2012 IEEE 4th International Conference on Cloud Computing Technology and Science

CRAMP: Cost-Efficient Resource Allocation for Multiple Web Applications with Proactive Scaling

Adnan Ashraf*†‡, Benjamin Byholm*, Ivan Porres*†
*Department of Information Technologies, Åbo Akademi University, Turku, Finland.
Email: adnan.ashraf@abo.fi, benjamin.byholm@abo.fi, ivan.porres@abo.fi
†Turku Centre for Computer Science (TUCS), Turku, Finland.
‡Department of Software Engineering, International Islamic University, Islamabad, Pakistan.

***Abstract*—This paper presents a prediction-based dynamic resource allocation approach for web applications called CRAMP (Cost-efficient Resource Allocation for Multiple web applications with Proactive scaling). The proposed approach provides automatic deployment and proactive scaling of multiple simultaneous web applications on a given Infrastructure as a Service cloud in a shared hosting environment. It monitors and uses resource utilization metrics and does not require a performance model of the applications or the infrastructure dynamics. The shared hosting environment allows us to share virtual machine (VM) resources among deployed applications, reducing the number of required VMs. The approach is demonstrated in a prototype implementation that has been deployed in the Amazon Elastic Compute Cloud.**

***Keywords*-Cloud computing; resource allocation; web applications; application server; proactive scaling; quality of service**

I. Introduction

Web applications are often deployed in a three-tier computer architecture, where the application server tier often uses a computer cluster to be able to process many user requests simultaneously. 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) on demand. IaaS offerings can be used to create a dynamically scalable server tier consisting of a varying number of VMs.

Determining the number of VMs to provision for a 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 that are specified in the Service Level Agreements (SLAs). Under-allocation leads to subpar service, while over-allocation results in increased operation costs. There are several research works [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] as well as some vendor-specific commercial solutions, such as AWS Elastic Beanstalk [12], which provide dynamic resource allocation. However, a common characteristic of these solutions is that they use dedicated hosting [13], where one VM hosts only one application. This is a reasonable approach if an application has enough user load to keep at least one VM sufficiently utilized. But in many cases, dedicated hosting may introduce unnecessary overhead and cost due to underutilization of servers. A recent study showed that $80\%-85\%$ underutilization is common in enterprises [14].

The underutilization of VMs becomes even more pertinent when deploying a large number of web applications of varying resource needs. The solution to this problem is to design a dynamically scalable application server tier that manages multiple applications simultaneously, while using shared hosting [13] to deploy multiple applications on a VM [15]. There are two possible approaches, namely reactive and proactive resource allocation. The proactive approach often makes better resource allocation decisions by using a prediction of the future load to act preemptively [2].

In this paper, we present a prediction-based dynamic resource allocation approach for web applications called CRAMP (Cost-efficient Resource Allocation for Multiple web applications with Proactive scaling). It provides proactive resource allocation for the application server tier, while simultaneously supporting dynamic scaling of multiple web applications on a given IaaS cloud in a shared hosting environment. In a broader context of ARVUE PaaS (Platform as a Service) [16], CRAMP provides a dynamic resource allocation approach for PaaS clouds.

The rest of the paper is organized as follows. Section II outlines the main tasks and the most important characteristics of the CRAMP approach. Section III presents the system architecture. CRAMP algorithms are described in Section IV. Section V presents our prototype implementation and experimental results. Section VI discusses important related works and Section VII concludes the paper.

II. CRAMP Approach

The main tasks of the proposed approach are to provision and remove VMs for the application server tier and to deploy and undeploy applications from each VM. The most important characteristics of the CRAMP approach are that it is based on proactive control, it does not depend upon a performance model of the application or the infrastructure dynamics, it supports shared hosting, and it provides dynamic resource allocation for the application server tier along with dynamic scaling of multiple web applications.

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

IEEE computer society

A. Reactive and Proactive Resource Allocation

Many traditional resource allocation approaches, such as [4], [5], [6], [7], use reactive resource allocation, which monitors user load in the system and reacts to it. However, the primary shortcoming of the reactive approach is that it starts a resource allocation operation only after a significant increase in the load is detected [2]. Therefore, the new VMs can only be instantly used if the VM provisioning is instantaneous. However, in practice, VM provisioning takes a few minutes. Thus, the reactive approach may fail to ensure QoS due to the VM provisioning delay.

It is possible to develop better resource allocation algorithms by using a prediction of the future user load to act proactively [2], [3]. For proactive scaling, CRAMP uses a two-step load prediction method [17]. Moreover, instead of relying only on proactive scaling, CRAMP implements a hybrid approach that assigns certain weights to the measured and the predicted load.

B. Shared Hosting

For cost-efficiency, the proposed approach provides a finer deployment granularity than the smallest VM provided by the contemporary IaaS providers. Instead of adding or removing one full VM for a particular application, CRAMP effectively supports adding or removing a fraction of a VM for an application. This is especially important when running a large number of web applications, most of which may have very few users at a given time, while a few of them may have many users. Thus, CRAMP shares VM resources by supporting shared hosting. Consequently, fewer VMs are used to run several web applications and therefore unnecessary costs are avoided without compromising QoS.

C. QoS in Terms of Resource Utilization Metrics

QoS is often defined in terms of the commonly used software performance metrics, such as the response time and the throughput [18]. However, an application may have different expected response times for different request types. Therefore, the expected response time, and similarly the throughput, may be difficult to define for an application. Thus, we define QoS in terms of server resource utilization metrics. The rationale is that when the resource utilization of the bottleneck server resource exceeds its upper limit, the server becomes overloaded. An overloaded server fails to maintain its performance, which translates into subpar service (higher response time and lower throughput) [19]. Thus, in order to ensure a sustainable QoS, CRAMP strives to maintain the individual resource utilizations below their respective upper limits.

The proposed resource allocation approach uses two utilization metrics: CPU load average and memory utilization. Load average represents the average CPU load over a period of time. It is computed as the exponentially weighted moving average of running (on CPU) and runnable (waiting for CPU) processes on all cores. Memory utilization represents the amount of used memory in proportion to the amount of total memory.

III. System Architecture

The CRAMP dynamic resource allocation approach distributes end user sessions for web applications to a scalable application server tier, which is proactively scaled up and down according to user load. The system architecture consists of a number of components, as shown in Figure 1. The *application server*, *local controller*, *application repository*, *global controller and resource allocator*, *cloud provisioner*, and *HTTP load balancer* are described in detail in [15] and [16].

The *predictor* predicts future load on an *application server*. It uses current and past resource utilization data of the server to predict a few steps ahead in the future. Our load prediction approach is detailed in [20]. It consists of a load tracker and a load predictor [17]. CRAMP uses exponential moving average (EMA) for the load tracker and a simple linear regression model [21] for the load predictor.

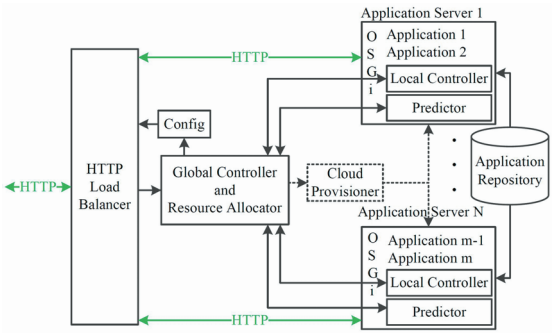

Fig. 1. CRAMP system architecture

IV. CRAMP Algorithms

The CRAMP approach provides an algorithm for proactive resource allocation of the application server tier and an algorithm for dynamic scaling of web applications. The second algorithm is similar to the application-level scaling algorithm in [15]. Therefore, we only present the first algorithm here.

The resource allocation decisions are based on the states of the application servers. The states are determined by comparing load average and memory utilization with their upper and lower threshold values. These threshold values depend upon a tradeoff between QoS and cost. The states are: *overloaded*, *non-overloaded*, *underutilized*, and *long-term underutilized*.

The algorithm maintains a fixed minimum number of servers representing the base capacity N_B. Moreover, to ensure performance under sudden peak loads, it uses a small number of additional servers. The number of additional servers $N_A(k)$ is determined based on the number of total servers $|S(k)|$, number of overloaded servers $|S_o(k)|$, and aggressiveness parameter for additional VM capacity $A_A \in [0, 1]$ [15].

The algorithm is designed to prevent oscillations in the number of provisioned VMs [4]. This is desirable due to the inevitable VM provisioning delay, as explained in Section II-A, which may lead to deteriorated performance. Moreover, since some IaaS providers, such as Amazon EC2, charge on an

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

IEEE computer society

hourly basis, oscillations will result in a higher provisioning cost. Therefore, the algorithm counteracts oscillations by delaying new scaling operations until previous scaling operations have been realized [7]. Furthermore, it terminates only those VMs that have been constantly underutilized for a certain amount of time.

Instead of provisioning or terminating one VM at a time, the algorithm provisions or terminates one or more VMs based on the states of the existing servers. The number of VMs to provision $N_P(k)$ is based on the proportional factor $P_P(k)$ and the derivative factor $D_P(k)$

$$N_P(k) = \lceil P_P(k) + D_P(k) \rceil \tag{1}$$

The proportional factor for VM provisioning $P_P(k)$ is calculated as the product of the number of overloaded servers $|S_o(k)|$ and the aggressiveness of control for VM provisioning A_P, $P_P(k) = |S_o(k)| \cdot A_P$, where $A_P \in [0, 1]$ is a real number. For example, $A_P = 1$ suggests to provision as many new as overloaded servers. The derivative factor for VM provisioning $D_P(k)$ is computed as $D_P(k) = |S_o(k)| - |S_o(k-1)|$.

Likewise, the number of VMs to terminate $N_T(k)$ is based on the proportional factor $P_T(k)$ and the derivative factor $D_T(k)$

$$N_T(k) = \lceil P_T(k) + D_T(k) \rceil - N_B - N_A(k) \tag{2}$$

where the proportional factor for termination $P_T(k)$ is the product of the number of long-term underutilized servers $|S_{lu}(k)|$ and the aggressiveness of control for VM termination A_T, $P_T(k) = |S_{lu}(k)| \cdot A_T$, where $A_T \in [0, 1]$ is similar to A_P. The derivative factor for VM termination $D_T(k)$ is computed as $D_T(k) = |S_{lu}(k)| - |S_{lu}(k-1)|$.

Unlike a standard proportional-derivative (PD) controller [22], the proportional and derivative factors in CRAMP do not depend on a performance model of the application or the infrastructure dynamics, and support proactive resource allocation for the application server tier with dynamic scaling of web applications in a shared hosting environment, as described in Section II-B. The resource allocation based on the proportional and derivative factors enables handling of sudden peak loads, since provisioning one VM at a time may be too slow due to the VM provisioning delay. Likewise, in deallocation, it saves VM provisioning costs, because terminating one VM at a time may result in unwanted, longer provisioning periods.

The resource allocation algorithm for the application server tier is given as Algorithm 1. For the sake of clarity, the concepts used in the algorithm and their notation are summarized in Table I. Although the algorithm uses load average and memory utilization metrics, we omit memory utilization for brevity. The first step (line 2) deals with the monitoring of the server resource utilizations and using them to predict resource utilizations a few steps ahead in the future.

As described in Section II-A, the proposed hybrid approach assigns certain weights to the measured and the predicted utilizations. Therefore, the algorithm computes weighted load average by using the weighting coefficient for CPU load average $w_c \in [0, 1]$, which is calculated based on the prediction error in the average predicted load average of all servers. We use the normalized root-mean-square error (NRMSE) [23]

$$NRMSE = \frac{\sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}}{max - min} \tag{3}$$

where y_j is the measured utilization, $\hat{y}_j$ is the predicted utilization, n is the number of observations, and max is the maximum value of both measured and predicted utilizations, while min is analogous to max. The weighted load average $C_w(s, k)$ is used to determine the state of each server.

The algorithm then makes a set of overloaded servers $S_o(k)$ (line 3) and a set of non-overloaded servers $NS_o(k)$ (line 5). From the set of overloaded servers, it makes a set of overloaded applications $A_o(k)$ (line 4). If it finds at least one overloaded and at least one non-overloaded server, it deploys each overloaded application on another existing server according to the application-to-server allocation policy (line 8). However, if it finds that all servers $S(k)$ except the additional VM capacity $N_A(k)$ are overloaded, the algorithm allocates resources to the application server tier by provisioning one or more servers according to $N_P(k)$ (line 12) and then deploys each overloaded application on each new server (line 16).

For deallocation, the algorithm first determines the state of each server based on the weighted load average $C_w(s, k)$. It then makes a set of underutilized servers $S_u(k)$ (line 19),

TABLE I
SUMMARY OF CONCEPTS AND THEIR NOTATION

$A_o(k)$	set of overloaded applications at k
$NS_o(k)$	set of non-overloaded servers at k
$S(k)$	set of servers at k
$S_{lu}(k)$	set of long-term underutilized servers at k
$S_n(k)$	set of new servers at k
$S_o(k)$	set of overloaded servers at k
$S_t(k)$	set of servers selected for termination at k
$S_u(k)$	set of underutilized servers at k
$C(s, k)$	measured load average of server s at k
$\hat{C}(s, k)$	predicted load average of server s at k
$C_w(s, k)$	weighted load average of server s at k
w_c	weighting coefficient for CPU load average
$N_A(k)$	number of additional servers at k
N_B	number of servers to use as base capacity
$N_P(k)$	number of servers to provision at k
$N_T(k)$	number of servers to terminate at k
C_L	server load average lower threshold
C_U	server load average upper threshold
UC_T	underutilization count threshold for a server
$d\_a(s, k)$	applications deployed on server s at k
$delay()$	delay function
$deploy(a)$	deploy application a as per allocation policy
$deploy(a, S)$	deploy application a on a set of servers S
$migrate(S)$	migrate user sessions for servers S
$provision(n)$	provision n servers
$select(n)$	select n servers for termination
$sort(S)$	sort servers S on server utilization metrics
$terminate(S)$	terminate servers S
$under\_u\_c(s)$	underutilization count of server s
$startUp$	server startup time

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

IEEE computer society

Algorithm 1 Resource allocation and deallocation

1: **while true do**
2: $\forall s \epsilon S(k) | C_w(s,k) := w_c \cdot C(s,k) + (1 - w_c) \cdot \hat{C}(s,k)$
3: $S_o(k) := \{\forall s \epsilon S(k) | C_w(s,k) \geq C_U\}$
4: $A_o(k) := \bigcup_{s \epsilon S_o(k)} d\_a(s,k)$
5: $NS_o(k) := S(k) \setminus S_o(k)$
6: **if** $|S_o(k)| \geq 1 \wedge |NS_o(k)| \geq 1$ **then**
7: **for** $a \epsilon A_o(k)$ **do**
8: $deploy(a)$
9: **end for**
10: **end if**
11: **if** $|S_o(k)| \geq (|S(k)| - N_A(k)) \wedge N_P(k) \geq 1$ **then**
12: $S_n(k) := provision(N_P(k))$
13: $S(k) := S(k) \cup S_n(k)$
14: $delay(startUp)$
15: **for** $a \epsilon A_o(k)$ **do**
16: $deploy(a, S_n(k))$
17: **end for**
18: **end if**
19: $S_u(k) := \{\forall s \epsilon S(k) | C_w(s,k) \leq C_L\}$
20: $S_{lu}(k) := \{\forall s \epsilon S_u(k) | under\_u\_c(s) \geq UC_T\}$
21: **if** $(|S_{lu}(k)| - N_B - N_A(k)) \geq 1 \wedge N_T(k) \geq 1$ **then**
22: $sort(S_{lu}(k))$
23: $S_t(k) := select(N_T(k))$
24: $migrate(S_t(k))$
25: $S(k) := S(k) \setminus S_t(k)$
26: $terminate(S_t(k))$
27: **end if**
28: **end while**

from which a set of long-term underutilized servers $S_{lu}(k)$ is obtained (line 20). Since the aim of the algorithm is to reduce VM provisioning cost, which is a function of number of VMs and time, it terminates any redundant VMs as soon as it is appropriate. Therefore, when the algorithm finds at least one long-term underutilized server, while excluding the base capacity N_B and the additional capacity $N_A(k)$, it sorts them in an increasing order based on their weighted resource utilizations (line 22) and selects one or more servers according to $N_T(k)$ (line 23). The rationale of the sorting is to ensure that the least loaded servers are selected for termination, which also aids in reducing the number of migrations.

Since the servers selected for termination might still be running a small number of active user sessions on them, the next step (line 24) ensures that the termination of the selected servers does not abandon any active sessions that were currently running on said servers. This is achieved by migrating all active sessions from the selected servers to other existing servers. Finally, the selected servers are terminated and removed from the application server tier (line 26).

V. Prototype and Experimental Results

The prototype is implemented in Java and has been deployed on the Amazon EC2 cloud. It has initially been tested for Java Servlet-based web applications [24]. When running multiple web applications on a Java application server, all web applications are usually placed in the same Java Virtual Machine (JVM). However, Java lacks some important features needed to safely run multiple third-party web applications in one JVM. This is partly addressed by a widely adapted OSGi specification [25], which is extended for use in the security layer of the ARVUE PaaS [16]. There are several open-source as well as commercial implementations of the OSGi specification. Among the open-source implementations, the free, open-source Apache Felix [26] is certified to be compliant with the OSGi specification. Therefore, we used Felix for our prototype implementation.

A. Experimental Design and Setup

The objective of the experiment was to demonstrate the most important characteristics of the CRAMP approach, while emphasizing its cost-efficiency. Moreover, for a comparison of results with the alternative approaches that use reactive control, we repeated the same experiment with the reactive approach in [15], here referred to as the *alternative* approach.

The ARVUE PaaS [16] is currently under development and therefore real load traces are not yet available. Hence, we decided to generate and use synthetic load in the experiment, which was designed to generate a load representing a maximum of 1000 simultaneous user sessions. The sessions were ramped up from 0 to 1000 at a discrete rate of 25 new sessions per 10 seconds. After the ramp up phase, the number of sessions was maintained constant and then reduced at a rate of -25 sessions per 10 seconds. Each session was randomly assigned to one particular web application. The experiment used 10 web applications of varying resource needs. Application 1 was the most popular, having 50% of total user sessions, application 2 had 25% sessions, application 3 had 20% sessions, while the other 7 applications shared the remaining 5% sessions. Session duration was 15 minutes. Each session continuously generated HTTP requests on its web application. Each request was designed to cause the application server to do some work requiring up to 10 milliseconds of a dedicated CPU. User think time between consecutive requests varied between 0 seconds and 20 seconds. All random values were uniformly distributed. The ARUVE sampling period k was 10 seconds. The upper threshold for server load average C_U was 0.7 and for memory utilization M_U was 0.8, which are considered reasonable safety margins [27]. Likewise, the lower threshold for load average C_L was 0.2 and for memory utilization M_L was 0.3. The experiment used small (*m1.small*) instances from the Amazon EC2 cloud. The application-to-server and session-to-server allocation policies were set to *lowest load average* [15]. The load tracker and load predictor parameters were: $n = 20$, $q = 15$, and $h = 180$, taking the VM provisioning delay into account [20].

B. Results and Analysis

Now we compare the results of the CRAMP approach with that of the *alternative* approach. In order to highlight differences between the results of the two approaches while

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

IEEE computer society

using the same scale-size, we use a log-scale for response time, load average, and memory utilization plots. Figure 2 presents CRAMP results. The maximum average response time was 231 milliseconds, while the memory utilization was always less than 1.0. However, the maximum load average was 1.33. This was due to the lack of admission control [20], which led to the temporary overloading of some existing servers during the VM provisioning operations. The overloading of servers can also be mitigated based on the aggressiveness parameter for additional VM capacity A_A, which represents a tradeoff between cost and QoS [15]. The results also indicate that using additional VM capacity is a suitable solution for avoiding heavy overloading of the application servers due to the VM provisioning delay. The load average plot shows that the application server tier was able to quickly recover from the temporary overloaded state. The shared hosting of applications resulted in a reduced total number of VMs and improved utilization thereof. CRAMP used 5 VMs for running 10 applications, while dedicated hosting would have required a minimum of 10 VMs. Thus, under these circumstances, CRAMP operated at significantly less cost than the dedicated hosting approaches. It is also important to note here that the algorithm did not produce oscillations in the number of VMs. As explained in Section IV, CRAMP is designed to counteract such oscillations.

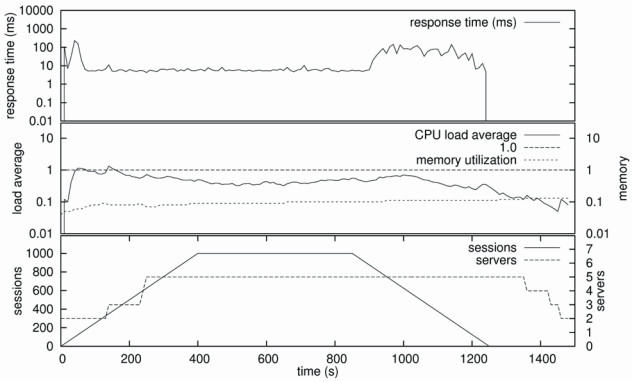

Fig. 2. Experimental results with CRAMP

The results of the *alternative* approach are shown in Figure 3. The *alternative* approach also implements shared hosting and therefore it also used 5 VMs for running 10 applications. However, due to the lack of proactive control, the maximum average response time was 2707 milliseconds and the maximum load average was 4.38. Thus, the experimental results demonstrate that CRAMP provides significantly improved QoS in terms of average response time and server resource utilizations.

VI. Related Work

Most of the existing works on dynamic resource allocation for web-based systems can be classified into two main categories: Plan-based approaches and control theoretic approaches [9], [10], [11]. Plan-based approaches can be further

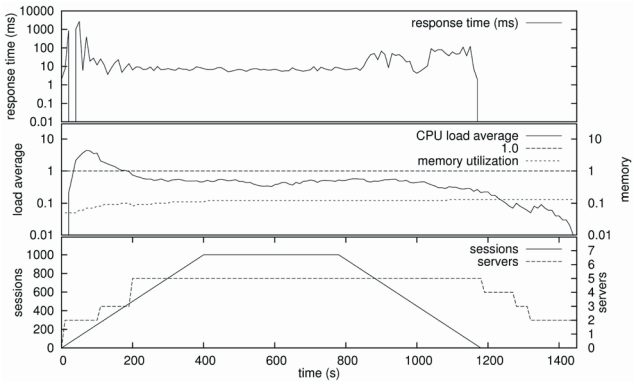

Fig. 3. Experimental results with the *alternative* approach

classified into workload prediction approaches [2], [3] and performance dynamics model approaches [4], [5], [6], [7], [8]. One common difference, between all existing works discussed here except [15] and CRAMP, is that CRAMP uses shared hosting. Another distinguishing characteristic of CRAMP is that in addition to proactive resource allocation for the application server tier, it also provides dynamic scaling of multiple web applications. In [15], we used shared hosting with reactive resource allocation. In contrast, CRAMP provides improved QoS with proactive resource allocation.

Ardagna et al. [3] proposed a distributed algorithm for managing Software as a Service (SaaS) cloud systems that addresses capacity allocation for multiple heterogenous applications. Raivio et al. [2] used proactive resource allocation for short message services in hybrid clouds. The main drawback of their approach is that it assumes server processing capacity in terms of messages per second, which is not a realistic assumption for HTTP traffic where different types of requests may require different amounts of processing time. Nevertheless, the main challenge in the prediction-based approaches is in making good prediction models that could ensure high prediction accuracy with low computational cost. CRAMP uses a two-step prediction method with EMA, which provides high prediction accuracy under real-time constraints. Moreover, it implements a hybrid approach that gives more or less weight to the predicted utilizations based on the NRMSE.

TwoSpot [4] supports hosting of multiple web applications, which are automatically scaled up and down in a dedicated hosting environment. The scaling down is decentralized, which may lead to severe random drops in performance. Hu et al. [5] presented an algorithm for determining the minimum number of required servers, based on the expected arrival rate, service rate, and SLA. In contrast, CRAMP does not require knowledge about the infrastructure or performance dynamics. Chieu et al. [6] presented an approach that scales servers for a particular web application based on the number of active user sessions. However, the main challenge is in determining suitable threshold values on the number of user sessions. Iqbal et al. [7] proposed an approach for multi-tier web applications, which uses response time and CPU

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

IEEE computer society

utilization metrics to determine the bottleneck tier and then scales it by provisioning a new VM. Han et al. [8] proposed a reactive resource allocation approach to integrate VM-level scaling with a more fine-grained resource-level scaling. In contrast, CRAMP supports proactive resource allocation with proportional and derivative factors to determine the number of VMs to provision.

Dutreilh et al. [9] used control theoretic models to design resource allocation solutions for cloud computing. Pan et al. [10] used proportional-integral (PI) controllers to provide QoS guarantees. Patikirikorala et al. [11] proposed a multi-model framework for implementing self-managing control systems for QoS management. In contrast to the control theoretic approaches, CRAMP also uses proportional and derivative factors, but it does not require knowledge about the performance models or infrastructure dynamics.

VII. Conclusions

In this paper, we presented a cost-efficient resource allocation approach called CRAMP for a dynamically scalable application server tier to deploy and scale multiple web applications on a given IaaS cloud. We also presented a prototype implementation and experimental results using the Amazon EC2 cloud. The results indicate that based on the user load on application servers, the CRAMP approach maintains a suitable number of VMs so that the over and under provisioning of VMs can be lessened. It uses shared hosting of applications, which reduces the number of required VMs. For ensuring performance under highly varying loads, CRAMP provides proactive scaling that uses a prediction of the future load to act preemptively. The experimental evaluation compared CRAMP against an existing approach that uses reactive scaling. The results showed that CRAMP provides significantly improved QoS in terms of average response time and server resource utilizations.

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 a doctoral scholarship from the Higher Education Commission (HEC) of Pakistan.

References

[1] "Amazon Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2/

[2] 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*, ser. CLOSER '12, 2012.

[3] 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.

[4] 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.

[5] 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*, ser. CASCON '09. New York, NY, USA: ACM, 2009, pp. 101–111.

[6] 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.

[7] 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.

[8] 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.

[9] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From data center resource allocation to control theory and back," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, july 2010, pp. 410 –417.

[10] W. Pan, D. Mu, H. Wu, and L. Yao, "Feedback control-based QoS guarantees in web application servers," in *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, sept. 2008, pp. 328 –334.

[11] 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*, ser. SEAMS '11. New York, NY, USA: ACM, 2011, pp. 218–227.

[12] "AWS Elastic Beanstalk." [Online]. Available: http://aws.amazon.com/elasticbeanstalk/

[13] 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. 1:1–1:45, Feb. 2009.

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

[15] A. Ashraf, B. Byholm, J. Lehtinen, and I. Porres, "Feedback control algorithms to deploy and scale multiple web applications per virtual machine," *38th Euromicro Conference on Software Engineering and Advanced Applications*, 2012.

[16] T. Aho, A. Ashraf, M. Englund, J. Katajamäki, J. Koskinen, J. Lautamäki, A. Nieminen, I. Porres, and I. Turunen, "Designing IDE as a service," *Communications of Cloud Software*, vol. 1, no. 1, December 2011.

[17] M. Andreolini and S. Casolari, "Load prediction models in web-based systems," in *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, ser. valuetools '06. New York, NY, USA: ACM, 2006.

[18] H. Liu, *Software Performance and Scalability: A Quantitative Approach*, ser. Quantitative Software Engineering Series. John Wiley & Sons, 2011.

[19] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguade, "Characterizing secure dynamic web applications scalability," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, april 2005, p. 108a.

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

[21] D. Montgomery, E. Peck, and G. Vining, *Introduction to Linear Regression Analysis*, ser. Wiley Series in Probability and Statistics. John Wiley & Sons, 2012.

[22] J. Hellerstein, *Feedback control of computing systems*, ser. Wiley interscience publication. Wiley-Interscience, 2004.

[23] P. Brandimarte and G. Zotteri, *Introduction to Distribution Logistics*, ser. Statistics in Practice. Wiley-Interscience, 2007.

[24] M. Grönroos, *Book of Vaadin*, 4th ed. Vaadin Ltd., Turku, Finland, 2011. [Online]. Available: http://vaadin.com/book

[25] The OSGi Alliance. OSGi service platform: Core specification, 2009, release 4, version 4.2.

[26] OSGi Felix. [Online]. Available: http://felix.apache.org/site/index.html

[27] J. Allspaw, *The art of capacity planning*, ser. O'Reilly Series. O'Reilly, 2008.

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

IEEE computer society