Service-Oriented Resource Management of Cloud Platforms

Xing Hu
Key Laboratory of High Confidence Software Technologies
Peking University
Beijing, China
huxing0101@pku.edu.cn

Rui Zhang
Key Laboratory of High Confidence Software Technologies
Peking University
Beijing, China
zhangrui12@sei.pku.edu.cn

Qianxiang Wang
Key Laboratory of High Confidence Software Technologies
Peking University
Beijing, China
wqx@pku.edu.cn

***Abstract*- How to deploy more services while keeping the Quality of Services is one of the key challenges faced by the resource management of cloud platforms, especially for PaaS. Existing approaches focus mainly on cloud platforms which mainly host small number of applications, and consider few features of different applications. In this paper, we present SORM, a Service-Oriented Resource Management mechanism on cloud platforms. The core of SORM is a service feature model which involves resource consumption and request variance of services. For each server, SORM deploys service instances with complementary resource consumption, so as to improve resource utilization. SORM also divides servers into three pools and deploys service instances onto different pools, mainly based on their request variance features, so as to reduce computational overhead of resource management and keep cloud platforms stable. We evaluate the effectiveness and efficiency of SORM by simulation experiments and find that: compared with one exiting approach. SORM can deploy 3.6 times more services with nearly 74.1% time cost.**

Keywords*- *service features, cloud computing, resource management, resource pool partition.

I. INTRODUCTION

Cloud computing platforms have been popular these years for its flexibility and convenience in providing different resources to the users. Such platforms, which are mainly categorized to IaaS (Infrastructure as a Service) and PaaS (Platform as a Service), can facilitate developers to deploy and manage their applications with less effort and thus to focus on the business logics of applications, and contribute to development with higher efficiency and quality.

Applications are usually deployed into Virtual Machines (VMs) in traditional IaaS cloud platforms. The Amazon EC2 [1], for example, allows users to deploy applications in VMs freely. Meanwhile, in recent years, a large number of applications are deployed on PaaS. For example, over 4 million applications have been created on Heroku which is the largest PaaS so far. However, hosting such large number of services and managing vast resources for them effectively is a great challenge for cloud platforms.

The servers in Twitter are managed by Mesos. [22] The CPU utilization is consistently below 20%. And the memory utilization is higher (40-50%) but still differs from the reserved capacity. Similarly, Google manages their servers by Borg [21], and Borg system consistently achieves CPU utilization of 20-35% and memory utilization of 40%. [23]

Fig.1 shows a typical deployment model for services on cloud platforms. A large number of heterogeneous services are deployed onto the servers in a cloud platform. These services have different features. Each service can run on multiple servers, and each server can host multiple services. They consume resources in different ways: different types of resources (CPU cycles, memory space, network bandwidth, etc., shown by different shapes and filled patterns in Fig. 1), and different amounts of resources (shown by different sizes in Fig.1). As the request rate to a service increases or decreases, it demands more or less resources accordingly. A typical way of adjusting resource supply for a service is to start or stop a number of instances for it, which lead to elastic resource management [6, 10].

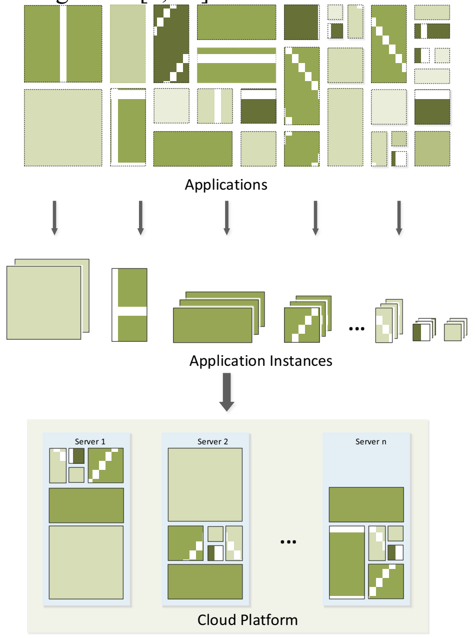

Fig. 1. Deployment model for applications on cloud platforms.

XIAO et al. [10] models resource management as a CCBP (Class Constrained Bin Packing) problem, which loads application instances into servers mainly based on their CPU usage. This approach can be implemented easily and achieve good effectiveness theoretically. However, it only considers applications' one or two types of resource as bottleneck resource and it is not easy to extend to more bottleneck resources. Therefore, it cannot achieve its best effect when applied to real cloud platforms with multiple types of resources as constraints when deploying applications.

In this paper, we present a Service-Oriented Resource Management mechanism (SORM) for cloud platforms. For services on cloud platform, we mean a huge number of services which are different in multiple aspects, e.g., programming language, resource consumption, request rate from users. To allocate resources effectively for these services, SORM acquires their resource demand characters (including resource demands on some request rate and how request rate changes), and divides servers into three pools. Based on the features of services, SORM deploys services with complementary features to each physical server, and deploys different service instances to different kinds of server pools.

The main contributions of this paper include:

- A three-pools based deployment mechanism, which can reduce computational overhead of resource management and keep the cloud platforms stable.
- A service-feature model which involves variance of request rate and resource consumption.
- A service oriented resource management mechanism, which can improve resource utilization.

The rest of the paper is organized as follows. Section 2 presents the architecture of our system. Section 3 formulates the resource management problem. Section 4 defines service features and illustrates how to acquire them. Section 5 describes the details of SORM. Experimental evaluation is presented in Sections 6. Section 7 describes the related work and Section 8 concludes this paper.

II. SYSTEM ARCHITECTURE

The architecture of our system is shown in Fig. 2. We partition servers in the whole cloud platform into three groups and each group is called a *Resource Pool*: Static Resource Pool, Weak Elastic Resource Pool and Strong Elastic Resource Pool. Partitioning resource pools and deploying the large number of service instances into these three pools and enforcing resource management for them respectively can reduce the overhead of resource management considerably and thus improve resource management efficiency. The instances of each service are deployed onto one or two of these resource pools, and their requests are forwarded by the *Dispatcher* to resource pools accordingly.

When a service instance is running, we acquire its resource usage information by the *Monitor* module, and also their past request information by the *Log Analyzer* module through analyzing service's access logs. Then the *Features Learner* module uses the information to learn service's features of resource consumption and request variance. Meanwhile, we predict the service's future request rate based on statistics of their past requests collected by the *Log Analyzer*. Combining resource usage information and predicted request rate, the *Scheduler* module makes resource provisioning decisions periodically based on service features learned from the *Features Learner*. Such decisions will lead to the following actions:

- Start new instances for some services which need more server resources.
- Stop existing instances for some services which need fewer server resources.

The decisions are then forwarded to *Resource Pools* to be enforced. The decision intervals of the *Resource Scheduler* are different in the three resource pools. The details are explained in the section 5.

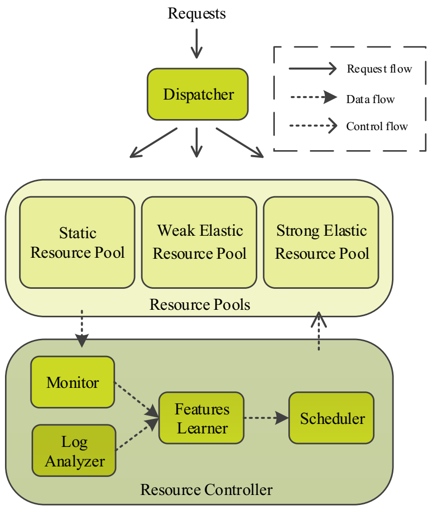

Fig. 2. Overview of the System Architecture

III. PROBLEM DEFINITION

In this section, we formulate the resource management problems in the *Scheduler* module in Fig. 2.

Most service instances consume multiple types of resources, such as CPU cycles, memory space, and network bandwidth. For convenience of illustration, in this paper we only deal with two types of resources: CPU and memory. Our approach can be extended to more types of resources easily.

If service instances are deployed without considering the features of resource consumption, the resources on some servers may be quite unbalanced: in some servers, CPU is exhausted by some services while few memory is used. Meanwhile, in some other servers, memory is exhausted by some services, while few CPU is used. Even the service feature is considered, if the service instances are not scheduled in a small enough granularity, the different types of resources may still be used unbalanced.

SORM introduces the concept of *Execution Unit,* which is similar to "page" concept in memory management mechanism of Operating System. *Execution Unit* is one kind of service instance, whose resource consumptions are fixed, and occupies less than half each of server resources.

The resource management problems are defined as follows. Suppose we have a fixed server set (denoted as S) on which we need to run services which come as a sequence (denoted as $A = \{a_1, a_2, \dots, a_n\}$) and the requests to them are changing. We assume that the servers are homogeneous with uniform capacity: the CPU capacity of a server is C, and the memory capacity is M.The CPU demand and the memory demand of each execution unit of service a ($a \in A$) are c_a andm_a. Let $D_{a,s}$ to be the service deployment matrix ($D_{a,s} = k$ means that service a has k execution units running on server$s$). We make resource provision decisions for services every decision interval or every time new services are coming. The inputs to *Scheduler* contain the current service deployment matrix $D_{a,s}$, the CPU and memory resource capacity of each server (C and M), and the required CPU and memory resource of each execution unit (c_a and m_a), which can be calculated from service features and their request rate. The outputs contain a new service deployment matrix and the number of services deployed (denoted as m). Our goal is to deploy more services while keeping the following constraints, which means that the total CPU and memory consumption of all execution units on a server cannot exceed its capacities of these resources:

$$\sum_{a \in A} c_a * D_{a,s} \leq C \forall s \in S, \tag{1}$$

$$\sum_{a \in A} m_a * D_{a,s} \leq M \forall s \in S. \tag{2}$$

IV. DEFINITION OF SERVICE FEATURES

A. *Definition of service Features*

Services deployed on cloud platforms are usually heterogeneous in resource demand because of their differences in user requirements, business logics and implementation details. An online report processing system, for example, mainly consumes servers' CPU cycles for reporting data computation. While a file storage system mainly consumes disk bandwidth and space. Services also have different features of resource demand and users' requests. For example, a tomcat-loaded web application's memory demand usually keeps stable, while its CPU demand and I/O demand change with users' requests. In other words, these services memory demand is rarely affected by its requests, while its CPU and I/O demands are closely related to how users' requests vary over time.

In this paper, we mainly consider two types of service features which are closely relevant to services' resource demand: 1) resource consumption, which describes how a service's demands for various resources change with users' requests; 2) request rate variance, which describes how the rate of users' requests change over time.

Service features can be acquired through various monitoring approaches, e.g., online monitor approaches [19] for resource consumption features, and log analyzing approaches for request variance features [11]. See section C for detail.

B. *Formal Description of Service Features*

1) *Formal Description of Resource Consumption Feature*

The resource consumption of some service usually tends to vary regularly, which we refer to as services' resource consumption features, donated as RC. Specifically, a service's RC is the demand for resources under some request rate: $RC = \{C_1(r), \ldots, C_n(r)\}$. In which *n* represents the number of resource types and $C_i(r)$(*1≤i≤n*) is one function of the *i*th resource on request rate r.

In this paper, to keep the issue simple, we assume that all servers of the cloud platform are homogeneous, which means they are composed of the same kinds of and the same number of resources. In these cases, the results will not be affected by the environment. We define a resource consumption vector for each server, denoted as *SR* (Server Resource consumption):SR $= \{R_1, \ldots, R_n\}$. In which R_i(1≤i≤n) represents the usage of the server's *i*th type of resource.

2) *Formal Description of Request Rate Variance*

Services' request rate variance features describe how their user request rate varies over time. We analyze the services' access logs and obtain the request variance features using Fourier transform.

We denote Request rate Variance features as RV. RV is expressed as a set of frequencies and amplitudes of request series' primary frequency components after Fourier transformation: $RV = \{(f_j, a_j, b_j) \mid j \in [1, m]\}$. In which m is the number of series' primary frequency components.

C. *Acquisition of Service Features*

1) *Acquisition of Resource Consumption Feature*

After a new service is deployed onto the Strong Elastic Resource Pool, we monitor it and record its usages of different kinds of resources. When its running becomes to stable, we obtain the function of the service's resource consumption with request rate by fitting method from the monitored resource usage information, thus obtaining resource consumption features for the service.

To facilitate the following resource management, we further identify a special value about services' resource consumption. As the request rate increases, a service's consumption for a certain type of resource would increase to a particular threshold (for example, 80%) prior to other types of resources. We define the service's request rate at this particular time as the *upper limit* for this service, denoted as R_{ul}. For instance, when the some service's request rate reaches 70 req/s, its CPU consumption reaches the threshold of 80%, while the consumptions of memory and I/O are still under the thresholds. Therefore, by definition, the sample service's R_{ul} is 70 req/s. R_{ul} provides a basis for identifying services' execution units, which we shall illustrate in Section 5.

2) *Acquisition of Request Variance Feature*

Log analysis is widely used for learning and mining features for applications [11], [12]. We acquire services' request varying features by analyzing their access logs generated during running process. For a service, we calculate its request rate and get a request-rate sequence for it firstly. Then we apply *Fourier* transformation to the sequence and get frequencies and amplitudes of the resulted Fourier series' primary frequency components, thus obtain the service's request variance feature as defined above.

A service's request variance feature reflects how the past request rate changes, based on which, we can predict services' future request rate and adjust resource allocation for them in advance. Note that characters of services' request rate have a substantial influence on the accuracy of the prediction. We can achieve a better prediction if the changes of request rate show stronger regularity.

V. RESOURCE MANAGEMENT BASED ON SERVICE FEATURE

We aim to ensure the quality of services deployed on cloud platforms and also to improve resource utilization and deploy more services on some limited number of servers. We adjust a service's resource allocation by increasing or decreasing its instances. In this paper, we present a resource pool partition based resource management mechanism. Also, to further improve the efficiency of elastic resource management, we design an *execution unit* based requests distribution approach, which divides large number of requests, and sends them to multiple execution units with fixed resource consumption.

In this section, we first describe the intuition and advantages of resource pool partition, then we give a detailed description of the two elastic resource pools.

A. Resource Pools

Managing the resources of cloud platform has to face with some challenges. First, it would be time costly to enforce resource management for thousands or even millions of services. Second, if vast services are deployed together in just one server cluster, it would be hard for the whole platform to keep stable. Third, instances that change quickly would affect those that change less quickly and reduce the effectiveness of resource management. Forth, some particular services require resource supply from the cloud platform on a long-term and stable basis. Therefore, it is necessary to provide enough instances for them and not to adjust their instances frequently.

To address these challenges, we partition the servers in the whole cloud platform into 3 resource pools: Static Resource Pool (SRP), Weak elastic Resource Pool (WERP) and Strong Elastic Resource Pool (SERP). Specifically, execution units of special services (for example, mission critical service) are deployed to SRP and respond to all the requests to these services. Execution units of regular services are deployed to WERP and SERP. Execution units in different resource pools respond to different parts of requests to the services.

As mentioned in Section 4, when we finish learning the features of services, we deploy their instances in the form of execution units onto particular resource pools based on their features. An execution unit is a logical instance whose requests processing capacity is limited by a maximal request rate and its resource consumption is fixed. We denote the maximal request rate as R_{max}. Execution units are implemented by adjusting the requests dispatcher to allocate the R_{max} number of requests in each requests distribution period. We acquire the value of R_{max} for a service's execution units by its R_{ul} mentioned in Section 4. $R_{max} = R_{ul}/k$, in which k is a positive integer, determined by the service's resource consumption features. Note that an execution unit is essentially a service instance with fixed resource consumption. Therefore, in the following sections we shall use indiscriminately the phrases of *service instances* and *execution units*).

In this paper we focus mainly on the elastic resource management and thus in the following sections we shall give a detailed description of these two elastic resource pools: WERP and SERP. Note that although SRP does not literally perform elastic resource management and we will not introduce it in depth, such a resource pool is a necessary complement for a real cloud platform.

We deploy execution units called weak elastic units for services on demand in WERP. They respond to fixed number of requests, to provide stable resource supply for services. We adjust number of execution units and locations in a coarse grained time basis (for example, adjust once per day) to improve the resource usage and the capacity of hosting services. As the interval for adjustment is relatively long, we can effectively reduce the number of adjustment actions and the overhead for resource management.

We also deploy execution units called strong elastic units for services on demand in SERP. They respond to frequently changing requests, to satisfy fluctuated resource demand. We adjust number of execution units and locations in real time to ensure the service quality for services. Because of the frequently fluctuated resource demand, we should make the adjustments in a fine-grained time basis (for example, adjust once per ten minutes). In other words, we enforce a strong elastic resource management in this resource pool, and thus we call it Strong Elastic Resource Pool.

By partitioning resource pool, we deploy services' execution units into different server clusters. In this way, we can achieve a better effectiveness of resource management in two aspects. On one hand, execution units that change with different frequencies are deployed separately in different resource pools and do not affect each other. On the other hand, dividing the large number of service instances into two groups and enforcing resource management for these two groups respectively can reduce the overhead of resource management considerably. We shall show the advantages of resource pools by simulation experiment in Section 6. The basic idea of partitioning resource pools is to divide the requests into two parts: one part with stable number of requests which are responded with execution units in weak elastic resource pool, and another part with frequently changed requests which are responded with execution units in strong elastic resource pool. As shown in Fig. 3, request rate is collected once every one hour for 48 times. We adjust the base line (horizontal lines) after 24 hours. Horizontal lines divide requests into two parts. Weak elastic resource pool is responsible for requests under the line, while strong elastic resource pool is responsible for the upper requests.

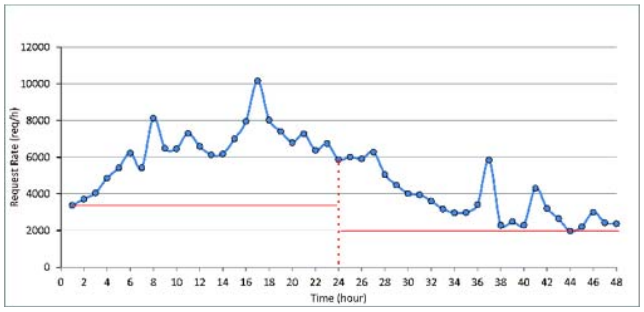

Fig. 3 Request rate of a sample service.

B. Weak Elastic Resource Pool

In Weak Elastic Resource Pool, we perform the following actions periodically: identify a stable request for every service, compute and adjust the numbers of services, and enforce collocation for weak elastic units.

1) Identify Stable Request Rate

Service execution units in Weak Elastic Resource Pool process services' stable amount of requests, which we define as *Stable Request Rate*, denoted as R_s. Ideally, R_s is service's lowest request rate in a period .

2) Compute and Adjust the Number of Weak Elastic Units

At the beginning of every weak elastic period, we compute the number of weak elastic units for every service based on its R_{max} and R_s(R_{max} means the maximal request rate of the execution unit's request capacity as mentioned above). We denote this number as N_w, then $N_w = \lceil R_s/R_{max} \rceil$.

We then adjust the numbers of weak elastic units to the corresponding N_w for every service. Specifically, assuming that a service's weak elastic units number now is N_c. If $N_c = N_w$, no adjustment is needed. If $N_c > N_w$, we stop $N_c - N_w$ units for it. If $N_c < N_w$, we start $N_w - N_c$ more units for it.

3) Collocate Weak Elastic Units

We collocate weak elastic units, i.e., deploy service execution units with different resource consumption features and request variance features together in servers. By this collocation operation, server resources can achieve a balanced usage and contributing the high utilization.

As the number of service execution units in Weak Elastic Resource Pool is large, it is costly to make adjustments for all of them. Therefore, we only do this for part of them, which we refer as execution units to be deployed, denoted as EU_t. As mentioned above, we add newly started service execution units into the set of EU_t. Also, we add exectution units of some CPU intensive services and memory intensive services into EU_t. We choose these two parts of service execution units because they are complementary in resource consumption, which means a better effectiveness when they are deployed together.

C. Strong Elastic Resource Pool

When the request rate to some service changed, we should adjust the number of execution units of that service. As the decision interval in WERP is relatively long and we cannot make timely adjustments in the resource pool, so we perform such adjustments in Strong Elastic Resource Pool, which are called Strong Elastic Actions. There are two types of strong elastic actions: *Scaling Up* and *Scaling Down*, which adds and reduces execution units services respectively.

The collocation for execution units in the set of EU_t is performed with the following algorithm. For each execution unit in EU_t:

a) Acquire resource consumption:
$RC_{R_{max}} = \{C_1(R_{max}), C_2(R_{max})\}$ (We focus only on the demanded CPU cycles and memory space here, which are denoted as C_1 and C_2 respectively).

b) List the SR for each server in Weak Elastic Resource Pool:
$SR = \{R_1, R_2\}$ (R_1 and R_2 represent the server's CPU consumption and memory consumption, respectively) and find all those that can host the execution unit to be deployed, which means $R_1 + C_1(R_{max}) \leq 80\%$ and $R_2 + C_2(R_{max}) \leq 80\%$ (as mentioned in Section 4, we set the threshold of consumption for each type of resource in a server as 80%).

c) For each server found by Step b, compute $\delta = |(R_1 + C_1(R_{max})) - (R_2 + C_2(R_{max}))|$ and select the one with the minimal δ to deploy the execution unit. If more than one server share the minimal δ, compute $\alpha = |(R_1 + C_1(R_{max})) + (R_2 + C_2(R_{max}))|$and select the one with the minimal α to deploy the execution unit.

1) Scaling Up

We monitor services' respond time in every strong elastic period. If a timeout occurs, which means the corresponding service cannot get enough resource to process its requests and some of the requests are delayed. We need to start new execution units for it. Meanwhile, we predict services' future request rate based on their past request information. If the predicted request rate of a service is too high to be processed by its execution units, we also need to add more units for it. For both of the cases mentioned above, we perform scaling up.

For the first case, the number of execution units that are to be added for a service is based on the number of its current weak elastic units: assuming the service has N weak elastic units, we first add N execution units for it; if the timeout still occurs for the service, we then add N/2 (at least one) execution units for it; repeat the above steps until there is no timeout.

For the second case, we can compute the number of execution units that are to be added by execution units' R_{max}. For example, a service has N execution units. Therefore, the request rate this service can handle at most is $N \times R_{max}$. If we predict the service's request rate is r ($r > N \times R_{max}$), we should add $\lceil r/R_{max} - N \rceil$ more execution units for the service.

Besides the number, we also need to identify proper locations for newly added EUs, which is, in essence, a collocation operation for them. The collocation algorithm just like the

Weak Elastic Units collocation algorithm mentioned above.

2) Scaling Down

Similar to scaling up, we make scaling down decisions to reduce execution units for services by monitoring and predicting future request rate.

We monitor the resource usage of services' strong elastic units in every period. If a strong elastic unit's CPU usage is 0 in m consecutive periods, which means it sits idle for a long time, we stop it if it is not the last strong elastic unit for the corresponding service.

Also, if the predicted request rate for a service is too low, we stop some of its execution units. For example, a service has N execution units. If we predict the service's request rate is r ($r \leq$ （$N-1$） $\times R_{max}$), we should stop$\lfloor N - r/R_{max} \rfloor$ more execution units for the service.

VI. EXPERIMENTAL EVALUATION

We evaluate the effectiveness and efficiency of SORM by simulation experiments. SORM is targeted to cloud platforms which host enormously many services and a large number of servers are required to build such a platform. Because building such a platform is too difficult, we choose to conduct a simulation experiment instead. Noted that our simulation takes such factors that influence experimental results into consideration. This ensures the fidelity of our simulation results.

A. Objective

We hope to evaluate the effectiveness and efficiency of our method and answer the following two questions:

1) Can more services be deployed into limited number of servers by our strong and weak elastic actions?
2) Can the management overhead be reduced by the resource pool partition?

For question 1), we compare the numbers of services between our SORM and existing approaches. For question 2), we compare the overheads between SORM and existing approaches.

B. Settings

We build a simulated experiment environment to evaluate our approach. We use a computer which has an Intel Core i7-3770 CPU, 4 GB memory and 1 TB disk. We also use such softwares as Oracle JDK 7.0.15 and Oracle JRE 7.0.400.43.

The data that we use in the experiment comes from NLANR [2], which includes real web services' access log.

We deploy a large number of services onto a cloud platform. We construct two kinds of services – memory intensive services and CPU intensive services, denoted as *Service-A* and *Service-B*, respectively. Services' request variance rate comes from the data set of NLANR that we denoted above. There are average three hundred thousand requests to every service in 24 hours. It corresponds to the request rate classic services on cloud platform.

Fig. 4 shows *Service-A*'s request rate. Its execution units' $R_{\max}$ is 11 req/s, CPU consumption is 3% and memory consumption is 10%. Fig. 5 shows *Service-B*'s request rate. Its execution units' $R_{\max}$ is 7 req/s, CPU consumption is 10% and memory consumption is 2%.

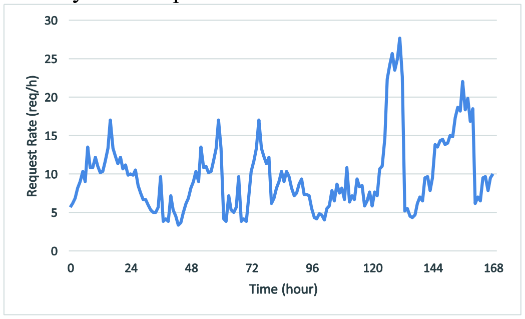

Fig. 4. Request rates of *Service-A*.

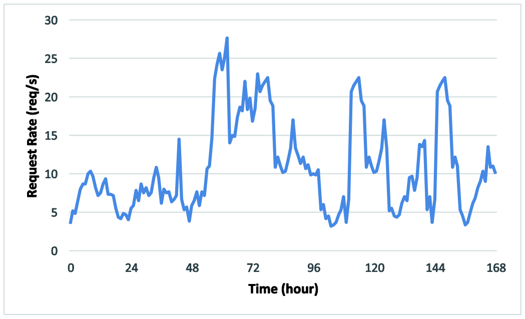

Fig. 5. Request rates of *Service-B*.

C. Results

To evaluate the effectiveness and efficiency of our method, we conduct the experiments and compare SORM with one approach (referred to as C approach in this paper) [10].

C approach developed a color set algorithm to decide the application placement and the load distribution. They made scaling decisions for applications mainly based on their CPU usage.

1) Effectiveness

We evaluate the effectiveness of resource management approaches by comparing the number of services' instances that different approaches can deploy on 100 simulated servers.

We deploy 10 instances of *Service-A* and 10 instances of *Service-B* firstly. The result is shown in the columns of *10:10* in Fig. 6, which illustrates the number of services deployed by different approaches. We can see that SORM can deploy 3060 services, 3.4 times as many services as C approach does (900). The reason behind this is simple: we take more types of resource into consideration to enforce elastic resource management and collocation for applications. As a result, the platform can achieve a higher resource utilization and more services can be deployed.

We further evaluate how different services influence the effectiveness of resource management, by varying the ratio between the instance numbers of *Service-A* and *Service-B,* e.g., 2:18, 6:14, 14:6, and 18:2. Results for all these 5 sets of experiments are shown in Fig. 6. The columns show the service number deployed by different approaches and the line shows the ratio between the numbers of services deployed by SORM and C.

We can see from Fig. 6 that service features indeed have influence for the effectiveness of resource management: better effectiveness can be achieved when services' resource usages are complementary. Also, Fig. 6 shows that SORM can always deploy more services than C and in average SOAM can deploy 3.6 times as many services as C can.

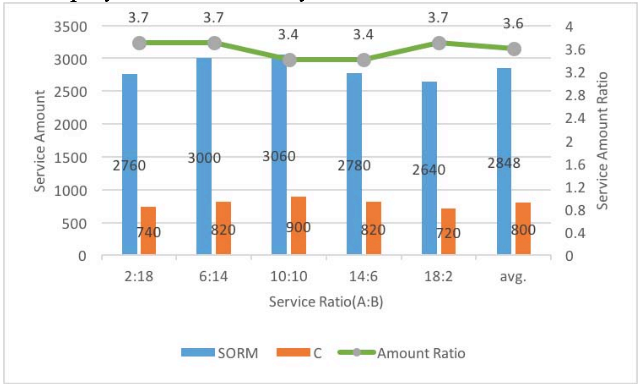

Fig. 6. Numbers of deployed services with various ratios by different resource management approaches.

2) *Efficiency*

To evaluate the efficiency of resource management approaches, we deploy *Service-A* and *Service-B* with different proportions to the cloud platform. We deploy instances of *Service-A* and *Service-B* in proportion every 10 minutes and compute the average overhead. Here we do not set a limit for the number of servers, so that we can deploy same number of services by different resource management approaches and compare their time overheads. During this process, we record the time that different approaches spend.

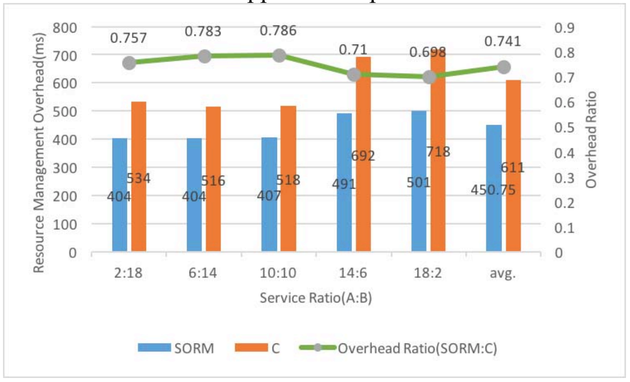

Fig. 7. Time overheads for different resource management approaches.

Fig. 7 shows the time ratio between SORM and C. The columns show the resource management overheads of different approaches and the line shows the ratio between the overheads of SORM and C. Fig.7 shows that SORM has lower overhead no matter what proportion between *Service-A* and *Service-B*. In average SORM costs 74.1% time of C. It is a great progress, in the millisecond level. This is mainly due to our resource pool partition policy, which reduces the numbers of service instances that need to be monitored. In this way, we reduce the computational overhead and thus reduce the time of deploying vast instances of services.

VII. RELATED WORK

The resource management problem has been extensively studied by many research works.

Some researchers modeled cloud platforms' resource management problems by such complicated algorithms as machine learning, control theoretic algorithms and etc. For example, Guo et al. [13] and Rao et al. [9] proposed a reinforcement learning approach that facilitated self-adaptive virtual machines resource provisioning, which was able to reach near-optimal resource provisioning decisions. However, such reinforcement learning approach had sufficiently high overhead when applied to cloud platforms with large number of applications hosted. Also, these approaches were relatively difficult to implement.

Researchers also presented relatively simple approaches to enforce resource management for cloud platforms. Some of them allocated resource for applications based on request rate prediction, as in [3], [4] and [5]. Gandhi et al. [14], for example, proposed a hybrid approach that proactively allocated resources for the predictable demand pattern of applications, which was power efficient and avoided unnecessary provisioning changes. In another work, Gandhi et al. [15] proposed to allocate power resource for servers hosting applications by an algorithm with both reaction and prediction.

Menarini et al. [16] presented SOPRA, a framework which dynamically predicted the resources required by the web service application, leveraging statistical data over web services usage patterns. Therefore, it could allocate resource in advance and improve energy efficiency of data centers. Gong et al. [17] applied signal processing technologies to acquire applications' resource usage features and predicted their future request rate by Markov Chain. Nguyen et al. [18] proposed AGILE, an application-agnostic, prediction-driven, distributed resource scaling system for cloud platforms, which could predict performance problems far enough in advance that they could be avoided.

Besides the above request-rate-based approaches, researchers also presented approaches that made resource provisioning decisions based on resource usage. Xiao et al. [10], for example, presented a color set algorithm that provided automatic scaling for applications in the cloud environment, which made scaling decisions for applications mainly based on their CPU usage. This approach does not apply to cloud platforms with multiple types of resources as constraints to applications' deployment.

Recent years, lightweight containers became more popular to support PaaS cloud platforms. For example, LXC (Linux Container) based on Docker is widely used as lightweight environment for applications. Google proposed Kubernetes [20] for automating deployment, operations, and scaling of containerized applications. It had similar design as Borg [21], which was a cluster management system to schedule, and monitor the applications that Google runs.

VIII. CONCLUSION

In this paper, we present SORM, a service-oriented resource management mechanism for cloud platforms. We first construct service feature model by combining monitored services' resource usage information and also their predicted request rate in the future. Then we partition servers into 3 resource pools, in which Static Resource Pool provides stable resource supply to services with special resource demand and Weak Elastic Resource Pool and Strong Elastic Resource Pool perform resource allocation tasks which change with different frequencies. Besides, for each server, we collocate services with different features together into it to improve resource utilization.

Our experiments show that compared with the exiting approaches, SORM can deploy more services with lower computational overhead.

In this work, we just consider the feature of services, but the cloud platforms are heterogeneous. In future work we will add the feature of cloud platforms to the feature model to support heterogeneous cloud platforms. In addition, in order to achieve a better resource management, we will try some other prediction approaches to acquire the request variance feature.

REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/. Accessed on July 10, 2015.

[2] NLANR(National Laboratory for Applied Network Research), Anonymized access logs. ftp://ftp.ircache.net/traces/, 1995.

[3] Islam S, Keung J, Lee K, Liu A. Empirical prediction models for adaptive resource provisioning in the cloud. Future Generation Computer Systems, 2012, 28(1): 155–162.

[4] Huang J, Li C, Yu J. Resource prediction based on double exponential smoothing in cloud computing. 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet). Yichang, China, 2012: 2056–2060.

[5] Iqbal W, Dailey M, Carrera D, Janecek P. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. Future Generation Computer Systems, 2011, 27(6): 871.

[6] Shen Z, Subbiah S, Gu X, Wilkes J. CloudScale: elastic resource scaling for multi-tenant cloud systems. Proc. of ACM Symposium on Cloud Computing (SOCC) in conjunction with SOSP 2011. Cascais, Portugal, 2011: 5.

[7] Han R, Guo L, Ghanem M, Guo Y. Lightweight resource scaling for cloud applications. 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). Ottawa, Canada, 2012: 644–651.

[8] Krioukov A, Mohan P, Alspaugh S. Napsac: Design and implementation of a power-proportional web cluster. ACM SIGCOMM computer communication review, 2011, 41(1): 102-108.

[9] Rao J, Bu X, Wang K, Xu C. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. 2011 IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS). Los Alamitos, CA, USA, 2011:45–54.

[10] Xiao Z, Chen Q, Luo H. Automatic scaling of internet applications for cloud computing services. IEEE Transactions on Computers, 2014, 63(5): 1111-1123.

[11] Xu W, Huang L, Fox A. Detecting large-scale system problems by mining console logs. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM. Big Sky, MT, USA, 2009: 117-132.

[12] Xu W, Huang L, Fox A. Online system problem detection by mining patterns of console logs. 9th IEEE International Conference on Data Mining, 2009, IEEE. Miami, FL, USA, 2009: 588-597.

[13] Guo Y, Lama P, Rao J, Zhou X. V-Cache: towards flexible resource provisioning for multi-tier applications in IaaS clouds. 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS). Boston, MA, USA, 2013: 88-99.

[14] Gandhi A, Chen Y, Gmach D. Minimizing data center sla violations and power consumption via hybrid resource provisioning. 2011 International Green Computing Conference and Workshops (IGCC), IEEE. Orlando, FL, USA, 2011: 1-8.

[15] Gandhi A, Chen Y, Gmach D. Optimal power allocation in server farms. ACM SIGMETRICS Performance Evaluation Review, ACM, 2009, 37(1): 157-168.

[16] Menarini M, Seracini F, Zhang X. Green web services: Improving energy efficiency in data centers via workload predictions. 2nd International Workshop on Green and Sustainable Software (GREENS), IEEE. San Francisco, CA, USA, 2013: 8-15.

[17] Gong Z, Gu X, Wilkes J. PRESS: predictive elastic resource scaling for cloud systems. 6th International Conference on Network and Service Management (CNSM). Niagara Falls, Canada, 2010: 9-16.

[18] Nguyen H, Shen Z, Gu X. AGILE: elastic distributed resource scaling for Infrastructure-as-a-Service. Proc. of the USENIX International Conference on Automated Computing (ICAC'13). San Jose, USA, 2013: 69-82.

[19] Shao J, Wang Q, Mei H. Model based monitoring and controlling for Platform-as-a-Service (PaaS). International Journal of Cloud Applications and Computing (IJCAC), 2012, 2(1): 1-15.

[20] Kubernetes. http://kubernetes.io. Accessed on May 6, 2016.

[21] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu David Oppenheimer, Eric Tune, John Wilkes: Large-scale Cluster Management at Google with Borg. In EuroSys'15, April 21–24, 2015, Bordeaux, France.

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI). Boston, MA, 2011

[23] Christina Delimitrou, Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.