iScale: An Intelligent Auto-Scaling Engine for Migrated Applications to the Cloud

Omid Bushehrian, Zahra Reisi

Department of Computer Engineering and IT, Shiraz University of Technology, Shiraz, Iran

bushehrian@stu.ac.ir, zreisi@stu.ac.ir

ABSTRACT

Migration of the legacy software systems to cloud environment is a very cost-effective solution for the service providers. It is primarily due to the elastic nature of the cloud environment: The migrated software is able to acquire and release resources on-demand based on the incoming workload and by doing this not only the service provider has to pay only for the resources they need, but also the QoS requirements of the end-users are always met. To achieve this, the migrated software has to be supported with an “Auto-Scaling Engine” to add or remove resources to/from it at runtime based on the incoming workload. However, Reacting to the transient workload changes always results in useless sequences of “acquire-release” actions of cloud resources which impose unwanted overhead costs on the service provider due to the configuration changes. In this paper an “Intelligent Auto-Scaling Engine” which recognizes the transient workload changes using a Learning Automata and only reacts to the stable ones is proposed. We observed that the proposed method is very effective in reducing the overall costs of the service provider.

Keywords: Migration to Cloud, Auto-Scaling, Learning Automata, Transient Workload

1. INTRODUCTION

Recently cloud computing has attracted a lot of attention from the IT industry due to its ability of providing the infrastructure, platform and software to the clients “on-demand”. The elasticity nature of the cloud computing is one of its main characteristics which enables the service providers to allocate more or less resources to their running applications in response to the changing workloads[1][2]. The amount of resources required by a cloud application during its execution time is a function of two important factors: (1) The established SLA’s(Service Level Agreement) between application users and the service providers and (2) the current workload of the application. A service provider has to estimate the required capacity at each specific workload such that the QoS requirements specified in the SLA’s are not violated and subsequently demands only the needed resources from the cloud provider.

Obviously the main objective of the service provider is to maximize its profit during the execution of its cloud application which can be achieved by minimizing two important cost drivers: first, the cost of resources (including virtual machines and software licenses) rented by the service provider and second, the cost of penalties that have to be paid to the application users due to violation of the previously established SLA constraints. While it is true that scaling up the application resources in response to workload increase is a way to avoid the fall of perceived application performance below the agreed values specified in the SLA which causes high penalty costs; however it also increases the resource costs rented by the service provider from the cloud provider. Therefore, to achieve the maximum profit, it is essential that the optimal amount of resources corresponding to each workload is specified (before migrating the application to the cloud) such that the total service provider cost becomes minimal. The outcome is specified as a set of scaling rules which are specified by the service provider and determine when the scaling is triggered and how much resources are allocated(or de-allocated)[3][4].

Commercial cloud providers such as Amazon expose API’s to developers in order to define and configure the auto-scaling rules. This is performed by defining an Amazon Cloud Watch Alarm and associating it with a policy [21]. An Alarm is an object that monitors a VM level metric (such as CPU utilization or network traffic) over a time period and triggers the scaling instructions defined within its associated policy if its value goes beyond a predefined threshold. The scaling instructions might be either adding (or deleting) the number of VM’s (horizontal scaling) or changing the current VM’s sizes (vertical scaling). However there are still open issues regarding the auto-scaling mechanisms:

1.1 Adaptation of Scaling Rules Based on runtime conditions

The scaling rule set may involve some probabilistic rules whose triggering probabilities change over time based on the stability or instability of the application workloads. For instance consider scaling rule saying that “if the workload changes to L_i then add one VM to current VM group”. However if the workload L_i is not stable and immediately vanishes, changing the current VM set and software deployment is useless. Generally, reacting to the transient workload changes always results in useless sequences of “acquire-release” actions of cloud resources which impose unwanted overhead costs on the service provider due to the configuration changes. Therefore it is essential to foresee the future conditions of the system workload by learning from previous events.

1.2 Supporting of the Application Reconfiguration during the auto-scaling
Creating the scaling rule set which is performed by the performance engineer is the outcome of capacity planning activity which estimates the amount of resources needed considering each known application workload such that the SLA’s are not violated. However in addition to determining the number and size of VMs, the deployment of application components over the VM’s is also an important issue and has to be specified particularly for legacy applications which are migrated to the cloud environment. Therefore the new configuration and topology of software components has to be considered during auto-scaling.

In this paper an “Intelligent Auto-Scaling Engine” which recognizes the transient workload changes using a Learning Automata[5] and only reacts to the stable ones is proposed. The main objective is to maximize the overall profit of the service provider by intelligently selecting the next best configuration of the application once a workload change happens.

2. RELATED WORK

Workload has a considerable impact on cost and performance as studied in [6]. In the field of auto-scaling, many of previous studies are focused on the workload forecasting techniques. A number of articles have been presented to overcome the horizontal and vertical scaling challenges. Dutta et al. [7] presented an automatic scaling framework called ‘Smart Scale’ that applies a combination of horizontal and vertical scaling to ensure that the programs are scaled in the best way considering both the costs of using resources and configuration changes. This framework rapidly scales the software by workload volume changes and applies a decision tree to find the optimal combination of horizontal and vertical scaling.

In [8] an automatic scaling controller to determine the optimal number of resources for properly responding to the workload changes has been proposed. They have formulated the problem as an optimization problem and solved it numerically. This method has reduced costs compared with the basic methods.

In [9] an automatic scaling model at the virtual machine level for web applications is proposed that predicts the number of web requests of the application using machine learning techniques. The aim has been to find the resource allocation with an optimal trade-off between cost and latency. The optimal amount of resources are identified using a M/M/m performance model which determines the relationship between the number of VMs and the response time of the web application. To predict the number of web requests, a linear regression using the auto-correlation function is used and a pattern is extracted.

In [10] Royet al. predicted workload using the second order Auto Regression Moving Average method (ARMA). They proposed an efficient model and allocated the resources according to the prediction outcome. The results showed that this method actually uses a small number of machines and provides the users their QoS requirements in addition to reducing the costs.

In [11] a new architecture for cloud services is presented. Linear regression model is used to predict the workload and based on the prediction outcome, scaling at different levels of cloud services is proposed. Scaling problem is considered as an integer programming problem and a greedy heuristic is used to solve it. In this work, scaling is conducted in two parts of runtime and pre-runtime and three scaling techniques are considered: self-healing scaling, resource level scaling and VM-level scaling. When the algorithm is started, an initial workload for each service is predicted and an appropriate number of VMs are allocated regarding the prediction outcome. Information about each service is extracted and on the basis of the obtained data, at run-time phase, resource scaling is carried out. The results show that lower costs are achieved as well as fewer SLA violations.

A two-step prediction approach is proposed in [12]. In the first step, a workload tracker function obtains an initial view of load trend from raw data using a Moving average (MA) method. In the second step, it applies a workload prediction algorithm based on the Exponential Moving Average (EMA) technique for workload forecasting. However the adaptation of this method with rapid workload changes is not considered in this paper.

A statistical learning algorithms and a state-driven approach to predict workloads is presented in [13]. It uses a discrete-time Markov chain to predict the service demand. In [14] a pattern matching for the workload forecasting based on the workload history is proposed. However the number of calculations is considered as a prohibiting factor in this method.

Despite the previous studies to overcome the challenges in the auto-scaling, there have been no studies so far regarding the impact of transient workloads on the total costs incurred by the service provider. In this paper a novel method, based on the Learning Automata technique, for predicting the transient workloads and allocating the resources according to durability of the forthcoming loads is proposed.

3. THE PROPOSED AUTO-SCALING ENGINE: iSCALE

Assuming that the migrated software initially is deployed on a set of virtual machines acquired from the IaaS provider, scaling up (or down) the system based on changing the workload means that the allocated resources to the software and the assignment of software components to the virtual machines may change and a new configuration is installed for the software. Prior to the migration process, the required configuration G is to support the known workload L has to be specified by the designer during an activity called Capacity Planning[15]. Here, based on the QoS requirements of the end-users specified in the Service Level Agreements (SLA), the
designer determines the required capacity (the number of virtual machines and the size of allocated resources on each one) and the optimal assignment of software components to virtual machines such that the SLA constraints are not violated for each known workload \( L_i \). The outcome of this activity is the Configuration Plan of the migrated software which is defined as follows:

Definition 1. A Configuration Plan \( P \) is a function which associates each workload \( L_i \) with its corresponding configuration \( G_i \) such that: 1) when the system workload is \( L_i \) and the software configuration is \( G_i \), none of the SLA constraints are violated and 2) \( G_i \) contains the least resources to satisfy the first condition.

Each configuration of the software is defined as follows:

Definition 2. A Configuration \( G_i \) corresponding to load \( L_i \) is defined as triple \((C,M,A)\) where \( C \) is the set of software components to be started during the execution of the software with this configuration, \( M \) is the set of VM’s on which the software components are deployed and \( A \) is a function that maps each component \( c_k \in C \) to a \( v_{mj} \in M \).

The Capacity Planning activity can be performed with analytical models such as Layered Queuing Networks (LQN)[16][17] or simulate-able formal models such as Labeled Transition Systems(LTS)[18][19] as studied thoroughly in previous researches. Based on the theses premises, the proposed architecture of iScale is illustrated in Figure 1.

As explained earlier, The Configuration in Figure 1 represents the set of virtual machines allocated to the application components. Moreover, the deployment of application components on the VM’s is also determined based on the current configuration \( G_i \) obtained from the Configuration Plan of the iScale regarding the current workload \( L_i \) detected by the Load Watch module of the iScale as shown in Figure 1. The Load Watch monitors the input traffic of the VMs and provides the Learning Automata with the newly detected workload. The responsibility of the Learning Automata is to decide whether to scale out/in the current configuration or ignore the workload change if it is learned as a transient workload.

There are durability thresholds corresponding to each workload computed at design time considering the cloud costs and SLA violation penalty costs by which the durability of a given workload is determined at runtime. If a given workload is not changed before its corresponding threshold, it is recognized as a “stable load” otherwise it is a “transient load”. The durability of each load change from \( L_i \) to \( L_j \) is learned by a Learning Automata \( L_{Ai} \) based on its durability history. If a load change from \( L_i \) to \( L_j \) is learned as “stable” by \( L_{Ai} \), it means that the act of scaling out/in the current configuration to the new configuration \( C_j \) is more probable than keeping the current configuration.

### 3.1 Cost model and Parameters

Assuming that the input workload is \( L_c \) and the current configuration of the application is \( C_i \) as shown in Figure 2, an increase in the workload to the new load \( L_j \) may be reacted by the auto-scaling engine either by scaling out the configuration \( C_i \) and installing the new configuration \( C_j \) or ignoring the load change and continue with the current configuration. In the former case, assuming that the new load persists \( t \) time units, the total costs incurred by the application provider is calculated as follows:

\[
\text{Scaling Cost}(t) = \int_{0}^{t} F_{\text{change}}(t) \, dt + \text{Cost}_{\text{SI}} \times (t - t_{\text{SI}}) + \text{Software License Cost}_{ij}
\]

Where \( t_{\text{SI}} \) denotes the reconfiguration time from \( C_i \) to \( C_j \), \( F_{\text{change}}(t) \) is the SLA violation function related to the horizontal configuration, \( \text{Cost}_{\text{SI}} \) denotes the cost of next configuration \( C_j \) per time unit, \( t_{\text{SI}} \) denotes the durability threshold regarding the current configuration \( C_i \) and new load \( L_i \) and Software License Cost \( \text{Cost}_{ij} \) is the cost of new software licenses in configuration \( C_j \).

If the load change is ignored and the current configuration is kept, the total cost incurred fort time units is computed as follows:

\[
\text{Ignoring Cost}(t) = \int_{0}^{t} F_{\text{SI}}(t) \, dt + \text{Cost}_{\text{SI}} \times t
\]

Where \( F_{\text{SI}}(t) \) is the penalty value at time \( t \) of the SLA violation period when the current configuration is
kept $C_i$ but the new load is $L_j$, and $C_{cost_i}$ denotes the current configuration cost.

Assuming that the penalty function $P_{violation}(t)$ is an ascending function and the cloud cost per time unit for configuration $C_j$ is constant along the time, we can find the time duration $t_{eq}$ where the two cost functions (1) and (2) are equal. Obviously if the new load duration is less than $t_{eq}$, the load is assumed as transient and the auto-scaling engine has to ignore the load change due to the fact that for any $t < t_{eq}$, the cost function in formula (2) has the less value than the cost function in formula (1). For load durations longer than the threshold value, the auto-scaling engine scales out/in the current configuration and installs the new one. The cost parameters are listed in Table 1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{eq}$</td>
<td>Reconfiguration time from $C_i$ to $C_j$</td>
</tr>
<tr>
<td>$t_{eq}$</td>
<td>Time Threshold for current configuration $C_i$ and new load $C_j$</td>
</tr>
<tr>
<td>$C_{cost_i}$</td>
<td>cloud cost per time unit for configuration $C_i$</td>
</tr>
<tr>
<td>$P_{change}(C_i)$</td>
<td>SLA violation penalty function during the configuration change</td>
</tr>
<tr>
<td>$P_{violation}(C_i)$</td>
<td>SLA violation penalty function during the load $L_j$ and configuration $C_i$</td>
</tr>
<tr>
<td>Software License Cost$_j$</td>
<td>Extra cost of software license when changing the configuration from $C_i$ to $C_j$</td>
</tr>
</tbody>
</table>

### 3.2 Learning Method

Learning Automata is a finite state machine that aims to apply the best action on environment through a learning process. The best action is the one that maximizes the probability of receiving rewards from the environment. LA chooses an action repeatedly based on the action probabilities and then updates the action probabilities considering the environment responses.

### 4. EXPERIMENTAL RESULTS

The experiments were designed to illustrate the effectiveness of the proposed method in reducing the overall costs of the application provider by ignoring the transient workloads and hence avoiding cost overheads due to the useless re-configurations. Using the Amazon EC2 as a reference [20] we used four types of VMs as listed in Table 2.
Table 2: Different VM types

<table>
<thead>
<tr>
<th>Configuration</th>
<th>VMs</th>
<th>Memory</th>
<th>CPUs</th>
<th>Price (per hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>1 VM</td>
<td>1.7 Gb</td>
<td>1</td>
<td>$0.065</td>
</tr>
<tr>
<td>Medium</td>
<td>1 VM</td>
<td>3.75 Gb</td>
<td>1</td>
<td>$0.130</td>
</tr>
<tr>
<td>Large</td>
<td>2 VM</td>
<td>7.5 Gb</td>
<td>2</td>
<td>$0.65</td>
</tr>
<tr>
<td>XLarge</td>
<td>2 VM</td>
<td>8 Gb</td>
<td>2</td>
<td>$0.70</td>
</tr>
</tbody>
</table>

Four classes of workloads are used in the experiments as listed in Table 3. Each workload $L_i$ has its corresponding configuration $G_i$. The workloads are generated using the Rain tool [21]. Rain is a Markov-chain based workload generator which can generate user defined workloads easily.

Table 3: Workload Classes

<table>
<thead>
<tr>
<th>Workload Class</th>
<th>Mean time between requests (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1$</td>
<td>100</td>
</tr>
<tr>
<td>$L_2$</td>
<td>150</td>
</tr>
<tr>
<td>$L_3$</td>
<td>200</td>
</tr>
<tr>
<td>$L_4$</td>
<td>300</td>
</tr>
</tbody>
</table>

We have implemented iScale in C# language. The class diagram of iScale is shown in Figure 4.

Three types of composite workloads: Stable, Mixed and Transient were used to evaluate the presented method. Each composite workload was generated for a time period of 60 minutes by composing the individual workloads listed in Table 3 for random times. The Stable workload only consists of durable load changes. The Mixed one consists of both durable and transient load changes and the Transient only consists of transient load changes. To compute the threshold values in our experiments the penalty function form was defined as follows:

$$P(t) = \begin{cases} 
  A*t + B, & t \leq C \\
  D*t + E, & t > C 
\end{cases}$$

Where $C$ denotes the severely threshold of the penalty function above which the penalty is paid with higher slope.

As shown in Figures 5, 6 and 7, the transient workloads were successfully diagnosed by iScale after passing the learning period which was shorter for the third scenario.
The proposed method was compared with the Simple Scaling Method (SSM) in which the re-configuration is performed after each workload change regardless of its durability. Two methods were compared based on the total costs each imposes on the service provider after 60 minutes of simulated workload. The experiment was repeated three times for each composite workload type.

As shown in Figure 8, for the Stable workload type no significant difference is observed between the iScale and SSM method however for the Transient and Mixed workload types the iScale method reduces the overall cost of the migrated application due to ignoring the transient workloads.

5. CONCLUSION

In this paper a novel scaling method to minimize the total costs paid by the application provider during the execution of a migrated-to-cloud application was presented. The proposed method successfully predicted the transient workloads using a LA-based algorithm and was able to avoid scaling out the system resources for the transient workload changes. It was observed that preventing the application from being scaled out/in when the forthcoming workload is not stable results in considerable reduction of the total costs incurred by the application provider. We also proposed novel criteria for discriminating the stable and transient workloads.

REFERENCES


