Robust Performance Control for Web Applications in the Cloud

Hector Fernandez, Corina Stratan, Guillaume Pierre

Robust Performance Control for Web Applications in the Cloud

Hector Fernandez[1], Corina Stratan[1] and Guillaume Pierre[2]
[1]*Vrije Universiteit Amsterdam, The Netherlands*
[2]*IRISA / University of Rennes 1, France*
{*hector.fernandez, corina.stratan*}*@vu.nl, guillaume.pierre@irisa.fr*

Keywords: Cloud computing, PaaS, Resource provisioning, Web applications.

Abstract: With the success of Cloud computing, more and more websites have been moved to cloud platforms. The elasticity and high availability of cloud solutions are attractive features for hosting web applications. In particular, the *elasticity* is supported through trigger-based provisioning systems that dynamically add/release resources when certain conditions are met. However, when dealing with websites, this operation becomes more problematic, as the workload demand fluctuates following an irregular pattern. An excessive reactiveness turns these systems into imprecise and wasteful in terms of SLA fulfillment and resource consumption. In this paper, we propose three different provisioning techniques that expose the limitations of traditional systems, and overcomes their drawbacks without overly increasing complexity. Our experiments conducted on both public and private infrastructures show significant reductions in SLA violations while offering performance stability.

1 INTRODUCTION

One of the major innovations provided by Cloud computing platforms is their pay-per-use model where clients pay only for the resources they actually use. This business model is particularly favorable for application domains where workloads vary widely over time, such as the domain of Web application hosting. Web applications in the cloud can request and release resources at any time according to their needs.

However, provisioning the right volume of resources for a Web application is not a simple task. Web applications are usually composed of multiple types of components such as Web servers, application servers, database servers and load balancers that distribute the incoming traffic across them. Complex performance behavior of these components makes it difficult to find the optimal resource allocation, even when the application workload is perfectly stable. The magnitude of the problem is further increased by the fact that Web application workloads are often very unstable and hard to predict. In the case of a sudden load increase there is a necessary tradeoff between reacting as early as possible to minimize the duration when the application underperforms because of insufficient processing capacity, and a slower approach to avoid situations where the load has already decreased when the new resources become available.

Faced with this difficult scientific challenge, the academic community has proposed a wide range of sophisticated resource provisioning algorithms (Dejun et al., 2011; Muppala et al., 2012; Urgaonkar et al., 2008; Vasić et al., 2012). However, we observe a wide discrepancy between these academic propositions and the very simple mechanisms that are currently available to cloud customers. These mechanisms are usually based on lower or upper thresholds on resource utilization. Crossing one of the thresholds triggers a pre-defined resource provisioning action such as adding or removing one machine.

We postulate three possible reasons why sophisticated techniques are not more widely deployed: *(i)* the gains of using sophisticated provisioning strategies are too low to be worth the effort; *(ii)* implementing and evaluating these techniques is a difficult exercise, which is why real cloud systems rely on simpler techniques; and *(iii)* academic approaches mostly focus on unrealistic evaluations using simple applications and artificial workloads (Do et al., 2011; Islam et al., 2012).

This paper investigate which of these possible causes are the real problems, and aims to propose automatic scaling algorithms which provide better results than the simple ones without overly increasing complexity. We implemented and installed several resource provisioning mechanisms in ConPaaS, an open source platform-as-a-service environment for hosting cloud applications (Pierre and Stratan, 2012). Our

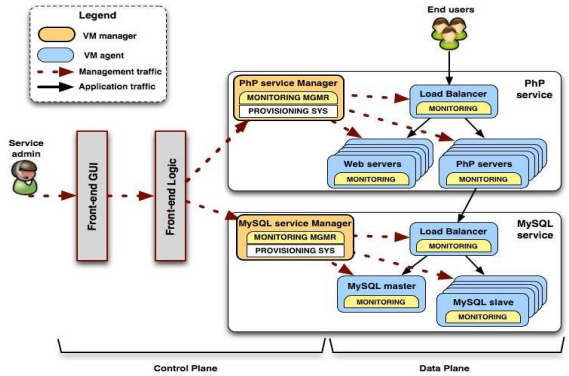


Figure 1: ConPaaS system architecture.

techniques use several levels of thresholds to predict future performance degradations, workload trend detection to better handle traffic spikes and dynamic load balancing weights to handle resources heterogeneity. We exercised these algorithms in realistic unstable workload situations by deploying a copy of Wikipedia and replaying a fraction of the real access traces to its official web site. Finally, we report on *(i)* implementation complexity; and *(ii)* potential gains compared to the threshold-based solution.

2 CONPAAS OVERVIEW

ConPaaS is an open-source runtime environment for hosting applications in Cloud infrastructures (Pierre and Stratan, 2012). In ConPaaS, an application is designed as a composition of one or more elastic and distributed *services*. Each service is dedicated to host a particular type of functionality of an application. ConPaaS currently supports seven different types of services: two web application hosting services (PHP and JSP); a MySQL database service; a NoSQL database service; a MapReduce service; a TaskFarming service; and a shared file system service.

Each ConPaaS service is made up of one manager virtual machine (VM) and a number of agent VMs.

Agent: The agent VMs hosts the necessary components to provide the service-specific functionality. Based on the performance requirements or the application workload, agent VMs can be started or stopped on demand.

Manager: Each service is controlled by one manager VM. The manager is in charge of centralizing monitoring data, controlling the allocation of resources assigned to the service, and coordinating reconfigurations so no loss of service is experienced when adding or removing resources.

Figure 1 shows the architecture of the ConPaaS deployment for a typical PHP web application backed by a MySQL database. The application is deployed using two ConPaaS services: the PHP service and the MySQL service. As illustrated in Figure 1, the manager VM of each ConPaaS service includes a performance monitoring and a resource provisioning system. The monitoring component is based on Ganglia (Ganglia monitoring system,), a scalable distributed monitoring system. The monitoring data is collected by the manager VM. For the PhP service, we implemented modules that extend Ganglia's standard set of monitoring metrics for measuring the request rate and response times of requests.

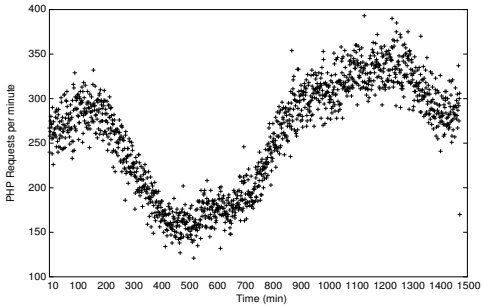
Each ConPaaS service can plug-in its own automatic resource provisioning policies, taking into account specificities of the service such as the structure of the service deployment, the complexity of the requests and service-specific monitoring data. So far our focus has been on the web hosting service, for which we implemented a number of automatic resource provisioning policies.

3 WIKIPEDIA APPLICATION

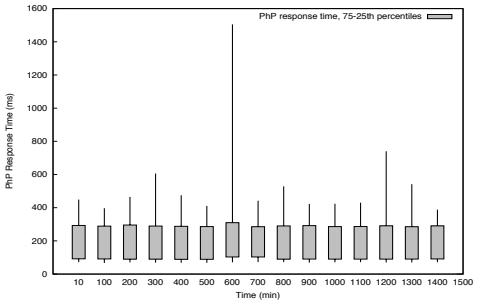
Most academic resource provisioning systems are evaluated using synthetic applications and workloads such as TPC-W, RuBiS and RuBBoS. However, these traditional benchmarks do not exhibit features of real Web applications such as traffic unstability as well as heterogeneous and constantly-changing request mixes (Cecchet et al., 2011). Instead of basing our evaluations on unrealistic workloads, we used the WikiBench benchmark (van Baaren, 2009). This benchmark uses a full copy of Wikipedia as the web application, and replays a fraction of the actual Wikipedia's access traces.

Hosting a copy of Wikipedia in ConPaaS requires two different services: a PHP web hosting and a MySQL service. The MySQL service is loaded with a full copy of the English Wikipedia articles as of 2008, which has a size of approximately 30GB. In the PHP service, the configuration was composed of one load balancer, one static web server and one or more PHP servers. We then use WikiBench to replay access traces from 2008 (Urdaneta et al., 2009). The trace contains requests for static and dynamic pages. When replaying it, the most important performance bottleneck is the application logic of the application: PHP requests are processed an order of magnitude slower than simpler static web pages. In this article we therefore focus on the automatic scaling of the PHP tier.

As an example, in Figure 2(a), we show the PHP workload sampled from one trace, as the number of PHP requests per minute during approximately one day. Besides the obvious variations in number of re-



(a) Workload intensity



(b) PHP request complexity

Figure 2: One day of Wikipedia trace

quests per minute, requests are also heterogeneous in complexity. To illustrate this heterogeneity, in Figure 2(b) we present the distribution of the response time values for the PHP requests during the execution of the trace shown in Figure 2(a). For this experiment we used a fixed number of 5 PHP servers, which are sufficient for handling the workload from the access trace even at its peaks; this is why in this case the response time is not influenced by the intensity of the workload. However, the results show a relatively wide dispersion of the response time. The main reason for this dispersion is that the Wikipedia articles vary in complexity, requiring different amounts of information that needs to be retrieved from the database and assembled in a page.

We also conclude from this experiment that, since the response time of the application varies and cannot be properly predicted even when the request rate is known, a provisioning algorithm aiming to keep the application's response time under a certain limit should take into account several monitoring parameters beyond the request rate and the response time.

4 PROVISIONING ALGORITHMS

We evaluate three resource provisioning algorithms: a trigger-based algorithm representing a baseline comparable to systems currently in production, and two new algorithms based on feedback techniques that alleviate the drawbacks of trigger-based systems by reacting in advance to traffic changes.

Algorithm 1: Trigger-based provisioning

Data: Pre-defined metric threshold ranges, *SLO_threshold*
Result: Scaling decisions

```
while auto-scaling is ON do
    Collect monitoring data of each metric, data_i;
    if no recent scaling operation then
        if avg(data_i)>=SLO_threshold_max_i for at least one metric then
            ADD resources;
        else if avg(data_i)<SLO_threshold_min_i for all metrics then
            REMOVE resources;
        end
    end
    Wait until the next reporting period (5 minutes) ;
end
```

4.1 Trigger-based provisioning

Existing cloud infrastructures adjust the amount of allocated resources based on a number of standard monitoring parameters. These are simple trigger-based systems which define threshold rules to increase or decrease the amount of computational resources when certain conditions are met. As an example, the Auto Scaling system offered by Amazon EC2 (Amazon Elastic Compute Cloud (EC2),) can be used to define rules for scaling out or back an application based on monitoring parameters like CPU utilization, network traffic volume, etc. Similar trigger-based techniques are also used in platforms such as RightScale (RightScale,) or OpenShift (OpenShift,).

For the sake of comparison, we designed and implemented a trigger-based provisioning mechanism in ConPaaS. This technique monitors the percentage of CPU usage and application response time metrics, and assigns a lower and upper threshold for each metric. It then creates or deletes VMs whenever one of the threshold is crossed, as described in Algorithm 1. Analogous to traditional trigger-based systems, the thresholds are statically defined by the user before execution.

Even though this algorithm is simple and widely used in cloud platforms, we will show that it is too reactive. This is due to two main factors:

1) Workload heterogeneity: For some web sites, the workload fluctuates following an irregular pattern, with occasional short traffic spikes. An excessively reactive algorithm cannot handle these situations well, and will cause frequent fluctuations in the number of allocated resources; this has negative ef-

fects on the stability and performance of the system. Considering the pricing model of cloud providers that charges users on a per-hour basis, trigger-based systems increase the infrastructure cost because they frequently terminate resources before this hour price boundary.

2) Resources heterogeneity: The performance of virtual instances provided by current clouds is largely heterogeneous, even among instances of the same type (Dejun et al., 2009). Simple trigger-based provisioning systems do not take this heterogeneity into account, thus providing less efficient resource allocation.

We believe that trigger-based provisioning mechanisms can be improved without drastically increasing their complexity. We now present two techniques which aim at solving the aforementioned drawbacks by relying on predictive and more accurate methods.

4.2 Feedback provisioning

To handle the workload heterogeneity and mitigate the reactiveness of trigger-based techniques, we designed and implemented an algorithm that relies on three simple mechanisms: integration of several metrics and automatic weighting of each metrics according to its predictiveness; estimating the overall workload trend; and definition of two levels of threshold ranges instead of one.

Weighted metrics: As pointed out in (Singh et al., 2010), provisioning decisions solely made on the basis of request rate or percentage of CPU usage can incur errors by under- or over-provisioning an application. We propose instead to use a collection of metrics (e.g. response time, request rate and CPU usage) likely to indicate the need to scale the system. Our algorithm continuously measures the correlation between each metric's values and the actual scaling decisions. This method enables to better adapt the scaling decisions to the current workload requirements by identifying how metrics affect to the processing time. We thus assign a weight to each metric (correlation coefficient) according to its predictiveness. For example, when hosting the MediaWiki application, the hosting system needs to deal with requests having a wide diversity of computing complexity, and with significant variations in the ratios of simple vs. complex requests. Our algorithm therefore associates a higher weight to the CPU usage metric than to the request rate, as it affects to the response times.

Workload's trend estimation: a difficult problem in any autoscaling system is the fact that any decision has a delayed effect. Provisioning extra resources can take a couple of minutes before they are ready to process traffic. Besides, cloud computing resources are usually charged at a 1-hour granularity so the scaling decision has consequences at least during the first hours after provisioning. It is therefore important to estimate the medium-term trend of response times in addition to any short-term bursts. We therefore estimate the response time trend using past values over a period of the same order of magnitude (30 min in our experiments). We classify workloads in one out of three categories: stable, increasing or decreasing.

Two-level thresholds: Initially, the users define a fixed threshold range based on their performance requirements (max and min required response time), denoted by *SLO thresholds* (Service Level Objective) in

Algorithm 2: Feedback provisioning

```
   Data: Pre-defined metric threshold ranges, SLO_threshold
   Result: Scaling decisions
 1 Create a queue to store historical workload, q;
 2 Establish two-level thresholds: pred_thr and reac_thr ;
 3 while auto-scaling is ON do
 4   Initialize variables s_bck and s_out to 0;
 5   Collect monitoring data (last ~5min), data;
 6   Add the most recent response_time value to q;
 7   Estimate workload trend (last ~30min) of q, td;
 8       - Increasing, td = 1; Decreasing, td = 0; Stable, td = -1;
 9   for each metric in data do
10     // Compute the weight w_i of metric_i
11     w_i = correlation_coefficient(metric_i, response_time);
12     if avg(metric_i) >= pred_thr_max_i then
13       // Increment chances of scaling_out
14       s_out = s_out + w_i;
15     else if avg(metric_i) < pred_thr_min_i then
16       // Increment chances of scaling_back
17       s_bck = s_bck + w_i;
18       else
19         // Decrease the chances
20         s_bck =s_bck - w_i; s_out =s_out - w_i;
21       end
22     end
23     if no recent scaling operation then
24       if avg(metric_i) >= reac_thr_max_i and td = 1 and s_out > s_bck
         then
25         ADD resources;
26       else if avg(metric_i) < reac_thr_min_i and td = 0 and s_out <
         s_bck then
27         REMOVE resources;
28       end
29       Reset s_bck and s_out to 0 ;
30     end
31     Wait until next reporting period (5 minutes);
32   end
33 end
```

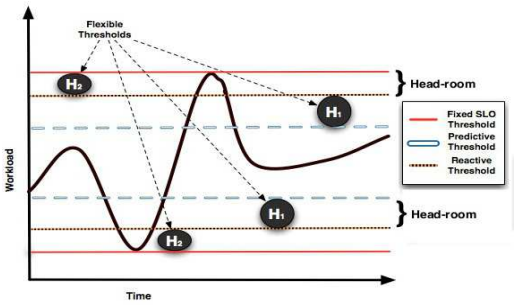


Figure 3: Two-level thresholds

Figure 3. However, these upper and lower bounds only indicate the resource under-utilization or when a SLO violation occurs. Thus, to prevent in advance these performance degradations, additional boundaries have to be defined. This mechanism establishes two levels of threshold ranges for each metric (CPU and response time) based on the bounds defined in the SLO by the user.

In Figure 3, these two extra thresholds called *predictive* and *reactive*, both with upper and lower bounds, create two "head-rooms" between the SLO threshold and them. The predictive head-room H_1 is intended to alert of future workload alterations when a metric exceeds its predictive bounds, thus increasing proportionally to its weight the chances to trigger scaling actions (denoted by *s_bck* and *s_out* in Algorithm 2 lines 12-17). Otherwise, the scaling chances will drop in the same proportion (Line 20). The reactive head-room H_2 is used to trigger scaling actions if the workload presents a *increasing or decreasing* variation (Lines 24-26), which may cause SLO violations if it keeps such a trend. This mechanism in conjunction with the workload trend estimation allow to better analyze the evolution of performance fluctuations, and as a consequence improves the accuracy of our decisions by reacting in advance. In the future, these two-levels of thresholds could be adjusted depending on the hardware configuration of each provisioned VM, as introduced in (Beloglazov and Buyya, 2010).

To sum up, the feedback algorithm triggers a scaling action when a series of conditions are satisfied: (i) no previous scaling actions have been taken over the last 15min; (ii) the recent monitoring data have to exceed the predictive and reactive threshold ranges; (iii) the workload trend has to follow a constant pattern (increasing/decreasing). Although the combination of these techniques improves the accuracy of our measurements, the heterogeneous nature of the VM instances requires more flexible provisioning algorithms, as pointed out in (Jiang, 2012).

4.3 Dynamic load balancing weights

The problem we consider here is the heterogeneity of cloud platforms. Different VMs have different performance characteristics, even when their specifications from the cloud vendor are the same. This issue can be addressed through various load balancing techniques, like assigning weights to the backend servers or taking into account the current number of connections that each server handles. Furthermore, the performance behavior of the virtual servers may also fluctuate, either due to changes in the application's usage patterns, or due to changes related to the hosting of the virtual servers (e.g., VM migration).

In order to address these issues in ConPaaS we implemented a weighted load balancing system in which the weights of the servers are periodically readjusted automatically, based on the monitoring data (e.g. response time, request rate and CPU usage). This method assigns the same weight to each backend server at the beginning of the process. The weights are then periodically adjusted (in our experiments, every 15min) proportionally with the difference among the average response times of the servers during this time interval. By adding this technique to the feedback-based algorithm, we noticed a performance improvement when running the benchmarks.

5 EVALUATION

To compare the provisioning algorithms described above, we ran experiments on two infrastructures: a homogeneous one (the DAS-4, a multi-cluster system hosted by universities in The Netherlands (Advanced School for Computing and Imaging (ASCI),)) and a heterogeneous one (Amazon Elastic Compute Cloud (EC2),). The goal of our experiments was to compare the algorithms by how well they fulfill the SLOs and by the amount of resources they allocate.

Testbed configuration: As a representative scenario, we deployed the MediaWiki application using ConPaaS on both infrastructures, and we ran the Wikibench tools with a 10% sample of a real Wikipedia access trace for 24hours. We configured the experiments as follows: a monitoring window of 5min, a SLO of 700ms at the service's side (denoted by a red Line in Figures) and the same statistically-chosen performance threshold ranges for response time and CPU utilization. Note that, the weighted load-balancing provisioning technique was only evaluated on the heterogeneous platform Amazon EC2, as it only brings improvements in environments where VMs may have different hardware configurations.

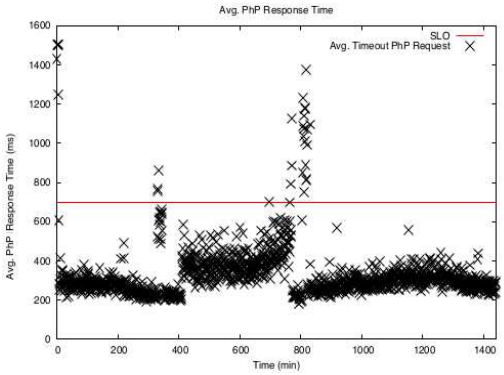


Figure 4: Response time on DAS4 – Trigger-based.

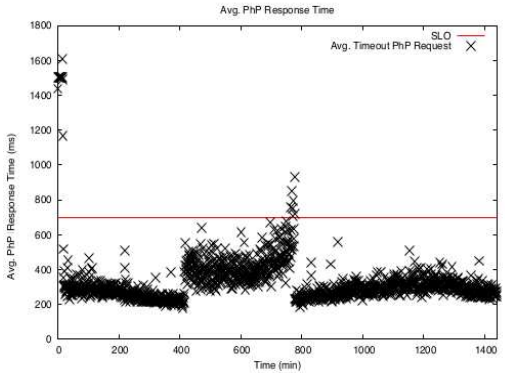


Figure 5: Response time on DAS4 – Feedback.

5.1 Homogeneous Infrastructure

Our experiments on DAS-4 rely on OpenNebula as IaaS (Sotomayor et al., 2009). To deploy the Wikipedia services, we used small instances for the PHP service (manager and agents) and a medium instance for the MySQL service (agent). In DAS-4, OpenNebula's small instances are VMs equipped with 1 CPU of 2Ghz, and 1GiB of memory, while medium instances are equipped with 4 CPU's of 2Ghz, and 4GiB of memory.

SLO enforcement. Figure 4 and Figure 5 represent the degree of SLO fulfillment of the trigger-based and feedback algorithms, indicating the average of response times obtained during the execution of the Wikipedia workload trace. In Figure 4 and Figure 5, each cross represents the average of response time obtained at monitoring window. The results from Figure 4 show that the trigger-based provisioning algorithm provokes an important amount of SLO violations at certain moments in time, due to its excessively reactive behavior. As we mentioned, this algorithm fails easily during traffic spikes, as it adds or removes VMs without evaluating the workload trend. The feedback algorithm, as shown on Figure 5, handles the traffic spikes better and can minimize the amount of SLO violations; specifically, there were 31.72% less SLO violations in comparison with the trigger-based algorithm.

Resource consumption. To better understand the behavior of both algorithms, we shall also focus on the resource consumption illustrated on Figure 6. The excessively reactive behavior of the trigger-based algorithm can be noticed in the time intervals around *t=350min* and *t=820min*, where two scaling operations under-provision the system during a short period of time. These provisioning decisions provoked the SLO violations that are visible in Figure 4 in the same intervals of time. Besides affecting the system's stability, such short fluctuations in the number of provisioned resource also raise the cost of hosting the application since more VM instantiations will be triggered. When using the feedback algorithm, the system makes provisioning decisions by analyzing the workload's trend. Scaling actions are only triggered when having *constant* alterations in the workload, thereby providing a more efficient resource usage. We can see that the provisioning decisions on Figure 6 match well with the workload variations depicted on Figure 2(a).

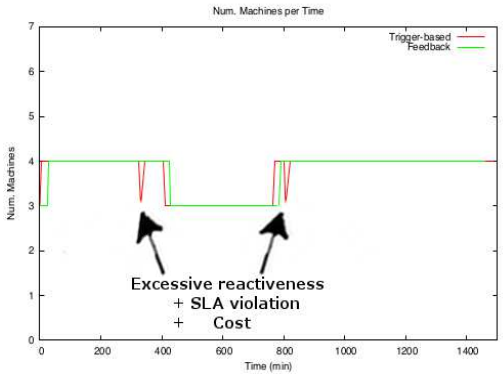


Figure 6: Resource consumption on DAS4.

Discussion. Both algorithms are best-effort regarding the SLO fulfillment, and thus they do not handle well short alterations of the workload (with duration in the range of minutes). The heterogeneity of the PHP requests, as well as the VM scheduling time (2-5min), are in part responsible of these SLO violations.

5.2 Heterogeneous Infrastructure

Our experiments on Amazon EC2 used small instances for the PHP service (manager and agents) and a medium instance for the MySQL service (agent). EC2 small instance are equipped with 1 EC2 CPU, and 1.7GiB of memory, while medium instances are

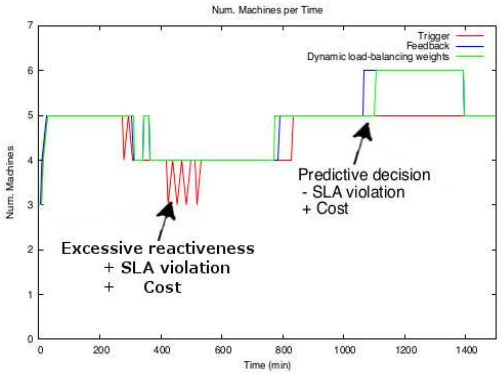


Figure 7: Resource consumption on EC2.

equipped with 2 EC2 CPU's, and 3.75GiB.

SLO enforcement. Figure 8, Figure 9 and Figure 10 show the system performance of the trigger-based, feedback and dynamic load-balancing weights algorithms, respectively. As depicted on Figure 8, the performance of the trigger-based algorithm is even more unstable than in the case of the homogeneous infrastructure. Two of the three peaks in response time, at *t=300min* and *t=820min*, can be explained by the variations in the Wikipedia workload shown in Figure 2(a). However, there is a third peak between *t=400min* and *t=500min* that corresponds to an interval of time in which the workload trace shows a significant drop in the request volumes. During this period of time, the algorithm attempts to scale back the system but the new lower number of resources cannot handle the load, and the system is scaled out again. As this algorithm does not keep any history information, after a short time it attempts again to scale back the system, with the same result; thus, some oscillations occur in the number of allocated resurces, causing also SLO violations.

Figure 9 and Figure 10 show that the feedback and dynamic load-balancing algorithm reduce the number of SLO violations and provide a more stable performance pattern. Specifically, with the feedback algorithm we obtained 41.3% less SLO violations than with the trigger-based algorithm, while the dynamic load balancing algorithm had 47.6% less violations than the trigger-based one. This improvement of 6.3% responds to an efficient distribution of the incoming traffic across the allocated resources.

Resource consumption. As explained above, the trigger-based algorithm sometimes initiates series of scaling back operations quickly followed by scaling out operations, due to the fact that it does not use any history information that could prevent these oscillations. This can be seen on Figure 7, in the time interval between *t=400min* and *t=500min*. The feedback and dynamic load balancing algorithms have similar and more stable patterns of resource consumption, which show the benefits of using trend estimations and two-level threshold ranges.

Discussion. These runs show significant differences between the trigger-based algorithm on one hand, and the feedback and dynamic load balancing on the other hand. The results show that using a few heuristics as in the last two algorithms can help in improving the performance and stability of the system. We did not obtain however significant differences between the feedback and the dynamic load balancing algorithm. The last algorithm uses the additional technique of dynamically adjusting the load balancing weights, but the VMs participating in the experiments had relatively similar performance and adjusting their weights only made a small difference. On other infrastructures that are more heterogeneous than Amazon EC2, the dynamic load balancing technique may bring a greater performance improvement.

We also note that the frequent scaling actions launched by the trigger-based algorithm generally increase the infrastructure cost. An explanation comes from the frequent creation/removal of VM instances occurring within a time interval of 1hour. Even though this trace shows a clear daily variation which represents the traditional traffic pattern of web applications, more traces have been utilized to validate our autoscaling system in (Fernandez et al., 2014)

6 RELATED WORKS

There is a wide literature on issues related to dynamic resource provisioning for cloud web applications. However, most of these models require a deep understanding in mathematics or machine learning techniques which are not easily interpreted by non specialists. Besides the traffic in web applications is shaped by a combination of different factors such as diurnal/seasonal cycles that follows an irregular pattern, thus making extremely challenging the design and development of realistic and accurate provisioning mechanisms.

These well-known drawbacks force to IaaS like Amazon EC2, or PaaS like RightScale and OpenShift, to design simple trigger-based auto-scaling systems, instead of relying on approaches from academic research. Unfortunately, these scaling systems are imprecise and wasteful in terms of resource consumption and cost savings (Ghanbari et al., 2011).

As a consequence, more relevant and realistic academic approaches have been proposed over the last years. (Urgaonkar et al., 2008) designed and imple-

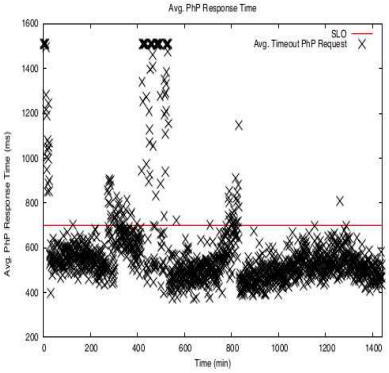


Figure 8: Response time on EC2 – Trigger-based.

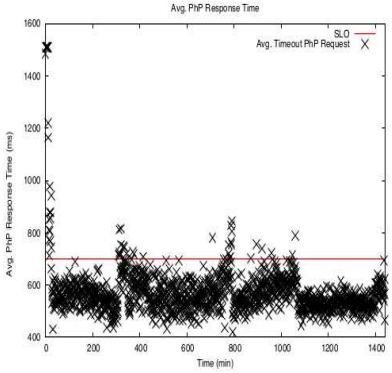


Figure 9: Response time on EC2– Feedback.

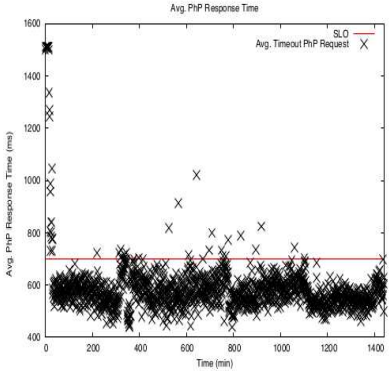


Figure 10: Response time on EC2– Load-balancing Weights.

mented a predictive and reactive mechanism using a queuing model to decide the number of resources to be provisioned, and an admission control mechanism to face extreme workload variations. Even though the use of admission mechanisms enforce the performance requirements, it reduces the QoS of the service, and therefore affects user experience. Differently, (Muppala et al., 2012) proposed offline training techniques to gather information about the resource requirements of the current workload, and thereby to improve the accuracy of the scaling decisions. In the same vein, DejaVu (Vasić et al., 2012) and CBMG (Roy et al., 2011) built similar mechanisms to classify the workload need by analyzing recent traffic spikes or the customer behavior. However, these approaches require additional resources to identify the workload requirements, and an exhaustive knowledge of the deployed applications; thus preventing its integration in existing autoscaling systems.

As a proposal closely related to ours, (Ghanbari et al., 2011) designed an autoscaling system using control theoretic and rules-based models. The authors claimed for simpler provisioning mechanisms in comparison with the sophisticated academic approaches. However, factors such as resource heterogeneity were not addressed in this system.

7 CONCLUSION

Our study demonstrated that traditional trigger-based provisioning mechanisms do not handle workload variation well and in some situations cause instability, by changing the number of provisioned VMs often in short time intervals. This behavior has a negative effect on the application's response time, leading to violations of the SLO.

By using a few techniques that are relatively easy to implement we showed that we can significantly reduce the number of SLO violations and improve the performance stability. Although there is still room for improvement in our techniques, from the experimental results we can draw the conclusion that implementing provisioning mechanisms that go beyond simple triggers is effective and should be considered when hosting a cloud application.

ACKNOWLEDGMENTS

This work is partially funded by the FP7 Programme of the European Commission in the context of the Contrail project under Grant Agreement FP7-ICT-257438 and the Harness project under Grant Agreement 318521.

REFERENCES

Advanced School for Computing and Imaging (ASCI). The Distributed ASCI SuperComputer 4. `http://www.cs.vu.nl/das4/`.

Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2`.

Beloglazov, A. and Buyya, R. (2010). Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proc. 8th International Workshop on Middleware for Grids, Clouds and e-Science*.

Cecchet, E., Udayabhanu, V., Wood, T., and Shenoy, P. (2011). BenchLab: An open testbed for realistic benchmarking of web applications. In *Proc. WebApps*.

Dejun, J., Pierre, G., and Chi, C.-H. (2009). EC2 performance analysis for resource provisioning of service-oriented applications. In *Proc. 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*.

Dejun, J., Pierre, G., and Chi, C.-H. (2011). Resource provisioning of Web applications in heterogeneous clouds. In *Proc. Usenix WebApps*.

Do, A. V., Chen, J., Wang, C., Lee, Y. C., Zomaya, A., and Zhou, B. B. (2011). Profiling applications for virtual machine placement in clouds. In *Proc. IEEE CLOUD*.

Fernandez, H., Pierre, G., and Kielmann, T. (2014). Autoscaling in heterogenous in cloud infrastructures. In *Proc. IEEE IC2E*, page (to appear).

Ganglia monitoring system. `http://ganglia.sourceforge.net/`.

Ghanbari, H., Simmons, B., Litoiu, M., and Iszlai, G. (2011). Exploring alternative approaches to implement an elasticity policy. In *Proc. IEEE CLOUD*.

Islam, S., Keung, J., Lee, K., and Liu, A. (2012). Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1):155–162.

Jiang, D. (2012). *Performance Guarantees For Web Applications*. PhD thesis, VU University Amsterdam.

Muppala, S., Zhou, X., Zhang, L., and Chen, G. (2012). Regression-based resource provisioning for session slowdown guarantee in multi-tier internet servers. *Journal of Parallel and Distributed Computing*, 72(3):362–375.

OpenShift. `https://openshift.redhat.com/app/flex`.

Pierre, G. and Stratan, C. (2012). ConPaaS: a platform for hosting elastic cloud applications. *IEEE Internet Computing*, 16(5):88–92.

RightScale. `http://www.rightscale.com//`.

Roy, N., Dubey, A., and Gokhale, A. (2011). Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. IEEE CLOUD*.

Singh, R., Sharma, U., Cecchet, E., and Shenoy, P. (2010). Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. ICAC*.

Sotomayor, B., Montero, R. S., Llorente, I. M., and Foster, I. (2009). Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14 –22.

Urdaneta, G., Pierre, G., and van Steen, M. (2009). Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830 – 1845.

Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. (2008). Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1–39.

van Baaren, E.-J. (2009). Wikibench: A distributed, wikipedia based web application benchmark. Master's thesis, VU University Amsterdam.

Vasić, N., Novaković, D., Miučin, S., Kostić, D., and Bianchini, R. (2012). DejaVu: Accelerating resource allocation in virtualized environments. In *Proc. ASPLOS*.