Dynamic Resource Allocation Using Performance Forecasting

Paulo Moura, Fabio Kon
Department of Computer Science
University of São Paulo
Brazil
{pbmoura, kon}@ime.usp.br

Spyros Voulgaris
Department of Computer Sciences
VU University Amsterdam
The Netherlands
spyros.voulgaris@vu.nl

Maarten van Steen
Centre for Telematics and Information Technology
University of Twente
The Netherlands
m.r.vansteen@utwente.nl

***Abstract*—To benefit from the performance gains and cost savings enabled by elasticity in cloud IaaS environments, effective automated mechanisms for scaling are essential. This automation requires monitoring system status and defining criteria to trigger allocation and deallocation of resources. While these criteria are usually based upon resource utilization, this may be inadequate due to the impossibility of identifying the actual amount of required resources. Consequently, most systems update resource allocation in small increments, e.g., one VM at a time, which may negatively affect performance and cost. In this paper, we propose a novel approach in which the system monitors workload, instead of utilization, and, by means of a scalability model, it makes predictions of resource demand and updates the allocated resources accordingly. We provide an implementation of this approach and describe experimental results that show its effectiveness.**

I. Introduction

Cloud computing is a large and growing market: 83% of CIOs (Chief Information Officers) are currently considering the use of cloud IaaS as an option, and workloads in cloud IaaS providers are growing faster than on-premises workloads [1]. Cloud Computing provides agility in setting up and maintaining systems with the key benefit of its simplicity in allocating and deallocating resources on demand. To stay competitive, it is crucial to benefit from this technology and, to achieve that, systems with self-scaling capabilities are becoming more and more common.

The standard approach to self-scaling is to closely monitor resource utilization and, when pre-established thresholds are crossed, make the corresponding adjustments. One limitation of this approach is that it is not possible to directly assess the actual demand for resources. Updates are performed incrementally, e.g., by adding or removing virtual machines from the system's pool, until a stable configuration is found. One downside of incremental updates is that, when facing a drastic increase in the workload, the time to reach an adequate state can be long, affecting performance in the meantime. This is aggravated by the fact that adding a new VM to a system's pool involves, at the least, initializing the VM, deploying the service and related dependencies, and migrating data, which may take several minutes. Thus, performing a sequence of small changes can lead to higher aggregated cost when compared to a single, larger change that immediately meets the demand.

Something similar happens when the load decreases drastically. If the system deallocates one VM at a time in each step, idle resources may remain allocated for a long time.

In this paper, we propose a novel approach based on estimates of resource demand variation caused by workload fluctuations. We adopted the Universal Scalability Law, developed by Neil Gunther [2], as a starting point. The relation between performance and resource availability provided by Gunther's model is used to estimate the resources needed to deliver the throughput required to process the current workload.

By doing so, we address the problem of auto-scaling systems running in cloud IaaS environments, enabling cost minimization while delivering adequate performance to users.

The paper is organized as follows. After a discussion of related work in Section II, we offer an overview of the Universal Scalability Law in Section III. Our proposal is discussed in Section IV and an experimental validation is presented in Section V. Finally, we summarize our results and contemplate future work in Section VI.

II. Related work

Much effort has been put on research to improve automated resource management in clouds. The general approach is to define rules based on utilization that trigger actions to request or release resources. Mamani et al. [3] worked with the cumulative difference between current utilization and pre-established limits, aiming at improved reaction to workload variation. Lim et al. [4] worked on proportional thresholds that are adapted based on cluster size to improve accuracy. But making elasticity decisions based on utilization may be imprecise because it may lead to states which are not ideal and require gradual updates.

Some alternative approaches are based on queue size. Salah et al. [5] proposed a model as such. But they consider that requests are enqueued in the load balancer, which is not usually the case and limits system capacity, as distributing the waiting queue among the servers increases the overall queue capacity. The model proposed by Aljohani et al. [6] considers a distributed queue. However, it is based on simulations in which the number of available servers is set a priori, which is contrary to the unlimited resources assumption typically advocated by cloud providers.

978-1-5090-2088-1/16/$31.00 ©2016 IEEE

Just as the works of Salah and Aljohani, approaches to automate resource management are frequently grounded on queuing theory and stochastic processes [7], [8], [9], [10], [11]. One common limitation of those approaches is the choice for simplistic models that may not capture the behavior presented by real systems and workloads. Some researches follow different approaches. Gong et al. [12] applied signal processing to find patterns in workload and resource usage, relying on Markov chains when no pattern was identified. Vasić et al. [13] experimented with numerous off-the-shelf machine-learning techniques, reporting good results with Bayesian models and decision trees. Both approaches are based on learning patterns of workload and the configuration to handle them. Thus, in the beginning of system execution, with no previous knowledge, they rely on other simpler techniques, such as utilization limits. Additionally, they do not provide good predictions for new, not yet experienced, workloads.

The decision process involving allocation and deallocation of resources generally requires communication. In the common approach, the control unit must get utilization measurements from all service instances. There are proposals whose process is yet more time demanding, with communication among the different services that compose the system, to identify which modification should be performed. Mencagli et al. [14] uses a model to estimate response times of each system component, based on the response time of the services it depends on, and makes a number of iterations, when they exchange those estimations, to find the optimal configuration. Dejun et al. [7], [8] models the system as an acyclic directed graph with a root node and communication is partly ordered, going from leaves to the root, which centralizes the resource management decisions.

We consider our proposal distinguishing as decisions are based on the workload, and are centralized in a control process running with the load balancer. This solution requires no communication neither considering a single service nor in a composition, because all information required to make the decisions are controlled by the load balancer. Also, we use a model that considers common limitations to scalability, as will be seen in the next sections.

III. Universal Scalability Law

As defined by Gunther, the Universal Scalability Law (USL) [15], [2] tries to explain performance improvement via parallelism but also considering the constraints that limit the speedups.

One such constraint is the assumption that workload processing is rarely completely parallelized. Even if each task execution is independent, there normally are managerial tasks, such as load balancing or splitting and merging data, that run sequentially. Also, often a large number of processes compete for the same processor and occasionally need to wait. Such sequential portions incur contention delays.

Fig. 1 shows how contention limits speedup obtained from parallelism. Without contention, changing a system architecture from one to four processes brings down execution time to one quarter. With contention, the reduction in execution time is limited. Contention limits system speedup through parallel processing because it does not improve the execution time of the sequential portions. The more parallel processes used, the larger the proportion of sequential execution time, because it is constant while the parallel execution time drops. Thus, there is a point after which there is no meaningful gain in increasing parallelism, as the total execution time is dominated by the sequential execution time, a fact first described by Amdahl [16].

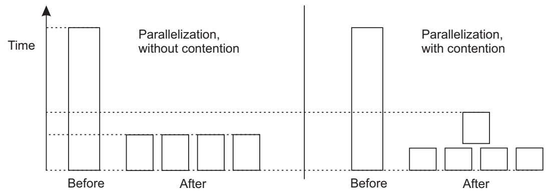

Fig. 1. Contention limits performance gains.

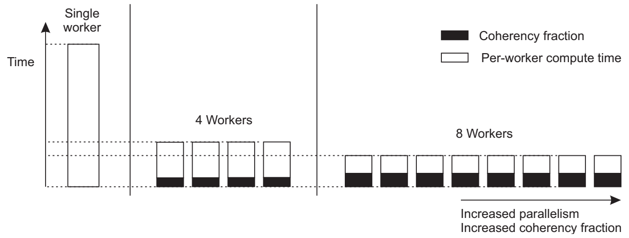

Fig. 2. Coherency limits performance gains and can cause degradation.

Besides contention, systems may need to deal with data exchange between tasks executing in parallel, referred to as coherency. Coherency delays are caused by the need to bring shared data into a consistent state. Whenever one of multiple independent processing units needs to save data, it must disseminate the operation so that all units update their data, maintaining consistency among them all. This demands extra time.

Coherency increases the execution time of each parallel processing unit, as depicted in Fig. 2. As all processing units must synchronize with each other, the number of messages exchanged for data synchronization increases quadratically with the number of processing units. Therefore, more parallelism means more time spent by each unit for synchronization. Hence, coherency constraints are more limiting than contention, because after a certain degree of parallelism, synchronization time is so high that performance drops.

According to the USL model, the relation between performance and parallelism is governed by the equation:

$$C(n) = \frac{n}{1 + a(n-1) + nb(n-1)}, \qquad (1)$$

where n is the number of parallel processing units, a is the contention factor and b is the coherency factor. C stands for capacity and is obtained by means of normalization, dividing the throughput reached with n units by the throughput of

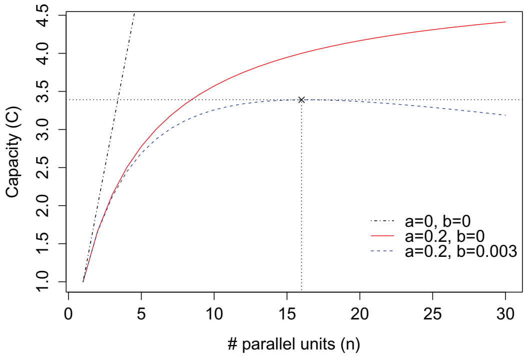

Fig. 3. The effects of contention and coherency on speedup.

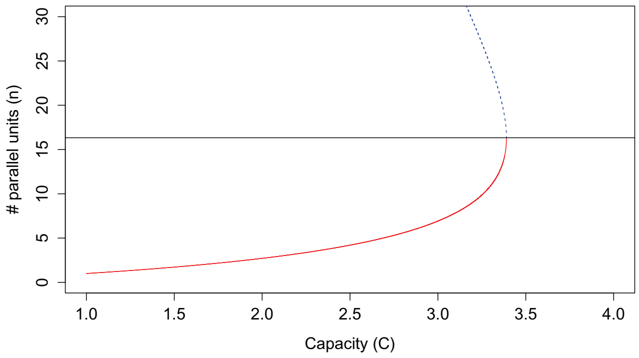

Fig. 4. Inversion of USL function

sequential execution – which is equivalent to the speedup. Contention and coherency are measured as fractions of the sequential execution time: contention is the fraction that must run sequentially and coherency is the fraction spent on one data synchronization operation.

Figure 3 shows how contention and coherency limit performance gains. As can be seen, in the presence of coherency (blue dashed curve), there is a performance peak followed by degradation. In such cases, the number of parallel processes that provide maximum throughput may be calculated as follows:

$$n_{max} = \left\lfloor \sqrt{\frac{1-a}{b}} \right\rfloor. \qquad (2)$$

As an example, with 0.2 of contention and 0.003 of coherency, we have that the maximum throughput is obtained with (n_{max}) 16 parallel processes, providing capacity of 3.39 ($C(n_{max})$), as shown in Fig. 3.

IV. Proposal

Many successful online services rely on the Service Oriented Architecture (SOA). It enables the creation of software systems based on the composition of a collection of small services, each a self-contained autonomous unit responsible for a given functionality. Two critical distinctions of SOA are message orientation – services interact by exchanging messages – and coarse granularity – services tend to use a small number of operations [17]. Thus, a service-oriented system is composed of a set of services, each with specific attributes, that communicate via message exchange.

One downside of SOA, in comparison to traditional layered systems, is that requests may need to go through more levels of the software stack (accessing many services), which can affect performance. On the other hand, functional partitioning is an important issue regarding scalability [18], which may be achieved by adding more servers running service instances and distributing the workload by means of a load balancer. This approach enables each service to scale independently.

Increasing the number of service instances potentially increases the processing capability of the service by the same amount. However, scalability is affected by the need to keep data consistent or balanced. Both sharding or replication require that instances exchange messages to keep data balanced or consistent among them all. Hence, our choice of a model that considers that performance varies with scale, as USL does: it takes into account the time that the services spend synchronizing with others.

Since we are interested in identifying the amount of resources needed to handle a given workload, we need to invert that function. This results in two partial functions (Fig. 4): one whose range goes up to n_{max} (red solid curve), and the other, from that on (blue dashed curve). We are interested only in the first portion, which provides values for the number of resources up to system capacity, which is modeled by Equation 3.

$$n(C) = \frac{b - a + 1/C - \sqrt{(a - b - 1/C)^2 - 4b(1-a)}}{2b} \qquad (3)$$

In order to handle varying workloads, we would like to modify a given system to keep throughput in equilibrium with the rate of requests (arrival rate). Otherwise, the system may become overloaded and, if the workload persists, the request queue may grow indefinitely, affecting the system's stability and performance. Since throughput capacity is a function of the number of service instances that are available, we use a control process to periodically monitor the load balancer and use the equation previously discussed (3) to estimate the adequate number of instances required to handle the current load. The goal of this control process is to, as much as possible, offer the best performance at minimum cost. The procedure is described in the next section.

A. Runtime algorithm

Resource demand is estimated following the algorithm in Fig. 5, periodically executed by the control process. The terms in monospaced font are managed by the load balancer. It will

```
1: $arrivalRate \leftarrow \texttt{arrivals}/\texttt{monitoringInterval}$
2: $capacity \leftarrow arrivalRate/\texttt{supportedArrivalRate}$
3: if $capacity >= C(n_{max})$ then
4:     $estimated\_n \leftarrow n_{max}$
5: else
6:     $estimated\_n \leftarrow ceil(n(capacity))$
7: end if
8: $diff \leftarrow estimated\_n - \texttt{current\_n}$
9: if $diff > 0$ then
10:     $addToResourcePool(diff)$
11: else if $diff < 0$ then
12:     if $\texttt{queueSize} > estimated\_n * \texttt{nodeCapacity}$ then
13:         $diff \leftarrow \texttt{queueSize}/\texttt{nodeCapacity} - n$
14:     end if
15:     $removeFromResourcePool(diff)$
16: end if
17: $\texttt{arrivals} \leftarrow 0$
```

Fig. 5. Runtime algorithm to estimate resource demand.

either update them along the execution or receive values as parameter during initialization.

The load balancer must keep track of two runtime metrics: the queue size (all current requests for the service) and the number of arrivals (new requests) since the last execution of the monitor. The load balancer must also know the monitoring interval, which is the time interval between two consecutive executions of the monitor.

In addition, the load balancer must be configured with a set of parameters related to the service characteristics: contention factor, coherency factor, maximum throughput achievable with one instance, and the average queue size to induce such throughput. In the next section we show how to estimate those parameters.

Lines 1 and 2 of the algorithm compute the arrival rate in the period since the last execution and estimate the required capacity to handle the new workload. If that capacity is higher than what USL estimates as the maximum achievable, the pool size is set to n_{max}; otherwise it is estimated using equation 3, whose result is rounded up because the pool size must be an integer and rounding down might cause under-provisioning (lines 3 to 7). Then, it computes the difference between the estimated and the current pool sizes, to update it accordingly. If it is necessary to increase the pool size, the required number of instances is promptly requested, in line 10. However, if the estimate points to a smaller pool size, it makes an additional verification in line 12. If the current queue size is too high to be handled by the estimated pool size, the control computes the minimum pool size to handle it. Then, excessive instances are released (line 15). Lines 12 and 13 are important to avoid releasing resources prematurely if there are still several requests in the queue, waiting to be processed.

B. Estimating service related parameters

We propose to estimate the service-related parameters required to execute the procedure presented in the previous section by means of load tests, in an initial profiling procedure. In a first phase, the goal is to identify the number of concurrent clients supported by a single working instance. We run a sequence of load tests, keeping the service configuration static, with a single instance processing all the requests, and varying the workload.

The simplest approach for doing that is starting with a single simulated client and adding more clients at each subsequent run. But we can use additional information to reduce the number of load tests. For instance, we can measure resource utilization caused by a workload to estimate the maximum workload supported and limit the load tests around that level.

At the end of this step, we identify the number of concurrent clients supported by the service in such circumstances ($nodeCapacity$) and estimate the throughput for that workload, which is equivalent to the supported mean arrival rate ($supportedArrivalRate$).

Then, we can move to the second phase of tests. It comprises another sequence of runs, starting with one instance available in the service pool and increasing the pool size at each subsequent experiment. The workload should be $nodeCapacity$ in the first run and grow linearly with the pool size. We can then compute the throughput of each run and combine the results with the respective pool sizes to estimate contention and coherency, following the method proposed by Gunther [19]. The method does not require exhaustive experiments nor reaching capacity limit. The author suggests at least six data points.

Contention and coherency estimates are used by equations 1 and 3, used at lines 3 and 6 in the procedure shown Fig. 5, respectively.

V. Experimental validation

To evaluate our proposal, we executed a number of experiments on DAS-5[1], a cluster composed of 68 nodes with two 8-core processors working at 2.4GHz, 64GB of RAM, and interconnected via InfiniBand and Gigabit Ethernet.

We implemented a basic setup composed of a load generator, a load balancer, a pool manager, and workers, interacting as shown in Fig. 6. The load balancer receives requests from the load generator and distributes them among the workers. Workers process requests and exchange synchronization messages. The monitoring loop, running in the load balancer, requests and releases workers from/to the pool manager. For the experiments presented in this paper, the parameters estimated via the profiling procedure as described in Section IV-B are listed in table I. The source code for the system and experiments are available as open source at https://github.com/pbmoura/scalability_experiments.

Parameter estimation was obtained with a simplified implementation of the load balancer which distributes requests in a round-robin fashion, with neither the monitoring loop nor interaction with the pool manager. This is because, during the experiments for parameter estimation, the system must remain

[1] http://www.cs.vu.nl/das5/

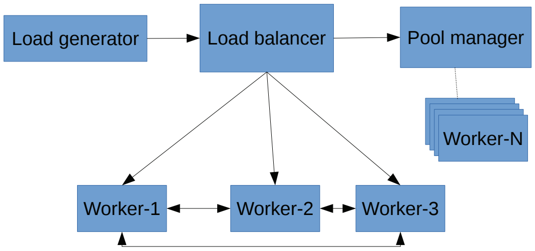

Fig. 6. Experiment setup

TABLE I
SETUP ESTIMATED PARAMETERS

Parameter	Value
$nodeCapacity$	4
$arrivalRate1$	0.769
a	0.0398
b	0.0031

at the specific configurations defined in the load tests and should not auto-scale. The load generator produced uniform workloads, varying the number of simultaneous processes to simulate concurrent clients. The elasticity experiments were executed with a single thread of requests, but varying the inter-request interval along the execution.

Here, we describe the experiments used to observe how the method proposed in this paper takes advantage of the cloud's elasticity, adapting the system's pool size to the workload. We compare our proposed method to an implementation based on resource utilization. To do so, we implemented a load balancer that requests and releases resources based on overall utilization, and configured it to remove one instance from the pool when utilization is below 35% and request one instance when utilization surpasses 80%.

A. Experiment 1

We start by presenting an experiment to show the difference in reaction time between our approach and a utilization-based, single-step approach, in which the resource pool is updated one VM at a time. To simulate a drastic change in the workload, we begin the experiment with a workload of 1 request per second and increase it to 4 requests per second. This variation demanded an increase in the pool size, and the workload remained at that level long enough for both approaches to stabilize the performance. The monitoring period is 20 seconds. Figure 7 shows the variation in the queue sizes along the experiment, in comparison to the variation in the workload. While the single-step approach took 194 seconds to stabilize, our proposal, using USL, needed only 94 seconds. This better reaction time also guaranteed less queue growth.

Better performance is an effect of the different strategies for resource allocation, as shown in Fig. 8. Even the allocation of more resources in the beginning of the higher load period maintained the overall cost lower, for two reasons. Due to its delayed response, the single-step approach needed to add more instances to the pool to deal with the larger increase in queue size. When the workload dropped, USL provided a faster adjustment to the pool size, avoiding wasting resources.

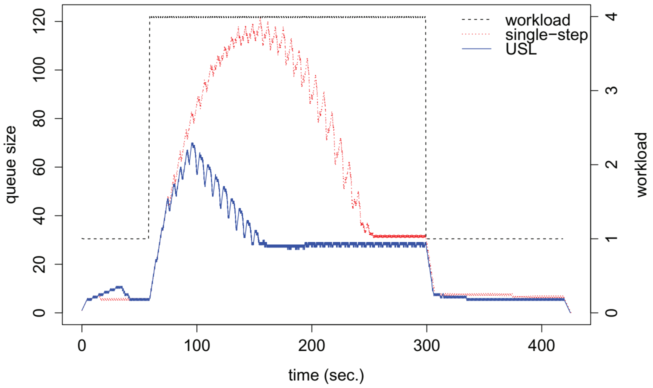

Fig. 7. Experiment 1: Queue size variation

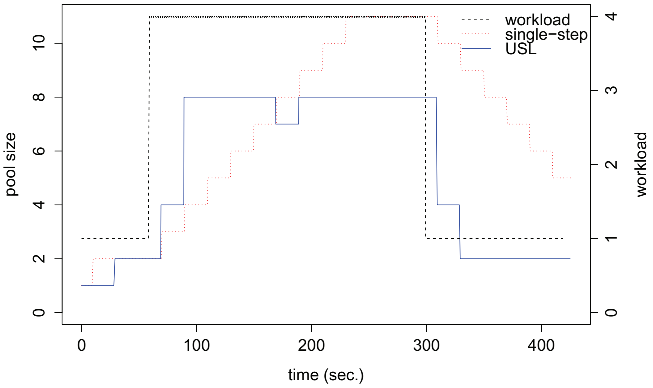

Fig. 8. Experiment1: Pool size variation

The different reactions to workload variation led to differences in the response times. Fig. 9 shows the response time of each request executed in the experiments, ordered by departure time. As can be seen, performance delivered to users by our algorithm is better. Just for the sake of an example, if we consider an SLA of 13 seconds, the number of SLA violations with the single-step approach is more than five times higher than that with our USL-based approach. Table II summarizes a comparison of the approaches. We consider cost as the total time of VM allocation, computed as the area under the respective pool size lines of the graph presented in Fig. 8.

B. Experiment 2

We were considering whether a small monitoring period would make the single-step approach as effective or even better than our approach. Thus, to evaluate how the monitoring period affects the performance of the two approaches, we ran a new experiment using a shorter duration for the increased workload of 4 requests/s. After 30s at 0.5 requests per second, we maintained the load at 4 requests per second for 60s, and switched back to 1 request per second for 30s. With this workload profile, we ran a sequence of experiments increasing

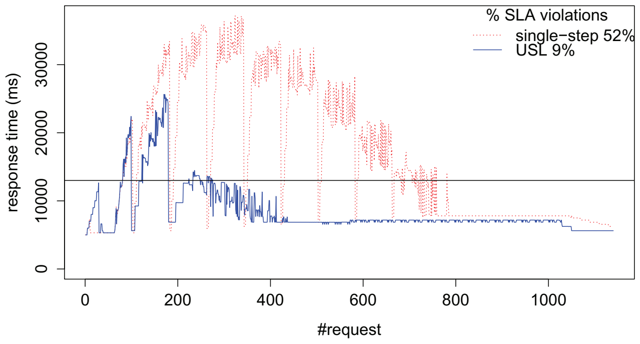

Fig. 9. Experiment1: Response time variation

TABLE II
EXPERIMENT 1 METRICS COMPARISON

Metric	USL	Single-step
Max queue size	70	122
Time to stabilization	94	194
Max pool size	8	12
Avg pool size	5.18	6.88
Cost	2200.96	2929.1
Avg service time (std dev)	8.45 (3.6)	16.44 (9.82)
SLA violations %	11	64

the monitoring period from 1s to 30s. Table III shows the differences in terms of SLA violations, considering an SLA of eight seconds.

We can see that increasing the monitoring period quickly degraded the performance provided by the single-step approach, while with USL the degradation is slower. Indeed, with a monitoring period of 1s, which might incur into a large overhead in many situations, the single-step approach performs better. But with all other monitoring periods, it is worth to predict the required amount of resources with the USL-based approach.

The efficacy of a pool-update policy depends on when, in the period between two monitoring iterations, the workload has changed. If that happens right after the previous iteration, the system will run overloaded for a longer period, resulting in more SLA violations.

TABLE III
EXPERIMENT 2 - MONITORING PERIOD VS. SLA VIOLATIONS

Monitoring period	USL	Single-step
1	11	0
2	2	5
3	1	17
4	4	66
5	6	72
7	13	70
10	13	80
13	22	83
15	66	85
20	27	85
30	71	88

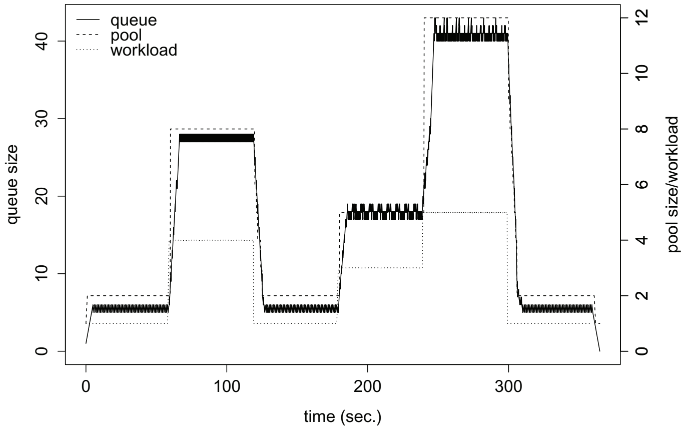

Fig. 10. Experiment 3: Baseline

That explains, for instance, why USL provided worse performance with 15s than with 20s of monitoring period. The workload jumped from one to four requests per second after 30 seconds of execution. Thus, when experimenting with 15s intervals, the system was overloaded during almost all of the 15s period of the third inter-monitoring cycle. In contrast, in the case of 20s intervals, the overload happened during the last 10s of the second cycle, yielding fewer violations. The same is also valid for the monitoring period of 30s. Still, in both cases, performance was better than using single-step.

Looking at the single-step approach, the results suffer from the additional time the system was overloaded before the reaction. Making decisions simply based on utilization does not make it possible to determine what is the amount of demanded resources, forcing the system to incrementally add a single instance per step. More time is needed to achieve an adequate pool size.

C. Experiment 3

To observe what would be the behavior of both approaches in a more complex situation, we now experiment with a workload with two spikes at different frequencies of requests/s. The frequency of requests in this workload follows a pattern that stays 60s in each of the following levels of requests/s: 1, 4, 1, 3, 5, 1, as can be seen in the dotted line in Fig. 11.

The focus is on the second spike. The intention is to observe the performance of each method with a sudden drop in the workload followed by a new increase. To compare the performance of both approaches with a close-to-optimal allocation of resources, we use, as a baseline, an experiment with an 1s monitoring period using USL to estimate demand. As can be seen in Fig. 10, this baseline provides a quick reaction to workload increases and a pool size reduction accompanying the queue size, avoiding overload in both cases. Thus, we can consider the system was never overloaded.

We compare, then, executions with the USL and single-step approaches using a monitoring period of 20s. Figures 11 and 12 show the variations of the queue and pool sizes with both

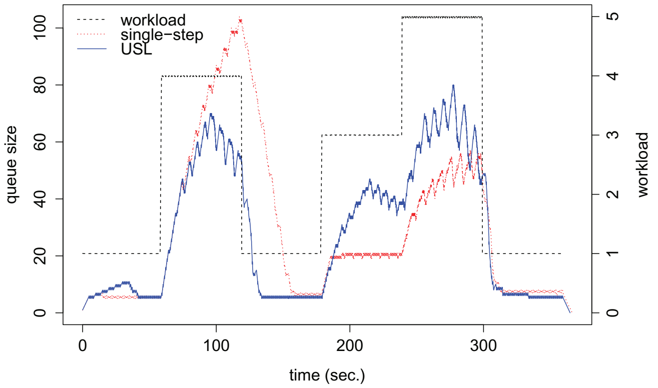

Fig. 11. Experiment 3: Queue size variation

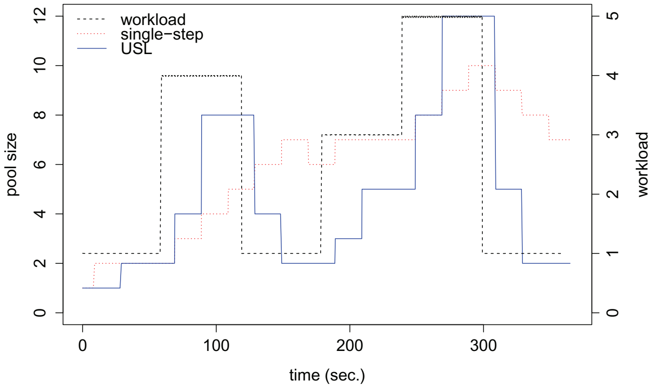

Fig. 12. Experiment 3: Pool size variation

approaches. Regarding the second load spike, we see that, due to the delayed pool size reduction, the single-step approach has still a large pool of service replicas when the workload increases, avoiding overload. Thanks to that, it provides better performance, resulting in less SLA violations (Fig. 13). This is an interesting case in which the slower reaction time of the single-step approach resulted in a better overall performance, because it was lucky to have the second spike that used the resources it still had. But, as shown in Figure 12 and Table IV this better performance has a price as it comes at an extra cost in terms of resource allocation.

Table IV shows a comparison of USL, single-step, and the baseline. Our proposal produced a better average performance in comparison to the single-step approach, which can be observed in the average queue size and response time. Regarding average pool size and cost, our proposal performed better than the baseline. That can be explained by the fact that, when there is a workload increase, it is partially captured by the control process in its subsequent execution. The average workload is estimated based on a portion of the workload before the increase and another portion after the increase. Only in the next iteration is the workload increase fully captured. Then, instead of increasing the pool size in a single step, the system does it in two, reducing the average pool size and, consequently, the cost of running the system. The downside is the system overload and SLA violations in the meantime.

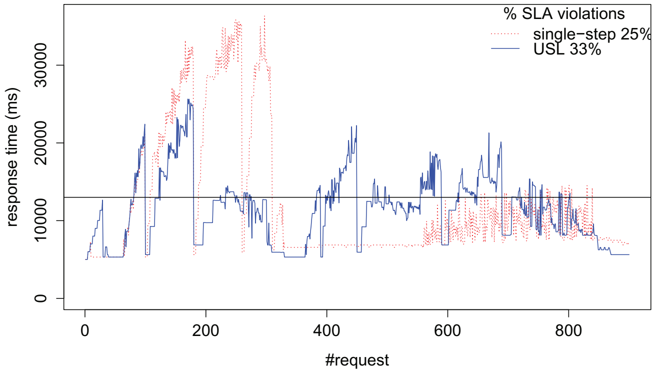

Fig. 13. Experiment 3: Response time

TABLE IV
EXPERIMENT 3 METRICS COMPARISON

Metric	USL	Single-step	Baseline
Avg queue size	28.16	29.87	16.91
Avg pool size	4.76	5.93	5.22
Cost	1735.57	2171.38	1904.56
Avg service time (std dev)	11.38 (4.44)	12.14 (8.26)	6.85 (1.05)
SLA violations %	33	25	0

Increasing the interval between the spikes so that the single-step approach has enough time to reach the same pool size set with USL makes both spikes present similar effects, as shown in Fig. 14. In this case, USL also provides fewer SLA violations with 29% of the requests violating the SLA against 56% with single-step.

Our results show the benefit of a quick reaction to workload increment and highlight that the extent of such reaction is also relevant. Promptly setting an adequate pool size shortens the time to stabilize the system and, in case of continued higher load, reduces the total amount of required resources.

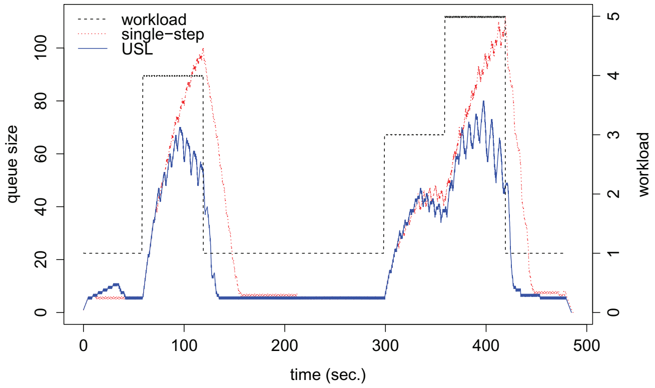

Fig. 14. Experiment 3: Queue size variation with long interval between spikes

The choice of the length of the monitoring period should take into consideration the cost of running the control process, in terms of utilization and time. Applying the procedure proposed in this work has the benefit that decisions are centralized, with no need for processing or communication from the worker instances. For this to be correct, though, the resources must be homogeneous.

Regarding workload decrease, we also favor a quick reaction, but there are two distinguishing scenarios: when dealing with occasional workload spikes, our proposal presents itself as a good option; in case of frequent workload variations, however, our proposal focused on cost reduction, which did not appear to be the best option, as that may affect performance.

We must also consider that, when hiring a third-party cloud infrastructure, the billing model may also affect the decision to deallocate virtual machines. With some IaaS providers, charges are calculated per hour, without the possibility to pay for fractions. Accordingly, a better approach might be to tag VMs for deallocation near the completion of full hours. This simple change might already bring benefits to our proposal in case of frequent variations in the workload, as it can postpone reductions in the pool size without affecting costs.

VI. Conclusion

One of its major attributes of cloud computing is the ease to scale up and down the resources allocated to a given system, i.e., elasticity. In this paper, we explored the application of the Universal Scalability Law, a scalability model that allows estimating relations between performance and resource demand, to accelerate the allocation and deallocation of virtual machines in response to workload fluctuations. This is in contrast with current standard approaches for auto-scaling, which are normally based on utilization.

We showed that the proposal is beneficial in terms of maintaining performance and lowering cost in cases of workload increments. In case of workload reduction, although always more economic, in a few cases the quick reaction may be harmful in terms of SLA violations, when consecutive workload increases and decreases are frequent. As future work, we expect to be possible to define additional tuning parameters to minimize this effect.

We are currently working to validate this proposal by means of trace-driven simulations with realistic Web workloads that are non-uniform and with higher arrival rates.

The next step in this research is to work on the dynamic runtime refinement of the model to improve the precision of estimates. This is of interest specially for systems with low contention and coherency. Initial experiments at low load could produce a model that makes poor predictions at workload levels far from those used in the estimates. Thus, we plan to work on a feedback loop that observes deviations of measured performance from estimates and uses the new measurements to update the model at runtime.

Acknowledgment

This research was funded by CAPES and CNPq Proc. 305566/2013-0.

References

[1] Gartner, "Gartner says worldwide cloud infrastructure-as-a-service spending to grow 32.8 percent in 2015," May 2015, http://www.gartner.com/newsroom/id/3055225.

[2] N. Gunther, "A General Theory of Computational Scalability Based on Rational Functions," *arXiv preprint arXiv:0808.1431*, pp. 1–14, 2008.

[3] E. L. C. Mamani, L. A. Pereira, M. J. Santana, R. H. C. Santana, P. N. Nobile, and F. J. Monaco, "Transient performance evaluation of cloud computing applications and dynamic resource control in large-scale distributed systems," *2015 International Conference on High Performance Computing \& Simulation (HPCS)*, pp. 246–253, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7237046

[4] H. Lim, S. Babu, J. Chase, and S. Parekh, "Automated control in cloud computing: challenges and opportunities," in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, 2009, pp. 13–18.

[5] K. Salah, K. Elbadawi, and R. Boutaba, "Estimating service response time for elastic cloud applications," *2012 1st IEEE International Conference on Cloud Networking, CLOUDNET 2012 - Proceedings*, no. January 2016, pp. 12–16, 2012.

[6] A. Aljohani, D. Holton, and I. Awan, "Modeling and Performance Analysis of Scalable Web Servers Deployed on the Cloud," *2013 Eighth International Conference on Broadband and Wireless Computing, Communication and Applications*, pp. 238–242, Oct. 2013.

[7] J. Dejun, G. Pierre, and C.-H. Chi, "Autonomous resource provisioning for multi-service web applications," in *Proceedings of the 19th international conference on World wide web - WWW '10*, New York, New York, USA, 2010.

[8] J. Dejun, G. Pierre, and C. Chi, "Resource Provisioning of Web Applications in Heterogeneous Clouds," in *USENIX Conference on Web Application Development*, 2011.

[9] A. Harbaoui, B. Dillenseger, and J.-M. Vincent, "Performance characterization of black boxes with self-controlled load injection for simulation-based sizing," in *French Conference on Operating Systems (CFSE)*, 2008.

[10] A. Harbaoui, N. Salmi, B. Dillenseger, and J.-M. Vincent, "Introducing Queuing Network-Based Performance Awareness in Autonomic Systems," *Sixth International Conference on Autonomic and Autonomous Systems*, pp. 7–12, Mar. 2010.

[11] N. Salmi, B. Dillenseger, A. Harbaoui, J.-m. Vincent, and O. Labs, "Model-based Performance Anticipation in Multi-tier Autonomic Systems : Methodology and Experiments," *International Journal on Advances in Networks and Services*, vol. 3, no. 3, pp. 346–360, 2010.

[12] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic reSource Scaling for cloud systems," in *Proceedings of the 2010 International Conference on Network and Service Management, CNSM 2010*, 2010, pp. 9–16.

[13] N. Vasić, D. Novaković, S. Miucin, D. Kostić, and R. Bianchini, "DejaVu: Accelerating Resource Allocation in Virtualized Environments," in *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[14] G. Mencagli, M. Vanneschi, and E. Vespa, "Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments," *Proceedings of the 2013 International Conference on High Performance Computing and Simulation, HPCS 2013*, pp. 11–18, 2013.

[15] N. Gunther, "A Simple Capacity Model of Massively Parallel Transaction Systems," in *CMG-CONFERENCE*, 1993.

[16] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483—-485. [Online]. Available: http://dl.acm.org/citation.cfm?id=1465560

[17] W3C, "Web services architecture," Feb. 2004, http://www.w3.org/TR/ws-arch/.

[18] D. Pritchett, "Base: an Acid Alternative," *ACM Queue*, vol. 6, no. 3, pp. 48–55, 2008.

[19] N. Gunther, *Guerrilla Capacity Planning*. Springer, Berlin Heidelberg, 2007.