Augmenting Elasticity Controllers for Improved Accuracy

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*—Elastic resource provisioning is used to guarantee service level objectives (SLO) at reduced cost in a Cloud platform. However, 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 show that assuming predictable performance of VMs in a multi-tenant environment to scale, will result in long periods of SLO violations. We augment the elasticity controller to be aware of interference and improve the convergence time of scaling without over provisioning. We perform experiments with Memcached and compare our solution against a baseline elasticity controller that is unaware of performance interference. Our results show that augmentation can reduce SLO violations by 65% or more and also save provisioning costs compared to an interference oblivious controller.**

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

I. Introduction

With the rise in web services, application and content have moved from being static to more dynamic. This includes social media and networking services that see massive growth in the amount of community generated content. The shift to dynamic content along with increased traffic puts a strain on the sites providing these services. This has resulted in the evolution of caching systems such as Memcached [1], that runs in-memory and acts as a caching layer to deliver content at low latency. For example, popular content that is frequently accessed can be replicated in these 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 [2], [3] to complex models based on reinforcement learning [4], [5], control modelling [6], [7], [8], and time series analysis [9], [10] to drive elastic provisioning of resources. The major challenge is to decide when to add/remove resources 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 caused by interference. Existing research shows that interference is a frequent occurrence in large scale data centers [11]. Therefore, 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 affects the performance of another due to contention in the use of shared resources such as memory bandwidth, last level cache etc. Prior work indicates that, despite having (arguably the best of) schedulers, the public cloud service, Amazon Web Service, shows significant amount of interference [12], [13], [14]. Existing solutions primarily mitigate performance interference and guarantee performance in either one or a combination of these 3 ways: (i) Hardware partitioning (ii) at the scheduling level or (iii) by dynamic reconfiguration. Hardware partitioning techniques involve partitioning the shared resources to provide isolated access to the VMs [15], [16]. They may not be feasible for the existing systems and require changes to the hardware design. Scheduling mechanisms look at ways to place together those VMs or threads that do not contend for the same shared resource, essentially minimising the impact of contention [17], [18]. This knowledge is typically accrued through static profiling of the applications. Dynamic reconfiguration involves techniques such as throttling best effort applications or live VM migration upon detecting interference [19]. Reconfiguration techniques such as VM migration involves a huge overhead and throttling best effort applications are possible only if they are co-located on the same host. All these solutions look at ways to guarantee performance in a multi-tenant setting by either resorting to 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.

We show that when elasticity controllers are unaware of interference, it either results in long periods of unmet capacity that manifest as Service Level Objective (SLO) violations or results in 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 3 ways: (i) At the ingress point by load balancer reconfiguration (ii) by reducing the convergence time when scaling out and (iii) by taking informed decisions when scaling down/removing instances. We achieve this with the help of hardware performance counters to quantify the intensity of interference on the host. We do not rely just on the counters of the target application as they are insufficient to detect interference. We take into account the behaviour of the co-running VMs to quantify interference. 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. Our main contributions are as follows.

1. We show that the maximum workload any VM can serve, within a SLO constraint, is severely impacted by interference. An immediate consequence of this impact is an increase in the time taken for the scaling out process to converge which results in increased SLO violations. We also show that this resulting period of SLO violation is directly proportional to the time taken to spawn and prepare VMs. Our tests on Amazon Web Service (AWS) show that preparing VMs takes anywhere between 2 mins to 28 mins depending on the size of data to be transferred.

2. We design and develop a solution to augment elasticity controllers to be aware of interference. Our solution quantifies the capacity of a VM based on the interference experienced on the host by modelling the performance of the target application. With this we are able to reduce the impact of interference on SLO violations by reconfiguring the load balancer, reducing the convergence time when scaling out and by removing highly interfered instances from the cluster when scaling down.

3. We perform experiments with Memcached and compare our solution against a baseline elasticity controller that is unaware of performance interference. We find that with augmentation we can reduce SLO violations by 65% and also save provisioning costs compared to an interference oblivious controller.

II. 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. 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 and satisfying the additional capacity. Although modelling techniques [20], [21], [7], [8] 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. We demonstrate the consequences of this problem through a representative example shown in Figure 1.

Convergence of Scaling: Convergence time of scaling is the time it takes an elasticity controller to reach a stable desirable state. Figure 1 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 for additional capacity to serve the increased workload. It then 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.

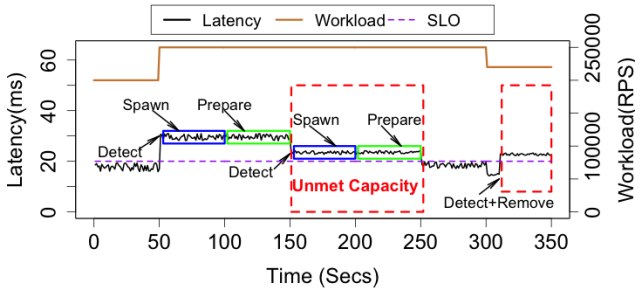


Fig. 1: Memcached service experiencing a long period of SLO violation during periods of scaling out and scaling down because of unmet capacity from performance interference.

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. More details on how interference affects capacity is explained in section III-A. 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. It 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 1), 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 which 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.

III. Experimental Analysis

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 [22] and benchmarks such as mbw [23] and Stream [24]. The experiment to study the time taken to prepare a VM is carried out on AWS, using 2 large instances. We choose to do the experiments of interference on our private testbed because of the difficulty in co-locating VM instances on the same host in AWS.

A. Interference reduces VM Capacity

In this experiment, we generate different amounts of interference on the memory subsystem when Memcached is serving workload. We set our SLO at 1.8ms and workout the maximum workload that can be handled just within this SLO, for different amounts of interference. The capacity of a VM, i.e., the maximum amount of workload it can handle without violating the SLO is inversely proportional to the amount of performance interference in the host. Figure 2 shows this behaviour. 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 will always spawn VMs assuming a much higher capacity than achievable (in the presence of interference) leading 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.

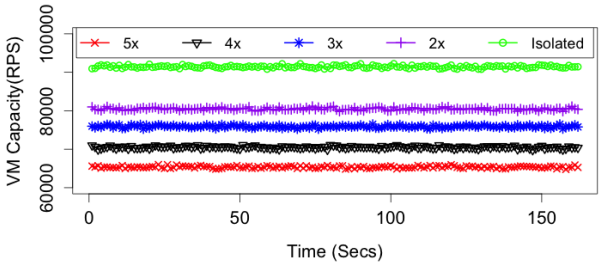


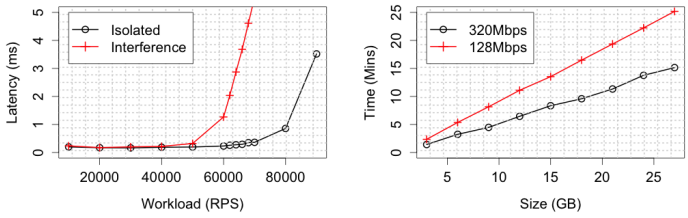
Fig. 2: VM capacity reduces with increasing performance interference. In this example the SLO is set to 1.8 ms.

B. Interference vs. Load

In this experiment, we run a standalone instance of Memcached at different loads for a fixed amount of interference. Interference has a significant impact only at high loads. Figure 3a shows the results from the experiment. Each data point in the figure is an average of a 10 minute run. The impact of interference is negligible until 55000 requests per second and they overlap with each other but as the load increases ($>$ 55000) latency begins to rise sharply in the presence of interference. This observation implies that the impact of interference can be mitigated to a large extent by reducing the workload served on impacted VMs. For example, the load balancer can be configured with weights corresponding to the interference to minimize the impact of interference.

C. Preparation Time

Recall from the previous section that the period of unmet capacity directly depends on the time taken to spawn and prepare a VM instance. We found that the time taken to spawn a VM on AWS takes between 1 to 2 minutes. In this experiment, we evaluate if the time taken to prepare the instance with necessary data is long enough to be a significant problem. Our results from AWS are shown in figure 3b. For a caching layer, data loading is done either by the backend



(a) (b)

Fig. 3: (a) Impact of interference for different load on Memcached. Interference impacts performance only at higher loads. (b) Time taken to load Memcached data on a AWS large instance for different data sizes.

server or by any of the other caching instance. This process of data migration by itself affects the performance of the cluster. We loaded data at 2 different data rates, one at full capacity of the data generator and another at a throttled rate. We find that the time taken to load is significant enough and is in the order of minutes. It takes anywhere between 2 to 28 minutes which is significant enough to be considered a problem when met with SLO violations for that period. In the next section we present the design and implementation of the approach to augment elasticity controllers for improved accuracy.

IV. Solution Overview

Our solution primarily consists of 2 components: i) An Augmentation Engine (AE) and ii) a Middleware Interface to quantify capacity (MI). 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 Augmentation Engine orchestrates with the MI to make scaling decisions.

First the AE receives the decision from the elasticity controller (EC) to scale out/scale down. It also receives the number of instances to add/remove respectively. In essence, the AE acts as a delegator to act on the decisions of the EC. Before acting on the input, the AE consults the MI on all hosts to quantify the capacity of each host. AE takes a decision by itself only when it has enough historical evidence to believe that the interference is sustained. Based on this knowledge of host capacity, along with the current workload, AE 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, then the decision from the EC is overruled and the AE reconfigures the load balancer. This is the first level of reconfiguration. However, such a plan may not always be possible. If the AE learns that a rebalancing solution alone cannot maintain the SLO, it then directly acts on the decision of the EC. 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. If the decision is to add new instances, the AE adds only so many instances as directed by the EC. 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 AE 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 AE 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.

The results from augmenting the elasticity controller in figure 1 is shown in figure 4. 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}).

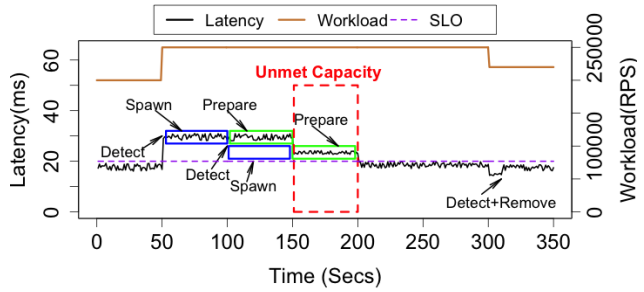


Fig. 4: Unmet capacity from figure 1 pruned by augmenting the elasticity controller to detect and quantify interference.

V. Middleware Interface to Quantify Capacity

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 target application from interference and to translate this performance degradation to capacity. We say that the storage system *co-runs* with other applications when they all run on different cores of the same physical host; we refer to all these applications as *co-runners*.

Contentiousness vs. Sensitivity: Contention can be seen as the amount of pressure an application puts on different shared resources such as memory-bandwidth or the cache. Each application may have a different amount of cache and memory-bandwidth usage and this determines the contentiousness of the application. However, sensitivity to contention depends on the application's reliance on the shared memory subsystem and how much an application progress benefits from this reliance. In our case, we are interested in the contentiousness of the co-runners and the sensitivity of the target system. Prior works [25], [26], [27], [28] use the target systems last level cache (LLC) miss rate/ratio as an indicator to detect contention and classify application for contention aware scheduling. While LLC miss rate/ratio can be a good indicator of contention, it suffers from the following limitation: An application can have varying run-time behaviour and depending on the application access patterns, LLC misses can vary over time making it difficult to attribute if contention or application behaviour is the cause for LLC misses. While it may be possible to detect contention in some cases based on LLC misses, it still cannot quantify the amount of degradation.

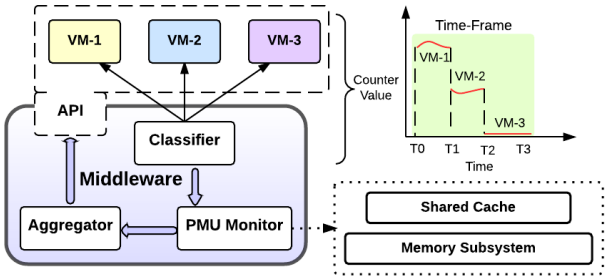


Fig. 5: Architecture of the Middleware Interface (MI)

PMU based approximation: For the reasons mentioned above, instead of relying on the behaviour of the target system alone, we take into account the co-runners behaviour for quantifying the contention. We use performance monitoring units (PMU) to approximate this behaviour. 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.

Middleware: Figure 5 shows the architecture of the middleware to quantify contention on the memory subsystem. The middleware provides an API that can be queried to access information about the contention from the co-runners. The different components of the middleware provide the following function: The classifier is responsible for identifying the VMs that need to be monitored for contention. It optimises the number of co-runners to be monitored. The role of the classifier is to only select those VMs that are potential candidates for creating contention at the memory subsystem. This minimizes the overhead of unnecessarily analysing VMs that are idle or not memory-subsystem intensive. The classifier maintains a moving window of the average CPU utilization of different VMs and selects only those VMs that consistently have a CPU utilization over a certain threshold. This is because any application that is intensive on the memory-subsystem has a high CPU utilization. In our experiments we set our threshold to 30%. The list of selected VMs are then passed on to the PMU Monitor. The PMU Monitor monitors the performance counters of the VM using the "burst-approach" as explained in the next paragraph. The measured counter values are then passed on to the aggregator that calculates a metric called the interference-index (explained in section V-A) . Interference-index approximates the performance drop from interference suffered by the target system. The aggregator subsequently

makes them available for the API along with the monitored counter values to allow for a user specific composition for quantifying contention. By exposing different counter metrics through the API, it also allows the users to compose their own index of pressure for any subsystem.

Burst-Approach: Typically performance counters are saved/restored when a context switch changes to a different process, which costs a couple of microseconds. Since these counters can hold context for only a single process at a time, monitoring the behaviour of the co-runners during runtime requires the middleware to adapt to this limitation. We circumvent this limitation by using a "burst-approach" where different co-runners are monitored in bursts and their values composed together within a single time-frame. A time-frame is defined as the period during which an application is assumed to have minimal variations in its behaviour. Consider, for example 2 VMs co-located on a physical host. In order to monitor their behaviour, the middleware collects the counters of each VM one after another in cycles. The time chosen to measure the counters of all the VMs exactly once defines a time-frame. Figure 5 shows one time-frame of execution. In our experiments, we monitor the counters of each VM for a period of 5 seconds.

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.

In our experiments, we compute the performance drop as follows: First, we measure the average latency L_i of the storage system when running in isolation. Then we measure the average latency L_c of the storage system when it co-runs with other processes. Performance drop suffered is $\frac{L_c - L_i}{L_i}$.

Sources of degradation: There are 2 main subsystems responsible for contention: the cache and the memory bandwidth. In order to assess the impact of contention on these subsystems, we use different system configurations that are designed to generate contention at different resources: the first configuration generates contention only on the cache, the second only on the memory bandwidth and the third one on both. Figure 6 shows the drop in performance experienced by Memcached for all the three configurations. It is clear that cache is the dominant source of performance degradation, contributing upto 15X drop in performance and bandwidth contributing less than 1X. The results show that Memcached benefits more from it's reliance on the cache than from memory bandwidth. What is also interesting to note is that the drop in latency increases linearly for the same co-runner. We only plot the characterisation of Memcached 60000 RPS and up, since, Memcached remains unaffected by interference below that workload (see figure 3a).

Properties that determine degradation: We investigate properties of the co-running application that cause performance degradation. In figure 6, all the different co-runners cause degradation in the same order; ie. lbm consistently causes the highest amount of performance degradation, followed by mbw, stream, and povray. In order to understand the properties that define the aggressiveness of the co-runners, we rely on PMUs. Since cache is the dominant source of degradation, 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. However, this alone is insufficient to model the sensitivity of Memcached. Upon deeper analysis, we find that the order of co-runners that cause performance drop (lbm>mbw=stream>libquantum>povray) does not correspond with their cache access rate. This is because cache access rate alone does not take into account the memory access patterns of the co-runners. 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-reference counter for Memcached. From our experiments, we find that these counters provide a good model to quantify sensitivity of Memcached. The final list of performance counters chosen to quantify sensitivity are shown in table I.

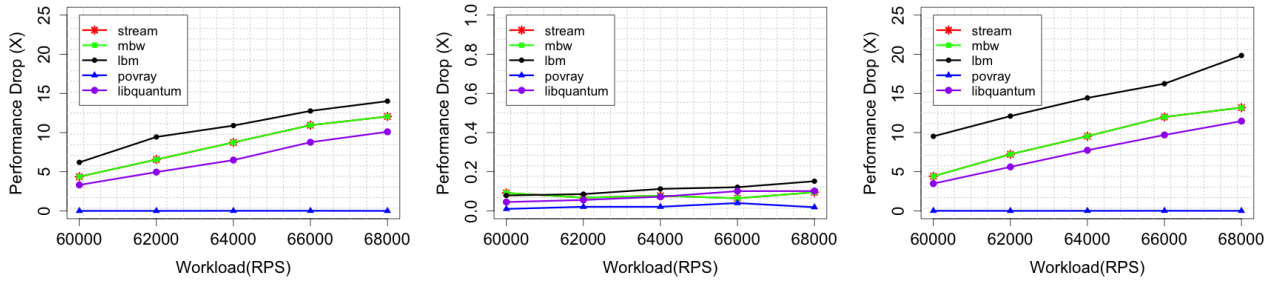
Interference-Index: The goal of characterising contention is to quantify the properties of the co-runners that lead to performance degradation of the storage system. We call this metric interference-index and it approximates the performance degradation suffered by the storage system. In order to be useful, the metric must correlate with the performance drop suffered by the storage system.

Name	Description	Name	Description
cpu-clk	Reference cycles	inst-retired	Instructions retired
cache-ref	References to L3 cache	cache-miss	L3 cache misses
llc-prefetch	L3 prefetches	llc-prefetch-miss	L3 prefetch misses

TABLE I: Performance counters included in characterising contention

We derive a set of N representative performance counters (shown in table I) $WS = m_1, m_2, ..., m_N$ where m_i represents the metric i. Using these counters 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 on these counters to construct the interference index. Figure 7 shows the interference-index constructed for Memcached. Since the modeling is data-driven, the interference index generated is application-dependent. We however do not view this as an issue since modeling can be fully automated. From figure 7, we see 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 stream, milc and omnetpp 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. Note that because of the linear trend in performance drop above 60000 RPS (figure 6), knowing a workload and interference index, it is possible to estimate the drop in latency for any other workload.

Capacity: We apply a binary classifier used in the state of the art approaches [20], [21] to approximate the capacity of a VM, which defines its ability to handle workloads under



(a) Contention for L3 cache. (b) Contention for memory bandwidth. (c) Contention for both resources.

Fig. 6: The drop in performance of Memcached for contention at different levels of memory subsystem. Memcached is run alongside multiple instances of different co-runners.

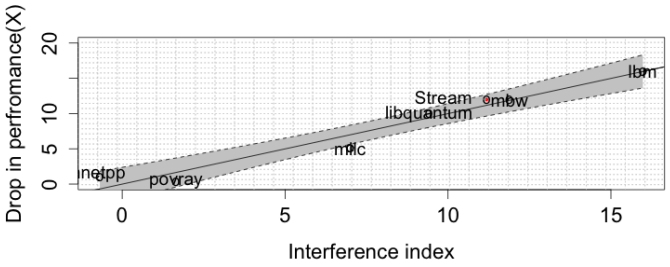


Fig. 7: Interference-index quantifies the performance drop suffered by Memcached based on the behaviour of the co-runner.

the SLO. In state of the art approaches, models are built by finding correlations between Monitored Parameters (MP), which reflects the operating state of the system, and the Target Parameter (TP), in which the SLO is defined. The MPs are chosen by analyzing whether they have remarkable influences on the SLO. Previous works adopt CPU utilization, workload intensity and workload composition, including read/write ratio, data size etc., as typical MPs. 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, which outputs satisfying (true) or not satisfying (false) the SLO for a given workload intensity and interference index. Then, we are able to obtain the maximum supported workload intensity under the current MPs while satisfying the SLO. The maximum supported workload intensity defines the capacity of a VM.

The model used in our paper is trained offline using the training data collected when running Memcached under different intensities of the interference and workload. Various kinds of workload, including varying compositions and RPS, are generated for Memcached. Multiple intensities of interference and workload provide sufficient variance in the interference index, which covers most of the operating space of the model. With little modifications, the model can also be adapted and automated to get trained online.

Overhead: Our middleware has a very minimal overhead since it only samples the counters every few seconds. It has a negligible CPU consumption of less than 3% and does not perform any instrumentation to the application that result in performance loss. Augmentation incurs very negligible overhead since the only additional step involved is the construction of interference index, which in itself relies on non-intrusive monitoring.

VI. Augmentation Engine

The AE operates with the estimation of the capacities of VMs using an empirical model explained in Section V. The capacity of a VM defines the ability of the VM to serve workload, in terms of requests per second, under the current workload composition. Without knowing the interference index of a VM, the capacity estimation is the most optimistic and is denoted by $\underline{C}$, which is a function of the current workload composition and the VM flavor ($\underline{C} = f(W_t, F)$). Taking into account the interference index, the capacity of a VM is represented by C, which is a function of the current workload composition, the VM flavor and the interference index ($C = g(W_t, F, I_t)$). Let L_t denote the total amount of workload received by the system at time t and each procedure starts with n VMs in the cluster providing $\sum_{i=1}^{n} C_i$ capacity. The procedures during load balancer reconfiguration, scaling out and scaling down are described in Algorithm 1

Load Balancer Reconfiguration: The load balancer reconfiguration procedure is triggered when the AE receives a scaling out decision from the EC or after scaling out/down. Assuming the capacities of each VM in the current cluster setup are $C_1, C_2 \ldots C_n$ and the total capacity provided is $\sum_{i=1}^{n} C_i$. When $\sum_{i=1}^{n} C_i \succeq L_t$, the cluster is able to operate under the SLO using load balancer reconfiguration. Then, the AE reset weights $(W_1, W_2 \ldots W_n)$, that are proportional to the capacity of each VM $(C_1, C_2 \ldots C_n)$, in the load balancer. Otherwise, the AE continues with the scaling out approach.

Scaling out: During scaling out procedure, the AE spawns the number of VMs indicated by the EC. After the VMs are spawned, their interference index are assessed and their capacities $\Delta C_1, \Delta C_2 \ldots \Delta C_m$ under the current workload are estimated. In the meantime, these VMs start to prepare. Whether the current setup with proper load balancing is able to satisfy the current workload L_t under the SLO is evaluated by $\sum_{i=1}^{n} C_i + \sum_{j=1}^{m} \Delta C_j \succeq L_t$. If the above inequality holds, proper weights of both existing and additional VMs are reset in the load balancer and the scaling out procedure exits. Otherwise, the unmet capacity $C_{unmet} = L_t - (\sum_{i=1}^{n} C_i + \sum_{j=1}^{m} \Delta C_j)$ needs to be handled by spawning another batch of VMs, whose capacities are optimistically estimated $(\underline{C_1}, \underline{C_2} \ldots \underline{C_k})$, where

$\sum_{i=1}^{k-1} \underline{C_i} \preceq C_{unmet} \preceq \sum_{i=1}^{k} \underline{C_i}$. Then, the AE spawns the corresponding k VMs like executing scaling out commands from the EC. This procedure iterates until $C_{unmet} \preceq 0$.

Scaling down: When the EC issues a scaling down decision, the AE overrules the amount of VMs to be removed issued by the EC and judiciously removes the most interfered VMs. This is because the most interfered VM serves the least amount of workload with in the SLO for the same price as a non-interfered VM. The number of VMs to be removed by the AE satisfies $\sum_{i=1}^{m} \Delta C_i \preceq C_{extra} \preceq \sum_{i=1}^{m+1} \Delta C_i$, where $C_{extra} = \sum_{i=1}^{n} C_i - L_t$. By selecting and removing the highly interfered VMs, the AE usually removes more VMs than the amount issued by the EC, which saves the provisioning cost.

Algorithm 1 Augmentation Engine

```
1:  procedure LoadBalancerReconfiguration()
2:      if ∑_{i=1}^{n} C_i ⪰ L_t then
3:                          ▷ Calculate capacity of VMs C_1, C_2 ... C_n
4:          with C_i = g(W_t, F, I_t)
5:                                  ▷ Reset weights W_1, W_2 ... W_n
6:          with W_i = L_t / (∑_{j=1}^{n} C_j) * C_i
7:      else
8:          ScalingOut()
9:  procedure ScalingOut(m)
10:     Spawn m VMs as indicated by the EC
11:     Prepare these m VMs immediately and concurrently
12:                          ▷ Calculate extra capacity ∑_{j=1}^{m} ΔC_j
13:     with C_i = g(W_t, F, I_t)
14:                                  ▷ Calculate unmet capacity
15:     with C_unmet = L_t − (∑_{i=1}^{n} C_i + ∑_{j=1}^{m} ΔC_j)
16:     while C_unmet ⪰ 0 do
17:         Spawn extra k VMs
18:         with ∑_{i=1}^{k-1} C_i ⪯ C_unmet ⪯ ∑_{i=1}^{k} C_i
19:         Prepare these k VMs immediately and concurrently
20:                                  ▷ Update unmet capacity
21:         with C_unmet = C_unmet − ∑_{p=1}^{k} ΔC_p
22:     LoadBalancerReconfiguration()
23: procedure ScalingDown(m)
24:                                  ▷ Calculate extra capacity
25:     with C_extra = ∑_{i=1}^{n} C_i − L_t
26:                                  ▷ identify VMs to remove
27:     with ∑_{i=1}^{m} ΔC_i ⪯ C_extra ⪯ ∑_{i=1}^{m+1} ΔC_i
28:     where these m VMs are highly interfered
29:     Remove VMs
30:     LoadBalancerReconfiguration()
```

VII. Experimental Evaluation

A. Assumptions

In this work we focus on Memcached for elastic scaling. In [29], Nishtala et al. show how they scale Memcached at Facebook and maintain consistency across regions. Our approach is complementary to this work and enables elastic scaling under SLO constraints in a multi-tenant environment. Cache invalidation and consistency is orthogonal to this work and the storage infrastructure is expected to manage the same. We particularly focus on replication and the backend server is responsible for populating and preparing the VMs. As a result the Memcached instances already present in the cluster don't incur any additional overhead of migrating data.

B. Experiment Setup

Our experiment setup is the same as in section III. 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 co-located instances. MBW, Stream 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 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.

Since the design of elasticity controller is not the focus of the paper, we build a simplified version of an elasticity controller similar to [8]. The elasticity controller relies on sampled request latency instead of CPU utilization on each VM. Given our evaluation scenarios, we ensure that the elasticity controller is able to make ideal scaling decisions when interference is not present.

C. 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 standard approach without any augmentation? 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.

Server Timeline: Each experiment has a server timeline that is depicted to explain the following status of the servers (i) Whether a Memcached instance is running on the server. White indicates that there is no instance on the server. (ii) Blue indicates that the instance on the server is preparing the data and (iii) Gradient of red indicates the amount of interference on the server. Dark red indicates high interference. The gradient of interference intensity is plotted according to the interference index computed on each server. S1, S2 and S3 indicate the different servers used in the experiment. The server timeline graph is presented for both approaches; scaling with augmentation, and without augmentation, which we call the standard/baseline approach. The server timeline essentially aids in understanding and reasoning about the choices made from augmentation and helps compare the cost involved.

1) Load Balancer Reconfiguration: Our first experiment is focused at showing the advantages 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 around 120 secs in one of the servers. The results of the experiment is shown in Figure 8. Latency vs. time shows the comparison of latency when the cluster is augmented to reconfigure the load balancer against the standard approach (baseline) with no reconfiguration. The standard approach without augmentation experiences long periods of SLO violation. The same figure also depicts a server

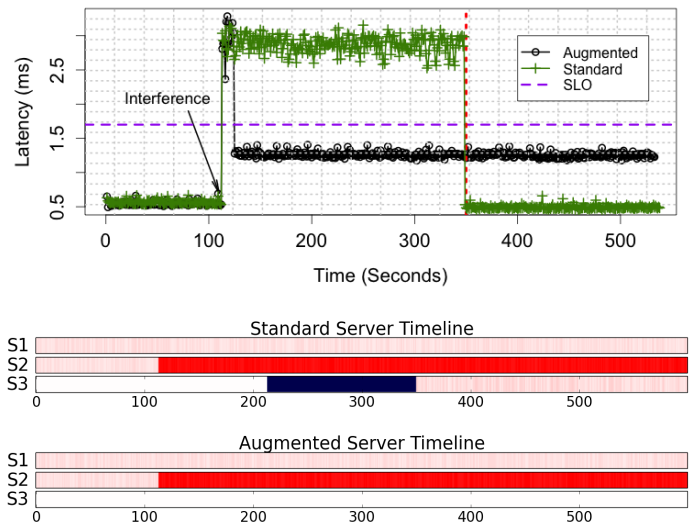


Fig. 8: Results demonstrating load balancer reconfiguration by AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference

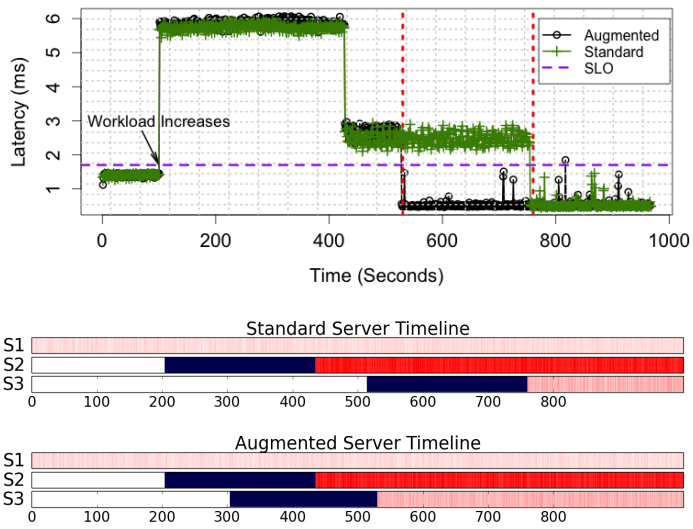


Fig. 9: Results demonstrating convergence time with AE and without AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference

timeline of all the servers (shown as S1,S2 and S3) to visualize the server status.

Looking at the standard server timeline, we see that Memcached instances are running only on S1 and S2 initially and they are able to serve the workload without violating the SLO until 120 seconds. The latency remains far below the SLO before 120 seconds. We note that the servers are not over provisioned. The latency stays far below the SLO because of the granularity of VM specification. The workload generated at these 2 machines cannot be served within the SLO by one server alone and having a additional server drops the latency far below the SLO. At 120 seconds, S2 experiences a high amount of interference that causes a spike in latency resulting in SLO violations. As soon as sustained SLO violations are detected, the standard approach spawns a new instance on S3 (not shown in the server timeline, as nothing resides on S3 to show spawning) and just after 200 seconds begins to prepare the instance on S3. This instance cannot serve any workload during the preparation phase and the SLO continues to remain violated. The preparation phase finishes around 350 secs and the Memcached instance on S3 is up and running and begins to serve the workload thereby maintaining the SLO.

The augmented approach behaves differently when compared to the standard approach. After around 120 seconds, when the augmentation engine (AE) detects sustained SLO violations, it first tries to generate a plan for load balancer reconfiguration. AE learns that there is excess capacity on S1 and reroutes the traffic to reduce the load on S2 which is highly interfered. We found that AE routed 60% of the workload to S2 and 40% to S1. This reconfiguration was enough to maintain the SLO. From the augmented server timeline in figure 8, we see that no Memcached instance was instantiated on S3. With augmentation, the AE was able to avoid instantiating a new instance, while minimising SLO violations without any additional cost.

2) Convergence when scaling out: Next, we highlight the problem of unmet capacity as a result of delayed convergence. Our experiment shows how augmentation converges quicker in the presence of performance interference while scaling out. In this experiment the total workload served to the cluster is increased so that it has to scale out to maintain the SLO. Figure 9 shows the results from the experiment. The workload increases at around 100 seconds and the SLO is violated. The elasticity controller decides to spawn a new VM to handle the increased workload. The standard server timeline in the same figure shows the status of the server. Only S1 was serving the requests initially, but when the workload increases, a new instance is spawned and prepared on S2. S2 is very highly interfered and although the latency dropped, it is still unable to meet the workload demands within the SLO. The elasticity controller learns this only after the instance is prepared and begins to serve the requests. Another instance is then spawned and prepared on S3 and the SLO is maintained.

Augmented approach functions similar to the standard approach up until the point of spawning S2. This can be seen from the augmented server timeline in figure 9. But as soon as S2 is spawned, AE learns that S2 is very highly interfered and does not have enough capacity to serve the workload within the SLO. In this particular scenario, even a load balancer reconfiguration does not help. So the AE immediately spawns and prepares another instance on S3 in parallel with the preparation phase on S2. By preparing the new instance in parallel, we minimise SLO violations proportional to the preparation time. In this experiment, the preparation time was a little over 3 minutes and SLO violations are reduced by 65%.

3) Scaling Down: In this experiment, we highlight how informed decision when scaling down can lower cost and also reduce SLO violations. Initially the cluster has 3 servers running Memcached instances, all of them serving the workload. Around 100 seconds later, the workload drops so much that a VM can be removed. In the standard approach without augmentation, the elasticity controller precisely does that when it sees sustained drop in latency and removes a VM. The results are shown in figure 10. From the standard server timeline in the same figure we see that the VM on S3 was removed. Since the elasticity controller has no information about the interference on different servers, it randomly chose a server to remove the VM. However, this decision ends up costing more in terms of SLO violations. This is because, the VM on S3 had a high capacity and also a lot of spare capacity. By removing the VM instance on S3, S2 and S1 have to serve a higher number of requests. Ideally, this should not have caused any SLO violations if all the VMs had the same capacity. But since S2 is highly interfered, it was already serving out on its max capacity and by removing S3, S2 didn't have enough spare capacity to serve the additional workload, resulting in SLO violations. Note that, in the standard approach the cluster is statically configured and all the servers receive equal number of requests. When the controller notices sustained SLO violations, it reinstantiates S3 and the SLO conditions are met.

By augmenting the elasticity controller, the AE knows the capacity of each VM and upon receiving a decision from the controller to remove a VM, it removes the VM with the least capacity, VM on S2 in this example. Since S2 was not contributing enough to serving the workload, removing S2 doesn't impact the cluster load significantly and the SLO is maintained. This can be seen in the augmented server timeline in figure 10. This experiment demonstrates that augmentation can help make informed decisions when scaling down and remove VM instances with smaller capacity, thereby reducing the cost.

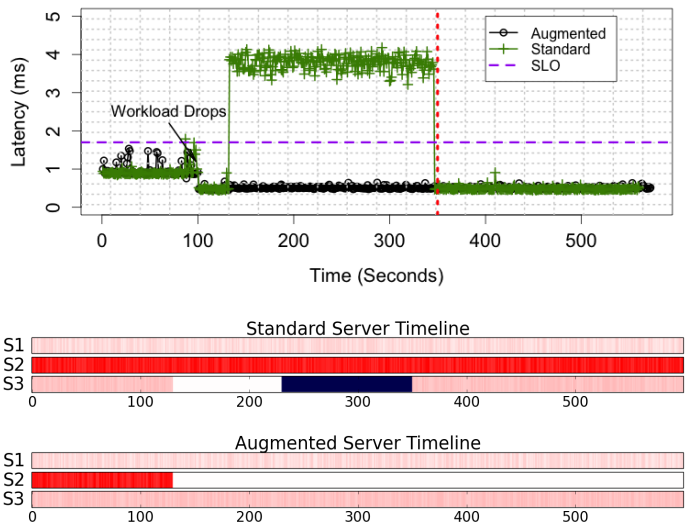


Fig. 10: Results demonstrating informed scaling down with AE. Server timeline shows the status of the server over time. White indicates that there is no instance on the server. Blue indicates that the instance on the server is preparing the data. Gradient of red indicates the amount of interference on the server. Dark red indicates high interference

VIII. Related Work

Performance Interference: DejaVu [30] relies on an online-clustering algorithm to adapt to load variations by comparing the performance of a production VM and a replica of it that runs in a sand-box to detect interference and learns from previous allocations the number of machines for scaling. We model contention from the behaviour of the co-runners. Our solution instead shows ways to quantify the interference-index and how this can be used to perform reliable elastic scaling. A similar system, DeepDive [31], first relies on a warning system running in the VMM to conduct early interference analysis. When the system suspects that one or more VMs are subjected to interference, it clones the VM on-demand and executes it in a sandboxed environment to detect interference. If interference does exist, the most aggressive VM is migrated on to another physical machine. Both these approaches require a sand boxed environment to detect interference.

Another class of work has also investigated providing QoS management for different applications on multicore [15], [16]. While demonstrating promising results, resource partitioning typically requires changes to the hardware design, which is not feasible for existing systems. Recent efforts [32], [33] demonstrate that it is possible to accurately predict the degradation caused by interference by prior analysis of workload. Paragon [18] profiles only a part of the application and relies on a recommendation system, based on the knowledge of previous execution, to identify the best placement for applications with respect to interference. Since only a part of the application is profiled, dynamic behaviours such as phase changes and workload changes are not captured and can lead to a suboptimal schedule resulting in performance degradation. ICE [28] is the closest work to our approach and mitigate the effects of interference by dynamically load balancing the requests away from highly interfered nodes. Our work, in contrast, considers elastic scaling and also reduces the convergence time when scaling out.

Elastic Scaling: Amazon Auto Scaling [2] is an existing production cloud system which depends on the user to define thresholds for scaling up/down resources. However, it is difficult for the user to know the right scaling conditions. Rightscale [3] is an industrial elastic scaling mechanism and uses load-based threshold to automatically trigger creation of new virtual instances. It uses an additive-increase controller and can take a long time to converge and know the requisite amount of machines for handling the increasing load.

Reinforcement learning are usually used to understand the application behaviors by building empirical models either online or offline. Simon [4] presents an elasticity controller that integrates several empirical models and switches among them to obtain better performance predictions. The elasticity controller built in [5] uses analytical modeling and machine-learning. They argued that by combining both approaches, it results in better controller accuracy. Although reinforcement-learning mechanisms converge to an optimal policy after a relatively long time, it reward mechanisms cannot adapt to rapidly changing interference as it is unaware of the amount of contention in the system.

Control theory aims to define either a proactive or a reactive controller to automatically adjust the resources based

on application demands. Previous works [6], [7], [8] have extensively studied applying control theory to achieve fine grained resource allocations that conform to a given SLO. However, the existing approaches are unaware of interference and will consequently fail to meet the SLO.

In time series based approach, a given performance metric is sampled periodically at fixed intervals and analysed to make future predictions. Typically these techniques are employed for workload or resource usage prediction and is used to derive a suitable scaling action plan. Gmach et al.[34] used a Fourier transform-based scheme to perform offline extraction of long-term cyclic workload patterns. PRESS [10] and CloudScale [9] perform long-term cyclic pattern extraction and resource demand prediction to scale up. Although these approaches account for performance interference inherently, they are known to perform well only when periodic patterns exist, which is not always true in a dynamic environment such as Cloud.

IX. 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. We design and implement a middleware that augment the decisions made by elasticity controllers. Evaluations have shown that, with our middleware, an elasticity controller is able to experience significantly less SLO violations and provisioning cost 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.

References

[1] Memcached. http://memcached.org/. accessed: April 2015.

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

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

[4] 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.

[5] 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.

[6] X Zhu, D Young, BJ Watson, Z Wang, J Rolia, S Singhal, B McKee, C Hyser, D Gmach, R Gardner, et al. Integrated capacity and workload management for the next generation data center. In *ICAC08: Proceedings of the 5th International Conference on Autonomic Computing*, 2008.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi 2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.

[12] Amiya K Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference*, pages 277–288. ACM, 2014.

[13] Christina Delimitrou and Christos Kozyrakis. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

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

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

[25] Ravi Iyer, Ramesh Illikkal, Omesh Tickoo, Li Zhao, Padma Apparao, and Don Newell. Vm 3: Measuring, modeling and managing vm shared resources. *Computer Networks*, 53(17):2873–2887, 2009.

[26] Richard West, Puneet Zaroo, Carl A Waldspurger, and Xiao Zhang. Online cache modeling for commodity multicore processors. *ACM SIGOPS Operating Systems Review*, 44(4):19–29, 2010.

[27] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Communications of the ACM*, 53(2):49–57, 2010.

[28] Amiya K Maji, Subrata Mitra, and Saurabh Bagchi. Ice: An integrated configuration engine for interference mitigation in cloud services. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pages 91–100. IEEE, 2015.

[29] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[30] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. *ACM SIGARCH Computer Architecture News*, 40(1):423–436, 2012.

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

[32] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 11–11. USENIX Association, 2012.

[33] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 129–142. ACM, 2010.

[34] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Capacity management and demand prediction for next generation data centers. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 43–50. IEEE, 2007.