Autoscaling Web Applications in Heterogeneous Cloud Infrastructures

Hector Fernandez, Guillaume Pierre, Thilo Kielmann

Autoscaling Web Applications in Heterogeneous Cloud Infrastructures

Hector Fernandez
VU University Amsterdam
Email: hector.fernandez@vu.nl

Guillaume Pierre
IRISA / Rennes 1 University
Email: guillaume.pierre@irisa.fr

Thilo Kielmann
VU University Amsterdam
Email: thilo.kielmann@vu.nl

***Abstract*—Improving resource provisioning of heterogeneous cloud infrastructures is an important research challenge. The wide diversity of cloud-based applications and customers with different QoS requirements have recently exhibited the weaknesses of current provisioning systems. Today's cloud infrastructures provide provisioning systems that dynamically adapt the computational power of applications by adding or releasing resources. Unfortunately, these scaling systems are fairly limited: *(i)* they restrict themselves to a single type of resource; *(ii)* they are unable to fulfill QoS requirements in face of spiky workload; and *(iii)* they offer the same QoS level to all their customers, independent of customer preferences such as different levels of service availability and performance. In this paper, we present an autoscaling system that overcomes these limitations by exploiting heterogeneous types of resources, and by defining multiple levels of QoS requirements. The proposed system selects a resource scaling plan according to both workload and customer requirements. Our experiments conducted on both public and private infrastructures show significant reductions in QoS-level violations when faced with highly variable workloads.**

I. Introduction

With the rise of cloud computing, modern enterprise software systems started to deploy their services over clouds. As a utility-oriented and service-based approach to computing, cloud infrastructures offer many attractive features to their customers. In particular, cloud providers allow tenants to rent resources in a *pay-as-you-go* fashion. This pricing model is specifically employed by enterprise software systems where the assurance of QoS requirements is crucial to boost the volume of customers, and hence their revenues. Typically these requirements are specified by the enterprise (customer) and affirmed by cloud provider in the form of a *service level agreement* (SLA), and vary depending on the size of the enterprise. Thus, large enterprises such as Animoto, *The Guardian* and Spotify, pay more to provide high assurance of availability and performance to their clients [1]; while small enterprises pay less to obtain an acceptable performance but with weaker service availability.

When dealing with websites, the fulfillment of the SLA requirements becomes more problematic, as the workload demand fluctuates as a result of sudden changes in the popularity and/or request mix, flash crowds, outages and network misconfigurations. For instance, on June 25th 2009, the news of Michael Jackson's death quickly crippled popular websites, such as *TMZ.com*, *The Angeles Times* or *Twitter*, and resulted in hours-long slowdown or outages [2]. These sudden traffic fluctuations are specifically difficult to handle by traditional resource management systems, thus causing long periods of violations of the SLA requirements [3]. Failure to comply with these requirements is often associated with significant financial penalties or other forms of loss of revenue such as decreases in the user base. Therefore, it is crucial to use resource provisioning systems that scale an application on demand while meeting the requirements.

Nowadays, cloud infrastructures provide elastic provisioning by supporting a variety of scaling mechanisms and different hardware configurations for rent, each with a different infrastructure cost. Even though the diversity of hardware configurations is common in cloud infrastructures, most resource provisioning systems focus on minimizing the infrastructure cost rather than selecting a suitable combination of resources [4], [5]. Moreover, when using these systems to scale in response to changing conditions, most of them restrict themselves to a single type of hardware configuration, ignoring the important avenues for cost/performance optimization. We believe that the selection of multiple types of resources with different performance capacity/cost characteristics can mitigate the degradations due to sudden workload demand fluctuations. As a consequence, a new challenge arises to autoscaling systems, as they have to decide which type of resources to choose for a particular workload. In the following, we use the term *scaling plan* to refer to this searching process to find the most appropriate resource combination.

Traditional autoscaling systems do not allow to adapt the selection criteria of scaling plans to customer preferences like service availability, cost or performance. From one customer to another, the tradeoff between cost and SLA fulfillment may vary, especially when handling flash crowds or other traffic anomalies. Therefore autoscaling systems have to choose the scaling plan that matches a customer's preferences best. As an example, large enterprises, able to pay more, will provision powerful resources to absorb traffic spikes, while small enterprises prefer to fine-tune their budgets by provisioning cheaper resources that have less slack to handle any eventual spike. There is a necessary tradeoff between the cost one customer is ready to spend and the performance guarantee that no SLO (Service Level Objective) violation will occur.

This paper presents an autoscaling system that benefits from the heterogeneity of cloud infrastructures to better enforce customer requirements, even under large and temporary workload variations. The selection of an appropriate combination of resources provides sufficient computing capacity to handle the traffic variations without drastically raising the cost. To achieve that, our system profiles each type of allocated resources to measure their capacity, and in conjunction with a

medium-term traffic predictor devises the *scaling plan* matching the workload requirements. In this system, each customer can tune its own cost/SLA fulfillment tradeoff using a "metal" classification scheme (for *gold*, *silver*, and *bronze* customers), which defines different criteria for the selection of scaling plans based on the respective customer's QoS preferences. To handle resource heterogeneity, the proposed system provides a weighted load balancing mechanism that enables the distribution of the incoming traffic across resources depending on their performance capacities. We evaluated our system using realistic, unstable workload situations by deploying a copy of Wikipedia and replaying a fraction of the real access traces to its site. This evaluation was conducted on both private and public clouds using different cost/SLA fulfillment configurations. Our results show significant reductions in SLA violations using our approach.

This paper is organized as follows: Section II discusses the related work; Section III presents our autoscaling system approach; Section IV explains how our system works; Section V presents our experimental evaluation; and Section VI closes this article with conclusions and future work.

II. Background

In the past few years, more and more cloud providers like Amazon EC2, designed and integrated dynamic resource provisioning systems into their infrastructures. Unfortunately, these systems are imprecise and wasteful in terms of SLA fulfillment and resource consumption, as pointed out in [6]. Consequently, more realistic academic approaches have been proposed. For instance, in [7] and [8], the authors designed provisioning systems that predict the future service demand to decide the amount of resources to provision. Specifically, [8] designed a pluggable and cost-aware autoscaling system that forecasts the future demand by analyzing metrics such as the request volume. However, even though these approaches proved to efficiently answer to the question *"When to provision?"*, they assumed that all resources perform similarly in a cloud infrastructure. As pointed out in [9], the performance of VM instances provided by current clouds is largely heterogeneous, even among instances of the same type.

To handle the resource heterogeneity of cloud infrastructures, a sensitivity analysis of the allocated resources is crucial to provide accurate scaling decisions. Accordingly, *online* and *offline* profiling-based techniques [10] have recently emerged as a solution to estimate the resource's throughput under a certain workload. This technique replicates at design-time (offline) or runtime (online) a server hosting an application, with a new server with profiling instrumentation. In [11], the authors uses an *offline profiling* technique for the definition of profiles at component-level. This mechanism measures the resource utilization of each component allocated to one resource to estimate the future demand under such resource. By doing so, the authors limits the profiling to only one specific type of resource. Similarly, *SmartScale* [12] provided an *offline* technique to analyze the CPU usage and memory consumed by one resource to decide the amount of identical resources to provision. However, the creation of one profiling server per type of resource (at design-time) limits the adoption of this technique in cloud infrastructures due to its high operational cost. For a complete *offline* training, these systems require to profile as many additional resources as VM instance's types are supported in the infrastructure. Using our proposed *profiling* technique, provisioned resources are profiled using real workload without need of setting a parallel environment.

Closer to our work, [13] built a cost-aware provisioning system that exploited the resource heterogeneity of cloud infrastructures prior to any resource selection. Although using an *offline profiling* technique, this work was able to estimate the maximum performance capacity of a resource by running an application on different resource types, and subjecting them to a gradually increasing synthetic workload. Like in [12], the necessity of *offline* training activities and a higher infrastructure cost prevents the integration of this approach in existing autoscaling systems. Moreover, the use of unreal application workload reduces the accuracy of the profiling results.

Using *online profiling* techniques, DejaVu [5] built a mechanisms to classify the workload need by analyzing recent traffic spikes and the performance behavior of the resources, respectively. To do that, DejaVu configures a parallel environment that distributes a fixed percentage of the current traffic to a specific resource's type for the definition of profiles. Even though this approach presents good results, it requires additional resources (one per resource type and one proxy) to identify the performance capabilities of each instance type offered by a cloud provider. Additionally, in this work, it is not clear how the scaling plan selection takes place when having profiles from different resource types.

Contrary to these efforts, our work goes one step further by measuring resource performance using real workload over the allocated resources. Moreover, it uses the resulting profiles to select the scaling plan that better handle the current and future workload demand according to the customer QoS preferences.

III. Autoscaling system architecture

The proposed autoscaling system scales a web application in response to change in throughput at fixed intervals, which we denote by reconfiguration intervals set to 10 minutes. This system operates alongside the services deployed by ConPaaS, an open-source runtime environment for hosting applications in Cloud infrastructures [14]. Nevertheless, it can also be easily integrated into any other PaaS, as it only relies on two common services provided by any platform: the resource manager and the monitoring engine. To feed this system, we use the monitoring engine that tracks the application-workload and system resources. Thus, as shown on Figure 1, the architecture of our system has the following key components:

Profiler: This component was designed to measure the computing capacity of different hardware configurations when running an application. To do that, this component creates profiles for each resource type (VM instance in the following) by analyzing its performance behavior (e.g. the percentage of CPU usage, request rate and response time). As we mentioned, the definition of profiles per-instance type improves the accuracy of the scaling actions, specifically in heterogeneous cloud infrastructures. Thus, the *Profiler* component calculates the *optimized throughput* of each type of provisioned configuration. Here an *optimized throughput* is the performance pattern under which a resource assures the QoS requirements while avoiding its under/over-utilization.

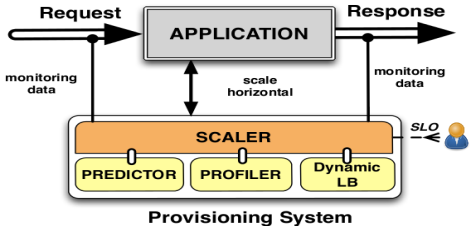

Fig. 1: Autoscaling system.

Predictor: To prevent SLA violations in advance, workload prediction is needed to estimate the incoming traffic. Inspired by [15], our *Predictor* component takes the monitoring data as input and uses different time-series analysis techniques to predict the future service demand for the next monitoring window. To provide accurate forecasting measures, this component utilizes the technique that exhibited the lowest cumulative error measure during the previous monitoring window. By doing so, the *Predictor* is able to adapt the predictions to the current type of workload. This component supports five distinct statistical models that fits four types of workload: *(1) Linear Regression* for linear trends [16], *(2) Auto Regression Moving Average* (ARMA) for linear with small oscillations [17], *(3) Exponential Smoothing Holt Winters* for daily and seasonal [18], and *(4) Autoregression* and *Vector Autoregression* for correlated trends [19].

Dynamic load balancer: In order to adapt our system to the heterogeneity of requests and cloud resources, this component dynamically adjusts weights to the backend servers to proportionally distribute the incoming traffic based on the performance capacities of each provisioned resource. As an example, a server with four cores is able to process a higher number of requests, thus its weight is usually greater than a server with only one core.

Scaler: This component governs our autoscaling system. As illustrated in Figure 1, the *Scaler* uses the *Predictor* and *Profiler* components to find the scaling plan that fulfills a pre-established SLO, and consequently better enforces customer preferences. Furthermore, this component constantly analyzes the behavior of each provisioned VM, triggering the *Dynamic load balancer* component when necessary.

In the following section, we focus on answering *when* to scale and *which scaling plan* to choose when running web applications. Note that for simplicity, detailed description of the *Predictor* and *Dynamic LB* components are out of the scope of this work.

IV. Provisioning on demand

As a central component of our autoscaling system, the *Scaler* is responsible for triggering the scaling actions. It supports horizontal scaling as a technique to add and remove resources to an application. *Horizontal scaling* enables to create a cluster of virtual machines associated to one application, which size is dynamically adapted to the workload fluctuations by adding or removing resources to the cluster. According to this technique, the *Scaler* evaluates whether the current performance behavior of an application requires triggering any of the following scaling actions:

- **Scale out:** Additional resources are provisioned if the loaded system state (response time or CPU utilization) exceeds an upper threshold (based on the QoS requirements), and the *Predictor* confirms that such traffic changes will remain at least during the next monitoring window. In our experiments, we define monitoring windows of 5 minutes as this interval is the minimum necessary time to collect fresh data from the monitoring engine provided by ConPaaS.
- **Scale back:** Resources are released if the loaded system state exceeds a lower threshold, and the *Predictor* confirms that such traffic changes will remain at least during the next monitoring window.

The logic behind these actions answers the question when to scale. But, the discovery of a proper scaling plan is the most important and challenging phase in a provisioning system. It is responsible of the selection of an appropriate combination of resources that fulfills the QoS requirements. Therefore, to decide which scaling plan to provision, the *Scaler* follows the next procedure:

1) *Measure the resource performance* to know the capacities of cloud resources when running the application.
2) *Monitor workload and generate medium-term predictions* to devise all the plausible scaling plans.
3) *Select the optimal scaling plan* based on the tradeoff cost/SLO guarantee defined by the QoS level selected by the customer.

A. Measuring VM instance performance

Even though the use of *online* profiling techniques is still under study, the configuration of a parallel environment, the use of synthetic workload and the heterogeneity of the cloud resources are the major drawbacks for its adoption. They increase the operational cost and do not necessarily improve the accuracy of the scaling decisions. As a consequence, to address these drawbacks we designed a novel online profiling technique that gives an estimation of the optimized throughput of each allocated VM instance type without the need for additional resources or parallel environments. To do that, the *Profiler* component uses the provisioned resources and the real workload for the analysis and estimation of the computing capacity of each available server configuration.

As detailed in Algorithm 1, once the *Scaler* component decides to trigger a scaling action, the *Profiler* component estimates the throughput of the allocated instances according to the following steps:

a) Collection of monitoring data: It collects the latest hour of monitoring data from each allocated VM instance type. This data contains information about monitoring metrics such as the request rate, total percentage of CPU usage and response time, which provide enough feedback for the definition of instance profiles. In fact, when provisioning web applications, these metrics are commonly used to decide whether to trigger scaling actions or not [4], [12]. Nevertheless, other metrics such as the network bandwidth and memory usage can also be collected to define instance profiles.

Algorithm 1: VM instance profiling algorithm

```
   Data:
   Service Level Objective, slo
   List of allocated VM instance types, inst_types
   Compute units per inst_type, compute_units_inst
   Result: VM instances performance classification, list_perf
1  while allocated instances to profile do
2      Collect profiling data of inst_type: req_rate, %cpu_usage and resp_time;
3      while profiling data to smooth do
4          // Perform smoothing percentiles technique ;
5          if resp_time_i > (slo * 0.25) and resp_time_i <= (slo * 0.75) then
6              if req_rate_i > 0 and %cpu_usage_i < 75 then
7                  Add %cpu_usage_i to %cpu_usage_data;
8                  Add req_rate_i to req_rate_data;
9                  Add resp_time_i to resp_time_data;
10             end
11         end
12     end
13     Initialize instance capacity Ideal_Throughput_inst to 0 ;
14     if resp_time_data, %cpu_usage_data, req_rate_data not empty then
15         Calculate average of %cpu_usage_data, %CPU_usage_inst ;
16         Calculate average of req_rate_data, Num_requests_inst ;
17         Ideal_Throughput_inst = ((%CPU_usage_inst / 100)) / Num_requests_inst ;
18         Store instance computing capacity, Ideal_Throughput_inst ;
19     else
20         Use historic value of Ideal_Throughput_inst for inst_type;
21     end
22     // Classify inst_type based on its computing capacity ;
23     if Ideal_Throughput_inst == 0 then
24         Use compute_units_inst of inst_type to rank inst_type in list_perf;
25     end
26     else
27         Use new value of Ideal_Throughput_inst to rank inst_type in list_perf;
28     end
29 end
```

b) Data smoothing: The *Profiler* performs a smoothing technique over the profiling data of each instance to remove the noise generated by traffic spikes. As pointed out in [20], when hosting web applications, sudden changes in the workload and interferences due to virtualization or OS activities may affect the precision of the profiling process. Hence, we decided to *smooth the profiling data* during the latest hour (or older if there is not enough data) to identify the performance capacity of each allocated VM instance type. This technique allows to identify the optimized throughput of each instance while enforcing the performance requirements and avoiding CPU saturation. In particular, the *Profiler* extracts the smoothed 75th and 25th percentiles from the response times below the SLO in correspondence with the *SLO threshold*; and 75th percentile from the percentage of CPU usage data-points. (See Lines 3-12). Note that, we only use the 75th percentile from the CPU usage due to lower values in the CPU usage do not imply a system free of SLA violations. As an example extracted from our experience, Amazon EC2 "*m1.small*" instances can throw violations even under percentages of CPU usage lower than 25%.

Figure 2 shows the profiling data and percentiles for one "m1.small" EC2 instance type during one hour. The gray areas represent the ranges of response times and CPU usage comprised by the percentiles. Black circles contain all data points that will be used to calculate the optimized throughput. In Figure 2, some data points are excluded as they identify periods of time on which resource suffered from under-utilization or over-utilization (denoted by white areas). Similarly, a short

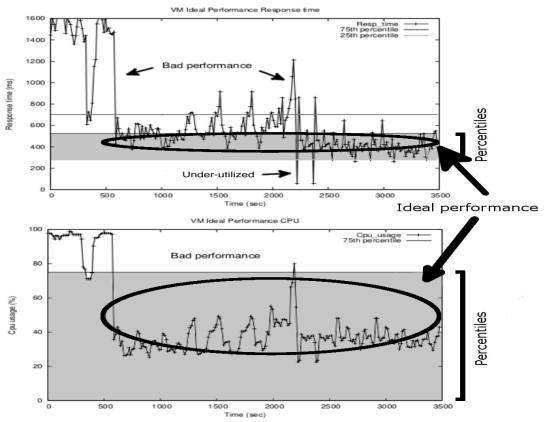

Fig. 2: Profiling data and percentiles of m1.small.

number of data points are comprised between the percentiles for the "m1.small" instance. Its poor hardware configuration makes this instance type more vulnerable to sudden changes in the workload.

c) Instance performance classification: The *Profiler* classifies the different instance types depending on its computing capacity. Using the profiling smoothed data of each instance type (refer to Lines 15-17), the *Profiler* computes a factor, named $Ideal\_Throughput_{inst}$, as the amount of clocks required to serve a specific amount of requests (clocks/requests) for one instance. In Equation 1, the $\%CPU\_usage_{inst}$ represents the average of percentage of CPU usage (clocks) and $Num\_requests_{inst}$ the average of request rate (requests) consumed during the last hour. Using this formula, we initially assume requests have the same complexity (processing time). Nevertheless, the *request heterogeneity* will be taken into consideration when calculating the workload requirements, as explained in Section IV-B1.

$$
\begin{aligned}
\%CPU\_usage_{inst} &= \frac{\sum_{i=1}^{N} (\%cpu\_usage\_data_i)}{N} \\
Num\_requests_{inst} &= \frac{\sum_{i=1}^{N} (req\_rate\_data_i)}{N} \\
Ideal\_Throughput_{inst} &= \frac{\left(\frac{\%CPU\_usage_{inst}}{100}\right)}{Num\_requests_{inst}}
\end{aligned}
\tag{1}
$$

The $Ideal\_Throughput_{inst}$ factor gives an estimation of the optimized performance capacity of one VM instance when processing the current workload. This formula only works in a stable state when the system is not thrashing, thereby we measure it using the percentiles. Based on the value of this factor per-instance, the *Profiler* classifies the different instance-types based on its computing capacity when running an application. (See Lines 22-27). The resulting classification gives an interesting feedback to the Scaler component, which is now able to identify the capacity of each instance type, and consequently to choose an optimal scaling plan. Note that, a lower value of the $Ideal\_Throughput_{inst}$ indicates a higher performance capacity. Initially, there are not instance profiles due to the lack of monitoring data, thereby the *Scaler* uses the number of compute units (see Table I) per instance as a priori classification of their performance capacities. Our system only considers the compute units, but memory or network

bandwidth can be also taken into consideration to classify the VM instances.

Finally, this profiling process allows to define a profile per instance type, thus facilitating the selection of an appropriate scaling plan that will satisfy the QoS requirements. Even though this profiling approach assumes that all the resources of the same type perform similarly, it can be easily extended to profile the allocated resources independently of its type.

B. Scaling plan decision-making

To discover the appropriate scaling plan, we should take into consideration three aspects: the workload requirements, the resource heterogeneity and the customer preferences. An exhaustive analysis of the current workload allows to identify the future demand that will determine the new configuration to use. On the other hand, cloud infrastructures offer a wide range of different hardware configurations that can be combined to better satisfy the requirements. Based on the customer preferences, the selection of one configuration or another can expose the application to availability or performance issues, specifically when handling workload variations or other traffic anomalies. Hence, the selection of an appropriate scaling plan becomes crucial to mitigate these penalties.

1) Analysis of the workload requirements: Prior to any resource selection, the *Scaler* has to measure the requirements of the current workload taken into account the traffic diversity of web applications. Traditional scaling systems gather monitoring data about the request volume or total percentage of CPU usage consumed to identify the workload requirements. However, the workload of web applications is highly heterogeneous being defined by sudden changes in the traffic as well as constantly-changing request mixes. As a consequence, our system computes the workload complexity considering both the total percentage of CPU usage and request rate served by the current scaling plan, as factors that affect to the final response time. They provide enough information to detect the causes of a performance degradation in web applications [12].

The *workload complexity* is a factor that identifies how much the current workload is affecting to the ideal performance behavior of the allocated resources. To calculate the *workload complexity* (denoted by $Workload_{complex}$), we first analyzes the monitoring data gathering the request rate (denoted by $Num\_reqs\_server_i$) and total percentage of CPU usage (denoted by $\%CPU\_usage\_server_i$) consumed by the allocated *N* resources during the monitoring window. Secondly, unlike the $\%CPU\_usage_{inst}$, we calculate the expected usage of CPU of each resource *i* (denoted by $CPU\_usage\_expected_i$) using its current request rate (denoted by $Num\_reqs\_server_i$) and its *optimized throughput* (denoted by $Ideal\_Throughput_{inst_i}$). By doing so, we are able to include into the $Workload_{complex}$ all the degradations that affect the ideal performance of the application. Finally we compute the $Workload_{complex}$ factor using the sum of CPU usage consumed to handle the current workload and the sum of expected CPU usage of all the resources, as illustrated in Equation 2.

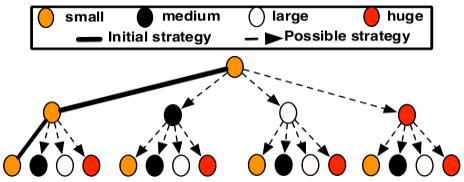

Fig. 3: Example of a decision tree created from an existing resource combination.

$$CPU\_usage\_expected_i = Num\_reqs\_server_i * Ideal\_Throughput_{inst_i}$$
$$Workload_{complex} = \frac{\left(\frac{\sum_{i=1}^{N} \%CPU\_usage\_server_i}{100}\right)}{\sum_{i=1}^{N} CPU\_usage\_expected_i} = \frac{(clocks)}{(clocks)} \quad (2)$$

Additionally, the future service demand of the application, in terms of total percentage of CPU usage and request rate consumed by the resources, can be estimated by the *Predictor* component for the next monitoring windows (in our experiments 30min). This enables to select in advance a scaling plan that will handle future fluctuations in the workload, and thereby saving cost. Note, a detailed explanation of how the *Predictor* component estimate future workload is not in the scope of this paper.

2) Calculating the different scaling plans: To decide which type and amount of VM instances to add or release, the *Scaler* uses an optimal decision tree that facilitates the discovery of all the possible scaling strategies. By going through this tree, our system evaluates all the possible resource combinations using horizontal scaling operations. As shown by Figure 3, in the scaling decision tree, each node represents each type of hardware configuration offered by a cloud infrastructure, where the nodes linked by a straight line represent the current scaling strategy. While the branches, denoted by a dotted line, define all the possible resource combinations to provision. In the following, we will use scaling plan or strategy indistinctly.

During the selection process of possible scaling strategies, the *Scaler* uses Equation 3 to compute how a strategy will distribute the current workload across the different resources taken into account their performance capacities. Equation 3 defines a factor, called *RU_Strategy*, that determines the resource utilization of a strategy when handling the current or future workload (depending on how the workload requirements were obtained). According to that, the *Scaler* selects a strategy that uses *N* resources to distribute the Num_reqs_{total} (total requests served by the current resources or estimated by the *Predictor*) with a workload complexity $Workload_{complex}$ across *N* instance types with different optimized throughputs $Ideal\_Throughput_{inst_k}$. As a function of the $Ideal\_Throughput_{inst}$ and Num_reqs_{total}, the $RU\_Strategy$ measures the resource utilization (clocks) to process the current workload when using a specific resource combination.

RU_Strategy = Resource utilization of the strategy

$$RU\_Strategy_i = \frac{\sum_{k=1}^{N} \left(\left(\frac{Num\_reqs_{total}}{N} * Workload_{complex} \right) * Ideal\_Throughput_{inst_k} \right)}{N} \quad (3)$$

Note that, the search space of plausible strategies can be as large as the number of available hardware configurations, and allocated resources of the current scaling strategy. It makes it difficult to find the best resource combinations in a reasonable short period of time, so that the *Scaler* defines two policies to filter the plausible strategies based on the following criteria:

- Define the maximum resource utilization consumed by a strategy when processing a particular workload (denoted by $Max\_RU\_Strategy$). It limits the number of possible plans where $RU\_Strategy_i \leq Max\_RU\_Strategy$. $Max\_RU\_Strategy$ specifies the maximum resource utilization at which the application starts to experience SLO violations or over-utilization of the resources. Its value can be extracted either from the Amazon EC2 recommendations for the maximum percentage of CPU usage (always lower than 75%), or based on the monitoring data (gathered by the monitoring engine) at which the application starts to experience SLO violations. As a result of this policy, the proposed scaling plans optimizes the current performance below values causing performance degradations.
- Include a cost policy to avoid choosing wasteful scaling strategies. Considering the pricing model of cloud providers that charges users on a per-hour basis, the *Scaler* includes a cost policy that rejects strategies releasing resources which have been recently started (5min $<$ time to the end of its hour $<$ 20). The *Scaler* releases resources which are closed to the hour are free under the cloud pricing model, and there is no gain from terminating them before this hour price boundary.

These two policies avoid to trigger new scaling actions in short time intervals (e.g. within less than one hour), and consequently reduce the operational cost.

Example: Figure 4 shows three possible scaling plans obtained by using our decision tree and Equation 3 (denoted by *Eq(3)*) over the initial strategy illustrated in Figure 3. This initial resource combination is composed of three *small* instance types which are not sufficient to handle the current workload. Using this strategy as reference and horizontal scaling operations, our algorithm adds and/or releases resources (leafs to the tree) as long as the resulting $RU\_Strategy$ (calculated for the new resource combination) is lower than $Max\_RU\_Strategy$ and do not propose to release resources recently added. As a result, three plans (trees) are proposed using different resource configurations and satisfying the performance requirements to handle the workload. This discovery process stops when all the possible combinations have been proved or there are not more combinations satisfying the aforementioned filtering criteria.

In summary, using the Equation 3, the Scaler is able to answer to the question *"How many and which instance-types to provision?"*.

3) Selecting the optimal scaling plan: Once the *Scaler* obtains a list of plausible scaling strategies, it computes the cost of applying each possible strategy how the cost incurred by its SLO guarantee and the infrastructure cost. The infrastructure cost (denoted by *Infra_cost*) specifies the cost required to provision a scaling plan for the next hour. While the SLO guarantee (denoted by *SLO_guarantee*) indicates the vulnerability of a scaling plan to experience SLO violations.

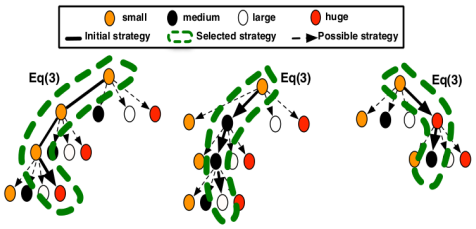

Fig. 4: Example of three selected scaling plans that satisfy the filtering criteria.

As defined in the Equation 4, the *SLO_guarantee* cost is calculated given the $RU\_Strategy_i$ of a strategy i and the $Max\_RU\_Strategy$. Obviously, higher values in the $RU\_Strategy$ imply an increment in the probability of having SLO violations under traffic fluctuations, so the *SLO_guarantee* will increase as well.

$$
\begin{aligned}
Infra\_cost_i &= \sum_{k=1}^{N} (instance\_price_k) \\
SLO\_guarantee_i &= \left(\frac{RU\_Strategy_i}{Max\_RU\_Strategy} \right) \\
Cost\_strategy_i &= \frac{SLO\_guarantee_i}{Infra\_cost_i}
\end{aligned}
\tag{4}
$$

The criteria for the selection of the optimal scaling plan is configurable, and thereby it can adapted to the customer preferences. For instance, large enterprises need to provide high availability and performance to their clients, disposing of a generous budget for such a mission. Therefore, based on the customer-tradeoff between performance/cost, the *Scaler* component will choose to provision one scaling plan or another depending on the value of its *Cost_strategy*.

To facilitate the specification of the performance/cost preferences to the customers, our system provides different predefined criteria that adapt the scaling decisions to these requirements. These selection criteria are defined thanks to a *metal classification* that classifies the different QoS levels based on the performance/cost preferences of the customers. Initially, we define three different QoS levels for customers following this *metal classification*. These three QoS levels namely, Gold, Silver and Bronze minimize the SLA violations with a different infrastructure cost.

- A *Gold* QoS level identifies those customers that pay more in order to get the best service at the cost of some extra over-provisioning. The *Scaler* then selects the strategy that has the highest *Cost_strategy*.
- A *Silver* QoS level includes those customers that prefer to get good availability but with a reasonable operational cost. As such, the *Scaler* calculates the median *Cost_strategy* of all the plausible strategies, and uses it as the strategy to provision.
- A *Bronze* QoS level represents those customers who are willing to obtain a reduced, but acceptable, SLO fulfillment but with very little over-provisioning; and thereby a low operational cost. The *Scaler* uses as the optimal strategy that one with the lowest *Cost_strategy*.

In our experiments, we will use this *metal classification* to show the benefits/drawbacks by selecting different scaling plans based on the QoS level selected by the customer.

V. EVALUATION

To evaluate our autoscaling system described above, we ran experiments on two infrastructures: a private one (the DAS-4, a multi-cluster system hosted by universities in The Netherlands [21]) and a public one (the Amazon EC2 cloud). The goal of our experiments was to evaluate and compare different scaling plans by how well they react under sudden workload changes (e.g. a sudden rise in requests received after a network outage). These scaling plans are chosen by our system to satisfy different QoS levels according to the *metal classification*. In particular, our evaluation focuses on SLO fulfillment, performance stability and amount and type of resources allocated to handle a traffic variation.

Testbed configuration: As a representative scenario, we deployed the MediaWiki [22] application using ConPaaS, on both public and private environments. To run the MediaWiki application, we used the WikiBench benchmark [23]. This benchmark uses a full copy of Wikipedia as the web application, and replays a fraction of the actual Wikipedia's access traces.

Given such a scenario, ConPaaS provides support for all the required services to host a copy of Wikipedia: a PHP web hosting and a MySQL service. The MySQL service is loaded with a full copy of the English Wikipedia articles as of 2011, which has a size of approximately 40GB. In the PHP service, the configuration was composed of one load balancer, one or more static web server and one or more PHP servers. For these experiments, we focus on the elasticity of the PHP tier, and in particular how the number and type of VM'es hosting PHP servers will change on demand. For monitoring-data analysis, ConPaaS provides a monitoring component based on Ganglia [24], a scalable distributed monitoring system. Moreover, for these experiments, we configured a monitoring window of 5 minutes, a minimum interval of 10 minutes for cool down between scaling actions and a fixed SLO of 700 milliseconds at the service's side.

Workload trace: Instead of using unrealistic and synthetic workloads generated by benchmark tools, such as TPC-W [25], we ran the Wikibench tools with a 10% sample of a real Wikipedia access trace from 2011. This trace contains requests for static pages as well as dynamic PHP requests. Figure 5 plots the PHP workload sampled from one trace, as the number of PHP requests per minute during approximately three days. One interesting aspect that we noticed about this trace is a sudden drop and rise in the load during a period of time around 17 minutes. By looking at the traffic logs, it seems that the access traces were not written into the logs due to an anomaly probably caused by a network outage or power shutdown.

A. Public Cloud

Initially, our experiments on EC2 used *m1.small* instances for the PHP service and one *m1.large* instance for the MySQL service. Table I details the hardware configuration and cost per-hour of the different sizes of EC2 instances employed during the experiments.

[1]A EC2 compute unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

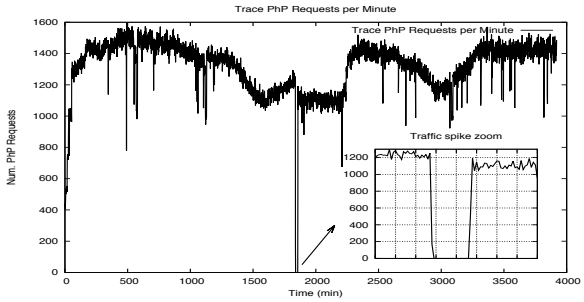

Fig. 5: Wikipedia trace workload.

Amazon EC2		
Size	**Configuration**	**Cost/hr**
m1.small	1-ECU [1] – 1.5Gb RAM	0.06$
m1.medium	2-ECU – 4Gb RAM	0.12$
c1.medium	5-ECU – 3Gb RAM	0.145$
m1.large	4-ECU – 8Gb RAM	0.24$
DAS4		
Size	**Configuration**	**Cost/hr**
small	1-core 2.4Ghz – 1Gb RAM	0.05$
medium	4-core 2.4Ghz – 4Gb RAM	0.23$
highcpu-medium	6-core 2.4Ghz – 3Gb RAM	0.28$
large	8-core 2.4Ghz – 8Gb RAM	0.46$

TABLE I: EC2 and DAS4 instance type characteristics.

1) Performance stability and SLO fulfillment: Figure 10 represents the degree of SLO fulfillment of three different QoS levels, indicating the response time data-points obtained during the execution of the Wikipedia workload trace. In particular, Figure 10 focus on the interval of time in which the outage occurred. The results show how the system scales back when the request volume drops, and quickly scales in when the load increases suddenly. As shown on Figure 10, the degree of SLO fulfillment gradually improves depending on the tradeoff performance/cost of each QoS level chosen to handle this traffic variation. Thus, the *Bronze* and *Silver* QoS levels need between 150 and 200 minutes to reach the performance stability, but the *Bronze* level experiences more SLO violations. While the *Gold* level requires less time to handle this traffic spike reducing at the maximum the number of violations, as detailed in Table II. The selection of different performance/cost configurations, and consequently different QoS levels, has important consequences, specially when a non-fulfillment of the requirements affects to the revenues of the customer.

2) Resource utilization: To better understand the impact of selecting a scaling plan adapted to the customer QoS preferences, we shall also focus on the resource consumption illustrated on Figure 11. While satisfying the requirements, the allocation of poor hardware configurations can easily expose the system to performance degradations, and consequently increases the time required to stabilize the system performance. This can be seen on Figure 11(left), in the time interval between t=338min and t=400min. The *Bronze* level provisions a low-cost configuration, that compromises the performance and raises an important amount of SLO violations during a long period of time. Unlike the *Silver* and *Gold* levels utilize along the whole execution powerful configurations that enables to minimize these degradations. Specifically, the *Gold* level allocates a costly resource combination (mainly composed of

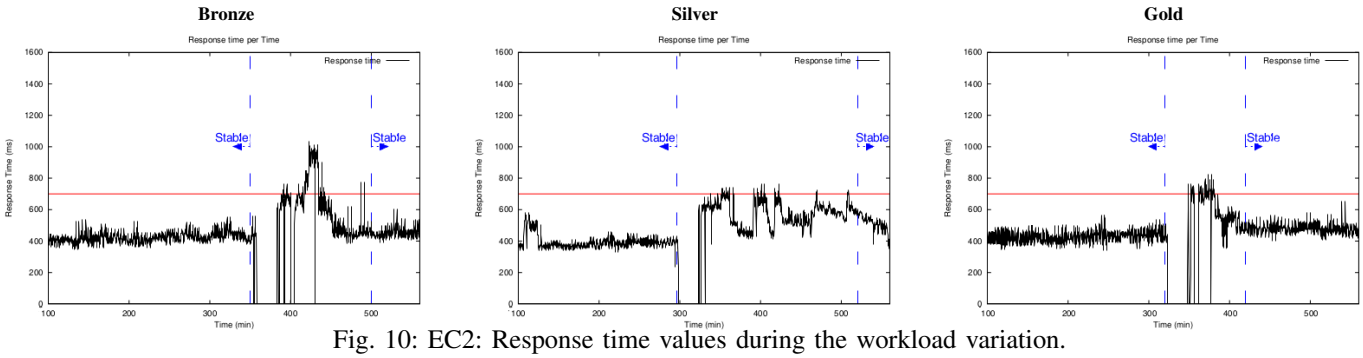

Fig. 10: EC2: Response time values during the workload variation.

m1.large instance types) which is able to handle such traffic changes while avoiding the under-utilization of the resources. This lack of under-utilization can be depicted in Figure 10, where the processing time remains close to the SLO after the new scaling plan is provisioned.

3) Analysis of results: The results of the three runs are summarized in Table II, where we show the amount of SLO violations, the number of provisioning decisions taken by the autoscaling system and an estimation of the cost of running the VMs in Amazon EC2.

As depicted on Figure 10 and Figure 11, the use of *Bronze* settings shows the vulnerability of selecting a configuration with low operational cost, as it easily becomes unstable when handling bursty workloads. Hence, poor hardware configurations increase the probability of experiencing SLO violations under sudden traffic spikes, outages or other traffic anomalies.

When using a mixed configuration like *Silver*, the *Scaler* explores the tradeoff between performance capacity and cost by combining different hardware configurations to handle the load with an acceptable cost. However, the discovery of such resource combination can be hard-to-find, affecting to the stability of the application due to frequent scaling actions, as shown on Figure 10(center). Finally, with a *Gold* level, the Scaler provisions resources that offer better performance in exchange for a higher infrastructure cost. With little over-provisioning, this class of customer configuration enforces the QoS requirements and improves considerably the stability and performance of the system by distributing the new request volume across the most powerful resources. Unfortunately, as illustrated in Table II, the *Gold* settings also experiences few SLO violations due to scheduling and booting operations (e.g. turning machines ON/OFF) when provisioning new EC2 resources. These operations can take around 3-6 minutes to be completed, thus causing SLO violations during this period of time.

Note that due to synchronization time lag between the scaling actions and the displayed information in the logs, time values are shifted for a few minutes during the whole execution, as shown in Figure 10 and Figure 11.

B. Private Cloud

Our experiments on DAS-4 rely on OpenNebula as IaaS [26]. To deploy the Wikipedia services, we initially provisioned *small* instances for the PHP service and one *large* instance for the MySQL service. Additionally, for these

QoS level	SLO Violations	Decisions	Cost
Bronze	117	6	6.9 $
Silver	74	14	8,6 $
Gold	63	4	10.8 $

TABLE II: Analysis of results on EC2

experiments, we specified a cost per-hour for each one of the available resources, and extended from three to five the initial QoS levels classification: *Platinum, Gold, Silver, Bronze* and *Copper*. Additionally, to select the strategies of these new five QoS levels, the *Scaler* additionaly calculates the 25th and 75th percentiles of the list of plausible strategies. Table I detailed the hardware configurations of the VM sizes in DAS-4.

1) Performance stability and SLO fulfillment: Figure 12 shows the performance behavior of the scaling plans selected for five types of QoS levels. As depicted in Figure 12, the performance stability is mainly affected by two aspects: frequent scalings operations and traffic spikes. Thus, the *Platinum* level presents an unstable performance pattern, due to numerous provisioning actions aiming to find the optimal scaling plan. In contrast, the performance behavior of the *Gold* level shows no oscillations even during the variation, which can be explained by the selection of an appropriate resource combination that do not under-utilizes the resources.

Regarding the traffic spikes caused by the new workload, it is clear how configurations, like *Copper, Bronze and Silver*, are vulnerable to traffic changes, thus causing SLO violations. As shown on Figure 12, the scaling decisions that prioritizes the infrastructure cost than performance are directly affected by the effects of the request volume increment. This can be seen in the time interval between t=510min and t=536min for the *Copper* level, and between t=498min and t=519min for the *Bronze* level, respectively. By selecting an appropriate scaling plan, the performance stability and SLO fulfillment can be better enforced without dramatically raising the cost.

2) Resource utilization: As explained above, frequent or inappropriate scaling actions can affect to the performance stability. This can be seen in Figure 13, where the *Copper* and *Bronze* levels attempt to scale back the system when the load drops, but the resulting resource combinations cannot handle the new request volume, and the system is scaled up again. At difference, the *Gold* level only de-provisions low-powered resources during the outage, which enables to distribute the new load among the current resources, while new resources are allocated to avoid future resource over-utilization.

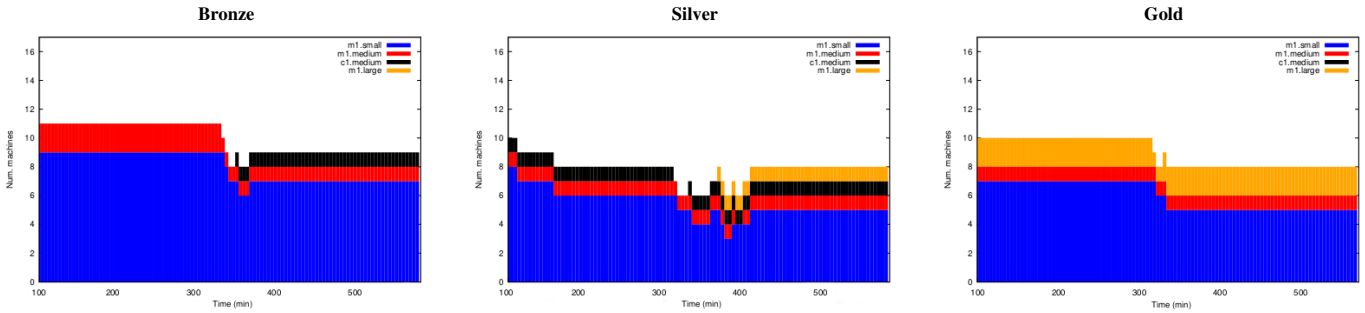

Fig. 11: EC2: Cloud instances provisioned to handle the workload variation.

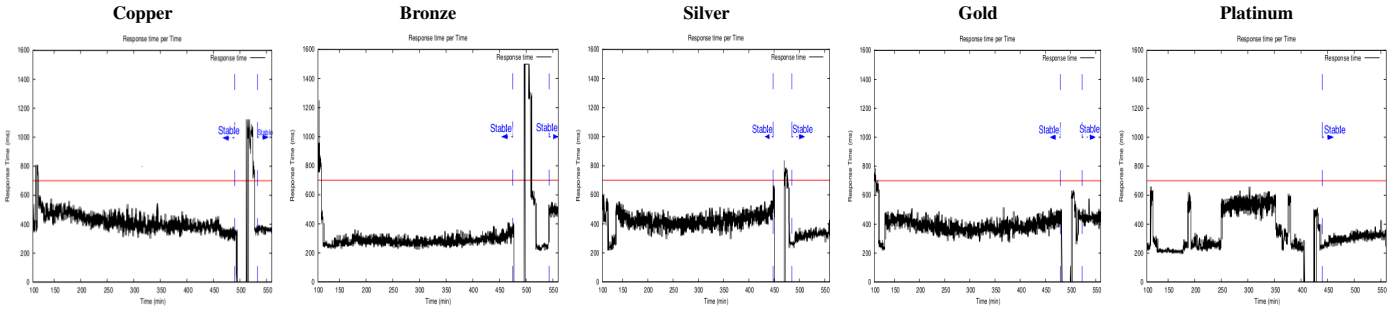

Fig. 12: DAS4: Response time values during the workload variation.

QoS level	SLO Violations	Decisions	Cost
copper	138	6	6.4 $
bronze	129	8	9.5 $
silver	82	10	10.4 $
gold	0	4	8.3 $
platinum	0	24	12.8 $

TABLE III: Analysis of results on DAS4

Once the outage occurred, another interesting aspect to notice is the type of resources employed to minimize the number of SLO violations. While the *Silver* level provisions powerful machines (e.g. *highcpu-medium*) that quickly mitigate the degradations, the *Copper* level allocates *small* resources that employ more time to handle the workload. When using the *Gold* or *Platinum* settings, the system barely appreciates the changes in the workload, as shown on Figure 12. It demonstrates the importance of using autoscaling system that exploits the resource heterogeneity of cloud infrastructures to satisfy customer QoS preferences.

On the other hand, it is remarkable to point out that the amount of allocated resources may not be a subject of comparison between QoS configurations. The poor performance isolation and the variable network bandwidth can affect to the performance of the resources, and consequently the amount of allocated resources can vary even for the same QoS level.

3) Analysis of results: Table III summarizes the results of these experiments, where we show the amount of SLO violations, provisioning decisions and total infrastructure cost of the allocated resources.

Firstly, we noticed a progressive minimization of the SLO violations related to the QoS level, and therefore due to the hardware configuration provisioned at the moment of the traffic increased. During the outage, the *Copper* and *Bronze* levels used poor hardware configurations which were insufficient to handle the new workload. In contrast, the *Gold* and *Platinum* levels allocated powerful and suited resources that handled the new load successfully.

Secondly, it is remarkable how depending of the scaling plan the required time to stabilize the system vary, and as a consequence the amount of SLO violations. An explanation comes from the quickly allocation of powerful resources, instead of a gradual allocation of standard or less-powered resources. This decision has special importance in cloud platforms aiming to reduce the penalties paid due to SLO violations or server errors. Finally, we also note some performance unstability with the *Platinum* level that was provoked in some situations due to wrong or delay balancing of the incoming traffic across the heterogeneous resources.

Note that due to synchronization time lag between the scaling actions and the displayed information in the logs, time values shown in Figure 12 and Figure 13 are shifted for a few minutes during the whole execution.

VI. Conclusion

Nowadays cloud infrastructures offer a plethora of distinct hardware configurations for rent and price. We argue that autoscaling systems can use this diversity to better fulfill the SLA requirements without dramatically raising the operational cost. This has special importance for enterprises where a decrease in the user base leads to a reduction in revenue, or for cloud providers where penalties paid due to SLO violations also have a revenue impact. We proposed a novel autoscaling approach for web applications that allocates the

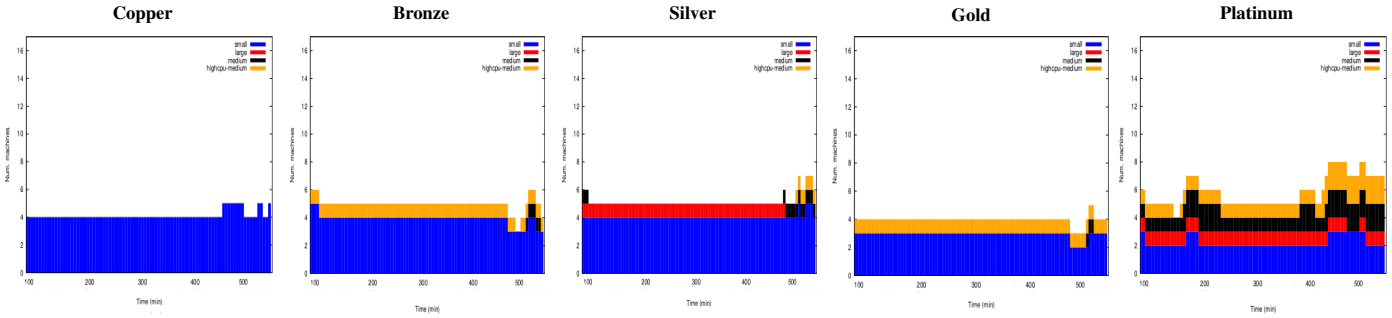

Fig. 13: DAS4: Cloud instances provisioned to handle the workload variation.

most appropriate resource combination to enforce the QoS requirements even under spiky workloads. This system benefits from cloud heterogeneity and online profiling techniques to find the suited scaling plan. Moreover, a metal classification was also proposed to allow customers to tune its own tradeoff *SLO guarantee/cost* that better adapt to their requirements. We implemented and successfully integrated our autoscaling system in a real cloud platform. In *multiple clouds*, we evaluated our system in a realistic scenario showing the benefits by selecting one QoS level or another (*e.g. Copper-Gold*) to mitigate the performance degradations caused by traffic spikes. In the future we intend to extend our system by supporting a wider number of hardware configurations and a more efficient dynamic load balancing mechanism.

ACKNOWLEDGMENTS

This work was 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

[1] Case studies of application hosting in AWS., http://aws.amazon.com/solutions/case-studies/.

[2] L. Rawlinson and N. Hunt, "Jackson dies, almost takes internet with him," *CNN*, Jun 2009.

[3] Y. Baryshnikov, E. Coffman, G. Pierre, D. Rubenstein, M. Squillante, and T. Yimwadsana, "Predictability of web-server traffic congestion," in *Proc. in. WCW*, 2005, pp. 97–103.

[4] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1–39, Mar. 2008.

[5] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "DejaVu: accelerating resource allocation in virtualized environments," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 423–436, 2012.

[6] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *Proc. IEEE CLOUD*, 2011.

[7] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Proc. NOMS'12*, 2012, pp. 204–212.

[8] C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hegde, and A. Srivastava, "A pluggable autoscaling service for open cloud PaaS systems," in *Proc. UCC*, 2012, pp. 191–194.

[9] J. Dejun, G. Pierre, and C.-H. Chi, "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*, 2009.

[10] N. Kaviani, E. Wohlstadter, and R. Lea, "Profiling-as-a-Service: adaptive scalable resource profiling for the cloud in the cloud," in *Proc. intl. conf. on Service-Oriented Computing*, 2011, pp. 157–171.

[11] N. Roy, A. Dubey, A. Gokhale, and L. Dowdy, "A capacity planning process for performance assurance of component-based distributed systems," in *Proc. Intl. Conf. on Performance Engineering*, 2011, pp. 259–270.

[12] S. Dutta, S. Gera, A. Verma, and B. Viswanathan, "Smartscale: Automatic application scaling in enterprise clouds," in *Proc. IEEE CLOUD*, 2012, pp. 221–228.

[13] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proc. ICDCS*, 2011, pp. 559 –570.

[14] G. Pierre and C. Stratan, "ConPaaS: a platform for hosting elastic cloud applications," *IEEE Internet Computing*, vol. 16, no. 5, pp. 88–92, 2012.

[15] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: A distributed resource performance forecasting service for metacomputing," *Journal of Future Generation Computing Systems*, vol. 15, pp. 757–768, 1999.

[16] S. Muppala, X. Zhou, L. Zhang, and G. Chen, "Regression-based resource provisioning for session slowdown guarantee in multi-tier internet servers," *Journal of Parallel and Distributed Computing*, vol. 72, no. 3, pp. 362–375, Mar. 2012.

[17] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proc. CLOUD*, Jul. 2011, pp. 500–507.

[18] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan, "Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers," in *Proc. intl. conf. on Services Computing*, Jul. 2010, pp. 514–521.

[19] A. Chandra, W. Gong, and P. Shenoy, "Dynamic resource allocation for shared data centers using online measurements," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 300–301, Jun. 2003.

[20] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah, "Hybrid resource provisioning for minimizing data center SLA violations and power consumption," *Sustainable Computing: Informatics and Systems*, vol. 2, no. 2, pp. 91–104, Jun. 2012.

[21] "Advanced School for Computing and Imaging (ASCI), The Distributed ASCI SuperComputer 4." [Online]. Available: http://www.cs.vu.nl/das4/

[22] Wikipedia Foundation, mediaWiki: a free software wiki package. http://www.mediawiki.org.

[23] E.-J. van Baaren, "Wikibench: A distributed, wikipedia based web application benchmark," Master's thesis, VU Amsterdam, 2009.

[24] Ganglia monitoring system,. [Online]. Available: http://ganglia.sourceforge.net/

[25] Transaction Processing Performance Council, http://www.tpc.org/tpcw/.

[26] B. Sotomayor, R. Montero, I. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, Oct. 2009.