Elastic Virtual Machine for Fine-grained Cloud Resource Provisioning

Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel

Hasso Plattner Institute,
Potsdam University,
Potsdam, Germany
`firstname.lastname@hpi.uni-potsdam.de`

Abstract. Elasticity is one of the distinguishing characteristics associated with Cloud computing emergence. It enables cloud resources to auto-scale to cope with workload demand. Multi-instances horizontal scaling is the common scalability architecture in Cloud; however, its current implementation is coarse-grained, while it considers Virtual Machine (VM) as a scaling unit, this implies additional scaling-out overhead and limits it to specific applications. To overcome these limitations, we propose Elastic VM as a fine-grained vertical scaling architecture. Our results proved that Elastic VM architecture implies less consumption of resources, mitigates Service Level Objectives (SLOs) violation, and avoids scaling-up overhead. Furthermore, it scales broader range of applications including databases.

Key words: virtualization, elasticity, auto-scaling, cloud computing

1 Introduction

The rapid growth of E-Business and the frequent changes in the sites contents and the customers interest pose the need for rapid and dynamic scaling of resources. Fortunately, Cloud computing Infrastructure as a Service (IaaS) model, based on Virtualization technologies such as Xen [4]; VMWare [3]; and KVM [14], enables agile and dynamic provisioning of resources. However, current scalability implementation offered by IaaS providers, e.g. Amazon EC2 [5] and GoGrid [1], has the following limitations: first, it uses VM as a scaling unit, which is course-grained scaling that can cause unnecessary over-provisioning. Second, it implies running additional load-balancer VM instance. Third, multi-instances scaling architecture is limited to specific application and can't scale-out applications like databases which do not support clustering or synchronization; applications which are not designed to be distributed (singleton applications); or applications with expensive licenses.

Typically, web applications and services consist of three tiers: web tier, application tier, and database tier. The incoming requests go through these tiers to get back with the results. According to the application characteristics, each tier may cast an intensive demand to specific tier resources making it a bottleneck. For example, Amza et al. implemented a benchmark that simulates the behavior of online bookstore such as amazon.com, bulletin board websites, and auction websites [6]. Analyzing these benchmarks shows that the CPU of database tier of online bookstore is the bottleneck, while for auction sites and bulletin board, the CPU of the web-tier is the bottleneck. Unfortunately, the dependency between these tiers propagates degradation in performance of one tier to the whole application, therefore, to cope with traffic load and eliminate bottlenecks of multi-tier applications, the first step is to detect the bottleneck tier, and then to scale it dynamically.

In this paper, we propose a fine-grained scaling architecture. That is Elastic VM architecture. It implements the scalability characteristic into VM resources level (e.g. Number of cores, CPU capacity (%), and Memory (MB)). The architecture can be implemented into any tier. If any of these tiers turned to be a bottleneck, the Elastic VM scales vertically to cope with workload demand. To compare our architecture with current multi-instances architecture, we implemented both architectures locally, and considered the parameters that control both *Amazon Elastic Load Balancing* and *Amazon Auto Scaling models.* For workload simulation, we installed RuBBoS benchmark [6] which is a simple bulletin board benchmark implemented as two-tiers system.

In the next section, we explain in details the proposed Elastic VM scaling architecture and compare it to the conventional multi-instances scaling architectures. In section 2, we describe our experiment setup and analyze the results. In section 4, we compare our research to related work. Finally, in section 5, we conclude our work and point out our extended research.

2 Scalability Architectures

This section starts with an overview of the current scalability architecture in the cloud infrastructure using multi-instances. This is followed by an overview of our proposed Elastic VM scalability architecture. Afterwards, we compare both architectures performance analytically. Finally, we discuss how the implementation of each architecture influences its performance.

2.1 Multi-Instances Architecture

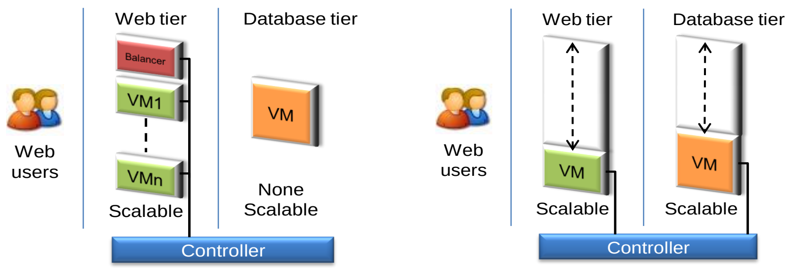
Implementation of scalability in the current (IaaS) providers, like Amazon and GoGrid, is illustrated in figure 1(a). Users' requests are directed to a load-balancer (i.e. balancer) which forwards it to the available web servers (i.e., VM1 to VMn) according to a specific load balancing policy (e.g., Round Rubin).

To maintain a determined QoS, a controller monitors instances of the web-tier, if the monitored performance metrics (e.g., CPU utilization) of current group of instances exceeded a user specific threshold, controller will provision more instances to maintain QoS. On the other hand, if the performance metrics exceed a user specified lower threshold, it will release some instances to reduce cost.

2.2 Elastic VM Architecture

Elastic VM is a VM that runs a modified kernel that supports on-the-fly resources scaling feature without interrupting the service or rebooting the system. In addition, the hypervisor is extended with interfaces that enable modifying VMs resources by programming languages.

Figure 1(b) illustrates implementation of Elastic VM into two-tier system (i.e., web-tier and database tier). As in multi-instances architecture, a controller monitors tiers performance and scale Elastic VM resource dynamically to cope with workload demand.



(a) Multi-instances Scaling Architecture (b) Elastic VM Scaling Architecture

Fig. 1: Multi-instances vs. Elastic VM Scaling Architecture

2.3 Multi-Instances vs. Elastic VM Response Time

In this section we analyze the average response time of both Multi-instances and Elastic VM using the queuing analytical model. The analyzed time is the waiting time spent by the packet in the queue until being served added to the service time. Each VM instance with one virtual CPU (vCPU) is modeled as a single-server queue (M/M/1), while the Elastic VM with several vCPUs is modeled as a single queue with multiple servers (M/M/c).

According to Kendall's notation [16], M is a notation for Markovian (exponential) distribution, which means that both the inter-arrival time and service time are exponentially distributed. The mathematical model that describes M/M/c model is as the following:

System utilization ρ is calculated by:

$$\rho = \frac{\lambda}{c * \mu} \tag{1}$$

The probability of having zero request in the system:

$$P_0 = \left[\sum_{n=0}^{c-1} \frac{\lambda^n}{n!} + \frac{\lambda^c}{c!(1-\lambda/c)} \right]^{-1} \tag{2}$$

Expected average queue length:

$$E(m) = P_0 \frac{\rho^{c+1}}{c.c!} \frac{1}{(1-\rho/c)^2} \tag{3}$$

Expected average number of requests in the system:

$$E(n) = E(m) + \rho \tag{4}$$

Expected average total time spent by a request in the system:

$$E(v) = E(n)/\lambda \tag{5}$$

Expected average waiting time spent by a request in the queue:

$$E(w) = E(v) - 1/\mu \tag{6}$$

An example to compare the average response time of Multi-instances architecture with Elastic VM architecture is as follows:

First, consider a multi-instances architecture running 4 VMs instances in parallel, each machine is modeled as a single queue served by one server (vCPU). Assuming that each vCPU has the capacity to serve 100 req/sec and the total traffic rate to the whole system is 320 req/sec distributed fairly by the load balancer to be 80 req/sec for each VM instance. In this case the VM utilization $\rho = \frac{80}{100} = 80\%$, hence, *the average response time* = 0.05 and the *waiting time* = 0.04 calculated by equations 5 and 6 in consequence where c=1, λ =80, and μ =100.

Second, consider the same traffic directed to Elastic VM with 4 vCPUs (i.e., c=4), in this case, the system utilization is calculated as $\rho = \frac{320}{4*100} = 80\%$, while *the average response time* = 0.017 and *the average waiting time* = 0.007 calculated by equations 5 and 6 where c=4, λ =320, and μ =100.

The above example, is repeated for different values of c while keeping systems utilization equals 80%, the results are presented in table 1.

Table 1: An Elastic VM with different number of vCPUs (i.e., c) compared with a Static VM with one vCPU (i.e., c=1)

c	1	2	3	4	5	6	7	8
λ	80	160	240	320	400	480	560	640
Average response time (seconds)	0.05	0.027	0.02	0.017	0.015	0.014	0.013	0.013
Average wait time (seconds)	0.04	0.018	0.01	0.007	0.006	0.004	0.003	0.003

As shown in table 1, a single VM with multiple vCPUs, as in Elastic VM implementation, performs better than multiple of VMs instances each with one vCPU running in parallel, even though there is the same total number of vCPUs in both systems. In section 3, we observe how this behavior could influence the system performance in real environments.

2.4 Elastic VM vs. Multi-Instances Architecture Implementation

By analyzing the scaling architectures implementations characteristics of both Elastic VM and Multi-Instances architecture, we can summarize pros and cons of each architecture as in table 2:

Table 2: Comparison between Mutli-instances and Elastic VM architecture implementation

Mutli-instances	Elastic VM
It implies running load-balancer (i.e., additional consumption of resources)	No need for running additional machine as a Load-balancer
Scalability is limited to specific tiers	It is applicable to any tier
It uses VM as a scaling unit (coarse-grained scale)[1]	It is fine-grained scaling while it implements scaling by the real units of resources
Scaling-down can interrupt sessions-based web connections	It supports sessions-based web connections
Booting time of VMs, to scale-out, increases the overhead	It eliminates the overhead caused by booting VMs
Scale-out overhead, cause SLO violation and decrease the throughput	Scale-up vertically maintains better performance, less violation of SLO, and higher throughput
Both software and hardware load-balancer can be a single point of failure	Elastic VM itself could be a single point of failure [2]
It supports all small, medium, and big business	Currently, Elastic VM scaling is limited to host machine capacity, which limits Elastic VM to small and medium business [3]

[1] One solution is to have smaller VM instances (e.g., Amazon micro-instance), however, this solution could reduce the probability of over provisioning, but do not eliminate it totally.

[2] Compared to static machines, Elastic VM ability to scale-up make it more resistance to failure that could be caused by overloading, this characteristic

makes it a recommended replacement to static load-balancer instances as an example.

[3] If a global policy is enabled to reallocate VMs according to host utilization, it will be possible to move VMs with lower load into another hosts to make more room for the overloaded Elastic VM to scale-up. Moreover, integrating both Elastic VM and Multi-instances architecture together can come up with fine-grained scaling architecture which is unlimited to one physical host.

One of the challenging issues in Elastic VM is that some applications could be unaware of dynamic resources scaling, especially memory scaling, fortunately, many researches are directed to optimizing applications parameters according to available resources like [9], [8], [15], [19], and [18]. Such approaches can be incorporated into our architecture to tune applications parameters for optimum performance after each scaling.

3 Experimental Setup

For the sake of the practical comparison between multi-instances and our Elastic VM scaling architectures, we implemented a similar architecture to Amazon Elastic Load Balancing and Amazon Auto Scaling models considering the same parameters that supposed to be defined by user. These parameters and the corresponding values, which are used within our experimental setup, are listed in table 3.

Table 3: Most significant parameters that control Amazon Scaling Model

Parameter description	Value
Minimum number of running instances	1
Maximum number of running instances	4
Monitored metric	CPU Utilization
Monitored metrics' measurement period	5 seconds
Lower threshold of measured metric	80
Upper threshold of measured metric	90
Breach duration	60 seconds
Lower breach increment	-1
Upper breach increment	1

Table 3 can be translated by our implementation as follows: The minimum number of running instances at anytime is 1 VM instance, and the maximum is 4 instances. The measured metric is CPU utilization, which is measured every 5 seconds. Before any scaling-down, a check for the new utilization to be less than 80% should be done with the following simple equation:

$$CPU^{util}_{next} = CPU^{util}_{current} * number\_of\_instances/(number\_of\_instances - 1) \quad (7)$$

The scaling-out is triggered when VM instances utilization exceeds 90%. To prevent oscillating which maybe caused by changing CPU utilization quickly, a specific period of time (Breach duration) is left to give the system a chance to reach a stable state after each scale. Finally, the last two parameters in table 3 determine the scaling step size.

In addition to above parameters, during our experiments, we discovered many modifications that can improve the performance of current auto-scaling implementation within the cloud. First, the current implementation of the Multi-instances auto-scaling in Amazon Ec2 considers only one breach duration value for both scaling-out and scaling-down. In our implementation, it is split into two values: breach-out duration for scaling-out, and breach-down duration for scale-down. The small breach-out duration value enables a rapid scale-out. Rapid scale-out is very import to cope with sudden surge of the traffic. On the other hand, breach-down duration should always be larger than breach-up duration to prevent uncertain scale-down, which could be more harmful to the system performance.

The same aforementioned parameters described are applied to Elastic VM scaling architecture but instead of changing VMs instances number, Elastic VM is scaled by changing the number of vCPUs and the memory size.

Testbed Setup: Our experiment conducted on a testbed of two machines (Client and Server) connected by 1 Gbps Ethernet. Server machine has Intel Quad Core i7 Processor, 2.8 GHz and 8GB of Memory. It runs Xen 3.3 with kernel 2.6.26-2-xen-686 as hypervisor. On the hypervisor, VMs with Linux Ubuntu 2.6.24-19 are hosted. Some of these hosted VMs run Apache 2.0 as a web server in prefork mode. One of them has additional mod_proxy extension installed to enable load balancing. Furthermore, a single VM machine runs MySql to host benchmark database.

Workload Generation: To evaluate our architecture, we installed RuBBoS benchmark [6]. RuBBoS is a bulletin board benchmark that implements the essential bulletin board features of the Slashdot site [2]. The benchmark is implemented as two tiers: front-end tier which is a web-server that enables PHP modules, and a back-end tier which is a database stores users information, posts, and comments. The user discussions are started as threads, each thread has a story at its root and many comments for that story. There are two types of users in RuBBoS: *Regular users* who mainly browse, start threads, and submit comments. *Moderators* who can review stories and rate comments in addition to the basic features available to regular users.

The workload is generated by RuBBoS clients which is written in Java. Each RuBBoS client emulates hundreds of HTTP clients. An HTTP client issues a sequence of requests with thinking time of 7 seconds, according to clause 5.3.1.2 of the TPC-W v1.8 specification [17]. One of the important parameters in RuBBoS benchmark client is the *workload_number_of_clients_per_node* parameter. It determines number of running threads that will be initiated in each RuBBoS benchmark client machine. Each thread emulates one user behavior, so for generating variant workload, we set different value of *workload_number_of_clients_per_node* for each workload step.

Utilization window: During the experiments, we noticed small periods of low CPU utilization appear among high CPU utilization values. The bad influence of such values appears when the system exceeds breach-down period. In this case, any uncertain low CPU utilization can instantly trigger a false scaling-down. The false scale-down reduces the system capacity and ability to cope with current workload. From our observation, this sudden low CPU utilization values are unavoidable, while it could be one of the workload characteristics, as we will seen in section 3.1. As a solution, we had a window of 10 values that stores a sorted list of the last CPU utilization. The second highest value of this window (i.e., 90th) is considered as the current CPU utilization. This maintains rapid scale-out but add confidence to scale-down triggers. Each point of the utilization window is measured by calculating the average CPU utilization for 5-seconds period of time.

3.1 Web-tier Scalability

The goal of the following experiments is to compare web-tier scalability using Multi-instances with web-tier scalability using Elastic VM scalability architecture. The comparison includes total throughput and SLO violation. For this experiment we assume (20 ms maximum response time) as a SLO. It is measured as the difference between the moment of packet arrival to network interface of physical host until the moment of getting out with the result. Response time is relatively small; this is because it excludes any delay caused by the network.

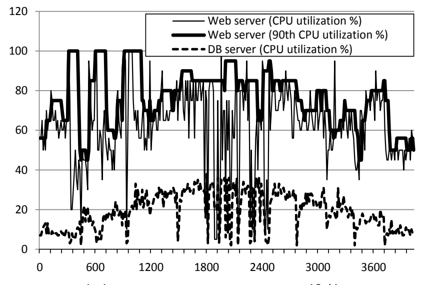
To experiment web-tier scalability without any influence from database-tier, we ran RuBBoS benchmark [6] with *browse_only_no_search_transitions* traffic pattern. This traffic pattern is selected because it casts high workload to web-tier and a low workload to database-tier which prevents database-tier from being a bottleneck at any stage of the experiment.

Web-tier Scalability implemented by Multi-instances: In this part of the experiment, we implement the scalability into web-tier using Multi-instances scalability architecture. To experiment this setup, we ran the step traffic seen in figure 2(d). Each step shows the number of users (sessions) initiated by RuBBoS client host. The number of sessions seen in figure 2(d) is scaled-down by 4 to fit in the figure. Each instance of web-tier VMs is set up with one virtual CPU, and 1 GB of memory.

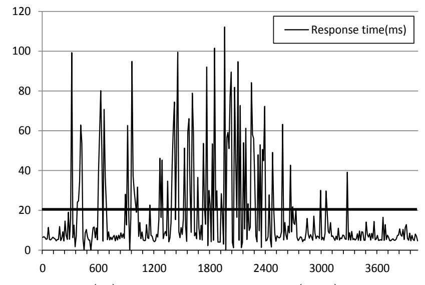
Load-balancer: The load-balancer is a VM instance set up with one virtual CPU and 512MB of RAM. The load-balancer software is Apache web server with *mod_proxy_balancer* module enabled. Apache load-balancer has a static configuration files that describe available web servers. During load-balancer running, it checks periodically these servers availability. If one of these web servers is turned down, the load-balancer will spend time to discover and stop forwarding traffic to it. Therefore, to improve the load-balancer response with the possible rapid change of the available VMs, we designed an interface that allows the controller to update the load-balancer configuration files dynamically with each change in number of VMs.

At the start of this part of the experiment, a single VM web-server instance was able to cope with workload of 200 sessions. But, at second 300, as in figure 2(d), when the sessions number jump from 200 to 400, the CPU utilization of the instance was very high. Accordingly, the controller provisioned more VMs instance rapidly to cope with traffic surge. At second #444, the CPU 90th utilization was 50% which allows the controller to scale-down to 3 VMs. At second #516, the CPU utlization still low, and the scale-down breach time period is passed (i.e. 60 seconds). This allows the contoller to scale down to 2 VMs. Along the remaining experiment run, the controller continue to scale-out rapidly and scale-down slowly to cope with traffic's variation.

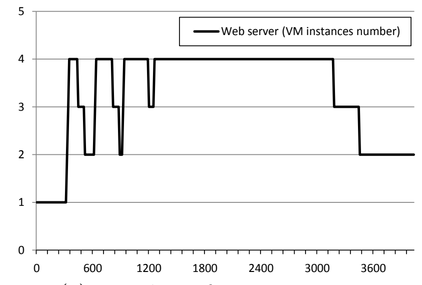
Utilization window: With the help of utilization window, the uncertain low values of the CPU utilization (e.g., CPU utilization at seconds 444, 936, and 1884) are filtered by utilization window. This saves the system from the oscillating and adds more stability to scaling-down process.



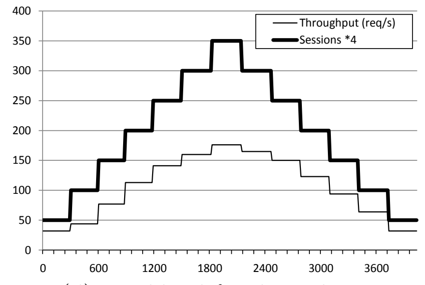
(a) CPU Utilization (%)



(b) Response time (ms)



(c) Number of VM instances

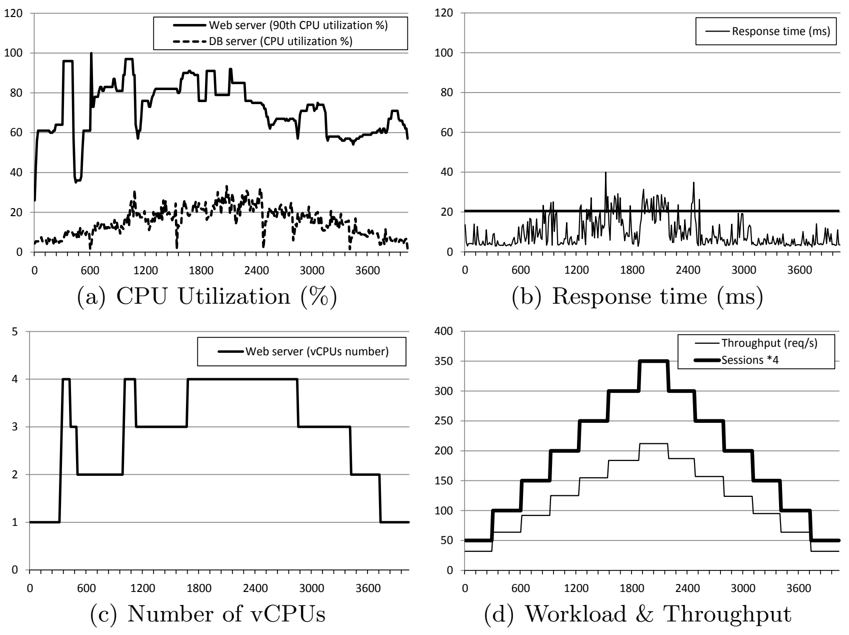


(d) Workload & Throughput

Fig. 2: Multi-instances Architecture implemented into web-tier

Analyzing response time curve at figure 2(b) shows a high number of violations to SLO, it reaches 23.4% of the total experiment run time. SLO violations are caused by the booting time overhead of the VMs instances which delays its ability to cope quickly with the traffic surge. Also, figure 3(d) shows a decrease in the throughput of the ramp-up step traffic compared with the ramp-down step traffic. For example at step 400 sessions, the throughput for ramp-up step is 44 req/sec, while it is supposed to be 64 req/sec. This is also because of the VMs booting time overhead.

Web-tier Scalability implemented by Elastic VM: In this part of the experiment, we implement web-tier scalability using Elastic VM. To test this setup scalability, we ran the same step traffic described in the first part of this experiment.



(a) CPU Utilization (%) (b) Response time (ms)

(c) Number of vCPUs (d) Workload & Throughput

Fig. 3: Elastic VM implemented into web-tier

Observing response time at figure 3(b) shows that proposed Elastic VM architecture was able to mitigate SLO violation, (i.e., 12.5%) of experiment run time. It is half the SLO violations in case of Multi-instances architecture. Also, figure 3(d) shows that average throughput (req/sec) for each ramp-up step almost equals average throughput of the equivalent ramp-down step. In other words, symmetry of throughput curve in figure 3(d) means that the scaling-up with Elastic VM does not degrade the throughput.

To compare the two architectures performance, we assume that the Elastic VM is the reference, and calculate the degradation in performance in the case of Multi-instances architecture. The results are illustrated in table 4.

Table 4: Throughput degradation in Multi-instances architecture

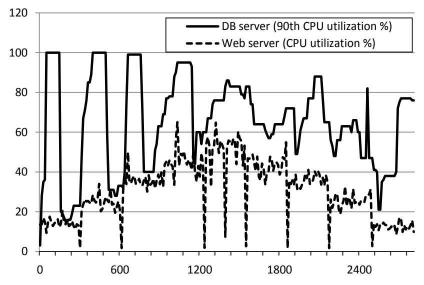
Sessions Number	Elastic-VM throughput (req/sec)	Multi-instances throughput (req/sec)	Multi-instances throughput degradation (%)
200	32	32	0
400	64	44	31
600	92	77	16
800	125	113	10
1000	155	141	9
1200	184	160	13
1400	212	176	17
1200	187	165	12
1000	157	150	4
800	124	123	1
600	95	94	1
400	64	64	0
200	32	32	0

By analyzing results in table 4, we come to the following observations: First, Elastic VM throughput is always better than Multi-instances architecture in case of ramp-up traffic (left side of figure 2(d)), because it copes faster with traffic surge. Second, for high level values of traffic (i.e., 1000, 1200, and 1400), Elastic VM throughput is also better than Multi-instances, this observation confirms the theoretical analysis in section 2.3 where an Elastic VM with 4 vCPUs implies less response time compared with 4 VMs instances running in parallel, this also enables Elastic VM to serve more requests and maintain higher throughput.

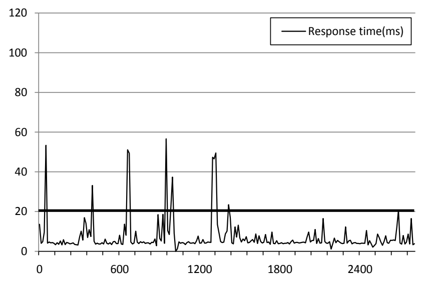
3.2 Elastic VM as a Scalable DB Server

In this part of experiment, we will study Elastic VM scalability as a database server. To make database tier the bottleneck, we tuned RuBBoS benchmark with traffic pattern that implied more search requests to put more workload on the database, therefore, the requests type was as the following: StoriesOfTheDay (12%), ViewStory(14%), SearchInStories (25%), and the remaining was distributed as same as the default_user_traffic pattern designed by RuBBoS benchmark.

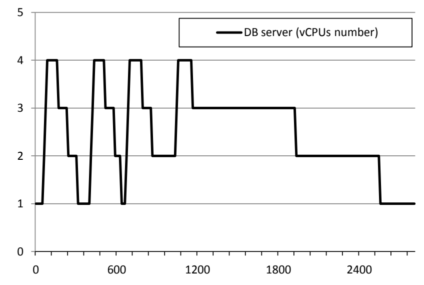
Figure 4(a) shows that web server was never the bottleneck of any of workload sessions, while database server CPU utilization touch a high value of CPU utilization many times, the controller reacts by increasing number of vCPUs rapidly to cope with traffic surge, and then slowly decrease them to a suitable value. In figure 4(d) we notice some violation to SLO accompany the workload increase, however, the violations did not exceed 5.2% of the experiment run time.



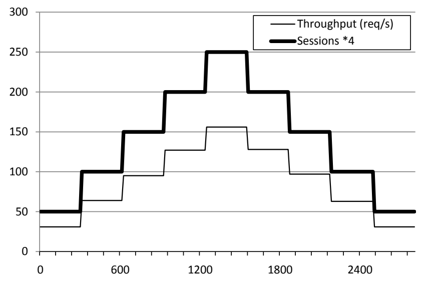
(a) CPU Utilization (%)



(b) Response time (ms)



(c) Number of vCPUs

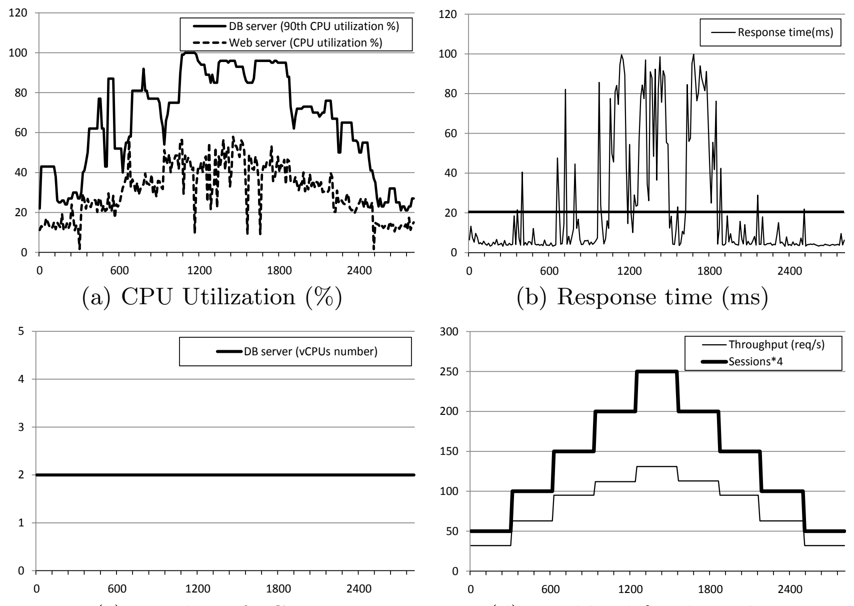


(d) Workload & Throughput

Fig. 4: Elastic VM implemented into database tier

To express the benefit of Elastic VM scalability in database-tier, we compare above result with non-scalable database tier. We initiated the same workload into the same setup but replaced Elastic VM with a static VM (i.e., two virtual CPUs). The number of the vCPUs is calculated by getting the average of the number of running vCPUs in the previous experiment after excluding intervals of high number of vCPUs which are caused by rapid scaling-up behavior of the controller. The average is 1.9 which can be rounded to 2 vCPUs.

Figure 5(b) shows many violations to SLO caused by the high utilization of database CPU for the traffic sessions higher than 800 as shown in figure 5(a). SLO violation time is 28% of the experiment run time, which is five times the SLO violation in case of Elastic VM as a database. The throughput also degrades for the traffic sessions 800 and 1000 by values 12% and 16% compared with the throughput of Elastic VM for the same traffic session number. In spite of the fact that the improvement in Elastic VM was on account of the price (number of vCPUs), but we should remind that we are manipulating the worst case (high traffic surge). Usually, real traffic contains periods of time (e.g., Midnight time) that implies low-workload. For these periods of time, Elastic VM reduces the resources consumption to the minimum.



(a) CPU Utilization (%) (b) Response time (ms)

(c) Number of vCPUs (d) Workload & Throughput

Fig. 5: Static VM implemented into database tier

4 Related Work

On the topic of detecting bottleneck tier, and provisioning adequate resources dynamically to mitigate SLO violation, the work of [11] implemented a prototype using Multi-instances scaling architecture, and developed a heuristic and active profiling of the CPU of virtual machines. The approach considers scaling database layer horizontally but do not discuss associated challenges that could affect the approach feasibility and performance specifically in database tier (e.g., data replication and synchronization). Moreover, the work presents scale-up algorithm with no scale-down.

To predict next workload and provision enough resources to cope with workload surge, the work in [13] implements two feedback controllers integrated with Kalman filter. The first controller, the Basic Controller (BC), predicts separately the necessary CPU allocation for each tier. The second controller, the Process Noise Covariance Controller (PNCC), extends BC by considering the coupling between multi-tier components. The experiment results express how PNCC cope better to the workload changes and therefore maintain better response time. However, the system is sensitive to the workload distribution, for example, it can not track the variables with noise that is not essentially normally distributed.

Many researchers presented analytical models to describe different tiers behavior, for example, [7] presents multi-tier model based on a network of queues, while each queue represents a different tier. The model is able to predict the mean response time for a specific workload. Scalability of this model is realized by dispatching new instances at each tier except the database tier which is not replicable in this model. The dispatching is initiated by a dispatcher at each tier. The dispatcher does not only provision or release VM instances but also balance the load.

By regression analysis of CPU utilization and service time to predict the bottlenecks, authors of [10] demonstrated an approach for performance modeling of two-tier applications (web and database). Despite the lack of dynamic scaling, this approach helps understanding application behavior for optimum capacity planning.

Using queuing theory models along with optimization techniques, [12] presented off-line techniques to predict system behavior and automatically generate optimal system configurations. The result is a rule set that can be inspected by human system administrators and used directly with a rule-based system management engines. In previous work [9], we implemented an online heuristic controller to tune apache controller for variant workload and dynamic resources provisioning, where the experiment results showed improvement in the system performance, reduction of SLO (e.g., response time) violation, and maintenance of a high throughput.

Amazon EC2 Spot Instances [5] is a way to provide VMs with a lower price. It is developed to serve customers who are in need for high computational power but for none online systems (e.g., Image and video processing, conversion and rendering; Scientific research data processing; and Financial modeling and analysis). Amazon EC2 Spot Instances was one of the motivating ideas to our research. Nowadays, Amazon static *Large EC2* instance costs $0.34 per hour, while static *Extra Large EC2* instance costs $0.68 per hour. Implementing Elastic VM can emerge the following service: *Elastic Large to Extra Large EC2* instance which costs for example $0.40 per hour. The lower case is to have a Large EC2 instance, and the upper case is to expand it to Extra Large EC2. As in Amazon EC2 Spot Instances, the idea behind the cost reduction is the dependency on the free capacity in cloud provider. Such approach depends on the probability of having free resources in the same zone, which is not guaranteed. However, having a global workload management plan that runs a complement workload on the same zone, considering the daylight differences around the world, increases the probability of having free resources in the same host.

5 Conclusion & Future Work

In this paper, we proposed an Elastic VM scalability architecture and compared it with existing Multi-instances scalability architecture. The presented work suggests modifications to current Multi-instances architecture that increases its stability and improve performance. Experiment results proved that Elastic VM reduces the provisioning overhead, mitigates SLOs violations, and maintains a higher throughput. Moreover, it enables scaling applications, such as databases, with lower cost and complexity.

Our immediate future work includes developing a global management policy for VMs management. It does not only consider scaling VMs in place but also relocation of VMs to physical hosts with less utilization. We also study integrating both Multi-instances and Elastic VM scaling architecture to enable rapid and fine-grained scaling architecture which is unlimited to one physical host. Moreover, we study the influence of the Elastic VM architecture on the current pricing models in cloud environments.

References

1. GoGrid, `http://www.gogrid.com/`
2. Slashdot, `http://slashdot.org/`
3. VMWare, `http://www.vmware.com/`
4. Xen hypervisor, `http://www.xen.org/`
5. Amazon: Amazon Elastic Compute Cloud, `http://aws.amazon.com/ec2/`
6. Amza, C., Cecchet, E., Ch, A., Cox, A.L., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., Zwaenepoel, W.: Bottleneck Characterization of Dynamic Web Site Benchmarks (2002)
7. Bhuvan Urgaonkar, G.P.: An analytical model for multi-tier internet services and its applications. In: In Proc. of the ACM SIGMETRICS2005. pp. 291–302 (2005)
8. Chess, Y.D., Hellerstein, J.L., Parekh, S., Bigus, J.P.: Managing Web server performance with AutoTune agents. IBM Systems Journal 42(1), 136–149 (Jan 2003)
9. Dawoud, W., Takouna, I., Meinel, C.: Elastic VM for Cloud Resources Provisioning Optimization, Communications in Computer and Information Science, vol. 190. Springer Berlin Heidelberg (2011), 10.1007/978-3-642-22709-743
10. Dubey, A., Mehrotra, R., Abdelwahed, S., Tantawi, A.: Performance modeling of distributed multi-tier enterprise systems. ACM SIGMETRICS Performance Evaluation Review 37(2), 9 (Oct 2009)
11. Iqbal, W., Dailey, M.N., Carrera, D.: SLA-Driven Dynamic Resource Management for Multi-tier Web Applications in a Cloud. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. pp. 832–837. CCGRID '10, IEEE, Washington (2010)
12. Jung, G., Joshi, K.R., Hiltunen, M.A., Schlichting, R.D., Pu, C.: Generating Adaptation Policies for Multi-tier Applications in Consolidated Server Environments. IEEE (Jun 2008)
13. Kalyvianaki, E., Charalambous, T., Hand, S.: Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In: Proceedings of the 6th international conference on Autonomic computing - ICAC '09. p. 117. ACM Press, New York, New York, USA (Jun 2009)
14. KVM: Kernel Based Virtual Machine
15. Liu, X., Sha, L., Diao, Y., Froehlich, S., Hellerstein, J.L., Parekh, S.: Online Response Time Optimization of Apache Web Server (2003)
16. Munir B., S., Abhik, C., S. L., N., K. T., S.: Novel Approach to Improve QoS of a Multiple Server Queue. Int'l J. of Communications, Network and System Sciences 3(1), 83–86 (2010)
17. TPC-W: Transactional web e-Commerce benchmark, `http://www.tpc.org/tpcw/`
18. Tran, D.N., Huynh, P.C., Tay, Y.C., Tung, A.K.H.: A new approach to dynamic self-tuning of database buffers. ACM Transactions on Storage 4(1), 1–25 (May 2008)

19. Wiese, D., Rabinovitch, G., Reichert, M., Arenswald, S.: Autonomic tuning expert. CASCON '08, ACM Press, New York, New York, USA (2008)