ is the additional pheromone amount that is given only to those tuples that belong to the global best migration plan in order to reward them. It is defined as

$$\Delta^+_{\tau_s} := \begin{cases} f(M^+), & \text{if } s \in M^+ \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

$\alpha \in (0, 1]$ is the pheromone decay parameter, and M^+ is the global best migration plan from the beginning of the trial.

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

$$\tau_s := (1 - \rho) \cdot \tau_s + \rho \cdot \tau_0 \tag{10}$$

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 approximate optimal $|M|$ and $|V|$

$$\tau_0 := (|M| \cdot |V|)^{-1} \tag{11}$$

Here, any very rough approximation of the optimal $|M|$ would suffice [22]. The pseudo-random-proportional-rule in ACS and the global pheromone trail update rule are intended to make the search more directed. The pseudo-random-proportional-rule prefers tuples with a higher pheromone level and a higher heuristic value. Therefore, the ants try to search other high quality solutions in a close proximity of the thus far global best solution. On the other hand, the local pheromone trail update rule complements exploration of other high quality solutions that may exist far form the thus far global best solution. This is because whenever an ant traverses a tuple and applies the local pheromone trail update rule, the tuple looses some of its pheromone and thus becomes less attractive for other ants. Therefore, it helps in avoiding stagnation where all ants end up finding the same solution or where they prematurely converge to a suboptimal solution.

The pseudocode in Algorithm 1 makes a set of tuples T using (1) and sets the pheromone value of each tuple to the initial pheromone level τ_0 by using (11) (line 2). The algorithm iterates over nI iterations (line 3). In each iteration, nA ants concurrently build their migration plans (lines 4–18). Each ant iterates over $|T|$ tuples (lines 6–16). It computes the probability of choosing the next tuple to traverse by using (6) (line 7). Afterwards, based on the computed probabilities and the stochastic state transition rule in (5) and (6), each ant chooses a tuple t to traverse (line 8) and adds t to its temporary migration plan M_k^m (line 9). The local pheromone trail update rule in (10) and (11) is applied on t (line 10), the used capacity vectors at the source VM $U_{V_{so}}$ and the destination VM $U_{V_{de}}$ in t are updated to reflect the impact of the migration (line 11), the objective function in (2) is applied on M_k^m, and if it yields a score higher than the ant's thus far best score Scr_k (line 12), t is added to the ant-specific migration plan M_k (line 14). Then, towards the end of an iteration when all ants complete their migration plans, all ant-specific migration plans are added to the set of migration plans MS (line 17), each migration plan $M_k \in MS$ is evaluated by applying the objective function in (2), the thus far global best application migration plan M^+ is selected (line 19), and the global pheromone trail update rule in (8) and (9) is applied on all tuples (line 20). Finally, when all iterations complete, the algorithm outputs the global best migration plan M^+.

Algorithm 1 Application consolidation algorithm

```
1: $M^+ := \emptyset$, $MS := \emptyset$
2: $\forall t \in T | \tau_t := \tau_0$
3: for $i \in [1, nI]$ do
4:     for $k \in [1, nA]$ do
5:         $M_k^m := \emptyset, M_k := \emptyset, Scr_k := 0$
6:         while $|M_k^m| < |T|$ do
7:             compute $p_s \quad \forall s \in T$ by using (6)
8:             choose a tuple $t \in T$ to traverse by using (5)
9:             $M_k^m := M_k^m \cup \{t\}$
10:            apply local update rule in (10) on $t$
11:            update used capacity vectors $U_{V_{so}}$ and $U_{V_{de}}$ in $t$
12:            if $f(M_k^m) > Scr_k$ then
13:                $Scr_k := f(M_k^m)$
14:                $M_k := M_k \cup \{t\}$
15:            end if
16:        end while
17:        $MS := MS \cup \{M_k\}$
18:    end for
19:    $M^+ := \arg\max_{M_k \in MS}\{f(M_k)\}$
20:    apply global update rule in (8) on all $s \in T$
21: end for
```

V. Experimental Design and Setup

A convenient and quick way of testing new algorithms and solutions involving complex environments is to write and run software simulations [26]. A special kind of simulations called discrete-event simulations are most appropriate for simulating and evaluating cluster, grid, and cloud computing environments and systems [28]. Therefore, we have developed a discrete-event simulation for the proposed application consolidation approach. Also, for a comparison of the results, we have developed a discrete-event simulation for a greedy application consolidation algorithm, here referred to as the *baseline* approach. The greedy consolidation algorithm is an extension of our previous works in [12] and [13] and serves as the base case for the comparison of the results. It periodically performs web application consolidation whenever at least one under-utilized VM is found. It marks each under-utilized VM as a source VM V_{so} and similarly each well-utilized VM as a destination VM V_{de}. Afterwards, it migrates each application instance from each V_{so} to a V_{de}, which is chosen according to the application-to-server allocation policy [12], and terminates each V_{so}.

TABLE II
ACS parameters in the proposed approach

α	β	γ	ρ	q_0	nA	nI	$\lvert N\rvert$
0.1	0.9	5	0.1	0.9	10	2	5

We considered two scenarios of interest in two separate experiments. Scenario 1 in experiment 1 used synthetic workload traces, while scenario 2 in experiment 2 used workload traces derived from a real web-based system. In both experiments, the ACS parameters that were used in the proposed approach are tabulated in Table II. Moreover, the remaining time lower threshold RT_L was set to 2 minutes and similarly the remaining time upper threshold RT_U was 15 minutes. These parameter values were obtained in a series of preliminary experiments.

1) Experiment 1: Synthetic Load Pattern: Experiment 1 used synthetic workload traces. It was designed to generate a load representing a maximum of 10000 simultaneous user sessions in two separate load peaks. In each peak, the sessions were ramped up to 10000. After the ramp-up phase, the number of sessions was maintained constant for a while and then reduced back to 5000 in the first peak and to 0 in the second peak. The two load peaks were similar, except that the sessions in the first peak were ramped up twice as quickly as in the second peak. Each session was randomly assigned to one particular web application. The experiment used $|A| = 10$ simulated web applications of varying resource needs.

2) Experiment 2: Realistic Load Pattern: Experiment 2 was designed to simulate a load representing a workload trace from a real web-based system. The traces were derived from SQUID access logs obtained from the IRCache[5] project. As the access logs did not include session information, we defined a session as a series of requests from the same originating IP-address, where the time between individual requests was less than 15 minutes. We then produced a histogram of sessions per second and used linear interpolation and scaling by a factor of 300 to obtain the load pattern used in the experiment. As in Experiment 1, each session was randomly assigned to one particular web application out of $|A| = 10$.

VI. Experimental Results

Now we compare the experimental results of the proposed web application consolidation approach with that of the baseline approach. Each result in Figure 2 to Figure 5 has the following plots: number of concurrent user sessions, number of VMs, total number of VM hours, total application migrations, CPU load average of all servers, and average memory utilization of all servers. The ultimate objective is

[5]http://www.ircache.net/

TABLE III
EXPERIMENT 1 RESULTS

	Baseline approach	Proposed approach
maximum $\|V\|$	39	38
total VM hours	3575	2612
total migrations	443	610
CPU load average	79% maximum	83% maximum

TABLE IV
EXPERIMENT 2 RESULTS

	Baseline approach	Proposed approach
maximum $\|V\|$	39	28
total VM hours	4453	3170
total migrations	1004	700
CPU load average	78% maximum	98% maximum

to minimize VM provisioning cost while maintaining CPU load average and average memory utilization below 100%. Moreover, since the VM provisioning cost is a function of the number of provisioned VMs and time, the comparison of the results is primarily based on the total number of VM hours.

3) Experiment 1: Synthetic Load Pattern: The results of the baseline approach are shown in Figure 2. The results show a dynamically scalable application server tier, which consists of a varying number of VMs. The baseline approach used a maximum of 39 VMs. The total number of VM hours were 3575 and the total number of application migrations were 443. The CPU load average and the average memory utilization were always below 100%.

Figure 3 presents the results of the proposed approach from experiment 1. The proposed approach used a maximum of 38 VMs. The total number of VM hours were 2612 and the total number of application migrations were 610. The CPU load average and the average memory utilization were always below 100%. The results show that the proposed approach used 26.94% less VM hours as compared to the baseline approach. Thus, it provided a more cost-efficient solution for web application consolidation in a cloud-based shared hosting environment. Table III presents a summary of the results from experiment 1.

4) Experiment 2: Realistic Load Pattern: Figure 4 presents the results of the baseline approach from experiment 2. The baseline approach used a maximum of 39 VMs. The total number of VM hours were 4453 and the total number of application migrations were 1004. The CPU load average and the average memory utilization were always below 100%.

The results of the proposed approach from experiment 2 are shown in Figure 5. The proposed approach used a maximum of 28 VMs. The total number of VM hours were 3170 and the total number of application migrations were 700. The CPU load average and the average memory utilization were always below 100%. The results from experiment 2 show that the proposed approach used 28.81% less VM hours as compared to the baseline approach. Thus, it provided a more cost-efficient solution for web application consolidation in experiment 2 as well. A summary of the results from experiment 2 is presented in Table IV.

VII. Conclusion

In this paper, we presented a novel web application consolidation approach for a cloud-based shared hosting environment. Our proposed application consolidation algorithm uses Ant Colony System (ACS) to build a web application migration plan, which is then used to minimize over-provisioning of VMs by consolidating web applications on under-utilized VMs. We presented a discrete-event simulation of the proposed approach along with an experimental evaluation involving synthetic as well as realistic load patterns.

The evaluation and analyses compared the proposed approach against a baseline, greedy application consolidation approach, which is based on an extension of our previous works on VM provisioning algorithms. We considered two different scenarios in two separate experiments. The first scenario used a synthetic workload pattern, while the second scenario used workload traces derived from a real web-based system. The results showed that the proposed approach provides a cost-efficient solution for web application consolidation in a cloud-based shared hosting environment. In comparison with the baseline approach, it provided significant improvements in terms of the total number of VM hours. In the first experiment involving a synthetic load pattern, the proposed approach used 26.94% less VM hours as compared to the baseline

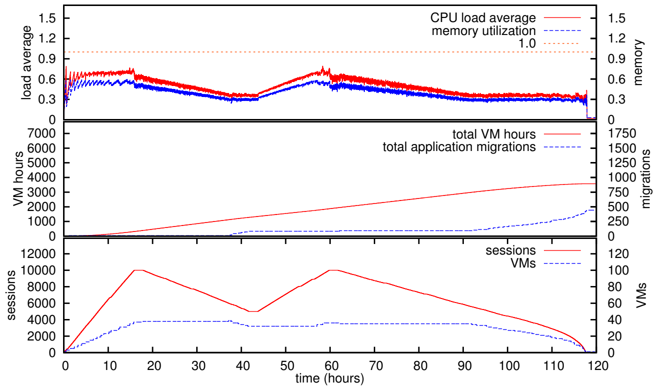

Fig. 2. Experiment 1: baseline approach with synthetic load pattern

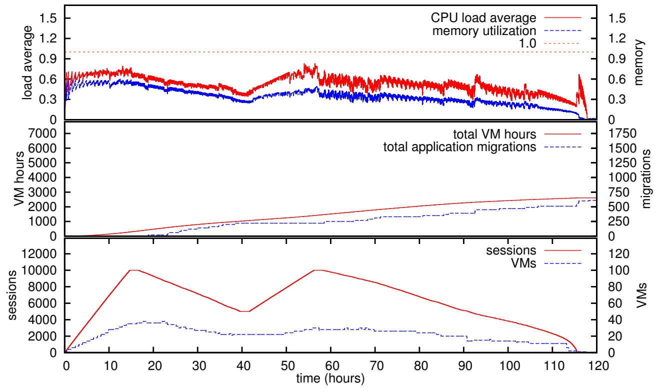

Fig. 3. Experiment 1: proposed approach with synthetic load pattern

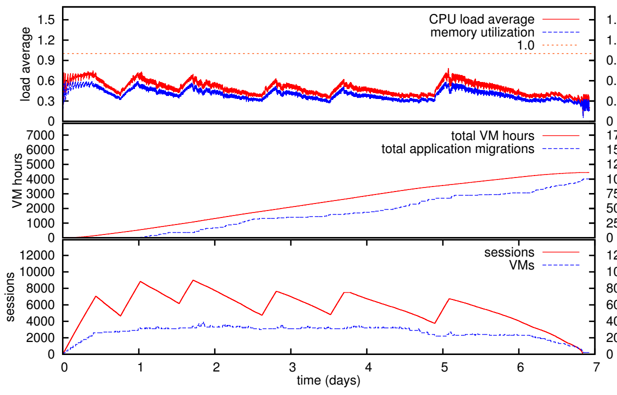

Fig. 4. Experiment 2: baseline approach with realistic load pattern

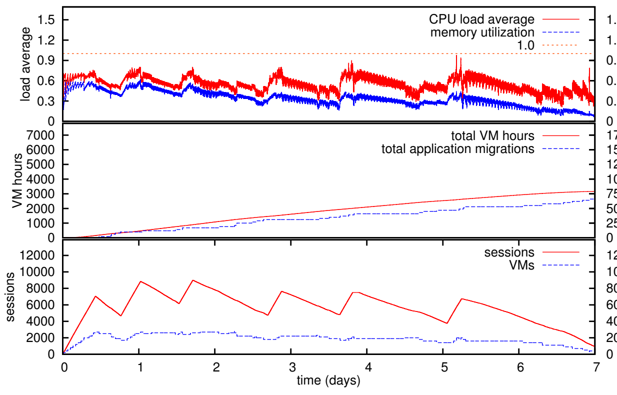

Fig. 5. Experiment 2: proposed approach with realistic load pattern

approach. Similarly, it used 28.81% less VM hours in the second experiment that was based on a realistic load pattern.

ACKNOWLEDGEMENTS

This work was supported by the Cloud Software Finland research project and by an Amazon Web Services research grant. Adnan Ashraf was partially supported by the Foundation of Nokia Corporation and by a doctoral scholarship from the Higher Education Commission (HEC) of Pakistan.

REFERENCES

[1] Y. Raivio, O. Mazhelis, K. Annapureddy, R. Mallavarapu, and P. Tyrväinen, "Hybrid cloud architecture for short message services," in *Proceedings of the 2nd International Conference on Cloud Computing and Services Science*. SciTePress, 2012, pp. 489–500.

[2] D. Ardagna, C. Ghezzi, B. Panicucci, and M. Trubian, "Service provisioning on the cloud: Distributed algorithms for joint capacity allocation and admission control," in *Towards a Service-Based Internet*, ser. Lecture Notes in Computer Science, E. Di Nitto and R. Yahyapour, Eds. Springer Berlin / Heidelberg, 2010, vol. 6481, pp. 1–12.

[3] A. Wolke and G. Meixner, "TwoSpot: A cloud platform for scaling out web applications dynamically," in *Towards a Service-Based Internet*, ser. Lecture Notes in Computer Science, E. di Nitto and R. Yahyapour, Eds. Springer Berlin / Heidelberg, 2010, vol. 6481, pp. 13–24.

[4] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu, "Resource provisioning for cloud computing," in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, 2009, pp. 101–111.

[5] T. Chieu, A. Mohindra, A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, oct. 2009, pp. 281–286.

[6] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871–879, 2011.

[7] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," *Cluster Computing and the Grid, IEEE International Symposium on*, pp. 644–651, 2012.

[8] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From data center resource allocation to control theory and back," in *Cloud Computing, 2010 IEEE 3rd International Conference on*, 2010.

[9] W. Pan, D. Mu, H. Wu, and L. Yao, "Feedback control-based QoS guarantees in web application servers," in *High Performance Computing and Communications, 10th IEEE International Conference on*, 2008.

[10] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A multi-model framework to implement self-managing control systems for QoS management," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2011.

[11] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in a shared internet hosting platform," *ACM Trans. Internet Technol.*, vol. 9, no. 1, pp. 1–45, Feb. 2009.

[12] A. Ashraf, B. Byholm, J. Lehtinen, and I. Porres, "Feedback control algorithms to deploy and scale multiple web applications per virtual machine," in *38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, 2012, pp. 431–438.

[13] A. Ashraf, B. Byholm, and I. Porres, "CRAMP: Cost-efficient resource allocation for multiple web applications with proactive scaling," in *4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE Computer Society, 2012, pp. 581–586.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011.

[15] W. Vogels, "Beyond server consolidation," *ACM Queue*, vol. 6, no. 1, pp. 20–26, Jan. 2008.

[16] E. Feller, C. Morin, and A. Esnault, "A case for fully decentralized dynamic VM consolidation in clouds," *Cloud Computing Technology and Science, IEEE International Conference on*, pp. 26–33, 2012.

[17] A. Murtazaev and S. Oh, "Sercon: Server consolidation algorithm using live migration of virtual machines for green computing," *IETE Technical Review*, vol. 28, no. 3, pp. 212–231, 2011.

[18] M. Marzolla, O. Babaoglu, and F. Panzieri, "Server consolidation in clouds through gossiping," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a*, 2011.

[19] C. Blum, J. Puchinger, G. R. Raidl, and A. Roli, "Hybrid metaheuristics in combinatorial optimization: A survey," *Applied Soft Computing*, vol. 11, no. 6, pp. 4135 – 4151, 2011.

[20] M. Harman, K. Lakhotia, J. Singer, D. R. White, and S. Yoo, "Cloud engineering is search based software engineering too," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2225 – 2241, 2013.

[21] 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.

[22] 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.

[23] T. Aho, A. Ashraf, M. Englund, J. Katajamki, J. Koskinen, J. Lautamki, A. Nieminen, I. Porres, and I. Turunen, "Designing IDE as a service," *Communications of Cloud Software*, vol. 1, pp. 1–10, 2011.

[24] The OSGi Alliance, *OSGi Service Platform Core Specification, Release 4, Version 4.3*, 2011.

[25] P.-Y. Yin and J.-Y. Wang, "Ant colony optimization for the nonlinear resource allocation problem," *Applied Mathematics and Computation*, vol. 174, no. 2, pp. 1438 – 1453, 2006.

[26] A. Ashraf, B. Byholm, and I. Porres, "A session-based adaptive admission control approach for virtualized application servers," in *The 5th IEEE/ACM International Conference on Utility and Cloud Computing*, 2012, pp. 65–72.

[27] F. Jokhio, A. Ashraf, S. Lafond, I. Porres, and J. Lilius, "Prediction-based dynamic resource allocation for video transcoding in cloud computing," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, 2013, pp. 254–261.

[28] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.