Performance Interference-Aware Vertical Elasticity for Cloud-hosted Latency-Sensitive Applications

Shashank Shekhar, Hamzah Abdel-Aziz, Anirban Bhattacharjee, Aniruddha Gokhale and Xenofon Koutsoukos
Dept of EECS, Vanderbilt University, Nashville, TN, 37235, USA
Email: {shashank.shekhar, hamzah.abdelaziz, anirban.bhattacharjee, a.gokhale, xenofon.koutsoukos}@vanderbilt.edu

***Abstract*—Elastic auto-scaling in cloud platforms has primarily used horizontal scaling by assigning application instances to distributed resources. Owing to rapid advances in hardware, cloud providers are now seeking vertical elasticity before attempting horizontal scaling to provide elastic auto-scaling for applications. Vertical elasticity solutions must, however, be cognizant of performance interference that stems from multi-tenant collocated applications since interference significantly impacts application quality-of-service (QoS) properties, such as latency. The problem becomes more pronounced for latency-sensitive applications that demand strict QoS properties. Further exacerbating the problem are variations in workloads, which make it hard to determine the right kinds of timely resource adaptations for latency-sensitive applications. To address these challenges and overcome limitations in existing offline approaches, we present an online, data-driven approach which utilizes Gaussian Processes-based machine learning techniques to build runtime predictive models of the performance of the system under different levels of interference. The predictive online models are then used in dynamically adapting to the workload variability by vertically auto-scaling co-located applications such that performance interference is minimized and QoS properties of latency-sensitive applications are met.**

***Index Terms*—Cloud Computing; Data Center; Multi-tenancy; Workload variability; Latency Sensitive; Performance interference; Vertical elasticity; Virtualization; Linux Containers; Docker; Online predictive models.**

I. Introduction

Elastic auto-scaling is a hallmark resource management property of cloud computing which to date has focused mostly on horizontal scaling of resources wherein applications are designed to exploit distributed resources by scaling them across multiple servers using multiple instances of the application. However, spawning new virtual machines on-demand for horizontal scaling requires initialization periods that can last several minutes and the spawned instances must adhere to the cloud provider-defined instance types. This may lead to quality of service (QoS) violations (and hence violation of service level objectives – SLOs) in applications. To avoid QoS violations and to account for workload variations, cloud-hosted latency-sensitive applications, such as online gaming, cognitive assistance, and online video streaming, are often assigned more horizontal resources than they need [1]. Unfortunately, maintaining a pool of pre-spawned resources and application instances often will waste resources.

Considering the recent and emerging advances in hardware including the ever growing capacity of servers and the advent of rack-scale computing [2], vertical elasticity has become a promising area for dynamic scaling of applications and also a first choice for elastic scaling before horizontal scaling is attempted [3]. Vertical elasticity is the ability to dynamically resize applications residing in containers or virtual machines [4], [5]. It not only allows fine-grained assignment of resources to the application but also enables traditional applications that were not designed for purposes of horizontal scaling, to scale vertically according to its changing resource demands stemming from workload changes.

Vertical elasticity for latency-sensitive applications is often realized by co-locating them with batch applications such that they have some slack available to scale up or down on-demand and the resources are not wasted because the batch applications can utilize the remaining resources. Cloud service providers use virtual machine or container technologies to host multiple applications on a single physical server. Each latency-sensitive application has its own configuration and dynamically allocated resources that fulfill its application-specific demands and requirements.

Despite these trends, performance interference [6] between the co-located applications is known to adversely impact QoS properties and SLOs of applications [7]. Dynamic service demands and workload profiles further amplify the challenges for cloud service providers in (de)allocating resources on demand to satisfy SLOs while minimizing the cost [8]. This problem becomes even harder to address for latency-sensitive, cloud-hosted applications, which we focus on in our work. Therefore, any solution to address these challenges necessitates an approach that accounts for the workload variability and the performance interference due to co-location of applications.

To that end, we present a data-driven and predictive vertical auto-scaling framework which models the runtime behavior and characteristics of the cloud infrastructure, and controls the resource allocation adaptively at runtime for different classes of co-located workloads. Concretely, our approach uses a multi-step process where we first apply Gaussian Processes (GP) [9]-based machine learning algorithm to learn the application workload pattern which is used to forecast the dynamic workload. Next, we use K-Means [10] to cluster the system level metrics that reflect different performance interference levels of co-located workloads. Finally, we apply another GP model to learn the online performance of the latency-sensitive application using the measured data, which in turn provides real-time predictive analysis of the application performance. Our framework uses Docker container-based application deployment and control infrastructure that leverages the online predictive model in order to overcome run-time variations in workload and account for performance interference. We also periodically update the models in online fashion such that the dynamics of the target application workload and co-located applications are reflected in our predictions.

The rest of the paper is organized as follows: Section II compares our work with related research; Section III presents details of our approach; Section IV presents experimental evaluations; and finally Section V provides concluding remarks alluding to future work.

II. Related Work

We surveyed literature that focus on resource allocation strategies in cloud computing along the dimensions of workload prediction, performance interference, and vertical elasticity, all of which are key pillars of our research. We provide a sampling of prior work along these dimensions and compare and contrast our work with them.

Related research based on Workload Prediction:

To model different classes of workloads and applications, the Dejavu [11] framework computes and maps the workload signature to resource allocation decisions, and then periodically clusters the workload signature using K-means algorithm storing known workload patterns in the cache for rapid resource allocation. Likewise, [12] proposes an adaptive controller using Kalman filtering for dynamic allocation of CPU resources based on the fluctuating workloads without any prior information. In our prior work [13], we proposed a workload prediction model using autoregressive moving average method (ARMA). These works are based on linear models for QoS modeling; in contrast, cloud dynamics often illustrate nonlinear characteristics and incur significant uncertainty.

In [14], a non-linear, predictive controller is proposed to forecast workload using a support vector machine regression model. In contrast, our work uses a Gaussian Process (GP)-based model because it has relatively small number of hyper parameters so that the learning process can be achieved efficiently in an online fashion. Although some efforts [15], [16] use Gaussian processes to model and predict the query or workload performance for database appliances, they do not incorporate performance interference in their model. While most of the strategies for performance and resource management are rule-based and have static or dynamic threshold-based triggers [4], our system uses a proactive approach using GPs to learn parameters dynamically and perform timely resource adjustments for latency-sensitive applications.

Related research based on Vertical Elasticity:

Vertical elasticity adds more flexibility since it eliminates the overhead in booting up a new machine while guaranteeing that the state of the application will not be lost [4]. Several approaches are proposed to scale the CPU resources [3], [17]. Kalyvianaki et al. [5] proposed a Kalman filter-based feedback controller to dynamically adjust the CPU allocations of multi-tier virtualized servers based on past utilization. A vertical auto-scaler [18] is proposed to allocate CPU cores dynamically for CPU-intensive applications to meet their SLOs in the context of varying workloads. The authors offer a linear prediction model on top of the Xen Hypervisor to plug more CPU cores (hot-plugging) and tune virtual CPU power to provide the vertical scaling control. Controlling of CPU shares of a container based on the Completely Fair Scheduler is proposed in [19]. Vertical autoscaling techniques based on a discrete-time feedback controller for Containerized Cloud Applications are proposed in ELASTICDOCKER [4] that uses an approach to scale up and down both CPU and memory of Docker container based on resource demand. However, their decision triggering approach is reactive. In contrast, we use a more efficient Gaussian-based proactive method to trigger the scaling of resources.

Related research based on Performance Interference:

Prior research shows that model-based strategies are a promising approach which allow the cloud providers to predict the performance of running VMs or containers and to make efficient optimization decisions. DeepDive [20] is proposed to identify and manage performance interference between co-located VMs on the same physical environment. Q-Clouds [7] is a QoS framework which utilizes a feedback mechanism to model the interference interaction.

DejaVu [11] creates an interference index by comparing the estimated and actual performance. Based on matching the trained profile, it then provisions resources to meet application SLOs. Paragon [6] also classifies applications for interference and predicts which application will probably interfere co-located application performance for heterogeneous hardware based on collaborative filtering techniques. Unlike our approach, these efforts do not prioritize latency-sensitive applications due to interference from their co-located applications.

Bubble-flux [21] produces interference estimation by considering co-located applications on servers by continuously monitoring the QoS of latency-sensitive application and controlling the execution of batch jobs accordingly based on profiling. Heracles [22], which is a feedback controller, reduces performance interference by enabling the safe co-location of latency-sensitive applications and best-effort tasks while guaranteeing the QoS for the latency-critical application. Unlike these efforts, our GP-based model predicts the future latency in online fashion, and tunes the parameters on each iteration.

Our prior work [23] designed a performance interference-aware resource management framework that benchmarks the applications hosted on VMs. The server's performance interference level is then estimated using neural network-based regression techniques. However, hardware heterogeneity and every application's performance is not considered in the model. In another prior work [24], we benchmarked a latency-sensitive application with co-located applications on different hardware and develop its interference profile. The performance of the new application is predicted based on its interference profile which is obtained using estimators of an existing application for the same hardware specifications. A safe co-location strategy is decided by looking up the profile, however, we did not consider dynamic vertical scaling. In the current paper, we determine the vertical scaling strategy based on our online GP prediction model.

III. System Design for Proactive Vertical Elasticity

This section provides details of our solution for interference-aware vertical elasticity to support SLOs of latency-sensitive applications that are co-located with batch applications.

A. System Model

We target cloud data centers comprising multiple servers that host both latency-sensitive and batch-processing applications. The latency-sensitive applications have higher priority and need assurance of their SLOs while the provider also needs to ensure the remaining resources are utilized by the co-located batch processing applications such that there is minimal to no resource wastage. Docker is a container platform for application hosting with a growing user base with cloud service providers providing their own Docker deployment services, such as Amazon EC2 Container Service, Azure Container Service and Google Container Engine. We target cloud platforms hosting Docker containers natively. However, our solution can apply to any virtualized platform that allows rapid resource reconfigurability that is needed for vertical elasticity.

B. Problem Statement and Solution Approach

Cloud providers support multi tenancy by deploying applications in virtual machines or containers in order to provide a certain level of isolation. Moreover, to assure bounded latencies, cloud-hosted latency-sensitive applications are often assigned dedicated cores with the use of CPU core pinning [25], [26] which is the ability to run a specific virtual CPU on a specific physical core, thereby increasing cache utilization and reducing processor switches. Despite all these strategies, multi tenancy still leads to performance interference causing degradation in performance for latency-sensitive applications which can be particularly severe in the case of tail latency [27], i.e., 90th, 95th, 99th or similar percentile latency values. This is due to the presence of non-partitionable or difficult-to-partition resources such as caches, prefetchers, memory controllers, translation look-aside buffers (TLBs), and disks among others. The workloads for each such resource are referred to as *sources of interference* (SoIs) [6]. A SoI helps in identifying the interference that an application can tolerate for that resource before SLO violation occurs. Exacerbating the problem is the fact that different applications incur different levels of sensitivity to co-located workloads [28]. Figure 1 depicts an exemplar where the performance of a web search application is shown deteriorating significantly because of the presence of varying interference workload, even when they do not share the CPU cores. We observe that the 90th percentile latency is more than 51% worse when performance interference is present.

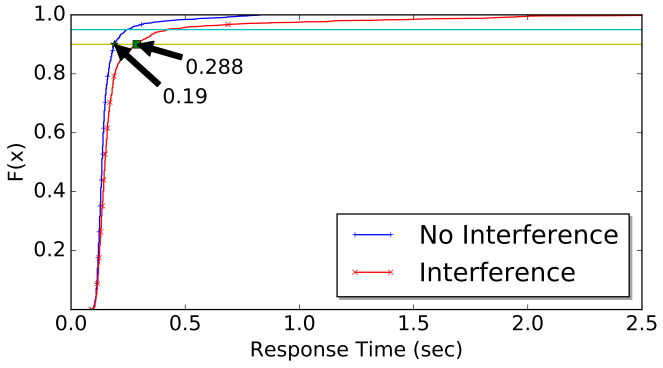

Fig. 1. CDF of Response Time for CloudSuite WebSearch with 6 Cores

Addressing these concerns warrants a solution that accounts for performance degradation due to interference and the workload variations so that SLO violations can be minimized while also minimizing cost to the cloud provider. Advances in container technologies such as Docker offer promise in allowing us the control and allocation of resources to rapidly adapt to workload variations and co-location interference. Prior efforts, such as Heracles [22], account for performance interference while also rapidly scaling the application for workload variations. They use feedback controllers where the best effort batch jobs are disabled if the demand from the latency-sensitive applications increases. However, the disabled jobs are still kept in memory thereby limiting the available resources for latency sensitive applications. To overcome this limitation, one can either checkpoint or migrate the co-located batch applications and restore them once the demand from latency sensitive application reduces. However, checkpointing and migration can take long durations, especially for memory-intensive applications where a large amount of state needs to be saved. Moreover, during this phase any additional resource allocation will not result in better performance. Reactive approaches also require very high rate of performance metric collection in order to react quickly to workload variations.

Consequently, an approach that can forecast the workload to proactively perform vertical scaling while accounting for interference imposed by co-located workloads is needed. Further, as the workload and interference level can vary dynamically, the solution should be able to forecast the required resources in an online fashion. Hence, we propose a model-predictive approach for vertical scaling which predicts the needed resources while accounting for workload variations at different levels of performance interference due to co-located workload. We use Gaussian Processes (GPs) to model the latency variations due to varying workloads forecasted using GPs We chose GPs over other learning techniques because they have relatively small number of hyper parameters. So the learning process can be achieved efficiently in online fashion. In addition, they are probabilistic models thus allowing us to model the uncertainty in the system while also being able to model nonlinear behavior of the underlying system.

C. Technique for Model-based Prediction

To optimize the resource utilization while maintaining the SLO guarantees for latency-sensitive applications, we need an accurate and online performance model of the latency-sensitive applications. There is also a need for an online model since the latency-sensitive application workload can vary dynamically. Moreover, the batch applications co-hosted on the same server can vary in their amount and nature of resource utilization. Finally, each application also incurs its own performance interference sensitivity to the co-located workload [6], [24]. Thus, the core component of our framework is the model predictor for which we have developed a per-application performance model in an online fashion that helps to rapidly adapt to changing levels of workload and co-location patterns.

We use Gaussian Processes (GPs) to model the performance of the latency-sensitive applications. GPs are non-parametric probabilistic models that utilize the observed data to represent the behavior of the underlying system [9]. A function $y = f(\mathbf{x}) : \mathbf{x} \in \mathbb{R}^d$ modeled by a GP can be expressed as: $f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}))$ where $m(\mathbf{x})$, $k(\mathbf{x}, \mathbf{x})$ are the mean function and the covariance functions of the GP model, respectively. Typically, a zero mean function and squared exponential (SE) covariance kernel are used for their expressiveness.

Given the training data with n data points ($\mathcal{D} = \{(\mathbf{x}_i, y_i) | i = 1, n\}$) where $\mathbf{x}_i$ are the training inputs and y_i are the training outputs, we train the GP model to identify their hyperparameters Θ so that they best represent the training data. In other words, we optimize the hyperparameters ($\hat{\Theta}$) of the GP model to maximize the log likelihood, i.e., $\hat{\Theta} = \arg\max_{\Theta} \log p(\mathbf{y}|\Theta, \mathcal{D})$ using the conjugate gradients optimization algorithm [9]. We define the test input at which we want to predict the model output as $\mathbf{x}_*$. Hence, the predicted output of the GP model (y_*) can be achieved by evaluating the GP posterior distribution $p(y_*|\mathbf{x}_*, \mathbf{X}, \mathbf{y})$ which is a conditional Gaussian distribution with a mean and a variance evaluated by:

$$\begin{aligned} \mathbb{E}[y_*|\mathbf{y}, \mathbf{X}, \mathbf{x}_*] &= \mathbf{K}_*^T \boldsymbol{\beta} \\ Var[y_*|\mathbf{y}, \mathbf{X}, \mathbf{x}_*] &= k_{**} - \mathbf{k}_*^T (\mathbf{K} + \sigma_\omega^2 \boldsymbol{I})^{-1} \mathbf{k}_* \end{aligned} \tag{1}$$

where $\mathbf{k}_* := k(\mathbf{X}, \mathbf{x}_*)$, $k_{**} := k(\mathbf{x}_*, \mathbf{x}_*)$, $\mathbf{K} := k(\mathbf{X}, \mathbf{X})$ and $\boldsymbol{\beta} := (\mathbf{K} + \sigma_\omega^2 \boldsymbol{I})^{-1} \mathbf{y}$.

In this work, we initialize the model with previously benchmarked metrics and then re-learn the model in an online fashion based on a moving window technique whenever new measurements are received. Since we emphasize online learning, we reduce the input features to our model to make the learning faster. First, we reduce the application level features using Pearson correlation analysis, and filtering out features with low correlation. Second, we cluster the system level metrics using K-Means, so that each cluster reflects a performance interference level caused by the co-located workloads. For each cluster, we segment its corresponding workload measurements and the container-level metrics to learn a distinct performance model of the latency-sensitive application. The cluster-based learning is very beneficial because it allows us to estimate the performance interference level caused by the co-located workloads. Moreover, it allows us to reduce the features dimension of the performance model (i.e., model input size) for fast online learning, since we learn independent models for each cluster using their corresponding workload measurements and container-level metrics in contrast to learning one model with all measurements including host-level measurements as inputs.

Figure 2 depicts the online performance model learning steps from our framework. The dashed lines indicate the learning steps and the solid lines map to prediction steps. In the first phase, we start by clustering the system-level metrics to estimate performance interference levels and use the associated workload measurements and the container-level metrics to learn a performance model, i.e., latency model using a distinct GP model for each estimated performance interference level. Furthermore, we learn a time-series GP model of the application workload, i.e., online users, so we can forecast the workload for the next time-step.

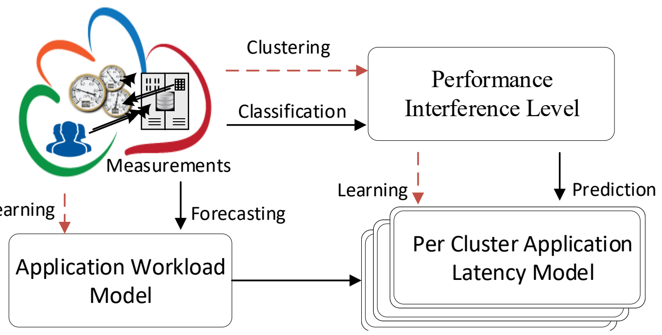
Clustering
Classification
Performance Interference Level
Measurements
Learning
Forecasting
Learning
Prediction
Application Workload Model
Per Cluster Application Latency Model

Fig. 2. Performance Model Learning and Prediction Steps

After learning the model, in the prediction phase, we use the current measurement as the model input to predict the application performance in terms of latency. Note that once we learn the initial model, the performance prediction happens first and the learned model is updated next. This ensures that performance prediction does not get delayed due to model update. We start with estimating the current performance interference level by classifying the current host-level metrics. In this step, the clusters' centroid that we obtain in the model learning step are used as the classifier centroid. Then, we use the corresponding GP model to predict the latency of the system. In addition, we forecast the application workload using the workload time-series model and pass it as one of the inputs to predict the latency using the GP model associated with the estimated performance interference level as described above.

D. System Architecture and Implementation

Figure 3 shows the system architecture. Our implementation comprises compute servers which host multiple containers or virtual machines. For our experiments, we used Docker as the virtualization mechanism, however, our architecture is generic

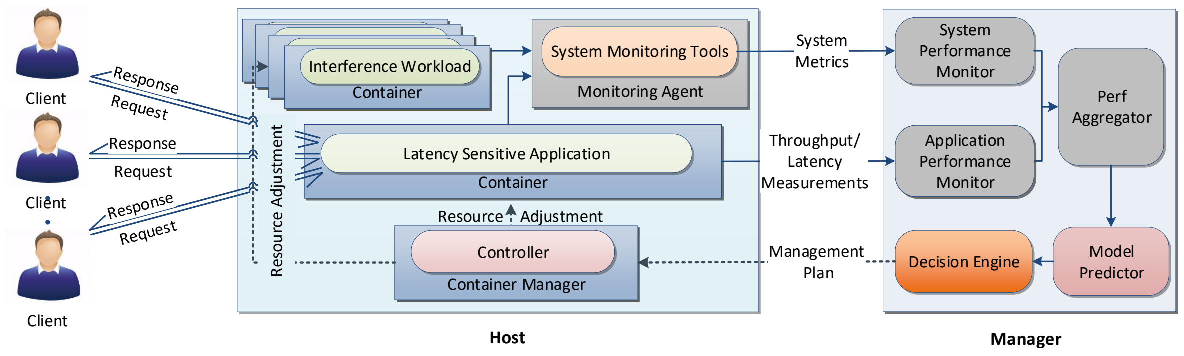

Fig. 3. System Architecture

enough to also include KVM or Xen-based hypervisors. The latency-sensitive application containers have dedicated cores assigned using CPU core pinning. The batch applications share the cores according to a defined overbooking ratio [29]. The CPU and memory allocation are controlled using the Cgroups features supported by Docker.

The performance of the entire system is monitored using a resource usage and performance interference statistics collection framework that we have developed called FECBench [24]. The measurements include both macro and micro architectural metrics, such as CPU utilization, memory utilization, network I/O, disk I/O, context switches, page faults, cache utilization, retired instructions per second (IPS), memory bandwidth, scheduler wait time and scheduler I/O wait time. Additionally, each latency-sensitive application reports its observed workload and response time to the Application Performance Monitor residing on the framework's Manager that is deployed on a separate virtual machine. Figure 3 illustrates the monitoring agent from FECBench residing on each of the hosts which periodically reports metrics to the Manager.

The system performance metrics are aggregated with the latency-sensitive application workload and latency data, and passed on to the model predictor (described in Section III-C). The model predictor predicts the performance of the latency-sensitive application and forwards the information to a decision engine. The decision engine then decides the action which can be *add/remove cores* to the latency-sensitive application and *remove/add cores* or *checkpoint/restore* for batch applications. Our control action is based on the fact that adding more cores not only provides more resources to process additional workload, but also alleviates performance interference due to larger share of resources, such as LLC and memory bandwidth [22]. The batch applications are checkpointed once they reach the overbooking ratio and the latency-sensitive applications need more resources. On the other hand, they are restored when it is found that restoring will still ensure that the latency-sensitive applications get their required resources and the overbooking ratio will not be exceeded. We leveraged the Checkpoint/Restore In Userspace (CRIU) feature for Docker to achieve checkpoint and restore of the containers.

IV. Experimental Validation

We present an experimental validation of our framework.

A. Evaluation Use Case

We consider a deployment scenario where latency-sensitive applications are co-located with batch applications whose workloads arrive in accordance to a distribution, emulated using real-world traces. Some of the popular cloud hosted latency-sensitive applications include web servers, search engines, media streaming among others.[1] For our system under test (SUT), i.e., the latency-sensitive application, we chose the CloudSuite WebSearch [30] benchmark since it fits our use case of varying workloads with low response time needs. The default version of this benchmark, however, can log only the runtime statistics to a file. Since we needed the runtime statistics to be published to a remote location to make adaptive resource allocation decisions, we modified the benchmark to publish the results using RESTful APIs for our data collection and decision making.

Since the CloudSuite WebSearch benchmark also does not provide workloads for experimentation, to emulate a real web search engine workload, we used the workload pattern for Wikimedia projects from the Wikipedia traces [31]. Specifically, we collected the page view statistics for the main page in English language for the month of September 2017 and scaled the first two weeks of data to our experimental duration. We used the scaled first week data for model training and the next week data for testing.

We used two batch applications to co-locate with the SUT: the first one is the Stream benchmark from the Phoronix test suite (http://www.phoronix-test-suite.com/), which is a cache and memory-intensive application, and the second one is a memory-intensive custom Java application.

B. Experimental Setup

Our experimental setup consists of a compute server with the configuration as defined in Table I. The server has Linux

[1]Latency-sensitive does not imply hard real-time applications but rather applications that have soft bounds on response times beyond which users will find the application behavior unacceptable. For instance, users expect a web search to complete within a specific amount of time.

kernel 4.4.0-98, Docker version 17.05.0-CE and CRIU version 2.6 for checkpointing and restoring Linux containers.

TABLE I
HARDWARE & SOFTWARE SPECIFICATION OF COMPUTE SERVER

Processor	2.1 GHz Opteron
Number of CPU cores	12
Memory	32 GB
Disk Space	500 GB
Operating System	Ubuntu 16.04.3 64-bit

We used the containerized version v3.0 of the Cloudsuite Web Search Benchmark as our SUT. The SUT is deployed on a server where it receives varying workloads over a period of time. The SUT is initially assigned 2 cores and 12 GB of memory. We vertically scale the number of assigned cores based on the output from the decision engine. The number of cores can vary from 2 to 10. One container is used to emulate the clients by varying their count as per the defined workload. This container also collects response times and throughput metrics. We deployed the client container on a separate host such that it does not have any effect on the experimental results similar to production deployment where the clients are located outside of the system. We aggregated and scaled the traces so that the number of users to the SUT change every 40 seconds.

The CloudSuite Web Search Benchmark relies on Faban [32] for time varying workload generation and statistics collection. We modified the Faban core used by CloudSuite benchmark so that it reports runtime metrics to the manager VM for model prediction and decision making. The metrics provided by CloudSuite include the throughput of the application, average latency which we use for the decision making and the 90th percentile latency which we consider as the tail latency used for measuring the efficacy of our system. Another key component of our experimentation is the FECBench framework that collects the application performance and system utilization metrics at an interval of 5 seconds and reports them to the Manager.

The manager VM is located on a separate machine which is responsible for each host's resource allocation decision making. The decision making and model update occurs every 15 sec which was chosen in order to avoid too frequent resource allocation modifications. Before deploying the system for online model prediction, we first perform offline analysis of the measured data set for the first week of the scaled Wikipedia traces. We applied the Silhouette [33] technique that determined two cluster centroids for performance interference level, which we used as the number of clusters in our online K-Means learning. For the online learning, we used 350 points as the K-Means window size and 200 points as the GP window size. We used the second week scaled data set for the experimental validation.

For the batch applications, the custom Java application is initially assigned 2 shared cores and 4GB of memory while the Stream test application is assigned 2 shared cores and 2.5 GB memory. We used Grid computing workload traces [34] to vary the batch application workload, which arrives according to a distribution to the same server as the SUT.

Our objective is to ensure that the latency-sensitive application adheres to the defined SLO guarantees while also allowing the batch applications to utilize the remaining resource slack. To achieve this, we need to appropriately assign resources to the latency-sensitive application and allocate the remaining resources to batch applications. While doing this, as the workload on the SUT increases, the resources allocated to the batch application need to be reduced. However, this reduction in resources for batch applications will increase the overbooking ratio (i.e., degree of contention for a specified number of resources). In our experiments, when the overbooking ratio reaches 2, the batch applications must be checkpointed in a way that does not incur the limitations of prior work where memory continues to be held by these batch applications. Later, when the workload on the latency-sensitive application reduces, the checkpointed applications are restored.

C. Experimental Results

The standard practice for cloud data center resource management involves threshold-based resource allocation. Approaches such as the ones defined in [35], [36] are reactive in nature and usually have thresholds based on request rate, response time or resource utilization. Thus, we compare our model predictive framework against two threshold-based reactive approaches. In the first approach, we set the threshold based on CPU utilization of the SUT container. The objective of the approach is to keep the CPU utilization within a target range.

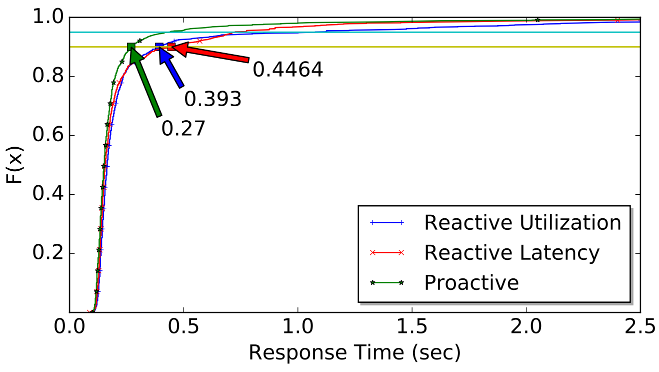

Fig. 4. CDF of Response Times for 3 different scenarios

We chose CPU utilization range of 50-70%, named as $Reactive_{Utililization}$. We make this choice as we do not want the server to be either under-utilized or become saturated. Whenever the utilization grows/reduces from the target range, we add or remove a core, respectively. In addition, if there is a sudden gain or drop of more than 20% utilization, we add or remove two cores. In the second approach, we put the threshold on average latency, which was chosen over tail latency because the latter characterizes transient and higher fluctuations, which is not needed for control actions. The target range was set to 70-100 ms. We also had higher bounds

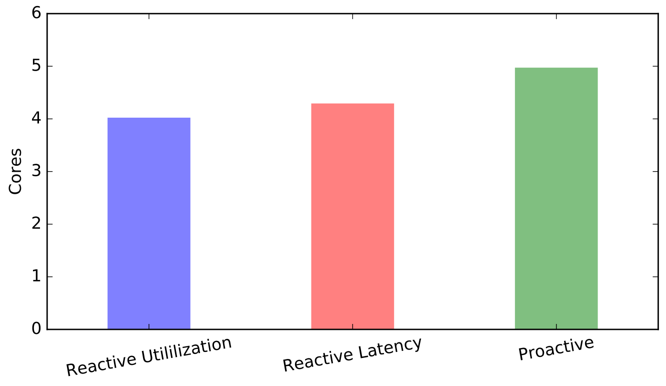

Fig. 5. Resource Utilization for 3 different scenarios

for adding/removing two cores of 50/150 ms. The configuration is called $Reactive_{Latency}$. We used the same bound as $Reactive_{Latency}$ for our proactive approach experiments, which is called $Proactive$.

Figure 4 compares the response time of the three scenarios listed above. We observe that the tail latency (90^{th} percentile) of our proactive approach is lowest at 270 ms. We also found the average latency to be 118, 102, 88 ms for $Reactive_{Utililization}$, $Reactive_{Latency}$ and $Proactive$, respectively. Figure 5 compares the average resource utilization for the duration of the experiment. Since our approach is a trade-off between resource utilization and the obtained latency, we observe that our proactive approach has higher resource utilization of 4.96 cores compared to 4.01 and 4.28 for the $Reactive_{Utililization}$ and $Reactive_{Latency}$ approaches, respectively. Thus, compared to the two approaches, at the cost of 19.15% and 13.7% extra resources, we achieve 39.46% and 31.29% better tail latency, respectively.

We also measured the efficacy of our model prediction. Figure 6 shows the model prediction results. We achieved a mean absolute percent error of 7.56%. For interference level, we found 2 clusters for our workload. The *Co-located Workload Clusters* part of Figure 6 displays different regions of co-located workloads. The other two subfigures compare our prediction against observed latency and request rate.

V. Conclusions

Dynamic vertical elasticity solutions for cloud platforms are increasingly becoming the first choice before using horizontal elasticity strategies. To that end, this paper presented a data-driven, machine learning technique based on Gaussian Processes to build a runtime predictive model of the performance of the system, which can adapt itself to the variability in workload changes. This model is then used to make runtime decisions in terms of vertically scaling resources such that performance interference is minimized and QoS properties of latency-sensitive applications are met. Empirical validation on a representative latency-sensitive application reveals up to 39.46% lower tail latency than reactive approaches.

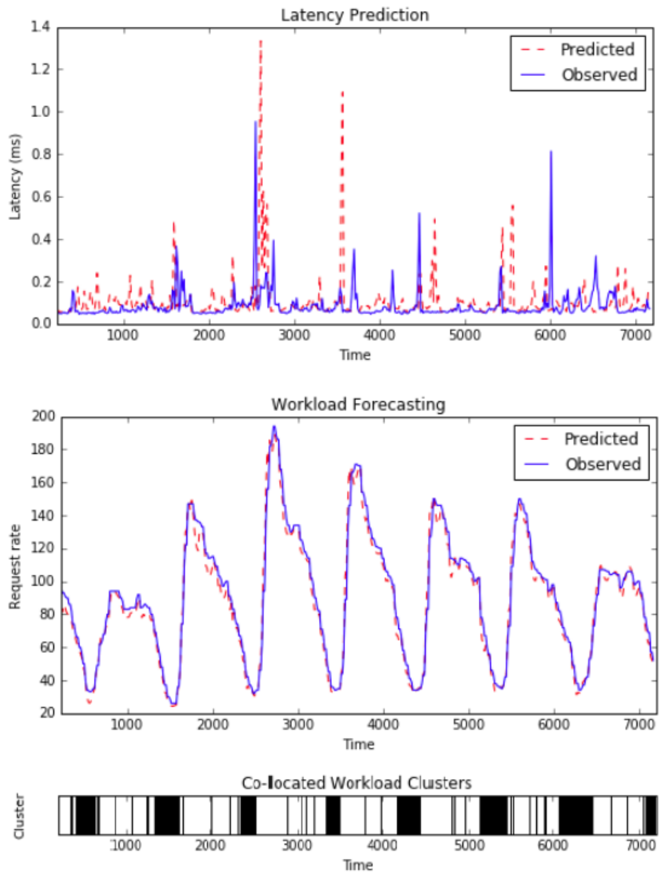

Fig. 6. Model Prediction

Our future work is informed by the following insights into unresolved problems that were gained from this research:

- Our work lacks finer-grained resource control such as managing CPU shares and memory, last-level cache and network allocation. To that end we are exploring the use of modern hardware advances, such as Intel's cache allocation technology and software-defined networking approaches to control network resource allocations.
- We viewed monolithic application design for latency-sensitive applications that are containerized. However, with applications increasingly being designed as distributed interacting microservices, distributed and coordinated vertical elasticity solutions become necessary.
- The thresholds for reactive approaches were chosen based on available literature. More experimentation is needed to compare against different thresholds and for different kinds of latency-sensitive applications.
- Presently, each of the clustered GP models executes inside the same VM. For future, we will perform distributed machine learning to reduce our online learning duration.
- Our future work will consider combining vertical scaling with horizontal scaling trading off along the different

dimensions based on application needs and incoming workloads. We will also include different workload categories, which can be both predictable and unpredictable.

The source code and experimental apparatus is available in open source at https://github.com/doc-vu/verticalelasticity.

Acknowledgments

This work is supported in part by NSF US Ignite CNS 1531079 and AFOSR DDDAS FA9550-18-1-0126. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF and AFOSR.

References

[1] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 301–312.

[2] "Rack-scale computing." [Online]. Available: https://www.microsoft.com/en-us/research/project/rack-scale-computing/

[3] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth, "Towards faster response time models for vertical elasticity," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2014, pp. 560–565.

[4] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *10th IEEE International Conference on Cloud Computing, IEEE CLOUD 2017*, 2017.

[5] E. Kalyvianaki, T. Charalambous, and S. Hand, "Adaptive resource provisioning for virtualized servers using kalman filters," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 2, p. 10, 2014.

[6] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[7] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 237–250.

[8] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[9] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[10] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[11] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: accelerating resource allocation in virtualized environments," in *ACM SIGARCH computer architecture news*, vol. 40, no. 1. ACM, 2012, pp. 423–436.

[12] E. B. Lakew, A. V. Papadopoulos, M. Maggio, C. Klein, and E. Elmroth, "Kpi-agnostic control for fine-grained vertical elasticity," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2017, pp. 589–598.

[13] 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*. IEEE, 2011, pp. 500–507.

[14] L. R. Moore, K. Bean, and T. Ellahi, "Transforming reactive auto-scaling into proactive auto-scaling," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*. ACM, 2013, pp. 7–12.

[15] M. B. Sheikh, U. F. Minhas, O. Z. Khan, A. Aboulnaga, P. Poupart, and D. J. Taylor, "A bayesian approach to online performance modeling for database appliances using gaussian models," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 121–130.

[16] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu, "Predicting completion times of batch query workloads using interaction-aware models and simulation," in *Proceedings of the 14th International Conference on Extending Database Technology*. ACM, 2011, pp. 449–460.

[17] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 5.

[18] L. Yazdanov and C. Fetzer, "Vertical scaling for prioritized vms provisioning," in *Cloud and Green Computing (CGC), 2012 Second International Conference on*. IEEE, 2012, pp. 118–125.

[19] J. Monsalve, A. Landwehr, and M. Taufer, "Dynamic cpu resource allocation in containerized cloud environments," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 535–536.

[20] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proceedings of the 2013 USENIX Annual Technical Conference*, no. EPFL-CONF-185984, 2013.

[21] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 607–618.

[22] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 450–462.

[23] F. Caglar, S. Shekhar, A. Gokhale, and X. Koutsoukos, "Intelligent, performance interference-aware resource management for iot cloud backends," in *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 2016, pp. 95–105.

[24] S. Shekhar, A. Chhokra, A. Bhattacharjee, G. Aupy, and A. Gokhale, "INDICES: Exploiting Edge Resources for Performance-Aware Cloud-Hosted Services," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, Madrid, Spain, May 2017, pp. 75–80.

[25] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao, "Who is your neighbor: Net i/o performance interference in virtualized clouds," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 314–329, 2013.

[26] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 1–10.

[27] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[28] W. Kuang, L. E. Brown, and Z. Wang, "Modeling cross-architecture co-tenancy performance interference," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 231–240.

[29] L. Tomás and J. Tordsson, "Improving cloud infrastructure utilization through overbooking," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/2494621.2494627

[30] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 122–132.

[31] wikipedia.org, "Page view statistics for Wikimedia projects," 2017, URL: https://dumps.wikimedia.org/other/analytics/, Last accessed: 2017-11-22.

[32] Faban - helping measure performance. URL: http://faban.org/, Last accessed: 2017-11-26. [Online]. Available: http://faban.org/

[33] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[34] A. Iosup and D. Epema, "Grid computing workloads," *IEEE Internet Computing*, vol. 15, no. 2, pp. 19–26, 2011.

[35] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," *Concurrency and computation: practice and experience*, vol. 26, no. 12, pp. 2053–2078, 2014.

[36] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia, "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *Journal of Network and Computer Applications*, vol. 52, pp. 11–25, 2015.