Elastic Scaling in the Cloud: A Multi-Tenant Perspective

Navaneeth Rameshan[†*], Ying Liu[*], Leandro Navarro[†] and Vladimir Vlassov[*]
[†]Universitat Politècnica de Catalunya, Barcelona, Spain
Email: rameshan@ac.upc.edu, leandro@ac.upc.edu
[*]KTH Royal Institute of Technology, Sweden
Email: yinliu@kth.se, vladv@kth.se

***Abstract*—Performance interference in the hosting platform introduces uncertainty in the performance guarantees of provisioned services. Existing elasticity controllers are either unaware of this interference or over-provision resources to meet the SLO. In this paper, we take a holistic view on elastic scaling from a multi-tenant perspective. We show that performance interference can significantly impact the accuracy of scaling and result in long periods of SLO violation. Using Memcached as a case-study, we show that making an elasticity controller interference aware can improve the accuracy of scaling decisions and significantly reduce the periods of SLO violation.**

***Keywords*—Elasticity, Performance interference, Cloud, Predictable Performance.**

I. Introduction

Enabling elastic provisioning saves the cost of hosting services in the Cloud, since Cloud users only pay for the resources that are used to serve their workload. For example, popular content that is frequently accessed can be replicated in caching layers to provide low latency access to multiple users. These caching layers must be able to adapt to the varying loads in traffic in order to save provisioning costs and to deliver content at high speeds. Cloud providers such as Amazon provide support for elastic scaling of resources to meet the dynamic demands in traffic. Previous works have proposed multiple models ranging from simple threshold based scaling [1], [2] to complex models based on reinforcement learning [3], [4], control modelling [5], [6], and time series analysis [7], [8] to drive elastic provisioning of resources. The major challenge is to decide when to scale and to decide the right amount of resources to provision. These challenges are further exacerbated by performance variability issues that are specific to cloud such as performance interference. None of the previous work consider performance variability from interference. Web services hosted in the cloud must be aware of such issues and adapt when needed.

Performance interference happens when behavior of one VM adversely affect the performance of another due to contention in the use of shared resources such as memory bandwidth, last level cache etc. Prior work indicate that, despite having (arguably the best of) schedulers, the public cloud service, Amazon Web Service, shows significant amount of interference [9], [10]. Existing solutions primarily mitigate performance interference and guarantee performance in either one or a combination of these 3 ways: (i) Hardware partitioning [11], [12] (ii) at the scheduling level [13], [14] or (iii) by dynamic reconfiguration [15]. All these solutions, look at ways to guarantee performance in a multi-tenant setting by either resorting to hardware modification, VM placement or reconfiguring the VMs. Elastic scaling in a cloud environment does not come with the convenience of choosing where to spawn a new VM and this further exacerbates the problem.

We take a holistic view on scaling from a multi-tenant perspective and show that when elasticity controllers are unaware of interference, it results in either of (i) Inaccurate scaling decisions about when to scale, (ii) long periods of unmet capacity that manifest as Service Level Objective (SLO) violations or (iii) higher costs from over provisioning. We augment them to be aware of interference to significantly reduce SLO violations and save provisioning costs. We consider Memcached for elastic scaling, as it is widely used as a caching layer, and present a practical solution to augment existing elasticity controllers in 4 ways: (i) By improving the accuracy of scaling decisions (ii) at the ingress point by load balancer reconfiguration (iii) by reducing the convergence time when scaling out and (iv) by taking informed decisions when scaling down/removing instances. We use hardware performance counters to quantify the intensity of interference on the hosts. This is achieved with the help of a middleware that exposes an API for VMs to query the amount of interference in the host. Decisions by the elasticity controller is then augmented with the help of this information.

II. Background

A scaling process involves 2 steps: making a decision on when to scale and choosing the right number of instances to be added/removed to serve the changing workload. Both these steps depend on the capacity of a VM. *Capacity* is defined as the maximum amount of workload a VM can serve within a certain level of QoS. A decision to scale-out is established when the elasticity controller detects that the workload exceeds the current capacity of the VMs. It then determines the additional capacity needed to serve the excess workload and spawns the required number of VMs. The accuracy of scaling thus depends on the agility of the controller in detecting workload changes (decision making) and in satisfying the additional capacity (actuation). In this section, we give a brief background on load-based elastic scaling and the required properties for the metrics used to drive the scaling decision.

A. Scaling Type

Load-based scaling handle variable loads by starting additional instances when the workload increases and stopping instances when workload decreases, based on any of load metrics, such as request intensity (RPS). It can be achieved in three ways: reactive control, proactive control and a combination of

reactive and proactive control. With reactive control, the system relies on reacting to changes in a system metric (indirect metric) such as workload intensity, intensity of I/O operations, CPU utilization, or direct metrics like latency to make scaling decisions. While this approach can scale the system with good accuracy, the system reacts only after the change occurs and is observed. This may result in SLO violation if the reaction is too late. Proactive control, on the other hand, explores the historic access patterns of the workload, in order to conduct workload prediction and perform model-predictive control. With this approach, it is possible to prepare the instances in advance and avoid any disruption in the service when auto-scaling. Despite their respective merits and demerits, both approaches require run-time measurement of a metric to make the scaling decision and to drive the elasticity control. In the following subsection we briefly explain the properties of a good metric that can be used to drive the scaling decisions.

B. Choice of Metrics

The right choice of metrics (control input) is critical for efficient elastic scaling since the performance, effectiveness and precision of the elasticity controller depends on the quality of the control input metric and the overhead in measuring and monitoring the control input [16]. In literature, authors have used a variety of metrics to make scaling decisions and to drive the elasticity control. An extensive list of those metrics is provided in [17]. A good choice of metric for the target environment should satisfy the following properties: (i) the metric should be easy to measure accurately without intrusive instrumentation because the controller is typically external to the guest application, (ii) the metric should be reasonably stable with little variations, (iii) should allow for quick reaction and (iv) the metric should correlate to the measure of level of quality of service (e.g, the service's average response time or latency) as specified in the SLO. Based on this the metrics can be further divided into:

Direct Metric: A straightforward approach to scale is to directly rely on the metric (latency, response-time) specified in the SLO to drive the scaling decisions. However, it does not satisfy the properties of a good metric, since monitoring latency/response-time may involve an overhead in measuring, needs instrumentation and reacts slower than an indirect metric. Some developers are however willing to incur the overhead in view of the benefits they accrue from easier scaling since the response variable to be tuned is measured directly.

Indirect Metric: Scaling using indirect metrics do not measure the response variable directly, instead use other metrics that correlate well the measure of service quality (latency) and satisfies the properties of a good metric. CPU utilization is one such widely used metric [18], [8], [7], [6], [19]. It can be obtained from the operating system or the virtual machine without instrumenting application code. CPU utilization also correlates well with the measure of service quality such as latency/throughput [6]. Another widely used metric for elastic scaling is workload in terms of Requests-per-second (RPS) [20], [5], [21], [22], [23]. RPS can be an important way to measure system performance and is mostly used for proactive control. Netflix developed a system called scryer [20] that uses workload to drive their proactive control for scaling decisions. Because CPU utilization and workload intensity are widely used in a large number of elasticity controllers, our work focuses on these two indirect metrics.

III. Problem Definition

Any scaling process involves 2 steps: making a decision on when to scale and choosing the right number of instances to be added/removed to serve the changing workload. In this section we focus on these two steps and present the challenges in the presence of performance interference.

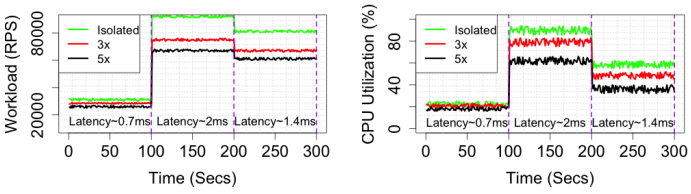
We perform experiments for studying the impact of interference on load and VM capacity on a private cloud testbed. It comprises of Intel Xeon X5660 nodes, each with 6 physical cores operating at 2.8GHZ and 12MB of L3 cache. We focus primarily on the performance of in-memory storage systems and run experiments using Memcached to study the impact of interference. Interference is generated using a slew of realistic applications from SPEC CPU benchmark [24] and benchmarks such as mbw [25] and Stream [26].

A. Decision Making

In the decision making phase, the elasticity controller has to accurately determine the need to scale-out/scale-down either based on a direct metric such as latency or an indirect metric such as CPU utilization or Workload intensity. When the controller relies on a direct metric, the decision making is highly accurate as it directly relies on the response variable itself. However, when relying on an indirect metric, the accuracy depends on how well the metrics correlate with latency. We focus on indirect metric in this section and show that in the presence of interference, the correlation gets skewed and can impact the accuracy of the decision making phase.

Need to attribute Interference: Figure 1 shows three stages of execution when the latency is maintained at 0.7ms, 2ms and 1.4ms. Figure 1a shows that workload correlates with latency, i.e., workload follows the same pattern as latency (First stage with lowest value, second stage with highest value and third stage with mid-range values). However, it correlates differently in the presence of interference. For simplicity, consider a simple threshold based scaling approach. If the SLO latency is set to 2ms (2nd stage in the figure), then the threshold setting of a VM modelled in an isolated environment is around 90000 RPS. However, in the presence of 5x interference the same latency of 2ms is achieved at a much lower workload of 65000 RPS. Without attributing and quantifying interference it is impossible for the elasticity controller to know the right threshold value to use for scaling out. Threshold value depends on the interference and needs to be dynamically modelled. Similar observations can be made for CPU utilization from figure 1b. Our results show that the two widely used indirect metrics, CPU utilization and workload, correlate differently when experiencing contention. It becomes imperative to quantify interference in order to make right scaling decisions.

Interference vs. Load: Interference has a significant impact only at high loads. Figure 1 shows this observation. The impact of interference is negligible under 30000 requests per second but as the load increases (2nd and 3rd stage) interference significantly degrades the capacity of the VM. This observation implies that the impact of interference can be mitigated to a large extent by reducing the workload served on impacted VMs. This is crucial for the decision making phase as it implies that SLO violations need not always be dealt with scaling out. For example, the load balancer can be configured with weights corresponding to the interference to minimise the workload served to VMs that are heavily impacted. This in turn saves the cost of unnecessary scaling out when the workload demands can be met with load balancing.



(a) (b)

Fig. 1: This figure demonstrates how interference skews the correlation between indirect metrics and latency. Experiment is divided into 3 stages where latency of each stage is maintained at the specified value. 3x and 5x represents 3 and 5 instances of interfering co-runners respectively. (a) Workload intensities at different levels of interference required to maintain the specified latency for each stage. (b) CPU utilization at different levels of interference required to maintain the specified latency for each stage.

B. Actuation

Actuation is the second phase in elastic scaling. Once the elasticity controller decides to scale out/scale down VMs, actuation is the phase where the actual scaling takes place. The Elasticity controller needs to decide how many VMs to launch or remove and should aim to minimise SLO violations as much as possible. The efficiency of scaling thus depends on the elasticity controller in satisfying the capacity requirements. Although modelling techniques [22], [23], [5], [6] can help determine the required capacity, it is difficult to identify the right number of VM instances required to meet the new workload demand. This is because, the capacity of a VM is not determined by the resource specification of a VM alone. Performance interference also impacts the VM capacity.

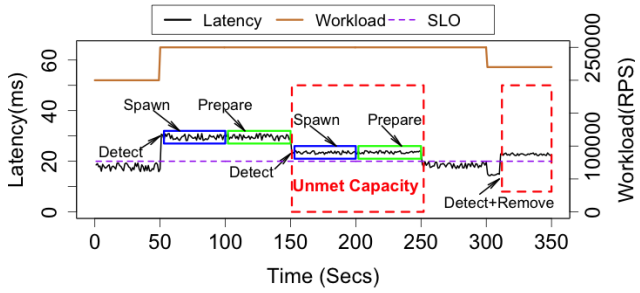


Fig. 2: Memcached service experiencing a long period of SLO violation during periods or scaling out and scaling down because of unmet capacity from performance interference.

VM Capacity: Figure 1a also shows that the capacity of a VM, i.e., the maximum amount of workload it can handle for a given latency (SLO) is inversely proportional to the amount of performance interference in the host. The VM has the highest capacity when running in isolation. As the amount of interference in the host increases, the capacity diminishes. This means that an elasticity controller that is unaware of performance interference, when scaling out, will always spawn VMs assuming a much higher capacity than achievable (in the presence of interference) that can lead to longer periods of SLO violations. With 5x interference the capacity diminishes by up to 35%. This means that, even if an elasticity controller over provisions every VM assuming the capacity at 5x interference, we end up effectively paying an unnecessary cost of 1 VM for every 3 VMs spawned (33% higher cost). We therefore need to augment the elasticity controller with knowledge of interference to minimize provisioning cost and SLO violations.

Convergence of Scaling-Out: Convergence time of scaling is the time it takes an elasticity controller to reach a stable desirable state. Figure 2 demonstrates the undue delay in convergence of a scaling process because of unmet capacity. From time 0-50 secs, the cluster is able to handle the workload and latency remains below the SLO. After 50 secs the workload increases and the elasticity controller detects the need to scale out to meet the capacity requirements of the increased workload. The elasticity controller spawns the required number of VM instances and prepares the instance with the necessary data to serve the additional workload. The number of instances spawned to serve the additional workload is based on its knowledge of additional capacity needed. We call this the first phase of scaling. Although latency reduces after the first phase of scaling, it still violates the SLO. The new instance (VM_{inter}) happened to be spawned on a server highly impacted by interference. As a result, the VM does not have enough capacity to serve the additional workload within the SLO. This is because performance interference reduces VM capacity. Had there been no interference, the SLO would have been maintained after the first phase of scaling. The elasticity controller is unaware of this interference and detects SLO violations at 150 secs and immediately spawns and prepares another instance to maintain the SLO. The period between 150 to 250 secs is the period of unmet capacity from interference and increases the convergence time of the elasticity controller. This period is directly proportional to the time it takes to spawn and prepare a new instance. Our experiments on Amazon AWS suggest that the time taken to prepare an instance is anywhere between 2 to 28 minutes depending on the size of data to be transferred. This delay results in SLO violations and can only be discovered by the controller after the first phase of scaling. This is because, the elasticity controller is oblivious to interference and cannot know the capacity of the VM before it starts serving the workload. Once another VM is spawned, the SLO is maintained. The cluster converges to a desirable state only after spawning and preparing this additional VM.

Scaling Down: In the same figure (figure 2), we see that the workload drops at around 300 seconds and the elasticity controller detects and removes an instance based on its model of capacity. However, removing the instance immediately results in SLO violations. This is because the controller is unaware of interference and randomly chose a VM to remove and it so happened to be a VM with high capacity (least interfered). The excess workload from removing this VM overwhelms VM_{inter} and exceeds the capacity VM_{inter} can handle. With augmentation, we make informed decisions on choosing which VM to remove.

IV. Solution Overview

Our solution primarily consists of a Middleware Interface (MI) to quantify capacity. MI resides on all the hosts in the cluster and exposes an API that can quantify the maximum amount of workload a VM on the host can handle without violating the SLO. MI computes this based on the amount of contention in the shared resources of the host. The elasticity

controller orchestrates with the MI to improve the accuracy of scaling.

Before taking a decision to scale, the Elasticity Controller (EC) consults the MI on all hosts to quantify the capacity of each host. The EC then recomputes the right threshold for scaling based on the interference quantified by MI. Based on this knowledge of host capacity, along with the current workload, EC decides to scale out only when the threshold is violated. A decision is taken only when the EC has enough historical evidence to believe that the interference remains sustained. In the event of a decision to scale-out, the EC first tries to generate a plan to balance the load among hosts without adding new VM instances. The plan aims to reduce the load on highly interfered VMs by diverting the workload to VMs that are less interfered. If such a plan is feasible, EC reconfigures the load balancer. This is the first level of reconfiguration. However, such a plan may not always be possible. If the EC learns that a re-balancing solution alone cannot maintain the SLO, it adds new instances computed by the additional capacity needed to serve the excess workload. It then waits for the new VMs to spawn. As soon as they are spawned, the MI on the hosts of the newly spawned VMs are consulted to learn the capacity of these VMs. Note that the MI need not wait for the VMs to finish preparing the instance to know the capacity. This is because the only hardware performance counter of Memcached that the MI relies on, to quantify interference, is the cache-reference rate (for other counters used from co-running VMs, see section V for more details) and this is unaffected by the hit rate of requests on Memcached. This counter only captures the effective rate at which the cache is accessed, which is only dependent on the rate at which the instance receives the requests. The preparation phase is briefly paused for a few seconds to determine the capacity based on the current level of interference. It is precisely because of this capability that the EC can determine if the newly spawned instances are enough to handle the increased workload even before the preparation phase is fully complete. If the newly added instances are incapable of maintaining the SLO, the EC spawns additional instances in parallel with the prepare phase of the previous instantiation. This process of parallel instantiation along side the prepare phase significantly reduces the duration of SLO violations. On the other hand, if the decision of the EC is to remove instances, it consults the MI, recomputes the overall capacity needed to serve the workload and removes the right number of highly interfered instances.

The results from augmenting the elasticity controller in figure 2 is shown in figure 3. Note how the periods of unmet capacity are significantly reduced. Spawning and preparing VMs in parallel with the preparation of the VMs in the first phase of scaling minimises SLO violations. Similarly, when scaling down, there are no SLO violations since the augmented approach is aware of interference and removes the VM that is highly interfered (VM_{inter}).

V. Middleware Interface to Quantify Capacity (MI)

The middleware interface (MI) exposes an API that is responsible for quantifying the capacity of the VM in terms of the maximum workload it can handle for a given SLO. In order to quantify the capacity, MI first needs to quantify the amount of interference in the host. The primary role of MI is to quantify the drop in the performance of a target application from interference and to translate this performance degradation to capacity.

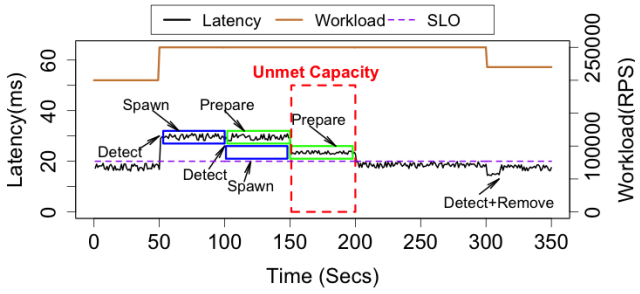


Fig. 3: Unmet capacity from figure 2 pruned by augmenting the elasticity controller to detect and quantify interference.

PMU based approximation: We use performance monitoring units (PMU) to quantify and approximate the amount of contention in a system. PMU's are special registers in modern CPUs that can collect low level hardware characteristics of an application without any additional overhead. The goals are two-fold: to identify the existence of contention from the co-runners and to quantify the amount of the pressure they exert on the target system. We monitor the performance counters of both the co-runner and the target application to quantify interference.

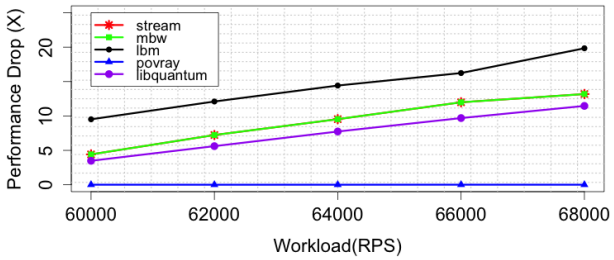


Fig. 4: The drop in performance of Memcached under contention. Memcached is run alongside multiple instances of different co-runners.

A. Characterising Contention:

In order to characterise contention, we choose in-memory storage system Memcached as a demonstrative target system to show the scaling of services under performance interference.

Properties that determine degradation: Figure 4 shows the drop in performance experienced by Memcached when interfered with different co-runners. It is interesting to note that the drop in latency increases linearly for the same co-runner. With low-level PMU analysis we discern the properties that define the aggressiveness of the co-runners. cache references and cache misses of the co-runner is a good indicator of cache contention. Intutively, higher cache references from the co-runners effectively reduces the cache space of Memcached, resulting in a drop in performance. LLC prefetches and LLC prefetch misses of the co-running application gives an approximation of the memory access patterns. Finally, the extent of degradation suffered by cache access reduction is captured by cache-reference counter for Memcached. The higher the degradation suffered, the lesser the cache-references of Memcached. From our experiments, we find that these counters provide a good model to quantify sensitivity of Memcached.

Interference Index: The goal of characterising contention is to quantify the properties of the co-runners that lead to performance degradation of the storage system. We call this metric interference-index and it approximates the performance degradation suffered by the target system. In order to be useful, the metric must correlate with the performance drop suffered by Memcached. Using the counters from our analysis and a training data set of co-runners, our system then builds a model that correlates co-runner properties with performance drop suffered by the storage system. We then use linear regression to construct the interference index. Figure 5 shows the interference-index constructed for Memcached. Since the modeling is data-driven, the interference index generated is application-dependent. Figure 5 shows that interference index correlates with performance drop suffered by Memcached. Higher the interference index, greater the performance drop suffered. In this example, our training set consists of lbm,mbw,libquantum and povray. We then fit omnetpp, milc and stream into the generated model. Stream causes similar degradation as mbw and they both correspond to similar interference indexes. milc causes a degradation that is greater than povray and omnetpp but lesser than mbw and is also captured by the model as expected. The value of interference index approximately corresponds to the drop in performance of Memcached. Once interference-index is quantified, it is then used as a control input to determine the capacity of the VM.

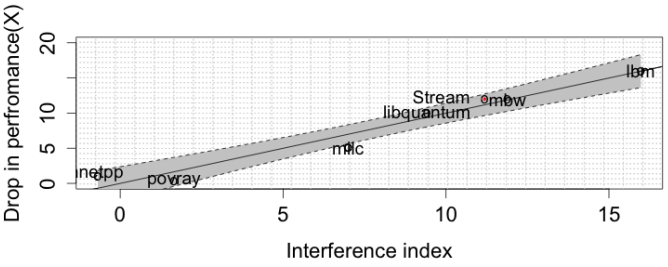


Fig. 5: Interference-index quantifies the performance drop suffered by Memcached based on the behaviour of the co-runner.

Capacity: We apply a binary classifier used in the state of the art approaches [22], [23] to approximate the capacity of a VM, which defines its ability to handle workloads within the SLO. Previous works adopt CPU utilization, workload intensity and workload composition, including read/write ratio, data size etc., as typical indirect metrics. In addition, our model includes the interference index, which, as shown in previous sections, significantly influences the SLO. With sufficient training data, the classifier uses SVM (support vector machine) to estimate a function for a given indirect metric and latency constraint, which outputs if it satisfies (true) or violates (false) the SLO. Then, we are able to obtain the maximum supported workload intensity or CPU utilization satisfying the SLO. Figure 6 shows how our interference aware approach is able to define the threshold to meet SLO for different interference index.

VI. Experimental Evaluation

A. Experiment Setup

We co-locate memory intensive VMs with the storage system on the same socket for varying degrees of interference by adding and removing the number of instances. MBW, Stream

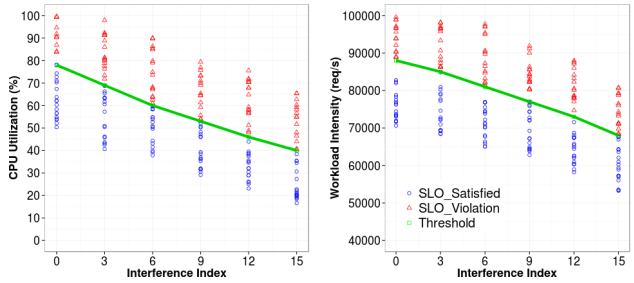


Fig. 6: Threshold values for CPU utilization and Workload intensity based scaling approaches when SLO latency is set to 1.8 ms. Interference index is used as one of the input to determine the right threshold.

and SPEC CPU benchmarks are run in different combinations to generate interference. We use HAproxy for load balancing the request. In all our experiments we disable Dynamic Voltage Frequency Scaling (DVFS) from the host OS using the Linux CPU-freq subsystem. Our middleware performs fine-grained monitoring by frequently sampling the CPU utilization and the different performance counters for all the VMs on the host and repeatedly updates the interference index every 10 secs.

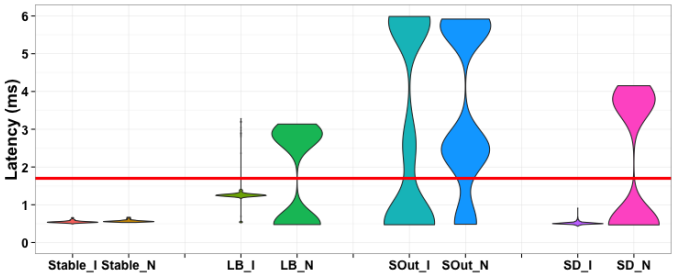


Fig. 7: Violin plot to demonstrate the spread of values satisfying and violating SLO. The horizontal line represents the SLO. In all the experiment groups, suffix I represents interference aware approach and suffix N represents normal approach that is oblivious to interference. The experiment stages are: Stable represents experiment without any interference. LB represents experiment to demonstrate load balancer reconfiguration. SOut represents experiment to demonstrate scaling-out convergence and SD represents experiment to demonstrate scaling down. In all stages, Interference aware approach suffers lesser SLO violations compared to normal approach.

B. Results

We design our experiments to highlight the inefficiencies in elastic scaling when performance interference is not taken into account and answer the following questions: i) How much reduction in SLO violations is achieved with augmentation as compared to the standard approach that is unaware of interference? ii) Can augmentation save cost of the hosting services? Specifically, we demonstrate the benefits of load-balancer reconfiguration, and the role of augmentation in minimising SLO violations and saving provisioning costs in a multi-tenant environment. Figure 7 shows the results of actuation with interference aware scaling and normal scaling.

In the first set of experiments categorised by the group stable in figure 7, no interference is generated throughout the experimental run. Both the interference aware approach and normal scaling approach achieve comparable performance. Specifically, interference aware approach does not suffer any significant overhead.

The second set of experiment categorised as LB is carried out to highlight the benefits of Load balancer reconfiguration when compared to a statically configured cluster (i.e. all the nodes in the cluster receive equal number of requests). In this experiment, the cluster receives a constant workload and interference is introduced in one of the servers. Figure 7 shows that LB_I (interference aware approach) reconfigures the load balancer to handle interference and meet the SLO. Normal scaling on the other hand suffers high SLO violation.

The third set of experiment categorised as SOut compares the scaling out behaviour of interference aware approach against a normal approach. This result shows the impact on convergence in scaling. When scaling out, the VM is spawned on a server that is highly interfered. The violations suffered by the interference aware approach is much lesser compared to the normal approach. This can be seen from the spread on data points above and below the SLO. The controller needs time to spawn and prepare data when scaling out; the violations in interference approach manifest from this period. However, it converges much faster and suffers lesser degradation. Normal scaling on the other hand suffer significantly higher violations.

The final set of experiment categorised as SD compares the scaling down behaviour of interference aware approach against a normal approach. This result highlights how informed decision can reduce SLO violations. Initially the cluster has 3 servers running Memcached instances, all of them serving the workload with one server highly interfered and the workload is dropped enough so that a VM can be removed. The normal approach detects the drop and happens to remove a VM on server with high capacity as it is unaware of interference, resulting in SLO violations as the interfered VM gets overwhelmed. Interference aware approach on the other hand makes an informed choice and removes the highly interfered VM, thereby scaling down without suffering SLO violations.

VII. Conclusion

In this paper, we show that an elasticity controller cannot make accurate scaling decisions under the interference imposed by co-runner applications sharing the infrastructure. It becomes imperative to be aware of interference to facilitate accurate scaling decisions. Evaluations with Memcached have shown that, by quantifying interference, an elasticity controller is able to improve the accuracy of decision making using indirect metrics and experience significantly lesser SLO violations under the presence of interference.

Acknowledgment

This work was supported by the Erasmus Mundus Joint Doctorate in Distributed Computing funded by the EACEA of the European Commission under FPA 2012-0030 and by the Spanish government under contract TIN2013-47245-C2-1-R. The authors would also like to thank the reviewers for their constructive comments and suggestions to improve the quality of the paper.

References

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[2] Right Scale. http://www.rightscale.com/.

[3] Simon J. Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 131–140, New York, NY, USA, 2011. ACM.

[4] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 125–134, New York, NY, USA, 2012. ACM.

[5] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: autonomic elasticity manager for cloud-based key-value stores. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 115–116. ACM, 2013.

[6] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, pages 1–10. ACM, 2010.

[7] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[8] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16, Oct 2010.

[9] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. Technical report, 2013.

[10] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 197–207. Springer, 2010.

[11] Miquel Moreto, Francisco J Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. Flexdcp: a qos framework for cmp architectures. *ACM SIGOPS Operating Systems Review*, 43(2):86–96, 2009.

[12] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 25–36. ACM, 2007.

[13] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

[14] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 41(1):77–88, 2013.

[15] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. Stay-away, protecting sensitive applications from performance interference. In *Proceedings of the 15th International Middleware Conference*, pages 301–312. ACM, 2014.

[16] Khalid Alhamazani, Rajiv Ranjan, Karan Mitra, Fethi Rabhi, Prem Prakash Jayaraman, Samee Ullah Khan, Adnene Guabtni, and Vasudha Bhatnagar. An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art. *Computing*, pages 1–21, 2014.

[17] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723. IEEE, 2011.

[18] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic vm for cloud resources provisioning optimization. In Ajith Abraham, Jaime Lloret Mauri, JohnF. Buford, Junichi Suzuki, and SabuM. Thampi, editors, *Advances in Computing and Communications*, volume 190 of *Communications in Computer and Information Science*, pages 431–445. Springer Berlin Heidelberg, 2011.

[19] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[20] Scryer. http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html.

[21] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, July 2011.

[22] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[23] Ying Liu, N. Rameshan, E. Monte, V. Vlassov, and L. Navarro. Prorenata: Proactive and reactive tuning to scale a distributed storage system. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 453–464, May 2015.

[24] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[25] MBW. http://manpages.ubuntu.com/manpages/utopic/man1/mbw.1.html. accessed: April 2015.

[26] Stream Benchmark. http://www.cs.virginia.edu/stream/. accessed: Feb 2015.