Modeling Performance of Elasticity Rules for Cloud-based Applications

Basem Suleiman Srikumar Venugopal

School of Computer Science and Engineering,
University of New South Wales, Australia
{basems,srikumarv}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201323
September 2013

Abstract

Many IaaS providers, e.g., Amazon Web Services, allow cloud consumers to define elasticity (or auto-scaling) rules to dynamically allocate and release computing resources on-demand and at per-unit-of-time costs. Modern enterprises are increasingly deploying their applications, e.g., internet banking and financial services, on such IaaS clouds so their applications can inherently become self-elastic to meet its variable workload. Defining elasticity rules for such applications, however, remains as a key challenge for cloud consumers as it requires choosing appropriate threshold values to satisfy desired applications and resources metrics. Achieving this empirically is expensive as it requires running large amount of empirical testing and analysis in real cloud environments. In this paper we propose novel analytical models that capture core elasticity thresholds and emulate how elasticity works. The proposed models also approximate primary metrics including CPU utilization, application's response time and servers usage cost for evaluating elasticity rules' performance. Based on our models, we develop algorithms that decide when and how to scale-out and scale-in based on CPU utilization and other thresholds and to estimate servers cost resulted from scaling actions. We validate the simulation of our elasticity models and algorithms with different elasticity rules' thresholds against empirical data resulted from experiments with the same elasticity rules thresholds with TPC-W application on Amazon cloud. The simulation results demonstrated reasonable accuracy of our elasticity models and algorithms in approximating CPU utilization, application's response time, number of servers and servers usage costs.

```
Monitor CPU Utilization (U) every 1 min.

IF $U > 80\%$ FOR 7 min.
 Add 1 server of small capacity //Scale out
 Wait 5 consecutive 1 min. intervals

IF $U < 30\%$ FOR 10 min.
 Remove 1 server of small capacity //Scale in
 Wait 7 consecutive 1 min. interval
```

Figure 1.1: Example of an Elasticity Rule

1 Introduction

Modern enterprise applications such as internet banking, retail and financial applications are increasingly provisioned as internet or cloud-based services. This is realized by deploying such enterprise applications, or parts of its business processes, on an Infrastructure as a Service (IaaS) cloud such as Amazon's Elastic Cloud Compute (EC2) and GoGrid's Cloud Hosting. The major driving factor of such IaaS clouds is elasticity or auto-scaling; a service quality that allows enterprises, or cloud consumers [1], to dynamically acquire (scale-out) and release (scale-in) computing resources (through internet-based self-service) on-demand and at per-unit-of-time service cost (typically per hour) [1, 2]. The IaaS Elasticity is crucial as it enables enterprises to make their applications self-elastic so it can meet its variable workloads and application's performance and cost objectives.

1.1 Controlling Elasticity Challenges

Many IaaS providers enable cloud consumers to control elasticity through implicit or explicit policies or rule-based mechanisms. For example, Amazon Auto Scaling [2] allows cloud users to set elasticity rules that define actions to be executed in response to triggers that are defined by users based on thresholds over measurable parameters. Similar facilities are provided by Microsoft through a library called the Windows Azure Autoscaling Block (WASABi [3] and by third party cloud management suites such as Scalr [4].

In such elasticity rules, a number of thresholds, e.g., CPU utilization thresholds, form the basis for the elasticity service to decide when to scale-out or to scale-in computing resources. Figure 1.1 presents an example of such elasticity rules. Here, the rule triggers if CPU utilization increases above 80% to scale-out or decreases below 30% to scale-in. Changing one or more threshold values can influence when a scale out/in action is triggered and therefore directly influence application's and performance and cost requirements [3]. For instance, setting low value for CPU utilization threshold can improve application performance

[1]In this context we use the term cloud consumers to refer to those enterprise owners who deploy their application on a IaaS cloud.

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

[3]http://msdn.microsoft.com/en-us/library/hh680945(v=pandp.50).aspx

[4]http://scalr.com/

but at the expense of high server usage costs and under-utilized servers [3]. In contrast, setting high value for the CPU utilization threshold can reduce servers cost but at the expense of poor application performance due to potential over-utilization of servers [3]. Both of these can have severe financial consequences for applications facing dynamic workloads and bound to rigid service-level objectives. Empirically verifying the thresholds for different types of expected workloads can also be expensive due to the expenditure involved in leasing cloud resources.

Existing auto-scaling services such as Amazon auto-scaling do not provide ways to support cloud consumers in evaluating and analysing the impact of changing elasticity thresholds on performance and cost metrics. Achieving this empirically is expensive as it requires exhaustive performance-cost testing and analysis in real cloud production environments. Many research studies ([4, 5, 6, 7, 8]) have focused on proposing dynamic provisioning mechanisms to efficiently allocate servers and meet application's performance targets. These techniques try to minimize server costs of virtualized data centers which are useful for cloud providers but not for cloud consumers. Other studies, e.g., [9, 10], provide models for predicting performance and cost of cloud applications. However, these studies do not consider modeling elasticity rules and the impact of tuning elasticity thresholds on applications and resource metrics.

1.2 Research Contributions

In this paper we present analytical models, based on queuing theory, that capture the core elements and emulate the behaviour of elasticity rules at the application tier of multi-tier applications deployed on IaaS cloud. Particularly, our models can approximate the values of crucial elasticity rules' metrics over time including CPU utilization, application's response time, number of servers and server usage costs. Based on these models, we also develop algorithms which simulate CPU-based scale-out and scale-in logic, i.e., when and how to trigger scale out/in actions, based on CPU utilization and other thresholds. In addition, we develop an algorithm that approximates the cost of server usage resulted from scale-out and scale-in actions and based on per-unit-of-time cloud leasing model.

Using this model, we have conducted a number of simulation experiments with different thresholds for elasticity rule and we analysed its performance in terms of CPU utilization, application response time, number of used servers and servers cost. We validated the simulation data against empirical data resulted from running the same elasticity rules experiments with TPC-W benchmark on Amazon EC2. Our empirical validation demonstrates reasonable accuracy of our elasticity models and algorithms in detecting CPU utilization, application's response time, number of servers and servers usage costs.

Our elasticity models and algorithms provide a tool that can support enterprises to perform performance-cost analysis and define appropriate elasticity thresholds to meet their resource and application's performance and costs metrics specified within service-level and budget constraints.

The rest of the paper is organized as follows. Section 2 compares and discusses the related studies to our work. Section 3 introduces the structure of elasticity rules. Then, it explains our analytical models and algorithms for CPU-based elasticity. Section 4 describes our experimental methodology and present the validation results of our simulation experiments against the empirical ones. Conclusions and future work are discussed in section 5.

2 Related Work

Due to its importance, a number of research studies have addressed elasticity, or auto-scaling, challenges in cloud environments. Particularly, studies such as [5, 6, 7, 8] proposed techniques for dynamic server provisioning for multi-tier internet applications running in data centers. One of the primary objectives of all these studies is to efficiently allocate servers for different internet applications hosted on a cloud environment while ensuring application's response time targets are satisfied. Xu *et. al.* [6] and Lama *et. al.* [5] proposed autonomic self-tuning resource controller that uses models based on fuzzy logic to achieve optimal resource allocation that satisfy application's response time requirements. The provisioning system proposed by Singh*et. al.* [7] considers non-stationarity in internet application workloads and proposed a clustering algorithm to detect the workload mix. They also used G/G/1 queueing model to determine number of servers needed to serve workload mix over time. The provisioning proposed by Malkowski *et. al.* [8] is based on automated learning and empirical models that require empirical measurements from previous application runs which are often hard to obtain. Ghanbari *et. al.* [11] also addressed the dynamic resource allocation but in a private cloud. Their focus is on optimal resource allocation through maximization of resource sharing (to minimize provider's costs) and meeting application's SLA requirements of all clients.

Our elasticity rules modeling share a common objective with these studies; i.e., when and how to provision and de-provision servers in cloud environments. Our work also simulates how elasticity rules works using queuing theory. However, none of these studies [5, 6, 7, 8, 11] considered modeling thresholds and its impact on performance-cost metrics from cloud consumers perspective. Particularly, our models and CPU-based elasticity algorithm can help cloud consumers to simulate different elasticity rules and perform off-line cost-performance analysis to decide on the best elasticity rules' thresholds.

Some research studies [12, 4, 10, 9] proposed analytical models, based on queuing networks, for performance and cost analysis of multi-tier internet applications [12, 4, 9] and for ERP applications [10]. Urgaonkar *et. al.* also proposed a dynamic provisioning techniques which were used by a server farm to determine capacity needed to serve application's workload and to predict the performance of an application running in certain server farm. In [4] they also proposed a predictive and reactive server provisioning techniques and analytical models based on queuing theory to capture multi-tier application's performance behaviour. Al-Azzoni *et. al.* [9] used service demand law and mean value analysis (MVA) algorithm to model CPU utilization and average response time respectively. They used the models to determine appropriate server capacity for web application running on Amazon cloud. Our analytical models have similar purpose of these models [4, 9, 10], to analyse the performance and cost of application's workloads. However, our modeling effort differs in terms of capturing elasticity metrics and thresholds and its influence on server's CPU utilization, application's response time, the number of servers triggered by an elasticity rule and servers cost. These studies have not modeled elasticity rules and its key

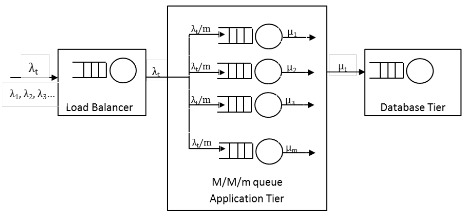
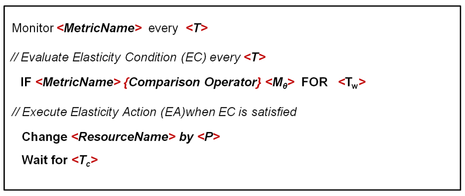


Figure 3.1: Queue Model of 3-tier Application Architecture



Monitor **<*MetricName*>** every **<*T*>**

// Evaluate Elasticity Condition (EC) every **<*T*>**

IF <*MetricName*> {*Comparison Operator*} <M_θ> FOR <T_w>

// Execute Elasticity Action (EA)when EC is satisfied

Change <*ResourceName*> *by* <*P*>

Wait for <T_c>

Figure 3.2: Key Elements of an Elasticity Rule

elements and the impact of tuning one or more threshold on the application performance and server costs.

Ghanbari *et. al.* [13] proposed an auto-scaling technique based on stochastic model predictive control to allocate and release computing resources in a way that application performance objectives are met and resource usage costs are minimized. Our elasticity rules modeling and algorithms are not aimed to perform auto-scaling at runtime on behalf of cloud consumers. Its main purpose is to support cloud consumers in performing performance-cost simulation and trade-off analysis with different elasticity rules to choose the best elasticity thresholds for their application workload.

3 Analytical Models for CPU-based Elasticity

A multi-tier architecture (often 3-tier; web, application and database tiers) is the most commonly used architecture in practice for web applications. In the context of our work, we assume that a multi-tier web application is deployed on a public cloud infrastructure such as Amazon EC2. We assume, as it is common, that each tier is deployed on separate servers in the cloud. We use queueing theory to model 3-tier application architecture as shown in Figure 3.1 [14]. Each server is represented as a queue at which requests are served. The web server balances the incoming requests across a pool of application servers in the application tier. Each application server sends one or more query to the database server to serve a request. The research presented in this paper focuses on modeling the elasticity of the application tier.

The general form of elasticity rules is shown in Figure 3.2. An elasticity rule consists of two main parts; a condition and action. The condition specifies a metric ($<MetricName>$) to be evaluated against a specific threshold value (M_θ). The metric can be any variable that is measurable through monitoring scripts provided by either by a cloud consumer or provider. In this paper, we are primarily concerned with monitoring application response time and CPU utilization metrics as these are the most important measures for determining the performance of an application. These metrics are measured at regular intervals (e.g. 1 minute), the length of which is denoted here as *Time Interval Length* (T).

Every interval, the measured value of the metric is compared against a user-defined threshold on the metric (denoted here as M_θ). If this condition holds for a time window (T_w), the action is triggered. T_w must be consecutive time intervals of length T. The action specifies the change in capacity (P) to be administered to a resource identified by <*ResourceName*>. After the action is executed, the elasticity action also specifies a cool-down time (T_c), for which the system has to wait before the elasticity condition is again evaluated.

We model the application tier as as an $M/M/m$ queue. The $M/M/m$ queue is a multi-server queuing model that consist of m servers at which the arrival of jobs (requests) are modeled as a Poisson process and the job service rate are exponentially distributed. Therefore, the parameters of the model defined with respect to a particular time interval t is as follows:

- *Number of servers* (m_t): is the number of servers at the application tier, which varies over time as scale-out and scale-in actions are triggered.
- *Request arrival rate*(λ_t): is the rate at which requests arrive into the system at the web tier. We consider that the requests are equally distributed among the servers in the application tier. Therefore, the arrival rate at each server is λ_t/m_t.
- *Mean service rate* (μ_t): the mean service rate which is also equal for all servers at the application tier as we assume each server to have the same processing capacity.
- *CPU utilization* (U_t): is the average CPU utilization of all servers at the application tier.
- *Average response time* (R_t): is the average response time of all requests at the application tier.

3.1 CPU Utilization

The average CPU utilization of m_t servers is used to evaluate the elasticity conditions at every time interval which in turn triggers elasticity actions. The total number of requests arriving and get served at the application tier during time interval t is $\lambda_t T$. Therefore, the total busy time of m servers is $B_t = \lambda_t T/\mu_t$ and the busy time of a server m is:

$$B_{tm} = \frac{(\lambda_t T/\mu_t)}{m_t} \tag{3.1}$$

According to the Utilization law, the CPU utilization of each server during time interval t is:

$$U_t = B_{tm}/T \tag{3.2}$$

By substituting B_{tm} in equation 3.2 then we get the average CPU utilization of the application tier's server:

$$U_t = \frac{\lambda_t}{m_t \mu_t} \tag{3.3}$$

Our experiments with different CPU elasticity rules [3] demonstrated that CPU utilization before triggering any scale-out actions (i.e., $m = 1$) increase significantly while request rate increase. This is because of the immediate start of concurrent user sessions that leads to sharp increases of the workload on the single server. This effect decreases quickly as soon as new servers are added by the elasticity rules as the increased workload is distributed between multiple servers. We capture this behaviour by applying a utilization ramp-up threshold (U_{r_θ}) which adjusts the approximated CPU utilization value (equation 3.3) as follows:

$$U_t = \begin{cases} U_t + U_{r_\theta}, & m_t = 1 \\ U_t, & m_t > 1 \end{cases}$$

The above CPU utilization models form the basis for CPU-based elasticity rules as it will be used to evaluate elasticity conditions to decide whether to trigger certain elasticity actions or not. The following illustrates the states in which elasticity conditions could be:

$$\begin{aligned} U_t &\geq U_\theta^u && \text{over-utilization state} \\ U_t &\leq U_\theta^l && \text{under-utilization state} \\ U_\theta^l \leq U_t &\leq U_\theta^u && \text{normal utilization state} \end{aligned} \tag{3.4}$$

3.2 Application's Response Time

. Response time is a crucial metric for evaluating the performance of elasticity rules. According to Little's law, the mean response time during time interval t can be approximated as shown in equation 3.5.

$$R_t = \frac{n_t}{\lambda_t} \tag{3.5}$$

Where n_t is the average number of requests in the system during time interval t. This can be divided into the number of requests being served and the number of requests queueing, i.e., $n_t = (n_s)_t + (n_q)_t$. This can be further represented in terms of m, U and probability of queueing as in the following equation:

$$n_t = (m_t U_t) + \frac{U_t \varrho_t}{1 - U_t} \tag{3.6}$$

Where ϱ_t is the probability that a request has to wait in a queue during time interval t and it can be estimated as follows:

$$\varrho_t = \frac{(m_t U_t)^{m_t}}{(m_t)!(1 - U_t)} P_0 \tag{3.7}$$

Where P_0 is the probability of 0 requests in the system which is estimated as follows:

$$P_0 = [1 + \frac{(m_t U_t)^{m_t}}{(m_t)!(1 - U_t)} + \sum_{n=1}^{m_t - 1} \frac{(m_t U_t)^n}{n!}]^{-1}$$

By substituting n_t in equation 3.5, we obtain the formula for the average response time during time interval t:

$$R_t = \frac{1}{\mu_t}(1 + \frac{\varrho_t}{m_t(1 - U_t)}) \tag{3.8}$$

The request mix is the the percentages of requests of different types relative to the total number of requests at certain time interval. Different request types (e.g., home page request, execute search request) put different demands on the CPU as each request has relatively certain execution code. Therefore, each request requires different processing times and therefore it influences the expected response times. This is a key factor that affects the performance of the web application and should be considered in the modeling of response time [7]. We capture the influence of request mix and types on response time as follows:

$$T_x = \sum_{i=1}^{n} D_i \lambda_{i_t} \tag{3.9}$$

Where T_x is the additional time resulted from request mix. D_i is the average demand a request of type i puts on the CPU and n represents the number of request types. The values of D_i can be obtained from real measurements for each request type. λ_{i_t} is the average request rate of type i during time interval t and it can be estimated as follows:

$$\lambda_{it} = \lambda_t Q_{p_i}$$

Where Q_{p_i} is the percentage of request type i relative to all requests. These percentages by can be defined by classifying the workload profiles. For example, the TPC-W industry standard benchmark for web applications classifies e-commerce into Browsing, Shopping and Ordering profiles in which the percentages of request types vary [15].

By adding the time resulted from request mix to equation 3.8

$$R_t = \frac{1}{\mu_t}(1 + \frac{\varrho_t}{m_t(1 - U_t)}) + \sum_{i=1}^{n} D_i \lambda_{it} \tag{3.10}$$

3.3 Modelling Infrastructure Constraints

The $M/M/m$ model assumes an ideal queue with frictionless elasticity, that is, servers are provisioned and de-provisioned instantly. However, a scale-out action involves a delay before a server becomes operational due to the time required to provision and boot a new server with all required operating system packages and software applications. For example, it has been shown that AWS cloud servers require in average about 5 minutes to start a cloud server of small size (m1.small) [3]. We term this as the *server provisioning lag time* and denote it by T_{sl}. In contrast, the effect of a scale-in or de-provisioning action can be considered as near instantaneous.

Cloud providers such as Amazon allow cloud consumers to specify a *cool-down interval* to take into account the effects of an elasticity action on the system, such as provisioning lag time and the time needed for a server to start serving requests, before evaluating elasticity conditions again. This interval is crucial as it allows appropriate time for a triggered scaling action to take effect and to see its impact on system performance before triggering a new action. We denote the cool-down time after a scale-out action as T_c^u and after a scale-in action as T_c^l.

Another important aspect that should also be considered in modeling elasticity is a user-specified limit on the maximum and the minimum number of servers as boundaries that must not be exceeded by elasticity actions. So, a scale out action will not be executed when a maximum number of servers S_{max} is reached. Similarly, a scale in action will not be triggered if a minimum number of servers S_{min} is reached.

3.4 CPU-based Elasticity Algorithms

Algorithm 1 CPU-based Elasticity - Scale-Out

```
Estimate $\lambda_t$, $\mu_t$, $U_t$, $R_t$
if $((T_{c_t}{}^u \leq 0)$ & $(T_{sl_t} \leq 0))$ then
    if $(U_t \geq U_\theta{}^u)$ then
        $overUtilTime = overUtilTime + T$
    else
        $overUtilTime = 0$
    end if
else
    $T_{sl} = T_{sl} - 1$
    $T_{c_t}{}^u = T_{c_t}{}^u - 1$
end if
if $((overUtilTime \geq T_w{}^u) \&\& (m_t < S_{max}))$ then
    $m_t = m_t + 1$
    $overUtilTime = 0$
    $T_{c_t}{}^u = T_c{}^u$
    $T_{sl_t} = T_{sl}$
else
    $m_t = m_{t-1}$
end if
```

Based on the model presented before, we have developed algorithms that simulate how CPU-based elasticity works. Listing 1 and 2 describe the logic of our scale-out and scale-in algorithms respectively. The scale-out algorithm (listing 1) is executed at each time interval t as follows. The values of request arrival rate at the application tier (λ_t), the mean service rate of each server (μ_t), the average server utilization at the application tier (U_t) and the average response time (R_t) are approximated using the equations explained earlier. λ_t and μ_t can be generated from Poisson and exponential distributions respectively. Alternatively, they can be generated from real workload and benchmarks. It first ensures that the cool-down time and server provisioning time do not hold. If these do not hold then it checks if the estimated CPU utilization value at current time interval is above the upper CPU utilization threshold. If so, it increases a time counter by one interval. If the estimated utilization goes below the upper threshold at any time it reset that time counter to zero to start counting again. If the time counter becomes greater than or equal to the upper monitoring time

Algorithm 2 CPU-based Elasticity - Scale-In

```
Estimate $\lambda_t$, $\mu_t$, $U_t$, $R_t$
if (${T_{c_t}}^l > 0$) then
    ${T_{c_t}}^l$=${T_{c_t}}^l - 1$
else
    if ($U_t \leq {U_\theta}^l$) then
        $underUtilTime$=$underUtilTime + T_i$
    else
        $underUtilTime = 0$
    end if
end if
if (($underUtilTime \leq {T_w}^l$)&&($m_t > S_{min}$)) then
    $m_t = m_t - 1$
    $underUtilTime = 0$
    ${T_{c_t}}^l = {T_c}^l$
else
    $m_t = m_{t-1}$
end if
```

window (T_w^u) and the number of servers has not exceeded the maximum limit then the number of servers are increased. In addition, the cool-down time and the server provisioning lag time are set the desired values so that no further scaling actions are taken until both cool-down and server provisioning lag times elapse.

The scale-in algorithm (listing 2) has similar logic but it does not have server provisioning lag time (as shutting down a server does not need time to occur). Another difference is the lower thresholds (e.g., CPU utilization threshold, cool-down threshold)and parameters (e.g., minimum number of servers) are used instead of upper thresholds and parameters.

3.5 Estimating Cost of Provisioning

Cloud (IaaS) providers mainly charge users based on the number of servers in operation and the time for which they have been used. Commonly, most providers charge on an hourly basis. Triggering of elasticity actions leads to changes in the overall cost of operation due to the provisioning and de-provisioning of servers. The cost of provisioning can only be calculated considering the entire time from the application's first deployment till the present. We denote this time period as τ and is the sum of all time intervals t. $Max(m)$ represents the maximum number of servers that have been provisioned for this application.

The algorithm shown in listing 3 describes the logic for calculating the cost of servers usage triggered by auto-scaling actions. The first two loops is to fill the number of minutes for each server in the S_{mins} (i.e., server minutes array) with dimensions maximum number of servers ($Max(m)$) and measurement time τ. For each server column a value of 0 or 1 is assigned depending wether that server is provisioned or not (the change in m over time determine here when a server is provisioned or de-provisioned). In the S_{mins} array a value 1 indicates that server is used for 1 time interval (e.g., 1 minute). The second loop block use server minutes array to compute the the total number of hours (rounded

Algorithm 3 CPU-based Elasticity - Servers Cost

```
$S_{mins}(Max(m), \tau) = 0$
for $i \leftarrow 1, \tau - 1$ do
    if $(m_{t+1} > m_t) \| (m_{t+1} < m_t)$ then
        $mCount = m_{t+1}$
    else
        $mCount = m_t$
    end if
    for $i \leftarrow 1, mCount$ do
        $S_{mins}(i, t) = 1$
    end for
end for
for $i \leftarrow 1, Max(m)$ do
    $totalMins = 0$
    for $t \leftarrow 1, \tau$ do
        $totalMins = totalMins + S_{mins}(i, t)$
    end for
    $S_c = S_c + (\lceil totalMins/60 \rceil * S_r)$ return $S_c$
end for
```

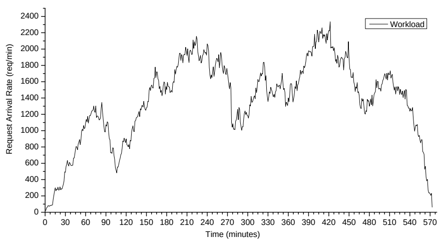


Figure 4.1: TPC-W Workload Used in all Experiments

to the next hour) each server is used. The number of hours is then used to compute the servers cost by multiplying total number of minutes with server charges (S_r) for each server and add the cost to the total cost of provisioning (S_c).

4 Validation

We have validated our elasticity models and algorithms empirically and by simulation. In this section, we first describe the design and methodology of our empirical and simulation environments (Section 4.1). We then compare and discuss the results of the empirical and simulation experiments in terms of CPU utilization, application's response time, number of servers and server costs (Section 4.2).

Table 4.1: Elasticity Thresholds Used in all Experiments

Rule/Threshold	${U_\theta}^u$	T_w	${U_\theta}^l$	T_w
CPU75	75%	5 min	30%	10 min
CPU80	80%	5 min	30%	10 min
CPU85	85%	5 min	30%	10 min
CPU90	90%	5 min	30%	10 min

4.1 Experimental Design and Methodology

In this section we describe the experimental setup and methodology for our (1) empirical experiments with TPC-W benchmark on Amazon EC2 and (2) simulation experiments in Matlab.

The Empirical Experiments

We have chosen TPC-W, an industry standard for transactional Web benchmark [15], as a representation of online retail applications. TPC-W has been widely used in cloud-related performance studies [3, 7, 9]. Based on 14 different web interactions, TPC-W specifications differentiate between three workload profiles; *Browsing, Shopping* and *Ordering* profiles. These profiles vary based on the percentage of each interaction in the *Browse* (read operations) and *Order* (write operations) groups. In all experiments we used *Browsing profile* as it stresses the application tier(i.e.,95% read operations and 5% write operations) which is the focus of our elasticity performance modeling. We have generated the workload using TPC-W user emulation software with *Browsing profile* as it stresses the application tier. The number of concurrent users and inter-arrival times have been generated from power-law (Zipf) and Poisson functions respectively. The resulting workload is shown in Figure 4.1.

We have used the open source Java implementation developed by Horvath [16]. We have deployed it on Amazon cloud as a 3-tier architecture; a typical architecture that is used for internet applications [4, 7, 9, 3]. The application tier consists of a pool of Amazon's Linux servers (*m1.small* instance). We installed JBoss2.3.2 and deployed the TPC-W bookstore application logic on each server. We configured the application tier as an *auto-scaling group* to scale out and scale in based on elasticity rules that can be configured by cloud consumers. We deployed the bookstore database on a separate Linux server (*m1.xlarge* instance) which runs MySQL5.1.92. We populated the database with 10000 books generated randomly according to TPC-W specifications [15]. We used Amazon's Elastic Load Balancer to distribute user requests among the pool of instances at the application tier. The TPC-W user emulation application was deployed on a separate Linux server (*m1.xlarge*) instance. All the servers were located in the same Amazon geographic region, US East (Virginia), to ensure reducing network overhead between servers at different tiers.

Using the above experimental setup we carried out 4 experiments each of which with different elasticity rules' thresholds as shown in Table 4.1. The naming of the rules is based on the upper CPU threshold (e.g., CPU75, CPU80). For all the elasticity rules the other parameters have been set as follows: ${T_c}^u$=5 minutes, ${T_c}^l$=5 minutes, S_{min}=1 and S_{max}=20.

In all experiments, we have configured Amazon CloudWatch [1] and we implemented a Java application to continuously collect measurements of important metrics including:

- CPU Utilization: the average CPU utilization of all servers at the application tier every minute interval.
- Response Time: the average response time of all requests at the application tier every minute interval.
- Number of Servers: the number of servers at the application tier every minute interval.
- Servers Cost: the usage cost of the servers at the application tier based on Amazon hourly charges ($0.08 for N.Virginia small instances).

We represented the CPU utilization and response time measurements using box plot as it provides useful statistics which can help analysing and comparing data points at glance. These statistics are; the mean, the median, the 1st, 25th, 75th and 99th percentiles as illustrated in Figure 4.2(c).

The Simulation Experiments

We implemented our CPU-based elasticity models and algorithms using Matlab. The key inputs to the simulation are the application workload λ and the threshold values of the elasticity rules to be evaluated. In all simulation experiments, we used the same workload traces that has been resulted from the empirical experiments(Figure 4.1). In this workload, T is 1 minute and τ is 573 minutes and therefore the workload was divided into 1-minute time intervals. Using this workload, we ran the same empirical experiments with the same elasticity rules thresholds described in the previous section and in Table 4.1. The measurements of all metrics are approximated using the CPU-based models and algorithms implemented in the simulation. Similarly, we collected these measurements we represented it in box plots statistics to compare it with the empirical box plot statistics.

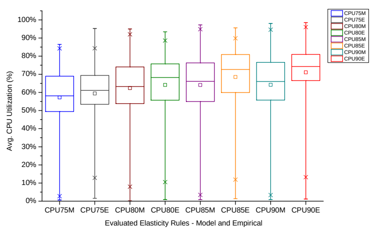
4.2 Experimental Results

In this section we analyse and discuss the results of the empirical and simulation experiments in terms of CPU utilization, application's response time, number of servers and servers usage costs at the application tier.

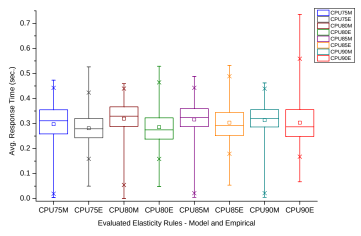
CPU Utilization

Figure 4.2(a) shows the box plots of CPU utilization of all elasticity rules resulted from the simulation and the empirical experiments. We have used the CPU upper threshold as a naming convention followed by character 'M' referring to results from the models simulation experiments or 'E' referring to results from the empirical experiments. As can be seen from Figure 4.2(a), the CPU utilization resulting from our simulation experiments is close to that resulting

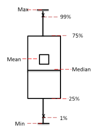
[1] `http://aws.amazon.com/cloudwatch/`



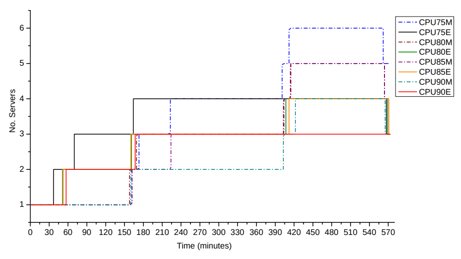
(a) CPU Utilization - Model and Empirical



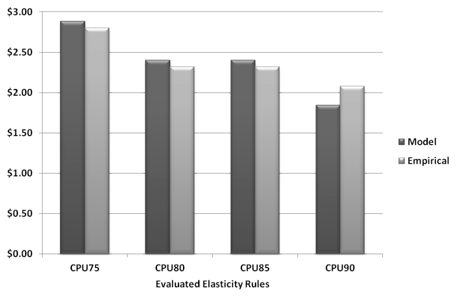
(b) Response Time - Model and Empirical



(c) Legend



(d) No. of Server Over Time - Model and Empirical



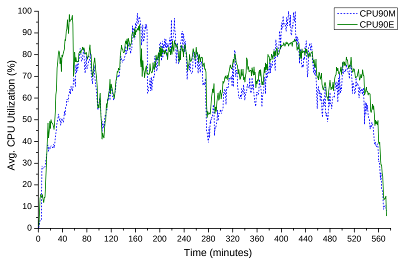
(e) Servers Usage Cost - Model and Empirical

Figure 4.2: Experimental Results of all Evaluated Elasticity Rules using Our Model and Empirically

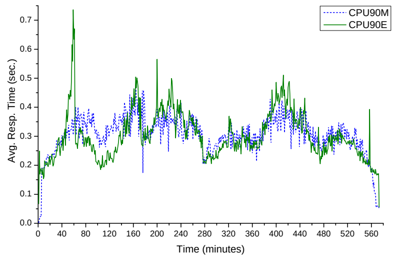
from the corresponding empirical experiments. For example, the difference between mean CPU utilization obtained via simulation and that obtained from empirical experiments ranges between 1.5% and 6.8%.

The raw data from the empirical experiments displays a spike in the CPU utilization within the first 80 minutes as shown in Figure 4.3(a). This pattern occurs in all elasticity rules experiments but the magnitude of such spike increase as the upper CPU threshold increase. As is illustrated in Figure 4.3(a), our CPU-based elasticity models have not precisely approximated such spikes. Such spikes occurred because of the continuous increase in request rate while one server was serving all incoming requests and before the first scale-out action was triggered [3]. Modeling workload spikes and its effect on CPU utilization require considering other system's factors. However, in a production environment, this effect is sensitively captured as all factors that could influence performance can naturally cause such effects. We can see that such spikes gradually disappear as the number of servers increase because of triggered scale-out actions over time. As shown in Figure 4.3(a), the CPU utilization data points of the simulation and empirical experiments have converged as the number of servers increase over time. This is because the increase in request rate was distributed between a number of servers instead of being handled by one server.

Another important observation from the CPU utilization results (Figure 4.2(a)) is that, under the same workload, the model has exhibited the same trend as the empirical results regarding the relationship between CPU upper threshold (U_{θ}^{u}) and the mean and median CPU utilization. Particularly, the mean and median



(a) CPU Utilization of CPU90- Model and Empirical



(b) Response Time of CPU90- Model and Empirical

Figure 4.3: CPU Utilization and Response Time Spikes of CPU90 Experiments

CPU utilization resulted from the simulation experiments increase as the upper threshold increases. Predicting such relationships will help cloud consumers in understanding the effect of changing thresholds on important metrics such as CPU utilization.

Response Time

Figure 4.2(b) shows the box plot statistics of all elasticity rules resulted from the empirical and simulation experiments. As shown in this figure, our model simulation approximated response times with average and median close to the empirical mean and median response times. The difference in mean response time between simulation and empirical results ranges between 11 milliseconds and 34 milliseconds. We can see some variation in terms of other statistics though. We believe such variations are due to the CPU utilization spikes which has been discussed above. Figure 4.3(b) shows the mean response times data points resulted from empirical and simulation experiments with elasticity rule CPU90. As shown in this figure, it is clear that the CPU utilization spike influenced response time by causing a spike in response time. This effect is also found in experiments involving all other elasticity rules but with a magnitude that increases as the upper CPU utilization threshold is increased. This effect has not been precisely approximated by our models as it does not consider situation under which such spikes could occur, i.e., fairly high increases in request rates over time while one server is serving requests and before other servers are added by scale-out actions. As shown in Figure 4.3(b), the response time spike effect had not hold when the number of servers is increased. Accordingly, the response time data points of the simulation experiments have converged and have become reasonably close to the empirical response time data points (see Figure 4.3(b)).

Another important observation about response time data is that the simulation results showed a relationship with the upper CPU utilization threshold (U_θ^u); specifically as U_θ^u increases, the average response time either increases or remains constant. We noticed that this trend is consistent with the same trend that occurred with the empirical response time results. This demonstrates that cloud consumers can rely on our CPU-based elasticity simulation as a tool for predicting such trends when evaluating the performance of different elasticity thresholds.

Number of Servers and Cost of Provisioning

As demonstrated in Figure 4.2(d), both the simulation and the empirical experiments of all elasticity rules have resulted in the provisioning of similar number of servers. But, in all elasticity rules, the model produces an extra server over the number provisioned during actual experiments. In addition, the pattern in the number of servers resulted from the simulation experiments while U^u_{theta} increasing is consistent with the number of servers pattern resulted from the empirical results.

Also, Figure 4.2(d) shows that the time when our simulation triggered scale-out actions (i.e., adding servers) does not match the time when scale-out actions happened empirically, especially in the first scale-out actions of all elasticity rules experiments. We believe both these effects are due to our models not being able to approximate the CPU utilization spikes which has caused triggering scale-out actions in the early stage of the experiments. This has led to delays in satisfying scale-out conditions in all elasticity rules in our simulation experiments.

Although the simulation has not approximated the exact number of servers and time of triggering servers, this does not have a significant impact on estimating provisioning costs as shown in Figure 4.2(e). The approximated provisioning cost resulting from the model for all elasticity rules, are very close to the costs resulting from the corresponding experiments. This is because the costs of provisioning are calculated based on the total server hours during which servers were utilized.

Figure 4.2(e) also reveals another important observation that relates U^u_θ and provisioning cost. Particularly, the server costs of the simulation experiments follow the trend of provisioning costs in the empirical experiment; i.e., as U^u_θ increases the provisioning costs decrease.

5 Conclusions and Future Work

Modern enterprises are increasingly deploying their applications on IaaS cloud so it inherently become self-elastic to meet its variable workload and performance-cost metrics. Many IaaS providers such as Amazon Web Services provide mechanisms to configure elasticity service based on a number of metrics and thresholds. However, defining appropriate elasticity thresholds that can lead to satisfaction of resource and application metrics remains as a primary challenge for cloud consumers.

In this paper we have presented analytical models, based on queuing theory, that emulate the behaviour of elasticity rules at the application tier of multi-tier applications deployed on IaaS cloud. These models capture the key parameters and thresholds of elasticity rules such as CPU utilization thresholds, monitoring time windows and cool-down time. In addition, our models approximate a number of metrics that are important for analysing the performance of elasticity rules. These metrics include CPU utilization, application's response time, number of servers triggered by elasticity and cost of servers usage. Based on these models, we have also presented our CPU-based elasticity algorithms; scale-out and scale-in algorithms which simulates when and how to trigger scale-out and scale-in actions and cost algorithm which estimate servers usage cost resulted from those scale-out and scale-in actions. We have also implemented these models and algorithms in Matlab and used them to evaluate different elasticity rules.

Using the proposed elasticity models and algorithms, we conducted simulation experiments with a number of elasticity rules with different CPU utilization thresholds. We have validated the resulting metrics against the same metrics that have resulted from the corresponding experiments we have conducted with the same elasticity rules with TPC-W application on Amazon EC2. The simulation results demonstrated reasonable accuracy of our elasticity models and algorithms in approximating CPU utilization, application's response time, number of servers and servers usage costs. It has shown ability in emulating the trends and relationship between changing CPU utilization thresholds and these metrics with acceptable variations.

The simulation of our elasticity models and algorithms provides a tool that can be used by aid enterprises to define appropriate thresholds for their application's workloads. Therefore, cloud consumers analyse the impact of changing elasticity thresholds on certain performance-cost metrics that are important to satisfy.

The research we have done has opened interesting future work. One future item is the modeling of CPU utilization and response time spikes before triggering any scale-out actions. This would help in improving the precision of our analytical models and algorithms for approximating elasticity and its key performance metrics. Another important future work is developing optimization models for elasticity thresholds. In this direction, we see the need for developing optimization models that allow cloud consumers to specify performance and costs objectives as functions so that an algorithm can finds the combination of elasticity threshold values that meet these objectives.

6 Acknowledgments

The cost of using cloud computing (EC2) resources for the experiments in this paper were covered by a grant provided by Amazon Web Services. However, Amazon was neither involved in nor had any influence on the research reported in this paper.

Bibliography

[1] B. Suleiman, S. Sakr, R. Jeffery, and A. Liu, "On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure," *JISA*, vol. 2, no. 3, pp. 1–21, 2011.

[2] S. Tai, P. Leitner, and S. Dustdar, "Design by units: Abstractions for human and compute resources for elastic systems," *IEEE Internet Comp.*, vol. 16, no. 4, pp. 84–88, 2012.

[3] B. Suleiman, S. Sakr, S. Venugopal, and W. Sadiq, "Trade-off analysis of elasticity approaches for cloud-based business applications," in *WISE 2012*, 2012, pp. 468–482.

[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. 1:1–1:39, Mar. 2008.

[5] P. Lama and X. Zhou, "Efficient server provisioning with control for end-to-end response time guarantee on multitier clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 1, pp. 78–86, Jan. 2012.

[6] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "On the use of fuzzy modeling in virtualized data center management," ser. ICAC'07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 25–34.

[7] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, "Autonomic mix-aware provisioning for non-stationary data center workloads," ser. ICAC'10. ACM, 2010, pp. 21–30.

[8] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann, "Automated control for elastic n-tier workloads based on empirical modeling," ser. ICAC'11. ACM, pp. 131–140.

[9] I. Al-Azzoni and D. Kondo, "Cost-aware performance modeling of multi-tier web applications in the cloud," in *Networked Digital Technologies*, ser. Communications in Computer and Information Science, 2012, vol. 293, pp. 186–196.

[10] H. Li, G. Casale, and T. Ellahi, "Sla-driven planning and optimization of enterprise applications," ser. WOSP/SIPEW'10. ACM, 2010, pp. 117–128.

[11] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Feedback-based optimization of a private cloud," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 104–111, Jan. 2012.

[12] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," ser. SIGMETRICS'05. ACM, 2005, pp. 291–302.

[13] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal autoscaling in a iaas cloud," ser. ICAC'12. ACM, 2012, pp. 173–178.

[14] R. Jain, *Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling.* New York, USA: Wiley Computer Publishing, 1991.

[15] T. P. P. Council, "Tpc benchmark web commerce specification (tpc-w)," Tech. Rep. 202, Feb 2002. [Online]. Available: http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf

[16] T. Horvath, "Tpc-w java implementation," http://www.cs.virginia.edu/~th8k/downloads/, Nov. 2008. [Online]. Available: http://www.cs.virginia.edu/~th8k/downloads/