2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-Tenancy

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 objective (SLO) with 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 to build an elasticity controller will fail if interference is not modelled. We identify and control the different sources of unpredictability and build Hubbub-Scale; an elasticity controller that is reliable in the presence of performance interference. Our evaluation with Redis and Memcached show that Hubbub-Scale efficiently conforms to the SLO requirements under scenarios where standard modelling approaches fail.**

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

I. Introduction

Services that are elastically provisioned in the Cloud are able to use platform resources on demand. Instances can be spawned to meet the Service Level Objective (SLO) during periods of increasing workload and removed when workload drops. 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. Virtualization is a key enabler for elasticity as it ensures operational isolation for Cloud users and provides management convenience for Cloud providers. However, it does not provide performance isolation on many shared resources, such as memory subsystem. In other words, consolidation of multiple VMs comes at the price of application slow down and VM performance interference in ways that cannot be modelled easily. VM performance interference happens when behavior of one VM adversely affects the performance of another due to contention in the use of shared resources in the system such as memory bandwidth, cache etc. Performance interference is well studied and works [1], [2] show that it is indeed a real problem in the cloud and can degrade the performance of an application significantly.

The already hard problem of building a general-purpose elasticity controller that guarantees SLO with adequate resource provisioning becomes exacerbated by the role of performance interference. Previous works have proposed multiple models ranging from simple threshold based scaling [3], [4] to complex models based on reinforcement learning [5], [6], control modelling [7], [8], [9], [10], and time series analysis [11], [12] to drive the scaling decisions. While every model comes with its host of benefits and demerits, the impact of performance interference on elastic scaling is often overlooked.

CPU utilization [13], [12], [11], [9], [14] and workload intensity [15], [8], [16], [17], [18] are two widely used indirect metrics for elastic scaling since they are easily available and correlate well with the measure of service quality such as latency. In this paper, we investigate if the performance interference from consolidation has a role to play on the metrics used for making scaling decisions. We find that it becomes imperative to quantify the contention in the system in order to achieve accurate scaling.

Our main contribution is Hubbub-Scale; an elasticity controller that achieves predictable performance in the face of resource contention without any significant overhead. We facilitate this by designing a middleware that provides an API to quantify the amount of pressure the co-running VMs put on the target system. Specifically our contributions are:

- We show that OS configuration, performance interference and power-saving optimisations stand in the way of predictable performance. While OS configuration and power-saving optimisations can be controlled, performance interference is inevitable in a multi-tenant system and needs to be modelled.
- In the presence of performance interference, indirect metrics used for elastic scaling cease to accurately reflect the measure of service quality, consequently affecting the scaling accuracy.
- In the presence of performance interference, even relying on direct metrics like latency to scale can lead to SLO violations.
- We build Hubbub-scale, an elasticity controller that is reliable in the presence of performance interference and achieves high resource utilization without violating the SLO.

II. Elastic Scaling

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

978-1-5090-2453-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCGrid.2016.71

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 such as request 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 to a workload metric change 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.

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 [19]. 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 [20]. 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.

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 involves 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 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 [13], [12], [11], [9], [14]. It can be obtained from the operating system or the virtual machine without instrumenting application code. CPU utilization is also a more stable signal than metrics like response time and correlates well with the measure of service quality such as latency/throughput [9]. Another widely used metric for elastic scaling is workload in terms of Requests-per-second (RPS) [15], [8], [16], [17], [18]. RPS can be an important way to measure system performance and is mostly used for proactive control. Netflix developed a system called scryer [15] 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. Motivation

There are 2 key aspects in elastic scaling that determine the effectiveness of the scaling model: when to add or remove instances (decision making phase), and, how many instances to add/remove (scaling phase). Any lapse in the accuracy of these two steps translates to SLO violations or increased cost from over-provisioning of resources. For example, a delay in adapting to an increasing load will result in SLO violations, and erroneously adding more instances than required will result in under utilized instances. All scaling models aim to minimise SLO violations and improve the resource utilization. However, existing scaling models fail to guarantee these properties in a multi-tenant setting. We explain this from two perspectives:

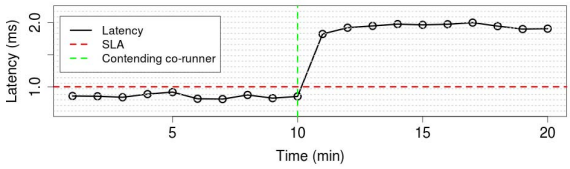
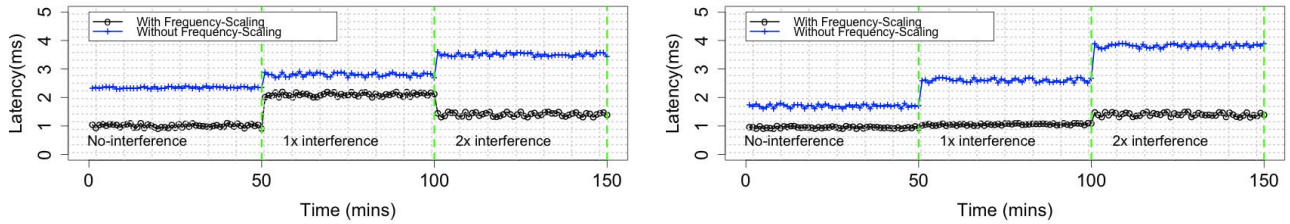


Fig. 1: Variation of latency over time for a constant workload (RPS). Until 10 mins, the application runs in isolation. After 10 mins, other applications are executed on the physical host, generating contention at the shared system resources.

Using an indirect metric to scale: For ease of explanation, consider a simple control model that reacts to indirect system metrics to scale an application. The model learns the relationship between the system metrics (workload intensity (RPS), CPU utilization) and latency during a characterization phase, and decides when to add/remove instances and how many instances to add or remove in order to conform to its SLO. Figure 1 shows the latency of the application for a constant workload (RPS). After 10 mins, the model learnt by the elasticity controller becomes void and violates the SLO in the presence of interference. This is because the system metrics do not accurately reflect the degradation caused by performance interference and the model is unable to discern the need to scale out. In other words, the system metrics correlate differently with latency in the presence of interference. It is important to note that a dynamic learning model will also fail in the presence of interference as these metrics cannot attribute accurately for resource contention. Without quantifying the amount of contention in the system, it is not possible to achieve accurate scaling in a multi-tenant environment.

Using a direct metric to scale: In the presence of interference, even though the effect of degradation from contention is reflected directly on latency, and accurately aids the decision making phase, it can lead to over/under subscription of resources during the scaling phase. Without any additional information about contention, it is not possible to know the exact number of instances needed to scale the system.



(a) Variation of latency when CPU utilisation is maintained between 65 and 70%

(b) Variation of latency when Workload (RPS) is fixed at 40000 RPS

Fig. 2: Frequency-scaling affects performance predictability. Once frequency-scaling is controlled, performance variation from interference becomes predictable as shown by blue line.

For example, first consider an isolated setting where an elastic application receives the workload W_c, handled by X instances, that corresponds to latency L_{SLO}. Without losing generality, we assume a round-robin load balancer. For an increased workload $W_c + n$, the total number of instances needed to ensure that SLO is not violated can be calculated as $\frac{W_c+n}{W_c} \times X$. Next, consider the case when the application is provided in a multi-tenant virtualized environment. In this case, the application can experience performance interference that causes its performance degradation. The same latency L_{SLO} is then reached by a smaller workload $W_{ic} = W_c - \delta$ because of performance degradation. If the latency-driven elasticity controller is unaware of interference, it will allocate $\frac{W_c+n}{W_c} \times X$ instances to handle the increased workload $W_c + n$. This means that each instance will receive a workload greater than W_{ic}, thereby under subscribing to resources and violating SLO. Without quantifying the interference (I) and knowing W_{ic}, the elasticity controller cannot make accurate scaling decisions even when using a direct metric.

IV. Sources of Unpredictability

Given the wide use of CPU utilization and workload intensity for driving the scaling decision, we set out to explore the reliability of these metrics in a multi-tenant environment. Our objective in this section is to answer if these metrics are reliable in the face of performance interference and to identify the different sources responsible for introducing unpredictability in the metrics. To enable effective and accurate model control for elastic scaling, the control input (CPU utilization, Workload intensity etc) should be reliably predictable.

In order to discern the reliability of CPU utilization and workload intensity (RPS), we perform experiments with Memcached [21] under a controlled setting. We execute SPEC CPU benchmarks [22] to create interference on the memory subsystem. The experiment is carried out in 3 phases: no interference, 1x interference (when 1 SPEC CPU benchmark instance runs alongside Memcached) and 2x interference (when 2 SPEC CPU benchmark instances run alongside Memcached).

A. Is CPU utilization and Workload Intensity reliable in the presence of performance interference?

Figure 2a shows the variation of Memcached latency when the CPU utilization of Memcached is controlled to remain between 65-70% by varying the workload. Figure 2b shows the variation of Memcached latency when the workload is set to 40000 RPS. In order to have a reliable model, the metric should correlate well with latency and be reliably predictable. Ideally, as the amount of performance interference increases, the latency should increase. Figure 2a and figure 2b shows that when frequency-scaling is enabled, the CPU driver can scale the frequency of the processor depending on frequency governor enabled on the operating system resulting in unpredictable performance. However, once the frequency of the processor is fixed (blue line in figure), Memcached behaves in a predictable manner with latency acting directly proportional to interference. Frequency-scaling introduces unpredictability to the correlation between input metrics (CPU utilization, workload intensity) and latency.

The observation that latency increases as performance interference increases indicates that the correlation between input metrics and latency gets skewed in the presence of interference. i.e., for the same amount of workload intensity/CPU utilization, Memcached can experience different latencies. Without quantifying interference it is not possible to attribute this variation in correlation. Therefore, it becomes imperative to not only take into account interference but also be able to quantify the same in order to make reliable scaling decisions. Performance interference just like frequency scaling skews the correlation unless modelled. Table I provides a summary of the different sources of variation, their impact on modelling and potential ways to minimize this unpredictability. The metrics (CPU utilization, Workload intensity) achieve more predictability when all the sources of variation are controlled. However, minimizing the unpredictability comes with significant tradeoffs and not all parameters can be completely controlled. Contention is one such source as it manifests in multi-tenant environments. While frequency-scaling and interrupt processing can be controlled, performance interference is unavoidable in a multi-tenant setting and needs to be modelled.

V. System Overview

If we are able to model contention, then it is possible to attribute the variation in correlation and still rely on these input metrics to achieve reliable scaling. Contention can be seen as the amount of pressure an application puts on different shared resources such as memory-bandwidth or the cache. Each

Source of Variation	How to minimize variation	Trade-offs involved	Impact on Modelling
Interrupt Schedule	Assign dedicated cores for interrupts	Lower throughput when running at low utilization	Minimal, since the interrupt schedule remains the same throughout the life cycle of the service
Frequency Scaling	Disable C-states and P-states	Results in increased power usage	High, Affects both CPU and Workload based modelling.
Contention	Over-provision resources	Results in underutilized machines	High, Affects both CPU and Workload based modelling

TABLE I: Source of variation in CPU Utilization and Workload based modelling

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. Contention and sensitivity need not always be correlated [23] and they need to be modelled separately during the run-time. In our case, we are interested in the contentiousness of the co-runners and the sensitivity of the target system. Prior works [24], [25], [26], [27] 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 is the sole 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.

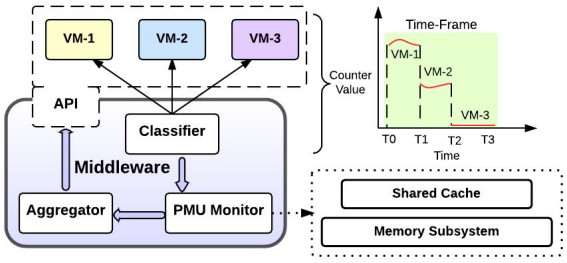


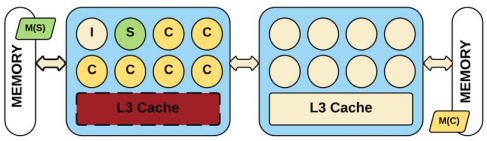
Fig. 3: Architecture of the middleware

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 exerted on the target system. It is important to quantify the amount of pressure exerted by the co-runners since this has a direct impact on the amount of performance degradation suffered by the target system. Different amount of contention causes different amount of degradation.

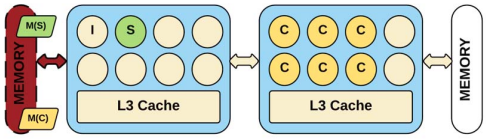
Middleware: Figure 3 shows the architecture of the middleware to quantify contention on the memory subsystem. The middleware provides an API that can be queried to access information about the contention from the co-runners. The different components of the middleware provide the following function: The 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 section. The measured counter values are then passed on to the aggregator that calculates a metric called the interference-index (explained in section VI) . Interference-index approximates the pressure the co-runners put on 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 3 shows one time-frame of execution. It is important to ensure that the chosen time-frame is neither too long nor too short. A very short time-frame can leave no time for the counters to be monitored since releasing and reacquiring the counters costs a couple of microseconds. Also, very short durations do not accurately capture the application behaviour. On the other hand, long time-frames aren't ideal either because it increases the probability of variation in the application behaviour and violates the assumption that the application experiences minimal variations. This is important

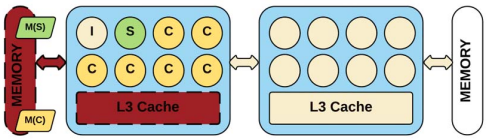
because the behaviour of every co-runner is composed together each time-frame. Major variations in the application behaviour during a time-frame can thus result in misleading conclusions.



(a) C's data is remote, hence S contends with C's only for L3 cache (config-a)



(b) C runs on a different processor, hence S contends with C's only for memory bandwidth (config-b)



(c) S and C share both memory bandwidth and L3 cache and hence contend for both (config-c)

Fig. 4: Configurations for generating contention at different resources. S denotes the storage system and M(S) denotes the memory allocation of S. C denotes the co-runners and M(C) denotes the memory allocation of C. I denotes the core serving interrupts.

VI. Characterising Contention

In order to characterise contention, we choose in-memory storage systems, i.e. Memcached and Redis, as demonstrative target systems to show the scaling of services under performance interference. We identify which resources, upon contention, degrade the performance of the storage system and the properties of the co-runners that determine the level of contention. In order to understand the properties of the storage systems and the memory sub-system they are sensitive to, we characterise them on a NUMA machine.

Co-runner (6X)	Cache References (millions)	Cache Misses (millions)	L3 Prefetch (millions)	L3 Prefetch miss (millions)	Memory bandwidth (GB/s)
mbw	150.1	54.1	73.76	68.29	20.3
stream	133.2	47.3	107.2	98.6	20.5
lbm	63.1	25.5	135.1	102.2	18.8
linearwalk	228	84.6	150.5	91.2	22.4
libquantum	242.5	91.6	93.3	57.7	21.2
randomwalk	1055.5	137.7	0.165	0.132	8.4
povray	24.6	0.015	37.6	0.009	0.01

TABLE II: Memory-subsystem behaviour of co-runners sorted by performance drop (highest to lowest) experienced by Redis and Memcached

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*. We denote the storage system by S and each of its co-runners by C. 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 $\frac{L_c - L_i}{L_i}$.

A. 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 three system configurations illustrated in figure 4. They 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 5 and 6 show the drop in performance experienced by Memcached and Redis respectively.

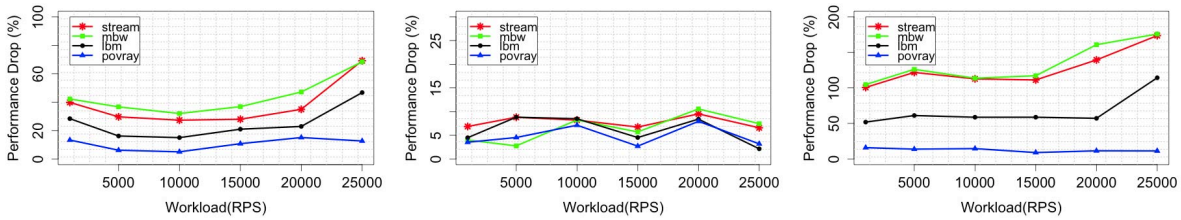
For both Memcached and Redis, it is clear that cache is the dominant source of performance degradation. In the case of Redis, cache contributes to a maximum of 30% drop in performance (figure 6a) while bandwidth only causes 8% (figure 6b). Similar observations can be made for Memcached, with cache contributing upto 65% drop in performance and bandwidth contributing less than 10%. However, the over all drop in performance drop of Redis is much higher in comparison to Memcached. Upon deeper analysis, we found that beyond 10000 RPS, Redis reaches a point of saturation in terms of available CPU. With proper configuration and optimization, it is possible to improve the throughput of Redis much beyond this limit. Since our intent is to demonstrate the impact of interference, we do not consider optimization or configuration set up to improve throughput. The results show that both the storage system benefits more from it's reliance on the cache than from memory bandwidth.

Our results are related to the conclusion drawn by running packet processing workloads on multicore platforms. The dominant contention source was found to be the cache [28]. As we will show, the difference comes from our observation that memory access pattern also impacts the performance of in-memory storage systems. On the contrary, SPEC CPU benchmarks are more sensitive on memory bandwidth [23].

B. Properties that determine degradation

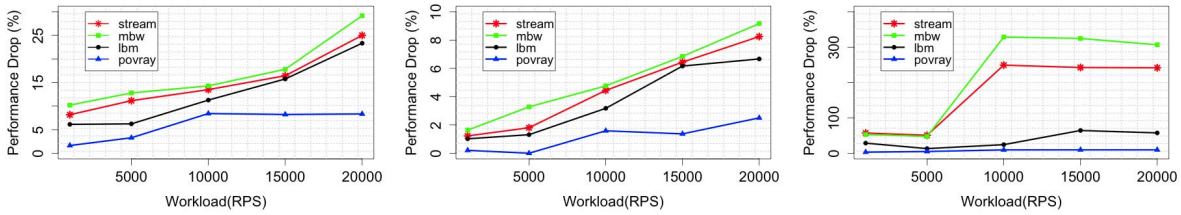
We investigate properties of the co-running application that cause performance degradation. In both figures 5 and 6, all the different co-runners cause degradation in the same order; ie. mbw consistently causes the highest amount of performance degradation, followed by stream, lbm, and povray. In order to understand the properties that define the aggressiveness of the co-runners, we rely on PMUs. L3 cache-references of the co-running applications were consistent with our observation and appears to mostly determine the degradation suffered. mbw has the highest number of cache-references and povray the lowest. This makes sense because higher cache references from the co-runners effectively reduces the cache space of the storage system, resulting in a drop in performance.

However, table II (sorted by descending order of performance degradation) shows that cache-references alone does not determine the performance drop of the storage system. For example: randomwalk, linearwalk and libquantum have



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

Fig. 5: The drop in performance of Memcached for different throughputs. Memcached is run alongside 6 instances of different co-runners.



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

Fig. 6: The drop in performance of Redis for different throughputs. Redis is run alongside 6 instances of different co-runners.

higher cache references than mbw, but they cause much lesser degradation. Linearwalk does a walk through the memory in a linear fashion being completely predictable, while randomwalk pseudo randomly walks within a page. Our results show that the sensitivity of the storage system also depends on the memory access patterns of the co-runner. We designed the application linearwalk and randomwalk precisely to study this property. Cache references do not capture the memory access patterns of the application. However, cache misses along with prefetch misses can provide hints about the memory access patterns of the application. In order to identify all relevant counters that affect the sensitivity of the storage system, we run a typical feature selection process that evaluates the effect of different performance counters on the sensitivity of the storage system. The chosen performance counters are shown in III. The chosen metrics correlate well with our observation on memory access patterns and cache access intensity of the co-runner. The feature selection process however indicated that DTLB loads and stores of the co-runner also reflect the sensitivity of the storage system. When we quantify the interference index we found that the impact of DTLB loads and stores are minimal and cache-references, cache-misses, prefetch and prefetch-misses are the strongest indicators to model sensitivity. Our model also includes cache-references of the target system to take into account different workloads. It inherently allows to generate a performance degradation model for different workloads. In the following section we explain the process of constructing the measure of degradation (interference-index) from these performance counters.

Summary: *Although the dominant contention factor is the cache, sensitivity of the storage system is not determined only by the number of cache references of the co-runners. The memory access pattern of the co-runner plays a significant role in determining the performance of the storage system.*

C. 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 for elastic scaling, 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 (Co-runner& Target-system)	References to L3 cache	cache-miss	L3 cache misses
llc-prefetch	L3 prefetches	llc-prefetch-miss	L3 prefetch misses
dtlb-loads	DTLB loads	dtlb-load-miss	load misses in DTLB that cause page walks
dtlb-store	DTLB stores	dtlb-store-miss	store misses in DTLB that cause page walks

TABLE III: Performance counters included in characterising contention

We derive a set of N representative performance counters $WS = m_1, m_2, ..., m_N$ where m_i represents the metric i. For applicability across heterogeneous machines, we rely only on generic counters to approximate this pressure. We find that even relying on a very coarse approximation to quantify contention can improve the accuracy of the scaling decisions

significantly. Using the counters in III 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 counter 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 the storage system. Higher the interference-index, greater the performance drop experienced by the storage system. Also similar interference-index should correspond to similar drop in performance. For example, mbw4 and stream4 indicate 4 instances of the co-runners mbw and stream respectively and are not included in the training set. stream4 causes similar degradation as linear walk and they both correspond to the similar interference indexes. mbw4 causes a degradation that is greater than linear walk but lesser than lbm and is also captured by the model as expected. We also test our model on a different set of co-runner, omnetpp from the SPEC benchmark. Our model is able to predict the drop with a good accuracy. Once interference-index is quantified, it is then used as a control input along with CPU utilization/workload intensity for the elasticity controller.

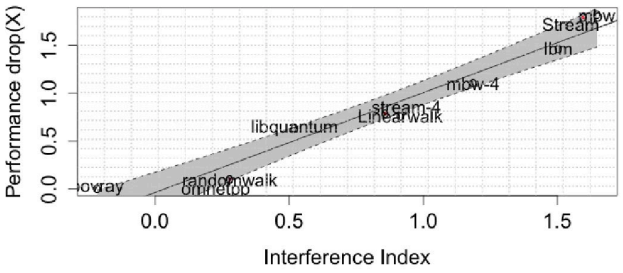


Fig. 7: Interference-index quantifies the performance drop suffered by Memcached based on the behaviour of the co-runner.

VII. Elasticity Controller

The main goal of an elasticity controller is to allocate adequate resource to a provisioned system in order to make the system operating in a healthy region that matches the control goal, e.g. Service Level Objective (SLO). The elasticity controller also optimizes the provisioning cost and prevents over-provisioning by allocating resources only when needed and freeing them when they are no longer needed. In our scenario, we define average service latency in small epochs (10 second) as our control goal, which is one of the common metrics specified in SLO between service providers and consumers. In the scenario of cloud computing, the amount of resources is translated to the number of virtual machines (VMs). As mentioned before, we target elasticity controllers that monitor system metrics, i.e. CPU utilization or incoming workload, i.e. read and write request rates, and model them as inputs. The former metric is commonly used to build an elasticity controller based on control theory [9], [3], [4] while the latter one is widely used to construct an elasticity controller using model-based control [18], [17], [8], such as Statistical Machine Learning (SML). Our observations from section IV show that the number of VMs in the system cannot be directly and linearly translated to the capability of the system to handle workload. Specifically, handling a workload under SLO constraint requires different numbers of virtual machines depending on the presence and intensity of VM interference. In Hubbub-Scale, we quantify the interference experienced in the system by querying the middleware API for interference index. Apart from this, the controller also takes CPU utilization and incoming workload intensity to model the load in the system.

Hubbub-scale is implemented as a centralized elasticity controller and its scaling decision is made by consulting control models that are built in an online fashion. There are two separate processes running in the Hubbub-scale controller that we call the model training process and the scaling process. Both processes run in parallel at different frequencies and do not interfere with each other. The model training process is used to continuously learn the application behaviours under different loads and intensities of interferences and update the control model. The load of the system is learnt from the sampled workload intensity or the CPU utilization on each VM. The monitored system behavior is narrowed down to the interested control goal, which is average service latency in small epochs. It is achieved by instrumenting the system to sample read and write latencies. We keep the monitoring overhead minimal by reducing the percentage of the sampled requests and the reporting frequency of the statistics to the model training process. The model training process updates the model with a simple data fading algorithm that uses weighted averages of service latencies from the most recent 10 epochs.

The scaling process consults the updated model with the monitored load of the system and the interference index from our API to make scaling decisions. To be specific, Hubbub-scale models the load of a system in two ways: workload-based modeling and CPU-based modeling The parameters used in workload-based and CPU-based models are firstly trained offline, and then improved during our online training process.

Workload-based Modelling: In workload-based modeling, the model is built and trained in a form of a binary classifier, which is widely used in recent works [18], [17], [8]. The classifier classifies the operational status of the modeled system. In our case, it models whether the system is operating in a state where its service latency violates the SLOs or not. Figure 8 shows a simplified version of Hubbub classifier, which uses finer data granularity, for explanation purposes. The binary classifier assumes that a certain VM is able to handle a specific load of the system, incoming read and write request rates, within the SLO constraint. The classifier is trained by having VMs operating with different intensity of workloads and monitoring the achievement of SLO. For example, when training the model without interference, an operating state with SLO violated is marked as a red cross in Figure 8 and an operating state complying SLO is marked as a green dot. The bounder of the red crosses and green dots is learnt using SVM (Support Vector Machine), which forms the classifier (the black border). Hubbub-scale provisions the underlying system to be operated just under the bounder of the classifier to

save the provisioning cost while satisfying the SLO. Hubbub-scale trains the classifier not only based on the incoming workload in terms of read and write request rate, but also takes into account the interference experienced on VMs, which is indicated from our interference index. Specifically, the amount of workload that can be handled by a VM is also learnt under different interference indexes with respect to the latency SLO constraint. The blue and pink borders in Figure 8 illustrate the learnt classifiers under 0.3 and 0.6 interference index. Thus, the binary classifier used in Hubbub-scale has 3 dimensions. It has an additional interference index dimension compared to the classifier proposed in [17], which has only 2 dimensions.

By obtaining the current workload and interference index, Hubbub-scale controller is able to calculate the number of VMs needed in the system using the following formula, where $AverageThroughputperServer$ denotes the throughput that can be handled by a server within the SLO constraint in the current level of the interference index.

$$NewNumberofServers = \frac{CurrentWorkload}{AverageThroughputperServer}$$

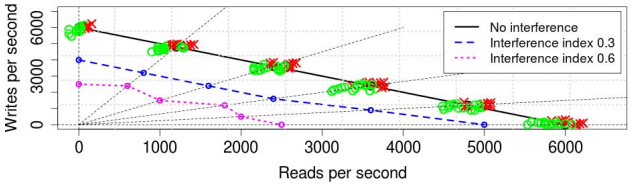


Fig. 8: Throughput Performance Model for different levels of Interference. Red and green points mark the detailed profiling region of SLO violation and safe operation respectively in the case of no interference.

CPU-based Modelling: Hubbub-scale can also model system's load using CPU utilization on each VM. A classical integral controller is built because of its self-correcting and provably stable performance in the application of a wide range of scenarios, and has been used successfully in state of the art systems [9], [29], [30], [31]. The core of the integral controller is the following formula:

$$a_{k+1} = a_k + K_i * (y_{ref} - y_k) \quad (1)$$

a_k and a_{k+1} are continuous integers that represent system capability at current control period and the next control period, which is then translated to the number of VMs that are needed in the system. K_i is the integral gain parameter [30]. y_k is the current input and y_{ref} is the desired input. The inputs are the monitored aggregated CPU utilization. Different values of desired CPU utilization y_{ref} are obtained with respect to a certain level of interference and the latency SLO. The controller obtains the desired VM numbers a_{k+1} from the previous time step a_k proportionally to the deviation between the current y_k and desired y_{ref} values of the CPU utilization in the current control period.

The difference between Hubbub-scale and a standard scaling approach is that Hubbub-scale takes into account the interference index in its model building. Standard scaling approaches rely on the standard modelling techniques, i.e., workload-based modeling and CPU-based modeling, used in the state of the art systems [17], [9], [18]. Specifically, standard workload-based modeling assumes a VM performs in an ideal scenario and is always capable of handling a specific workload without any knowledge of interference, similar to the implementation in [17], [8]. Standard CPU-based modeling only has one reference value (y_{ref}) in the model with respect to the latency SLO. In our evaluation, we show the inaccuracy of the standard modeling in the presence of interference and the accuracy of Hubbub-scale in conforming to SLO requirements.

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. Hubbub-Scale incurs very negligible overhead in comparison to standard modelling approaches since the only additional step involved is the construction of interference index, which in itself relies on non-intrusive monitoring.

VIII. Experimental Evaluation

We implemented our middleware on top of a KVM virtualization platform and conducted extensive evaluation using Memcached and Redis for varying types of workload and varying degrees of interference. This section describes our experiment setups and results.

A. Experiment Setup

All our experiments were conducted on the KTH private Cloud which is managed by Openstack [32]. Each host is an Intel Xeon 3.00 GHz CPU with 24 cores, 42GB memory and runs Ubuntu 12.04 on 3.2.0-63-generic kernel. It has a 12 MB L3 cache and uses KVM virtualization. The guest runs Ubuntu 12.04 with varying resource provisioning depending on the experiment. 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 and SPEC CPU benchmarks are run in different combinations to generate interference. 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 1 min. The time-frame chosen for monitoring the selected VMs after classification is 15 seconds and the counters are released for use by other processes for 45 seconds. The hosts running our experiments also run VMs from other users which introduces some amount of noise to our evaluation. However, our middleware also takes into account those VMs to quantify the amount of pressure exerted by them on the memory subsystem.

To focus on Hubbub-Scale rather than on the idiosyncrasies of our private Cloud environment, our experiments assume that the VM instances to be added are pre-created and stopped. These pre-created VMs are ready for immediate use and state management across the service is the responsibility of the running service, not Hubbub-Scale. Alternatively, interference generated from data migration can be accounted for by the middleware to redefine the SLO border to avoid excessive

SLO violations from state transfer. In order to demonstrate the exact impact of varying interference on Hubbub-Scale, we generate equal amounts of interference on all physical hosts and decisions for scaling out are based on the model from any one of the hosts. The load is balanced in a round robin fashion to ensure all the instances receive an equal share of the workload. We note that none of this is a limitation of Hubbub-Scale and is performed only to accurately demonstrate the effectiveness of the system in adapting to varying levels of workload and interference with respect to the latency SLO.

The control model of Hubbub-scale is partially trained offline before putting it online. Offline training is highly recommended but not mandatory. It identifies the operational region of the controlled system on a particular VM in an interference-free environment. Also, it improves the accuracy of the scaling during warm up phase. However, the Hubbub-scale control model can never be fully trained offline, because inter-VM interferences are hard to artificially produce as a cloud tenant. So, this part of the model can only get trained in an online fashion. The control models used in our evaluations are well warmed up by training them with different workloads and interferences online.

B. Results

Our experiments are designed to demonstrate the ability of Hubbub-Scale to dynamically adapt the number of instances to varying workload intensity and varying levels of interference, without compromising the latency SLO. The experiments are carried out in four phases, shown in figure 9i with each phase (separated by a vertical line) corresponding to a different combination of workload and interference setting. We begin with a workload that increases and then drops with no interference in the system. The second phase corresponds to a constant workload with an increasing amount of interference and later drops. The third phase consists of a varying workload with a constant amount of interference and in the final phase, both workload and interference vary. We carry out this experiment for 2 different types of control models: workload-based modelling and CPU-based modelling.

1) Scaling Out using a Workload based Model with/without Interference: Figure 9ii(b) and 9iii(b) compares the latency of a standard control model based on throughput performance modelling against Hubbub-Scale for all the four different phases for Memcached and Redis respectively. Without any interference (first phase), both systems perform equally well. However, in the presence of interference, the SLO guarantees of a standard control model begins to deteriorate significantly (figure 9ii(b), plotted in log scale to show the scale of deterioration). Hubbub-scale performs well in the face of interference and upholds the SLO commitment. The occasional spikes are observed because the system reacts to the changes only after they are seen. Figure9i(b) plots the interference index captured by the middleware during the run-time corresponding to the intensity of interference generated in the system. The index captures the pressure on the storage system for different intensities of interference. Certain phases of the interference index in the second phase do not overlap because of the interference from other users sharing the physical host (apart from generated interference). We found that during these periods services such as Zookeeper and Storm client were running alongside our experiments increasing the effective interference generated in the system. Figure 9ii(a) and 9iii(a) plots the number of active VM instances and shows that Hubbub-Scale is aware of interference and spawns enough instances to satisfy the SLO. In section VIII-C we show that Hubbub-scale does not over-provision instances to maintain the SLO.

2) Scaling Out using a CPU based Model with/without Interference: We construct a control model based on CPU as explained in section 5. Figure 9ii(d) and 9iii(d) plots the results from scaling out Memcached and Redis during the four different phases. Figure 9ii(c) and 9iii(c) plots the number of active VM instances as the workload intensity and interference intensity changes in four phases. Hubbub-Scale is aware of interference and adapts to it by spawning the right number of instances. Both Hubbub-Scale and the standard scaling perform equally well during the first phase and provision the same number of VMs to deal with the increasing workload in the absence of any interference. Hubbub-Scale adapts to the increasing interference and spawns more VMs to maintain the SLO requirement while standard modelling approaches fail. Figure 9i(c) shows the interference index captured during the runtime. Despite running a mix of different interfering applications, the index retains relative meaning and is robust enough to capture the pressure on the memory subsystem.

Our experiments indicate that a standard control model fails to capture the correlation between workload and latency in a multi-tenant scenario. Even a coarse approximation of resource contention is enough to drive the accuracy of the controller by a significant scale and minimize SLO violations.

C. Utility Measure

An efficient elasticity controller must be able to achieve high resource utilization and at the same time guarantee SLO commitments. Since achieving low latency and high resource utilization are contradictory goals, the utility measure needs to capture the goodness in achieving both these properties. While a system can outperform another in any one of these properties, a fair comparison between different systems can be drawn only when both the aspects are taken into account in composition. To this order, we define the utility measure as the cost incurred:

$$U = VM\_hours + Penalty$$

$$Penalty = DurationOfSLAViolations * penalty\_factor$$

$DurationOfSLAViolations$ is the duration through the period of the experiment the SLA is violated. We vary the penalty factor which captures the different cost incurred for SLO violations. Figure 10 shows the utility measure for 5 different scaling approaches. Ideal scaling represents the theoretical best scaling possible with right VM allocation and no SLO violations. Without any penalty for SLO violations, standard modelling incurs the lowest cost because it allocates only a few instances but results in SLO violations. But as the penalty for SLO violations increase, Hubbub-Scale achieves low utility (cost), which is much better than both standard scaling methods and comparable to ideal approach. Results with penalty=0 also shows that Hubbub-scale allocates a comparable number of VMs to ideal approach and does not achieve SLO guarantees by unfairly over-provisioning resources. We also note that this

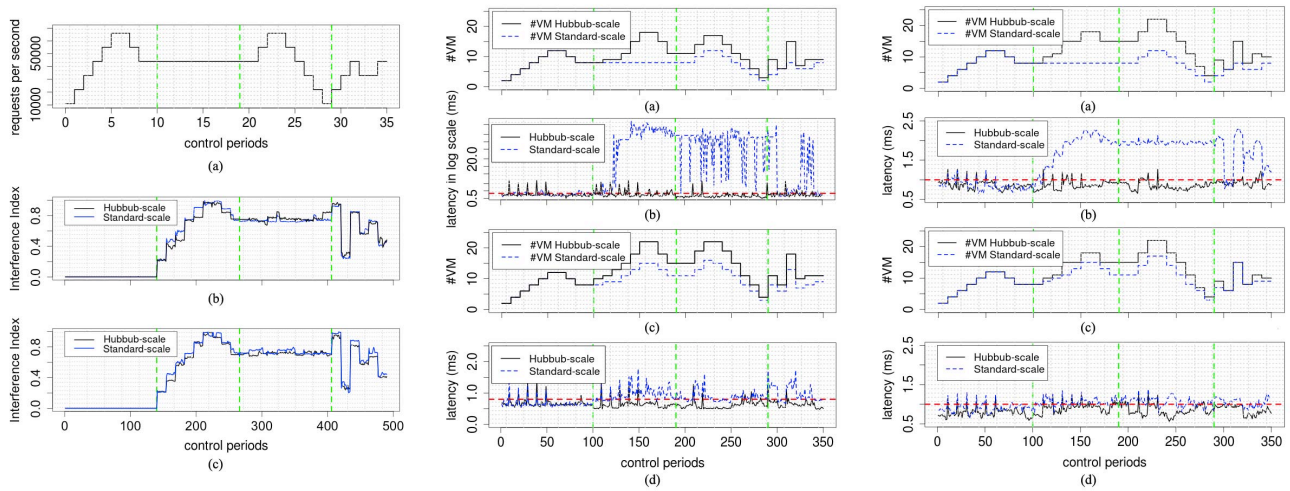


Fig. 9: (i) 9i shows the experimental setup. The workload and interference are divided into 4 phases of different combinations demarcated by vertical lines. 9i(b) is the interference index generated when running Memcached and 9i(c) is the interference index generated when running Redis. (ii) 9ii shows the results of running Memcached across the different phases. 9ii(a) and 9ii(b) shows the number of VMs and latency of Memcached for a workload based model. 9ii(c) and 9ii(d) hows the number of VMs and latency of Memcached for a CPU based model. (iii) 9iii shows the results of running Redis across the different phases. 9iii(a) and 9iii(b) shows the number of VMs and latency of Redis for a workload based model. 9iii(c) and 9iii(d) hows the number of VMs and latency of Redis for a CPU based model.

is a consequence of the way our experiments are carried out with interference on all physical hosts. With a round robin scheduler, the elasticity controller does over provision to some extent since each host roughly receive the same number of requests, and the maximum requests per server is capped by the lowest amount of workload that can be handled without violating the SLO. This over-provisioning can be mitigated by making the load balancer interference aware.

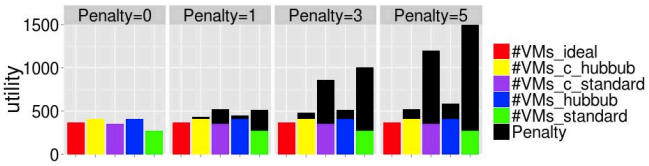


Fig. 10: Utility measure for different Scaling approaches. VMs_ideal represents the theoretically best scaling possible without any over-provisioning or SLO violations. VMs_c_hubbub and VMs_c_standard represents the utility measure of CPU based scaling using Hubbub and standard modelling respectively. VMs_hubbub and VMs_standard represents the utility measure of workload based scaling using Hubbub and standard modelling respectively.

IX. Related Work

Performance Interference: DejaVu [33] 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 [1], 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 as they do not consider the behaviour of the co-runners. Stay-Away [34] is a dynamic reconfiguration technique that throttles batch application proactively to minimise the impact of performance interference and guarantee QoS of latency critical services.

Another class of work has also investigated providing QoS management for different applications on multicore [35], [36], [37]. While demonstrating promising results, resource partitioning typically requires changes to the hardware design, which is not feasible for existing systems. Recent efforts [28], [38], [39] demonstrate that it is possible to accurately predict the degradation caused by interference by prior analysis of workload. In [40] the application is profiled statically to predict interference and identify safe co-locations for VMs. It mainly focuses on predicting which applications can be co-run with a given application without degrading its QoS beyond a certain threshold. The limitation of static profiling introduces a lack of ability to adapt to changes in application dynamic behaviour. Paragon [41] tries to overcome the problem of complete static profiling by profiling 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. Our work, in contrast, relies on quantifying contention in real time, allowing it to adapt to workload and phase changes.

Elastic Scaling: Amazon Auto Scaling [3] 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 [4] 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 is usually used to understand the application behaviors by building empirical models either online or offline. Simon [5] presents an elasticity controller that integrates several empirical models and switches among them to obtain better performance predictions. The elasticity controller built in [6] 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 [7], [8], [9], [10] 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. Chandra et al.[42] perform workload prediction using a histogram and auto-regression methods. Gmach et al.[43] used a Fourier transform-based scheme to perform offline extraction of long-term cyclic workload patterns. PRESS [12] and CloudScale [11] 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. Our proposed approach using a control model, combines both online and offline training to achieve efficient scaling plans.

X. Conclusion

We conducted systematic experiments to understand the impact of performance interference on CPU utilization and workload, two widely used metrics in elastic scaling. Our observations show that metrics become unreliable and do not accurately reflect the measure of service quality in the face of performance interference. Discounting the number of VMs in a physical host and the amount of interference generated can lead to inefficient scaling decisions that result in under-provisioning or over-provisioning of resources. It becomes imperative to be aware of interference to facilitate accurate scaling decisions. The implication of this observation introduces significant challenges in answering the following questions under multi-tenancy scenarios on: when to scale, how many VMs to launch, and where to place VMs.

We model and quantify performance interference as an index that can be used in the models of elasticity controllers. We demonstrate the usage of this index with CPU utilization and workload intensity by building Hubbub-scale, an elasticity controller that can reliably make scaling decisions in 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 sincerely thank Amiya maji and the reviewers for their critical and constructive comments and suggestions to improve the quality of the paper.

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Sujay Parekh, Neha Gandhi, Joseph Hellerstein, Dawn Tilbury, T Jayram, and Joe Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Systems*, 23(1-2):127–141, 2002.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 12–21. ACM, 2011.

[24] 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.

[25] 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.

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

[27] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 167–176. ACM, 2011.

[28] 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.

[29] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 289–302, New York, NY, USA, 2007. ACM.

[30] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Syst.*, 23(1/2):127–141, July 2002.

[31] Zhikui Wang, Xiaoyun Zhu, and Sharad Singhal. Utilization and slo-based control for dynamic sizing of resource partitions. In Jrgen Schnwlder and Joan Serrat, editors, *Ambient Networks*, volume 3775 of *Lecture Notes in Computer Science*, pages 133–144. Springer Berlin Heidelberg, 2005.

[32] Open Stack. http://www.openstack.org.

[33] 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.

[34] 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.

[35] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–355. IEEE Computer Society, 2007.

[36] 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.

[37] 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.

[38] 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.

[39] Jacob Machina and Angela Sodan. Predicting cache needs and cache sensitivity for applications in cloud computing on cmp servers with configurable caches. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[40] 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.

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

[42] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Quality of ServiceIWQoS 2003*, pages 381–398. Springer, 2003.

[43] 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.