2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing

Evaluating the Auto Scaling Performance of Flexiscale and Amazon EC2 Clouds

Filippo Lorenzo Ferraris*, Davide Franceschelli*, Mario Pio Gioiosa*, Donato Lucia*
Danilo Ardagna* Elisabetta Di Nitto* Tabassum Sharif†

{filippo.ferraris, davide.franceschelli, mario.gioiosa, donato.lucia}@mail.polimi.it,
{ardagna, dinitto}@elet.polimi.it, tsharif@flexiant.com
*Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
†Flexiant Limited, Edinburgh,UK

***Abstract*—Nowadays cloud computing is becoming one of the most used technological solution to achieve scalability and reduce costs. Scalability is a key point for the success of any business involving the Web and providing services to end-user requests that may vary drastically from one time to another. Sizing a system to provide performance guarantees under peak traffic can be cost prohibitive; the main advantage of cloud computing lays in the opportunity to allocate resources on-demand on a pay per use basis. In this paper we extend Flexiscale public cloud with auto scaling mechanisms and compare these mechanisms with those offered by Amazon. The analysis aims at identifying useful patterns for the execution of Web applications in the cloud and at underlining the critical factors that affect the performance of the two providers. We performed a large set of experiments that demonstrated the importance of tuning correctly the auto scaling parameters.**

***Index Terms*—Cloud computing, scalability, auto scaling, virtualization**

I. Introduction

The interest in cloud computing is increasing rapidly in the IT world. The term cloud computing refers to a set of technologies which provide access to resources as storage units, servers, and applications over the Internet. There are three main categories of cloud computing services: SaaS (Software as a Service), PaaS (Platform as a Service) and IaaS (Infrastructure as a Service). Our analysis concerns IaaS and, in particular, is focused on the performance of the auto scaling capabilities of the underlying infrastructure. Solutions which provide scalable and self-adaptive systems are key factors in the selection of a cloud service provider. In a cloud computing environment resources are allocated on-demand, allowing users to access to a potentially infinite amount of resources in a dynamic way. However, the resource demand usually is not uniform, but it changes according to end-user requests. Thus, the application should have the necessary resources to ensure adequate performance even in case of traffic peaks. In contrast, when the traffic decreases, it would be better to release the unnecessary resources in order to reduce cost. Therefore, dynamic scalability is a critical key point to the success of cloud computing [1].

Performing scaling activities manually is time consuming, so cloud systems provide solutions to automatically scale out or scale in the infrastructure on the basis of predefined policies. Policies are usually composed by a set of scaling rules, each of which is defined by one or more conditions and a set of actions to be performed when those conditions hold. Conditions are typically defined specifying thresholds over a set of performance metrics, including CPU utilization, disk I/O, bandwidth usage and so on. Whenever the observed performance metrics are above or below the thresholds, a predefined number of virtual machines (VMs) instances are added to or removed from applications, respectively. For example, a cloud administrator can define a trigger that launches an additional instance whenever the CPU usage exceeds 70% for five minutes. Although these performance metrics are suitable to monitor the health of the system, they cannot clearly reflect the quality of service of a cloud application. Choosing appropriate performance metrics and finding precise thresholds are not straightforward tasks. The problem becomes more difficult if the workload pattern is continuously changing. Moreover, considering only utilization information may not be sufficient to perform scaling activities. For example, scaling a cluster from 1 to 2 instances can increase capacity by 100%, while scaling from 100 to 101 instances capacity is increased by only 1%. So, such non-constant effects when adding/removing a fixed number of resources must be considered [2]. A possible solution to this problem is to take into account also the current number of active instances to define an adequate increment/decrement.

Another important element to consider in the analysis of the auto scaling mechanism is the start-up latency introduced to run a new VM. The instance is not immediately available as a result of a user's request; cloud infrastructures introduce a start-up delay due to activities such as accessing the required image, booting the VM, configuring network, etc. The start-up delay is one of the elements that we consider to compare different providers in our analyses.

In this paper we investigate the performance of the auto scal-

978-0-7695-4934-7/12 $26.00 © 2012 IEEE
DOI 10.1109/SYNASC.2012.58

ing characteristics of Amazon and Flexiscale cloud providers. The analysis aims at identifying useful patterns for the execution of Web applications in the cloud and at underlining the critical factors that affect the performance of the two providers. We performed a set of experiments demonstrating the importance of tuning correctly the cloud system parameters. The remainder of the paper is organized as follows. The next section introduces the auto scaling capabilities provided by the Amazon infrastructure and the framework we implemented on top of Flexiscale. Section III and IV describes the experimental setup and the results we obtained. Other studies proposed in the literature are discussed in Section V. Conclusions are finally drawn in Section VI.

II. Auto-scaling in Amazon and Flexiscale infrastructures

In order to enable the auto scaling mechanism an IaaS system has to implement at least load balancing and monitoring services. Monitoring services can be used to retrieve some relevant metrics on which alarms and triggers can be defined in order to execute custom actions on certain conditions. This allow to define scaling policies tailored to the current system status. Furthermore, an increment (scale out) or a decrement (scale in) of the server pool size must be transparent to the users. This is achieved by using a load balancer that allows to have a static single entry point for the application even if the server pool size changes due to the scaling activities. Amazon provides both load balancing (*Elastic Load Balancing* - ELB) and monitoring (*CloudWatch*) services and, moreover, the Auto Scaling tool allows to define scaling policies at a very high level. At the moment of writing, Flexiscale does not provide neither load balancing nor monitoring services, so we have developed a prototype environment implementing these services in order to compare the two IaaS solutions.

Auto Scaling [6] is part of the services offered by Amazon in its IaaS public cloud. The core concept of Amazon Auto Scaling is the Auto Scaling Group (ASG). An Auto Scaling Group is a set of different Amazon Elastic Compute Cloud (EC2) instances sharing similar characteristics and subject to the same scaling policies. Therefore, every machine in the group must have the same virtual image and the same hardware characteristics. This represents one of the limitations of the Amazon cloud, since the EC2 Auto Scaling service cannot replace automatically several "small" machines with one single more powerful (scale-up) machine or the other way round (scale-down). Auto Scaling groups are defined with a minimum and maximum number of EC2 instances taken from an EC2 Availability zone. Availability Zones are distinct locations within a Region that are engineered to be isolated from failures in other Availability Zones. Regions represent the geographic locations where Amazon's data centers are placed. CloudWatch [7] is a Web service offered by Amazon to monitor the cloud system and to keep track of various performance metrics. CloudWatch allows to define alarms for each metric to be checked. An Amazon CloudWatch Alarm is a trigger associated with a threshold on a specific metric. The threshold is associated with a comparison operator defining when to fire the alarm, e.g., when the metric is greater than or equal to the specified threshold. An alarm is always associated to an action executed when the alarm fires. This mechanism allows to define, for example, a simple notification system: it is sufficient to specify the action of sending emails whenever an alarm fires. Moreover, CloudWatch Alarms can be used to define scaling policies on instances belonging to the same ASG. For example, it is possible to specify a constant increment of one instance or a percentage increment of 100% of the current number of instances within the group, that is equivalent to doubling the group size. It is also possible to specify the cooldown period within which no other scaling activities can take place, so that the effects of a scaling activity can become visible avoiding collisions with other conflicting scaling policies.

To perform our analysis on Flexiscale we installed a load balancer and a monitoring system since they are not natively supported by the IaaS provider. The load balancer we used is Crossroads [19], a widely used open-source load balancer, that processes all the incoming requests and redirects them to active VMs. Since the load balancer potentially needs to manage a large number of requests without becoming the system bottleneck, it has been installed on the most powerful machine supplied by Flexiscale, i.e. an instance with 8 CPUs and 8 GB of RAM.

To monitor the performance of the server instances we developed a monitoring script, which collects the CPU usage values of the instances, computes the average and decides if it is the case to perform the scale in or scale out script. The monitoring script gathers data from monitoring agents running within each application instance and collecting raw performance data through the Linux SAR tool. Scale in and scale out scripts add or remove VM instances accessing the Flexiscale SOAP API and update the load balancer configuration accordingly [8]. Monitoring script, agents, and scale in and scale out scripts have been developed in Python.

III. Test Environment

This section describes the test environment we setup within Amazon and Flexiscale infrastructures. Experiments are based on a micro-benchmarking Web service application hosted within the Apache *Tomcat* 6.0 application server. The Web service is a Java servlet, we used also in [11], designed to consume a fixed CPU time according to several distributions, e.g., exponential but also log-normal, as observed for several real Web applications [10].

The workload is generated by Apache Jmeter v2.7 exploiting also Jmeter-Plugins v0.5.3, a set of Jmeter extensions which allow to define custom request distributions and to represent results in a graphical way. Jmeter allows to perform remote distributed load tests defining a local master/coordinator and several remote slaves/injectors. Remote injectors perform the requests specified by the coordinator, retrieve the results and send them back to the coordinator. This is a convenient way to perform intensive load tests by

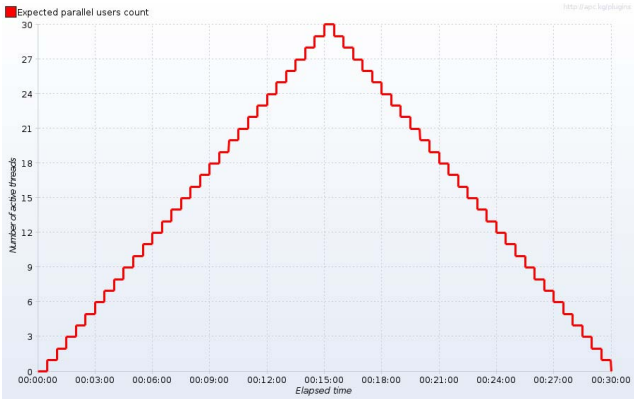

Figure 1. Incoming workload for scenario 1.

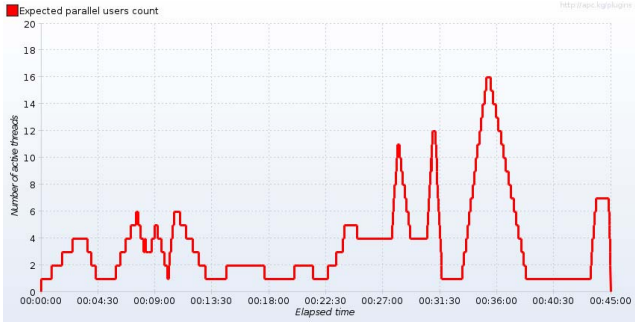

Figure 2. Incoming workload for scenario 2.

distributing the workload among multiple machines guaranteeing that the performance test results are not affected by any bottleneck on the client site.

We considered two test scenarios. In the first one, the incoming workload curve generated by each slave was a stepwise function with a triangular shape (see Figure 1). Even if a triangular function is not representative of a real workload it allows to test easily the behavior of the scale in and scale out activities of the underlying infrastructure. In the second scenario we have considered a synthetic workload obtained form the log trace of a large Web system (see Figure 2).

In order to make a fair comparison of the two providers, we adopted VM instances with similar characteristics in terms of computing power, configuration and geographic location. All tests have been run on Monday June 19th, 2012 at 15.00 CET. As for the auto scaling policy configuration, we have set the following parameters:

- Minimum pool-size: One instance
- Maximum pool-size: Five instances
- Scale in threshold: Average CPU Usage $\leq$ 20%
- Scale out threshold: Average CPU Usage $\geq$ 40%
- Cooldown period: Five minutes
- Average CPU Usage computation interval: One minute

In the next sections we will analyze in detail the configurations adopted and the results we have obtained on each provider.

A. Amazon Test Environment

The tests setup in Amazon required to create an instance for the Web server, an elastic load balancer, and to configure auto scaling policies using the tools provided by Amazon. Jmeter was deployed on four VMs, one dedicated to the coordinator and three for the remote workload injectors. The server instance was a 32-bit platform with 5 EC2 Compute Units[1] (2 virtual core with 2.5 EC2 Compute unit each) and 1.7 GB of memory, which corresponds to a "High-CPU Medium Instance" (c1.medium) [6]. The Jmeter coordinator instance was a 64-bit platform with 8 EC2 compute units (4 virtual core with 2 EC2 Compute unit each) and 15 GB of memory (m1.xlarge), while for each workload injector we used a 64-bit platform with 2 EC2 compute units (1 virtual core with 2 EC2 compute unit) and 3.75 GB of memory (m1.medium). The general schema of the Amazon test environment is shown in Figure 3. VMs were located in Ireland within the same availability zone.

For the health check configuration of the load balancer we changed the default, setting the timeout parameter to 20 seconds, the interval check to 120 seconds, the healthy and unhealthy thresholds to 5 successful and unsuccessful requests, respectively. An important point has been the choice of the health check interval value, that is the time interval used by the ELB to verify if the instances are still alive. The default value was 30 seconds, but we noticed that it was too short. Since the health check is an HTTP request, if an instance is overloaded it may be unable to respond to the load balancer and it is removed from the ASG. On the other hand if we set it to higher values, the load balancer would have taken too long time to figure out if an instance was still active, or if in the meantime another one had been added as a consequence of an auto scaling activity. In other words, increasing the health check value also increases the delay with which new instances are added to the ASG and start serving the incoming workload.

B. Flexiscale Test Environment

To setup the tests within Flexiscale, we have adopted a Web server instance with one single CPU equivalent to a 2.6 Ghz Opteron dual core processor and 2GB of memory. Flexiscale datacenters are located in South-East England. For the workload injection we used the same configuration adopted for Amazon tests. In order to have homogeneous results, also the scale in and scale out thresholds, as well as the the cool down period and the minimum and maximum number of instances within the auto scaling group, were the same as in Amazon. An overview of the Flexiscale test environment is shown in Figure 4.

[1]One ECU (EC2 Compute Unit) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. This is also the equivalent to an early-2006 1.7 GHz Xeon processor

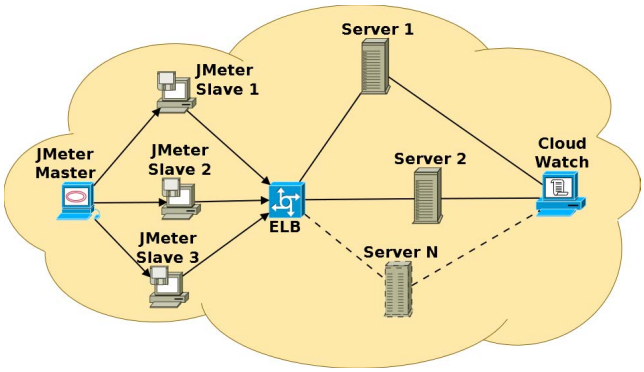

Figure 3. Amazon Test Environment.

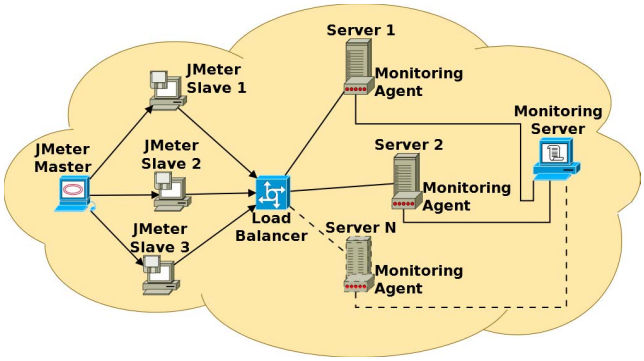

Figure 4. Flexiscale Test Environment.

We have also compared Amazon and Flexiscale costs which are reported in Table I. It is evident that Flexiscale server instances are cheaper, considering also that Flexiscale pricing is based on actual usage, i.e. the number of minutes effectively used are charged, while Amazon charges on a hourly basis. However, Flexiscale experiments required to run the load balancer and the monitoring server on dedicated instances and this may lead to higher costs with respect to the same Amazon services. Indeed, with Amazon pricing model ELB and monitoring costs depend on the served workload. Therefore in a real business scenario, the overall costs should be taken carefully into account in order to choose the most suitable cloud service.

Instance Type	Amazon	Flexiscale
Server	0.186 $/h	0.055 $/h
Load Balancer	0.028 $/h + 0.008 $/GB	0.264 $/h
Monitoring	0.005 $/h	0.11 $/h

Table I
COSTS COMPARISON BETWEEN FLEXISCALE AND AMAZON.

IV. TEST RESULTS

In the plots reported in this section a red bullet (•) denotes a scale out alarm firing, a green bullet (•) represents a scale in alarm firing and a black cross (×) represents the activation of a new instance.

In the first load test scenario we have considered an exponentially distributed service time with mean $D = 0.2$s. The workload peak is reached after 15 minutes injecting 90 requests per second. As expected, we noticed that the system rapidly run out of capacity. In the middle of the test period, the load balancer serves a large number of HTTP requests and most of them are lost due to the excessive load. Each HTTP request has a maximum timeout equal to 120 sec and this was the main cause related for request loss. Even if the request load is decreasing in the descent part of the curve, the queues still remain full and the system loses a significant number of the requests.

As regards the CPU, the Amazon average CPU utilization reaches a maximum of 55%-60%, while Flexiscale reaches 100% CPU load. This discrepancy is probably due to how we have measured the CPU utilization on Flexiscale instances. Indeed, we have exploited the SAR tool which computes the average CPU utilization over a specified time span. This tool operates at operating system level, while Amazon CloudWatch performs measurements at hypervisor level, which is more accurate. Concerning the maximum response time, on Amazon we registered 120 sec (i.e., the http request time out), instead on Flexiscale we registered 60 sec. Regarding the load balancer, within Amazon there is an health check every 120s and if the health check of one machine fails, the machine is considered unavailable and removed from the ELB. This is a tricky issue, since, if the queue is full but the instance is still working, the health check of that instance can fail because of timeout expiration, i.e., the instance is considered unhealthy. Vice versa the Crossroad load balancer we deployed in Flexiscale sends a TCP request instead of an HTTP request in order to perform the health check. In this way the health check is independent from the queue length of the Web server and this is one of the reasons why Flexiscale performs better than Amazon on average. However, using the Crossroads load balancer has a drawback: few HTTP requests are lost during the operation of adding and removing servers, while the Amazon load balancer did not lose any requests during auto scaling activities.

Amazon ELB requires a trade-off between short and large health check intervals. Indeed, if one chooses a short interval, a new instance can be attached to the load balancer in a very short time but, on the other hand, it is easier to reach the unhealthy request threshold, so that the instance could be wrongly considered unhealthy and detached from the load balancer. Otherwise, if one chooses a large interval, the probability that an unhealthy request threshold is reached when servers are still working properly but overloaded is lower. On the other hand it takes more time to attach a new instance to the load balancer, so that an instance could stay

idle until the healthy threshold is reached. Running our tests we have noticed that alarm firings are not always reported by Amazon CloudWatch. This happens when the system, after exceeding a threshold and performing the action specified in the policy, does not change health status and remains in a critical condition throughout the cooldown period and beyond. For example this happens when, despite removing an instance, the average CPU utilization remains below 20%, so the alarm does not come back to the normal status. This consideration is valid only for notifications, in fact the scaling actions are regularly executed according to the rules specified in the policies. This can be a serious problem when estimating the provisioning time of the instances. We did not experience such problems on Flexiscale since we can rely on our own scripts that keep track of every event and therefore we can estimate the provisioning time of each instance in a simpler and accurate way. In our experiments we have measured that Amazon instances require about 2.5 minutes to start up, while Flexiscale instances have a provisioning time of about 2 minutes. So the two systems perform more or less equally with respect to the provisioning times. The detailed result are shown in Figures 5 and 6 for Amazon, while Flexiscale results are shown in Figures 7 and 8.

Scenario 2, reported in Figure 2, can be split in three different parts. In the first one, from minutes 0 to 10, we are testing the scale out policy. In the second part, from minute 10 to 22, there is a long period with a low requests rate to test the scale in performance. In the last part the workload curve, from minute 22 to 45, has some peaks that allow to test the auto scaling adaptivity to a varying workload. On the server side we have considered a log-normal service time distributions with mean $D = 0.2$s and standard deviation $\sigma = 0.4$s representative of a moderate long-tailed distribution for Web applications [10].

In this test case we can notice significant differences between the two systems, as shown by the plots reported in Figure 9, 10 (for Amazon), 11, 12 (for Flexiscale). Overall, Flexiscale obtains better performance despite the incoming workload and auto scaling policy are the same. After the first period of heavy load, Flexiscale counts four active instances while Amazon counts only three active instances. In the following phase, from minute 12 to 23, when the number of requests is moderate, Amazon performs the scale in of two instances, while Flexiscale performs the scale in of a single instance. The choice to perform two scale in actions just before the following peak at minute 23 has been an unfortunate event for Amazon. In fact it faces not only an increasing number of requests with only one active instance, but it has also to wait the whole cooldown period until performing the next scale out operation. On the other hand, Flexiscale performs the scale out for the third time at 12:27 (3 minutes after the last scale out action in Amazon), just before requests rate decreases. After that, it cannot perform a scale in before the cooldown period is elapsed, although the average CPU usage is below 20%. In fact, Flexiscale performs the scale-in at 18:14 for the first time, staying with three instances. At minute 23, when the requests

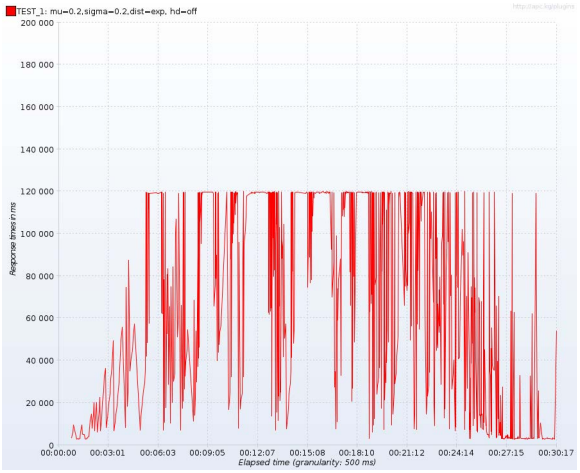

Figure 5. Amazon Scenario 1 Response time.

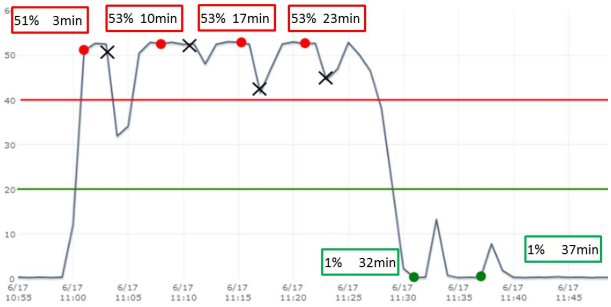

Figure 6. Amazon Scenario 1 Average CPU utilization.

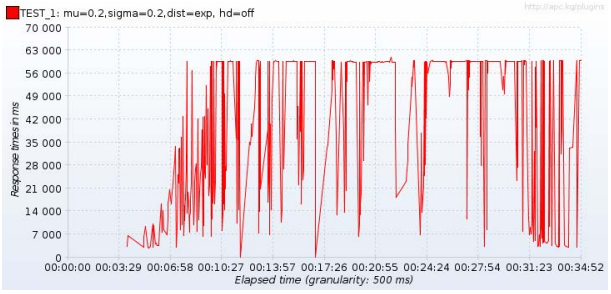

Figure 7. Flexiscale Scenario 1 Response time.

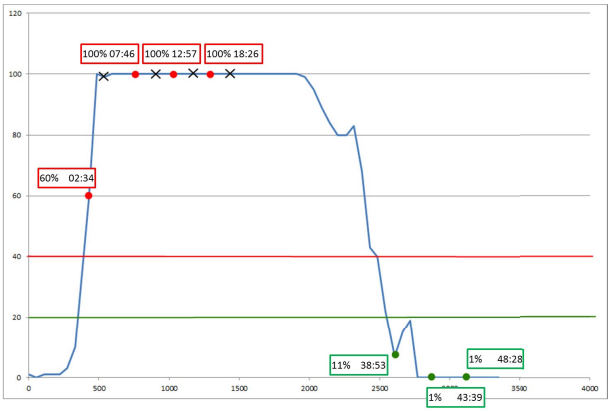

Figure 8. Flexiscale Scenario 1 Average CPU utilization.

rate increases, Flexiscale performs the scale out process just a minute later, when the cooldown period expires.
As a matter of facts, Amazon reaches a maximum response time of 120 seconds and presents a huge amount of packets losses, while the Flexiscale response time is limited to 40 seconds.

V. Related work

With the development of cloud systems, dynamic resource allocation techniques have received a great interest both within the industry and academia. The solutions proposed can be classified in centralized and distributed. In a centralized approach, a dedicated entity is in charge of establishing capacity allocation for the whole system and has a global knowledge of the resources state in the whole network [12], [13], [14].

Centralized solutions are not suitable for geographically distributed systems, such as the cloud or more in general massively distributed systems [15], [16], [17], since no one entity has global information about all system resources. Indeed, the communication overhead required to share the resource state information is not negligible and the delay to achieve state information from remote nodes could lead a centralized resource manager to very inaccurate decisions due to dynamic changes in system conditions, as well as resource consumption fluctuations or unexpected events [16].

Distributed resource management policies have been proposed to govern efficiently geographically distributed systems that cannot implement centralized decisions and support strong interactions among the remote nodes [15]. Distributed resource management is very challenging, since one node's decisions may inadvertently degrade the performance of the overall system, even if they greedily optimize the performance of its nodes. Sometimes, local decisions could lead the system even to unstable oscillations [18]. It is difficult to determine the best control mechanism at each node in isolation, so that the overall system performance is optimized. Dynamically choosing when, where and how allocate resources and coordinating the resource allocation accordingly is an open problem and is becoming more relevant with the advances of clouds [17].

There has been a huge effort on understanding and analysing optimal auto scaling policies in the cloud computing field taking into account different constraints [3], [4], [5], [2]. In [3], authors use a mechanism that scales up and scales down VM instances evaluating two points of view: performance and budget constraints. This mechanism uses integer programming to solve the instance startup plan generation as an optimization problem. In [5], cloud-based Web applications are considered and a specific metric to determine the scale up/down factor is proposed. The scaling algorithm that has been developed takes into account the number of active sessions. Their analysis has demonstrated the benefits of using a custom scaling system. In [2], authors showed that prior works concerning automatic control could not work when used in a cloud infrastructure, since scaling rules are based on the actual state of the system and consider only the current number of machines. In other words, the rules have to be set in a proportional way and not

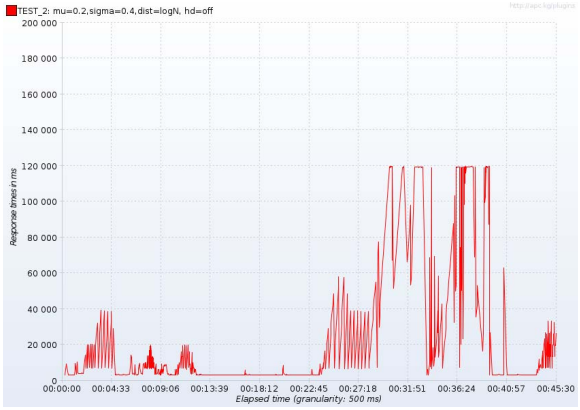

Figure 9. Amazon Scenario 2 Response time.

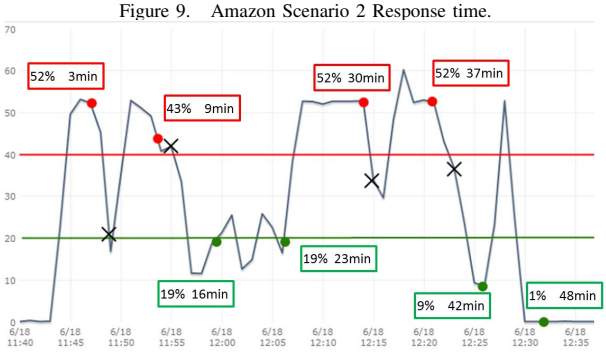

Figure 10. Amazon Scenario 2 Average CPU utilization.

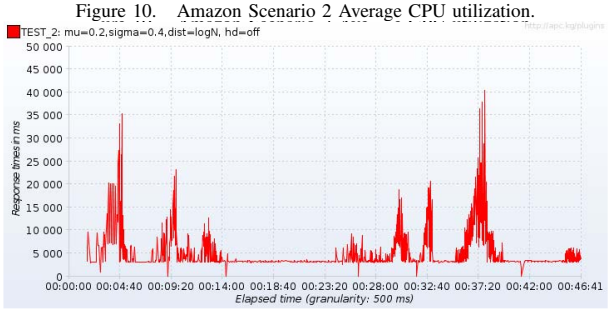

Figure 11. Flexiscale Scenario 2 Response time.

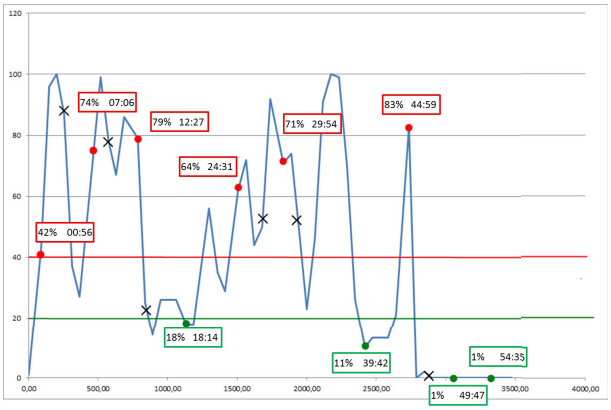

Figure 12. Flexiscale Scenario 2 Average CPU utilization.

in a constant one. There are two important limitations with their work: they do not consider different VM types and they ignore the total running cost.

The results of a performance and availability analyses on multiple Cloud providers have been reported in [9]. To the best of our knowledge, our work is the first one providing an in-depth experimental analysis of the auto scaling capabilities of IaaS solutions.

VI. Discussion and Conclusions

In this paper we have presented the results we achieved stressing the auto scaling mechanisms implemented by Amazon and Flexiscale cloud providers. Auto scaling is a key-point for the performance, availability and costs of a cloud environment. However, our experimentation has shown that the tuning of auto scaling parameters is challenging and more work need to be developed in order to implement a pro-active behaviour, since simple actions triggered after threshold violations cannot provide performance guarantees under critical workload conditions.

For what concerns the comparison between Flexiscale and Amazon, we can state that Flexiscale performance are better than Amazon, however a custom implementation of a load balancer and a monitoring framework need to be developed and the cost for load balancing could be higher under light load conditions.

Even if Amazon IaaS is one of the most mature in the market, Amazon auto scaling presents some limitations and probably a mechanism able to alert the load balancer when an instance is up and running, independent of the health check time period, should be developed.

In our research agenda we plan to improve and integrate the auto scaling mechanisms we have developed within the Flexiscale infrastructure and offer this services through the Flexiscale API. Furthermore, we plan to provide a performance model based framework able to provide end-to-end response time guarantees at the application level. Finally, the auto scaling analysis will be extended to include also major PaaS providers.

Acknowledgement

The experimentation on Amazon EC2 has been supported by the AWS in Education research grant. The work of Danilo Ardagna and Elisabetta Di Nitto has been partially supported by the European Commission, Programme IDEAS- ERC, Project 227977-SMScom.

References

[1] Caron, Eddy and Rodero-Merino, Luis and Desprez, Frédéric and Muresan, Adrian, *Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds*. Laboratoire de l'Informatique du Parallélisme - LIP.

[2] H. Lim, S. Babu, J. Chase, and S. Parekh, *Automated Control in Cloud Computing: Challenges and Opportunities*. In 1st Workshop on Automated Control for Datacenters and Clouds, 2009.

[3] Mao, Ming and Li, Jie and Humphrey, Marty, *Cloud auto-scaling with deadline and budget constraints*. IEEE, 2010.

[4] Eddy Caron, Luis Rodero-Merino, Frdric Desprez, Adrian Muresan, *Auto-Scaling, load balancing and monitoring in commercial and open-Source clouds*. INRIA, 2012.

[5] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, Alla Segal, *Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment*. IEEE Computer Society, 2009.

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

[7] http://aws.amazon.com/cloudwatch/

[8] http://www.flexiscale.com/reference/api/

[9] Cloud Performance form the End User. http://www.bitcurrent.com/downloading/?fid=1851

[10] E. Veloso, V. Almeida, W. M. Jr., A. Bestavros, and S. Jin. A Hierarchical Characterization of a Live Streaming Media Workload. *IEEE/ACM Transactions on Networking*, 14(1), 2006.

[11] D. Ardagna, M. Tanelli, M. Lovera, L. Zhang.. Black-box Performance Models for Virtualized Web Service Applications. *WOSP/SIPEW Proc*, 2010.

[12] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang. Energy-Aware Autonomic Resource Allocation in Multi-tier Virtualized Environments. *IEEE Transactions on Services Computing. 5(1), 2-19, 2012*.

[13] D. Ardagna, B. Panicucci, M. Passacantando. Generalized Nash Equilibria for the Service Provisioning Problem in Cloud Systems. *IEEE Trans. on Services Computing,* available on line.

[14] B. Urgaonkar, G. Pacifici, P. J. Shenoy, M. Spreitzer, and A. N. Tantawi. Analytic modeling of multitier Internet applications. *ACM Transaction on Web*, 1(1), January 2007.

[15] M. Andreolini, S. Casolari, and M. Colajanni. Autonomic request management algorithms for geographically distributed internet-based systems. In *SASO*, 2008.

[16] H. Feng, Z. Liu, C. H. Xia, and L. Zhang. Load shedding and distributed resource control of stream processing networks. *Perform. Eval.*, 64(9-12):1102–1120, 2007.

[17] H. Erdogmus. Cloud computing: Does nirvana hide behind the nebula? *IEEE Softw.*, 26(2):4–6, 2009.

[18] P. Felber, T. Kaldewey, and S. Weiss. Proactive hot spot avoidance for web server dependability. *Reliable Distributed Systems, IEEE Symposium on*, pages 309–318, 2004.

[19] http://crossroads.e-tunity.com/