Low Cost Quality Aware Multi-tier Application Hosting on the Amazon Cloud

Waheed Iqbal*, Matthew N. Dailey†, David Carrera‡
*Punjab University College of Information Technology, University of the Punjab, Lahore, Pakistan
waheed.iqbal@pucit.edu.pk
†Computer Science and Information Management, Asian Institute of Technology, Thailand
mdailey@ait.asia
‡Technical University of Catalonia Barcelona Supercomputing Center, Spain
dcarrera@ac.upc.edu

***Abstract*—Public cloud providers offer a variety of services and resources enabling users to host their applications. In order to determine the set of resources that will best minimize hosting costs while simultaneously meeting performance requirements, cloud users must be aware of the cost-performance tradeoffs of the available resources. This is particularly true of multi-tier Web applications using dynamic scaling strategies to sustain specific response time requirements. In this paper, we provide a cost-performance analysis of multiple scale out strategies using the three most economically feasible Amazon EC2 compute resources (micro, small, and medium virtual machine instances) for a typical multi-tier Web application. We find that the special CPU allocation policy of the micro instance makes it especially suitable for satisfying Web application performance requirements at minimal cost on the Amazon cloud.**

***Keywords*-Auto scaling; Dynamic resource provisioning; cost-effective provisioning; multi-tier applications; Amazon EC2.**

I. INTRODUCTION

Cloud computing has attracted a large user base mainly due to its pay-by-usage and on-demand resource provisioning features. Public cloud providers such as Amazon, Rackspace, IBM, Microsoft, and Google offer a variety of different options for the end user to acquire and host applications.

Multi-tier Web applications are typical cloud-hosted consumer-facing Internet applications consisting of at least a presentation tier, a business tier, and a data management tier. Multi-tier Web applications hosted on a specific fixed infrastructure can only service a limited number of requests concurrently before some bottleneck occurs. Once a bottleneck occurs, if the arrival rate does not decrease, the application will saturate, response time will grow dramatically, and eventually, requests will fail entirely. Such bottlenecks can be resolved using dynamic provisioning of the tiers to accommodate more users and requests [1], [2], [3], but it is very challenging to minimize cost while doing so [4].

When a user is selecting public cloud hosting options for a multi-tier Web application, he or she is faced with the daunting task of select the lowest cost option that provides acceptable performance. For example, on Amazon's Elastic Compute Cloud (EC2), the user must select virtual machine models having the right amount of CPU, memory, and storage resources.

To help guide such selections, in this paper, we perform a cost and performance analysis based on the three most economically-feasible EC2 instance types, namely micro, small, and medium, and we compare multiple rule-based scale-out strategies for a typical multi-tier Web application. Our rule-based strategies, namely CPU Reactive (scale out when CPU utilization thresholds are exceeded) and Response Reactive (scale out when response time thresholds are exceeded) are able to handle dynamically increasing workloads.

Our experimental evaluation shows that micro instances achieve very good performance at minimal cost. This surprising result is mainly due to Amazon's special CPU allocation policy for micro instances. The policy provides higher than usual CPU capacity to micro instances during short activity spikes. By keeping micro instances running at their limit, we presumably "steal" unused cycles from other virtual machines running on the same physical machine. As long as we are not unlucky in the assignment of our virtual machine to a physical machine, we can obtain excellent performance from micro instances at very low costs.

The main contributions of the paper are as follows.

1) We introduce a simple but effective strategy for benchmarking instance types for appropriate maximum CPU utilization thresholds.
2) We introduce and explore the performance of two different reactive scale out strategies for multi-tier applications hosted on public clouds.
3) We perform a cost/performance analysis of dynamic scaling strategies for multi-tier Web applications running on different types of Amazon EC2 instances.

In the rest of the paper, we present related work, an overview of Amazon's cloud services, our scale-out strategies, the experimental design, and results.

II. RELATED WORK

There have been several efforts towards adaptive allocation of cloud resources to satisfy performance metrics. For example, Bodik et al. [5] present a statistical machine learning approach to predict system performance for a single tier application and minimize the amount of resources required to maintain the performance of an application hosted on a cloud. Liu et al. [6] monitor the CPU and bandwidth usage of virtual machines hosted on an Amazon EC2 cloud, identify the resource requirements of applications, and dynamically switch between different virtual machine configurations to accommodate changing workloads. However, none of these solutions address the issues of multi-tier Web applications.

Thus far, only a few researchers have addressed the problem of resource provisioning for multi-tier applications. Urgaonkar et al. [7] present an analytical model using queuing networks to capture the behavior of each tier. Jia et al. [8] present an on-line method for capacity identification of multi-tier Web applications hosted on physical machines using hardware performance counters. However, this work does not include cost/performance analyses of public cloud infrastructure for multi-tier applications.

There have been several efforts to identify the performance of Amazon cloud services for high performance applications. For example, Jackson et al. [9] show that EC2 is six times slower than modern HPC systems. Ostermann et al. [10] evaluate the usefulness of Amazon EC2 instances for scientific computing and conclude that cloud resources should be improved to be used for scientific applications.

However, there has been less work on the performance of Amazon cloud services for multi-tier Web applications. One example is Jiang et al. [2], who present a dynamic resource provisioning approach for multi-tier Web applications hosted on clouds that attempts to ensure homogeneous performance from every Amazon EC2 small instance deployed to a tier.

Our work focuses on cost and performance analysis of the three most economically feasible EC2 instances (micro, small, and medium) for multi-tier Web applications using two different rule-based adaptive resource provisioning strategies. To our knowledge, this paper is the first study of different Amazon EC2 instances' ability to host scalable multi-tier Web applications. We examine the cost and performance of different EC2 instance types under multiple scale-out strategies in the context of a benchmark multi-tier Web application and dynamically varying loads.

III. Amazon Cloud Services

Amazon provides various types of virtualized infrastructure and application services to host applications, store data, enable high performance computing, and automate workflows over the cloud. In this section, we discuss the Amazon cloud services used in our experimental evaluation.

A. Elastic Compute Cloud (EC2)

Amazon Elastic Cloud (EC2) is a Web service providing a variety of compute resources (virtual machines) over the cloud. Different units of compute resources, such as "micro," "small," "medium," "large," and "extra large" are available for provisioning and use. The names of the instance types hint at the relative amounts of resources (CPU, memory, and storage) provided with each instance. EC2 uses standard units to describe memory and storage, but it introduces an abstract CPU allocation unit known as the "EC2 Compute Unit" (ECU) to describe the fixed amount of CPU resources allocated to a virtual machine. One ECU is equivalent to the CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. CPU and memory is dedicated to a given virtual machine, but network and disk subsystems are shared among the instances running on the same physical machine.

A variety of different options are thus available for the end user to acquire and host an application on Amazon's cloud. However, the most cost-effective choice for a particular user will depend on certain characteristics of the application being deployed and the workload it will serve. It is very difficult for the typical end-user to select the best set of resources for his or her specific application. Therefore, in this paper, we perform a cost and performance analysis based on the three most economically-feasible instance types, namely micro, small, and medium, for a typical multi-tier Web application.

Table I shows the targeted on-demand EC2 instance types, hardware resources (CPU and memory), and cost per hour for the `us-west-2` region. We briefly discuss each of the target instance types in the following sections.

1) Micro Instances: The EC2 micro instance is the least expensive option. Rather than providing a fixed level of CPU resources, micro instances have two different CPU allocation levels: *background level* and *max level*. Background level allocation provides a consistent baseline CPU resource allocation. Max level allocation is allowed for short periods of time to accommodate short spikes in CPU requirements [11]. The exact allocation of ECUs for the background and max levels is not published, but the EC2 documentation does specify that the maximum CPU allocation for a micro instance can be as large as two ECUs. This means that the micro instance may be feasible for low-throughput applications with occasional workload spikes.

2) Small Instances: The small instance is another inexpensive option for compute resources. Small instances have a fixed CPU allocation. Small instances may be a good choice for a low-throughput applications with known maximum CPU and memory requirements.

3) Medium Instances: EC2 provides two different types of medium instances (m1.medium and c1.medium). They are useful for different types of applications: m1.medium is best for applications requiring a large amount of memory and smaller amounts of CPU, whereas c1.medium is best for applications requiring small amounts of memory but large amounts of CPU. In this paper, we only use c1.medium instances; from here onward, we refer to our c1.medium instances simply as "medium" instances.

4) Elastic Block Storage (EBS) and Instance Storage: Amazon provides two types of storage for EC2 instances' root devices: Elastic Block Storage (EBS) and Instance Storage (IS). EBS is persistent irrespective of the instance's lifetime, whereas IS storage does not persist beyond the instance's lifetime. EBS-backed EC2 instances are thus more reliable in terms of persisting local instance data, but they incur extra costs. We use EBS-backed instances in all of our experiments, as EBS-backed instances are much faster than IS backed

TABLE I
TARGETED EC2 ON-DEMAND INSTANCE TYPE PRICES AND RESOURCE ALLOCATIONS FOR THE US-WEST-2 REGION.

Instance Type	**CPU**	**Memory**	**Price/Hour (USD)**
Micro	Up to 2 ECU	613 MB	$0.020
Small	1 ECU	1.7 GB	$0.065
Medium (c1.medium)	5 ECU	1.7 GB	$0.130
Medium (m1.medium)	2 ECU	3.75 GB	$0.130

instances to launch, requiring less than a minute.

5) CloudWatch: The Amazon CloudWatch service allows users to monitor various performance metrics for their AWS resources. As examples, a CloudWatch user can monitor the average, maximum, and minimum CPU utilization, the I/O utilization, or the bandwidth consumption of launched instances for specific durations. By default, CloudWatch monitors resources over five-minute intervals. However, it is also possible to perform more fine-grained monitoring over one-minute intervals by enabling "detailed monitoring," incurring extra costs.

Besides the built-in metrics, developers can also add application-specific metrics to the CloudWatch service through an API. We use the CloudWatch API to monitor CPU utilization of our instances, and we enable detailed monitoring. This allows us to quickly identify CPU saturation.

IV. SCALE-OUT STRATEGIES

A multi-tier Web application hosted on a cloud can satisfy specific response time requirements by performing horizontal scaling (scale-out) using a variety of policies, including rule-based methods and schedule-based methods. A rule base defines a set of rules for triggering scale out operations; for example, if a tier's virtual machine CPU utilization reaches 85% or its memory utilization reaches 90%, we may want to add an additional virtual machine to the tier. Schedule-based approaches, on the other hand, adjust the number of virtual machines allocated to an application based on a specific schedule, e.g., based on particular hours of the day or days of the week. More advanced techniques are also possible; for example, in ongoing work [4], we use a combination of heuristics and machine learning to scale out multi-tier applications. In this paper, with the aim of profiling the cost-effectiveness of the different Amazon EC2 instances for multi-tier application provisioning, we experiment with two fairly simple rule-based scale-out strategies. This section describes the two strategies in further detail.

A. CPU Reactive

Most Web applications that generate dynamic content require a significant amount of CPU capacity, and such applications often saturate their CPUs when attempting to serve an unexpectedly heavy workload. In preliminary experiments with our sample benchmark Web application on EC2, we established that CPU always saturates before the Web application's response times go beyond a specific threshold. This means that CPU is the main resource limitation under heavy workloads, so in our CPU Reactive strategy, we configure the system to trigger scale-out operations whenever average CPU utilization crosses a specific threshold. For our two-tier (Web and database) multi-tier Web application, the policy is as follows:

- Trigger a scale-out operation whenever the average CPU utilization of all virtual machines allocated to a specific tier goes beyond a specific threshold (α_{cpu}).
- Scale-out operations are performed by adding one virtual machine to the tier(s) triggering the operation.

This CPU Reactive approach may scale out none, one, or both tiers depending on the CPU utilization of the tier-specific virtual machines over a given monitoring interval.

B. Response Reactive

The Response Reactive strategy relies on response time monitoring rather than CPU utilization monitoring to trigger scale-out operations. This is sensible because ultimately, a maximum response time requirement would be an important service level objective (SLO) in the service level agreement (SLA) for any operations team setting out to manage an application's performance.

In the Response Reactive approach, we trigger scale-out operations whenever the application's response time exceeds a specific threshold. For our two-tier Web application, our Response Reactive technique is as follows:

- Trigger a scale-out operation whenever the 95^{th} percentile of the response time goes beyond a specific threshold (τ_{rt}).
- Scale-out operations are performed by adding one virtual machine to each tier(s) for which the average CPU utilization is higher than a specific threshold (α_{cpu}).

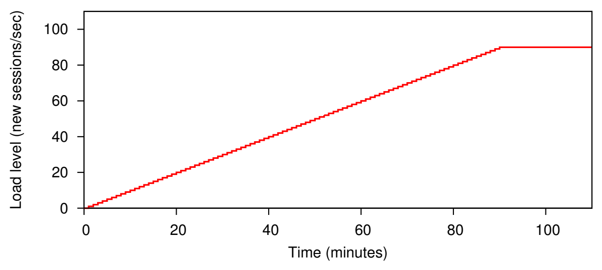
Like the CPU Reactive strategy, the Response Reactive strategy may similarly scale out none, one, or both tiers. Although the triggering event is different, the scale out operation itself is identical in the two cases.

V. EXPERIMENTAL DESIGN

In this section, we provide details of our application setup, workload generation methods, testbed cloud infrastructure, experimental details, and evaluation criteria.

A. Benchmark Web Application

RUBiS [12] is an open-source benchmark Web application for auctions. It provides typical core functionality of an auction site, such as browsing, selling, and bidding for items, and it provides three user roles: visitor, buyer, and seller. Visitors are not required to register; they are allowed to browse for items that are available for auction. We use the PHP implementation of RUBiS as a sample Web application for our experimental evaluation.



[t]

Fig. 1. Workload generation profile for all experiments.

Fig. 2. Testbed cloud infrastructure used for the experiments.

TABLE II
EXPERIMENTAL DETAILS.

Experiment #	Allocation Type	Scale-out Strategy	Parameter Value
Experiment 1	Micro	CPU Reactive	$\alpha_{cpu} = 100\%$
		Response Reactive	$\tau_{rt} = 500$ms $\alpha_{cpu} = 100\%$
Experiment 2	Small	CPU Reactive	$\alpha_{cpu} = 65\%$
		Response Reactive	$\tau_{rt} = 500$ms $\alpha_{cpu} = 65\%$
Experiment 3	Medium	CPU Reactive	$\alpha_{cpu} = 85\%$
		Response Reactive	$\tau_{rt} = 500$ms $\alpha_{cpu} = 85\%$

B. *Workload Generation*

We use `httperf` to generate synthetic workloads for RUBiS. We generate workloads for specific durations with a required number of user sessions per second. A user session emulates a visitor that browses for categories and regions and also browse bids on items up for auction. After every minute, we increment the load level by 1, from load level 1 through load level 90. After that, we maintain load level 90 for the last 20 minutes of the experiment (the entire experiment is thus 110 minutes). Each load level represents the number of user sessions per second; each user session makes 32 browser requests to simulate a user session. Figure 1 shows the workload distribution over time for all experiments.

C. *Testbed Infrastructure*

Amazon owns multiple geographical dispersed data centers around the world known as *regions*. Each region is divided into multiple locations known as *availability zones*. Users are able to place their EC2 instances in any region and any availability zone. We performed all experiments in the `us-west-2` region and the `us-west-2c` availability zone of the Amazon public cloud. Figure 2 shows the testbed cloud infrastructure used during the experiments. We use an EC2 medium instance (c1.medium) as the head node. The head node is configured to fulfill the following responsibilities:

- Generate the workload (user sessions).
- Act as a proxy server for the benchmark Web application.
- Load balance incoming requests among the servers in the provisioned Web server tier.
- Dynamically provision resources to the application.

The pool of dynamically-provisioned EC2 instances always contains at least one virtual machine in the Web tier and one virtual machine in the database tier of the benchmark application. We set the maximum number of dynamically provisioned instances to 18 for all experiments. We never observed any of the head node's resources (CPU, memory, I/O, or network bandwidth) saturate during the experiments.

We use Nginx as the load balancer for the Web tier because it offers detailed logging and allows reloading of its configuration file without terminating existing client sessions. Since RUBiS does not currently support load balancing over a database tier, we modified it to use round-robin balancing over a set of database servers listed in a database connection settings file, and we developed a server-side component to update the database connection settings file after a scaling operation has modified the configuration of the database tier. Our focus is on scaling multi-tier applications, and we assume that a mechanism such as [13] exists to ensure consistent reads after updates to a master database.

D. *Experimental Details*

We performed three sets of experiments to analyze the cost and performance of micro, small, and medium instances under the CPU Reactive and Response Reactive scale-out strategies for the same workload distribution. Table II shows details of the experiments. We repeated each experiment using both strategies. For each experiment, we initialize the system with one virtual machine in the Web tier and one virtual machine in the database tier of the benchmark application using the specific instances for the experiment. During the experiments, we monitor the 95^{th} percentile of response time and the CPU utilization of each provisioned virtual machine.

For the Response Reactive scale-out strategy, we configured the average response time threshold value (τ_{rt}) to 500 ms, an acceptable maximum response time requirement for Web applications.

To determine an appropriate value for α_{cpu}, we performed a pilot experiment to identify the maximum CPU utilization for each allocation type that does not affect the performance of the benchmark Web application. We allocated one virtual machine to the Web tier and one virtual machine to the database tier, then we executed the increasing workload shown in Figure 1 to find the latest point at which the 95^{th} percentile of the response time did not increase substantially.

Figure 3 shows the throughput (requests served per second), CPU utilization (%) in the Web and database tiers, and the 95^{th} percentile of the response time using micro instances. We observed dramatic growth in response time after the 7^{th} minute. We thus learned that micro instances continue to perform well with 100% average CPU utilization. Therefore, we set the average CPU utilization α_{cpu} to 100% for the experiments using micro instances.

Figure 4 shows the same data using small instances. We observed dramatic growth in response time after the 6^{th} minute.

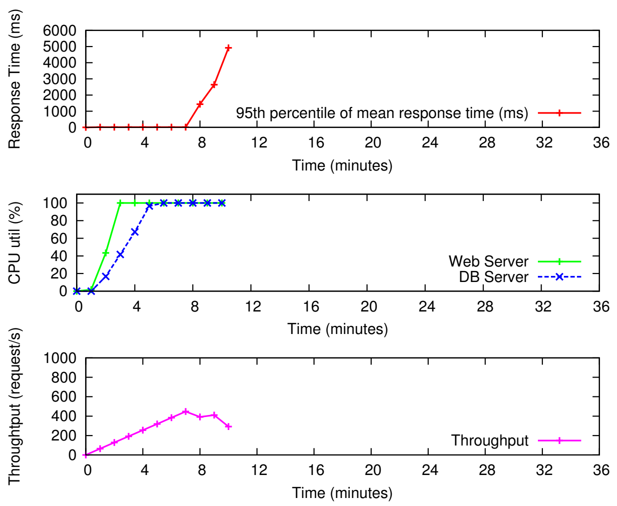


Fig. 3. Static micro allocation: throughput (requests served per second), CPU utilization (%) of the Web and the database tiers, and 95^{th} percentile of the response time.

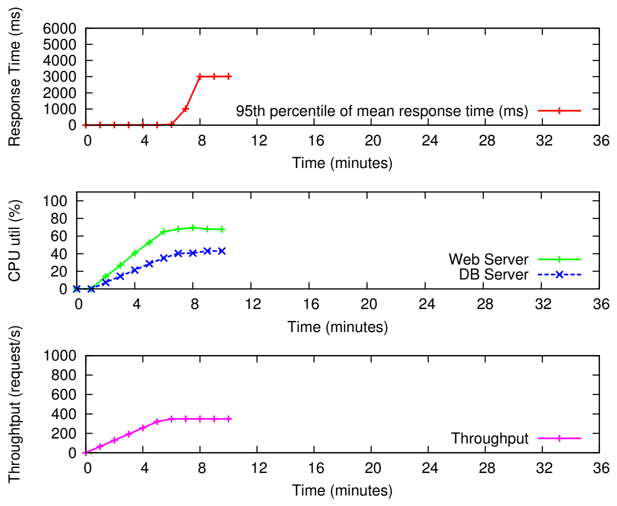


Fig. 4. Static small allocation: throughput (requests served per second), CPU utilization (%) of the Web and the database tiers, and 95^{th} percentile of the response time.

We thus learned that small instances perform well up to 65% CPU utilization before application performance decreases. Therefore, we set the average CPU utilization threshold α_{cpu} to 65% for the experiments using small instances.

Finally, Figure 5 shows the same data using medium (c1.medium) instances. We observed dramatic growth in response time after the 28^{th} minute. We thus learned that medium instances perform well up to 85% CPU utilization. Therefore, we set the average CPU utilization threshold α_{cpu} to 85% for the CPU Reactive phase of the medium instances allocation.

E. Evaluation Criteria

We evaluate the experiments by measuring the application's *performance* and the *cost* of the provisioned resources during

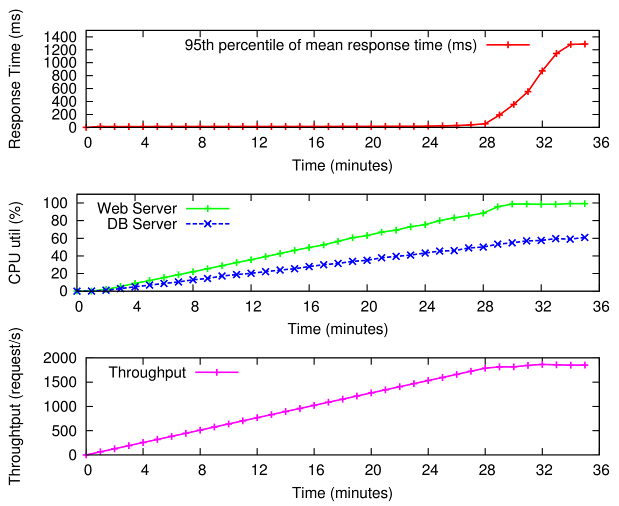


Fig. 5. Static medium allocation: throughput (requests served per second), CPU utilization (%) of the Web and the database tiers, and 95^{th} percentile of the response time.

each experiment. We measure application performance by calculating the percentage of requests missing the response time requirements (%SLA missing) during the experiment. Amazon cloud services have various costs, e.g. for data transfer, EBS storage, EBS I/O requests, CloudWatch API requests, detailed instance monitoring, and EC2 instances. Since it is difficult to identify the exact cost incurred during each experiment, to simplify the cost calculations, we only calculate cost based on the number of CPU hours allocated/used during each experiment.

VI. Experimental Results

In this section, we describe the results obtained in Experiments 1, 2, and 3 using the CPU Reactive and Response Reactive scale-out strategies, and we also discuss the performance and cost measurements obtained in each experiment.

For each experiment reported in this experiment, we provide graphs showing throughput (requests served per second), the 95^{th} percentile of the response time, the dynamic addition of instances in each tier, and the average CPU utilization of the Web and database tier instances for the specific allocation type and scale-out strategy.

A. Experiment 1: Micro Allocation

Figure 6 shows the results of experiment 1 (micro instance allocation) with the CPU Reactive scale-out strategy. Whenever the system detects a violation of the CPU utilization threshold, it uses the CPU Reactive scale-out strategy to identify the tier(s) to scale out, then it dynamically adds micro instances (virtual machines) to the identified tier(s). The system quickly reaches the maximum allocation limit (18 instances) during the experiment. By the 43^{rd} minute, all 18 instances consistently utilize 100% of their CPUs, but we never observe any significant growth in the 95^{th} percentile of the response time. The system throughput grows linearly in response to the growing workload throughout the full range

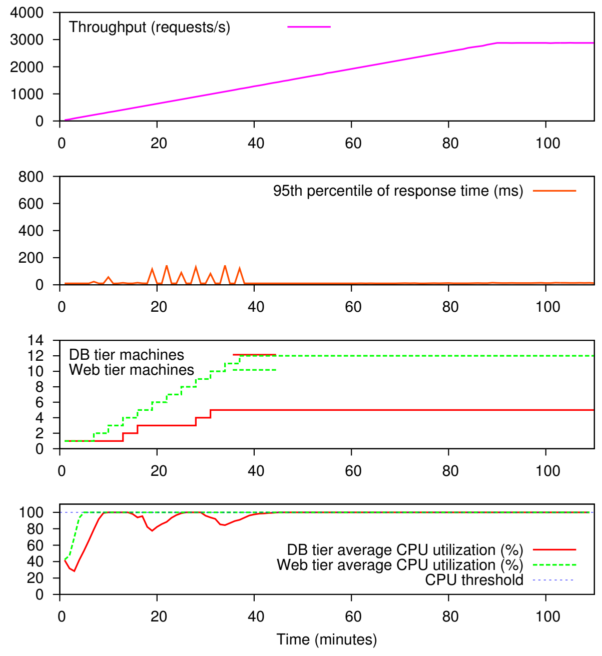


Fig. 6. Experiment 1 (micro allocation) results with CPU Reactive scale-out strategy. The graphs show throughput (requests served/second), 95^{th} percentile of response time, dynamic addition of instances in each tier, and CPU utilization of Web and database tier instances. Micro instances perform well with the CPU Reactive strategy even at 100% CPU utilization.

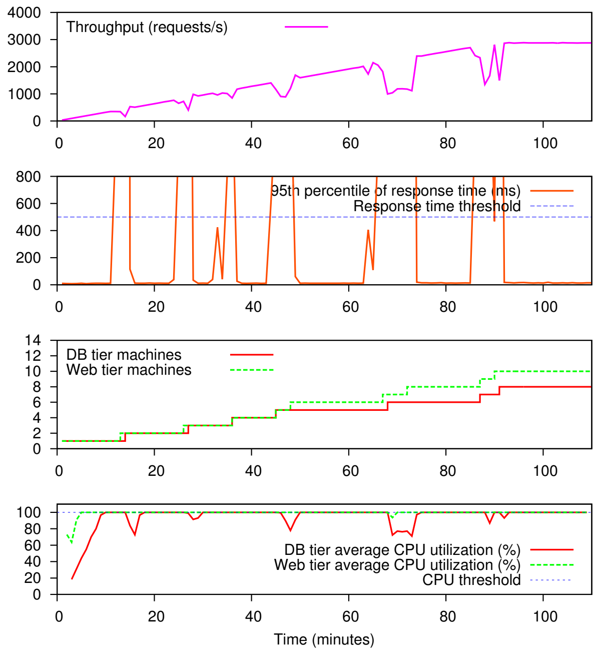


Fig. 7. Experiment 1 (micro allocation) results with Response Reactive scale-out strategy. The system is capable of reacting on performance violations and restoring application performance.

of load levels. This experiment shows that micro instances provide consistent performance even when the instances' CPU utilization reaches 100%.

Figure 7 shows the same measurements for micro instance allocation and the Response Reactive scale-out strategy. On response time requirement violations, the system uses the proposed Response Reactive scale-out strategy to identify the tier(s) to scale out, then it dynamically adds micro instances (virtual machines) to the identified tier(s). We observe that system is capable of reacting on performance violations and restoring application performance multiple times dynamically. We also observe that system throughput degrades whenever the response time saturates.

B. Experiment 2: Small Allocation

Figure 8 shows the results of experiment 2 (small instance allocation) with the CPU Reactive scale-out strategy. We observe multiple periods in which the 95^{th} percentile of the response time increases, but the CPU Reactive scale-out strategy works well, scaling out the appropriate tiers to restore system performance.

Figure 9 shows the results of experiment 2 (small instance allocation) with the Response Reactive scale-out strategy. The system is capable of reacting on performance violations and restoring application performance multiple times dynamically.

During this particular experiment, we observed an anomaly: from the 61^{st} minute of the experiment, a few of the Web tier virtual machines behaved inconsistently, with higher than normal response times compared to the other allocated virtual

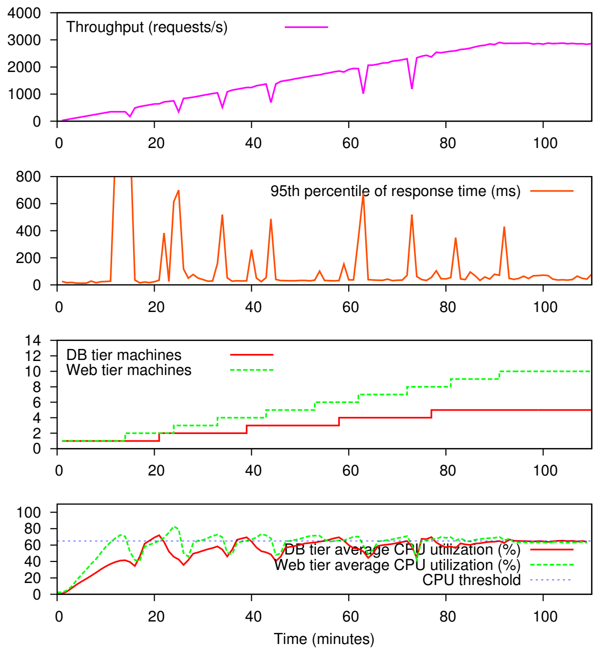


Fig. 8. Experiment 2 (small allocation) results with CPU Reactive scale-out strategy.

machines in the same tier. The average CPU utilization for the tiers did not cross the scaling threshold, however, so the system did not scale out the tier. However, by the 69^{th} minute, all of the virtual machines allocated to the Web tier returned to normal and performed well for the rest of the experiment, and the 95^{th} percentile of the response time

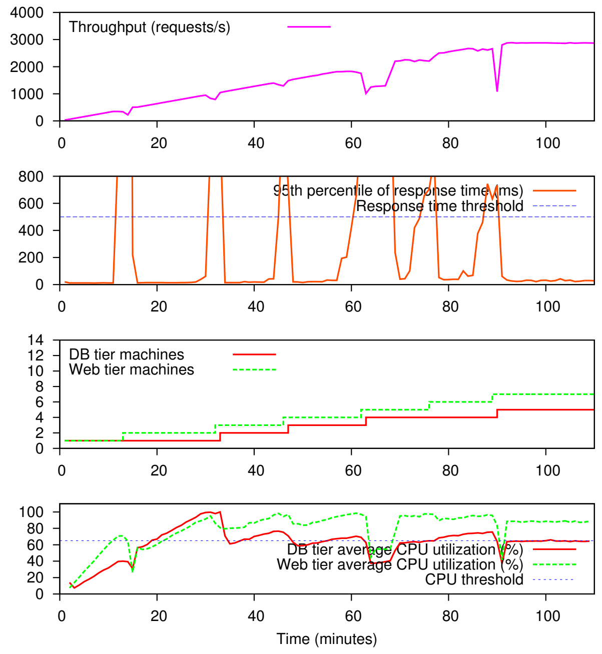


Fig. 9. Experiment 2 (small allocation) results with Response Reactive scale-out strategy.

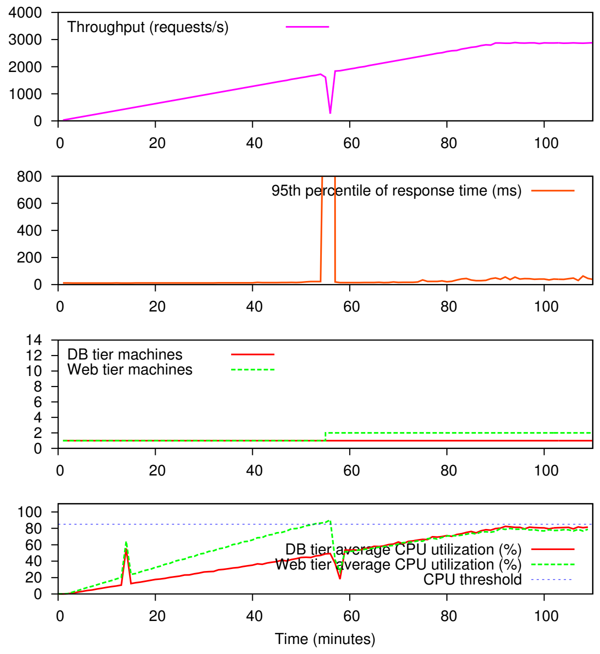


Fig. 10. Experiment 3 (medium allocation) results with CPU Reactive scale-out strategy.

decreased without adding more resources to the tiers. The anomaly was presumably due to contention for resources with other virtual machines on the same physical machine.

C. Experiment 3: Medium Allocation

Figure 10 shows the results of experiment 2 (medium instance allocation) with the CPU Reactive scale-out strategy. We observe only one case of reaching the CPU utilization threshold, in the Web tier. The system dynamically adds a second virtual machine to the tier, and the application performance is restored quickly.

Figure 11 shows the results of experiment 3 (medium instance allocation) with the Response Reactive scale-out strategy. We observe only one case of reaching the response time threshold. The system is able to scale out the Web tier of the application dynamically.

D. Summary

Table III summarizes the experimental results. The table shows the percentage of requests missing the targeted response time (%SLA Missing), the amount of CPU time used, in hours, and the resulting cost of the specific set of instances for each scale-out strategy in each experiment. Refer to Section V-E (Evaluation Criteria) for an explanation of the cost calculation. We see immediately that for our benchmark multi-tier Web application hosted on the Amazon cloud, the CPU Reactive technique is clearly better in terms of performance, because fewer requests miss the SLA response time target. However, the Resource Reactive scale-out strategy is consistently better in terms of cost, because it waits for the workload to begin to affect performance before it scales out any tier.

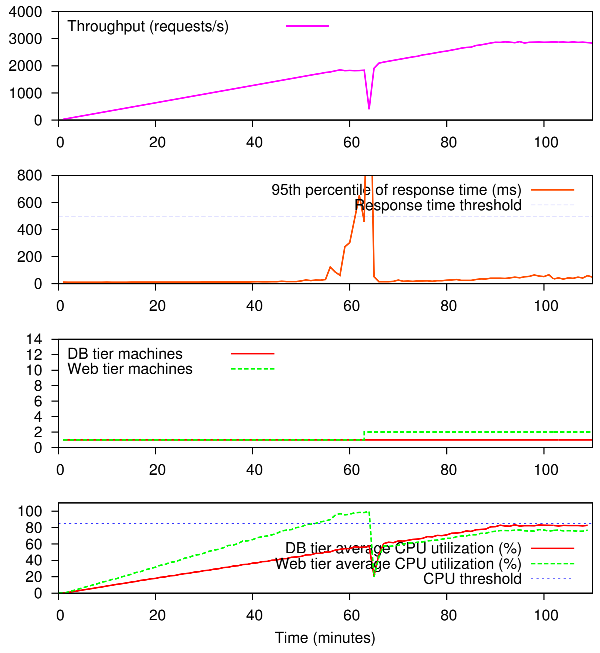


Fig. 11. Experiment 3 (medium allocation) results with Response Reactive scale-out strategy.

Micro instances are very low cost. They also perform well, particularly with the CPU Reactive strategy (no SLA violations at all). Under the Response Reactive strategy, micro instances do incur a moderate 1.1% SLA miss rate. Small instances are the worst performers, with the highest costs. Medium CPU Reactive instances are nearly as good as micro instances, but medium instances offer little benefit under the Response

TABLE III
EXPERIMENTAL RESULTS SUMMARY.

Allocation Type	Method	%SLA Missing	CPU Hours	Cost (USD)
Micro	CPU Reactive	0.00	25.92	$0.52
	Response Reactive	1.10	19.07	$0.38
Small	CPU Reactive	0.22	16.58	$1.08
	Response Reactive	2.11	12.98	$0.84
Medium	CPU Reactive	0.01	4.60	$0.60
	Response Reactive	0.23	4.47	$0.58

Reactive strategy.

Micro instances are 51.85% and 13.33% more cost effective than small and medium instances, respectively, using the CPU Reactive scale-out strategy and 54.76% and 34.48% more cost effective than small and medium instances, respectively, using the Response Reactive scale-out strategy.

Micro instances thus provide excellent performance at low cost in the context of linearly increasing workloads. However, they would most likely fail to provide high performance under sudden large increases in workloads. Reactive scale out using medium instances would probably be more appropriate under such circumstances. Therefore, a hybrid approach using a mix of micro and medium instances to accommodate different workload profiles might be more appropriate.

Finally, we note that our ability to squeeze good performance from a pool of micro instances depends on how Amazon gives CPU resources to micro instances, allowing up to two ECUs of CPU during short activity spikes. We are presumably "stealing" unused cycles from other virtual machines running on the same physical machine. Our ability to do that would necessarily depend on the other machines' workloads and Amazon not changing its policy for CPU allocation to micro instances.

VII. CONCLUSIONS

In this paper, we have presented an investigation into cost-effective choices for hosting multi-tier Web applications that are required to satisfy specific response time requirements on the Amazon public cloud for linearly increasing workloads. We experimented with the three most economically-feasible instance types, namely micro, small, and medium, for a typical multi-tier Web application. We used two rule-based scale-out strategies for dynamic scale out of allocated resources to maintain the application's performance. Our experimental study shows that micro instances provide the best combination of price and performance under linearly increasing workloads.

We are currently developing machine learning approaches to identify the resources required to satisfy response time requirements for multi-tier applications hosted on the Amazon cloud. We are also exploring the possibility of using a hybrid approach based on both micro and medium instances to accommodate large unexpected increases in workload.

REFERENCES

[1] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, p. 871, 2011.

[2] J. Dejun, G. Pierre, and C.-H. Chi, "Resource provisioning of Web applications in heterogeneous clouds," in *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[3] N. Bonvin, T. Papaioannou, and K. Aberer, "Autonomic sla-driven provisioning for cloud applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE, 2011, pp. 434–443.

[4] W. Iqbal, M. N. Dailey, and D. Carrera, "Minimalistic adaptive resource management for multi-tier applications hosted on clouds," in *Proceedings of the 2012 IEEE 26th International Conference on Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pp. 2546–2549.

[5] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *HotCloud'09: Proceedings of the Workshop on Hotp Topics in Cloud Computnig*, 2009.

[6] H. Liu and S. Wee, "Web server farm in the cloud: Performance evaluation and dynamic architecture," in *CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 369–380.

[7] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, vol. 33. ACM, 2005, pp. 291–302.

[8] J. Rao and C.-Z. Xu, "Online capacity identification of multitier websites using hardware performance counters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, 2010.

[9] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, , H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '10, 2010, pp. 159–168.

[10] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing," in *Cloud Computing*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 34, ch. 9, pp. 115–131.

[11] Amazon Inc, "Amazon EC2 Micro Instances," http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html.

[12] OW2 Consortium, "RUBiS: An auction site prototype," 1999, http://rubis.ow2.org/.

[13] xkoto, "Gridscale," 2009, http://www.xkoto.com/products/.