PRIMA: Subscriber-Driven Interference Mitigation for Cloud Services

Joydeep Mukherjee and Diwakar Krishnamurthy

***Abstract*—Network services, e.g., video streaming services, are increasingly being deployed on public cloud platforms. Such services often employ horizontal scaling where a group of resource instances, e.g., virtual machines (VMs), handle incoming workload. The response time of such services is often affected by interference, i.e., contention among resource instances belonging to multiple cloud subscribers for shared cloud resources. Most commercial cloud platforms do not support built-in mechanisms to detect interference and mitigate its impact. Consequently, subscribers of such platforms, i.e., network service providers, need to deploy their own mechanisms to ensure a specified end user response time target is continuously met even in the face of fluctuations in workload and interference. This paper describes PRIMA, our implementation of such a mechanism. PRIMA uses automated and controlled performance tests to build models that capture the joint impact of workload and interference on the response time of each resource instance employed by a service. It adapts the system to changing workload and interference conditions by using these models at runtime to control the number of instances in the system and the distribution of load among these instances. Unlike existing subscriber-oriented interference mitigation techniques in literature, PRIMA guarantees that a subscriber-specified response time threshold is satisfied at every resource instance assigned to a service. Furthermore, in contrast to these approaches PRIMA can help a subscriber avoid using more instances than necessary by automatically selecting at runtime the least number of instances required for handling the observed workload and interference. We experimentally validate the effectiveness of PRIMA in both private and public cloud environments. Results show that PRIMA outperforms competing approaches proposed by us and others, including those that are commonly used in practice. They also reveal that PRIMA can automatically calibrate its models at runtime to account for any model prediction errors.**

***Index Terms*—Software performance, cloud computing, quality of service, predictive modeling.**

I. Introduction

PUBLIC cloud providers often implement resource virtualization in their data centers by running multiple resource instances, e.g., virtual machines (VMs), on a shared physical machine (PM). Such virtualization can cause performance interference when multiple instances belonging to different cloud subscribers compete with one another for a shared PM resource, e.g., the processor or network bandwidth [1], [2], [3], [4], [5], [6]. Interference can be especially problematic for interactive network services. Specifically, the occurrence of interference can be unpredictable. When it happens, interference can manifest itself as higher response times, i.e., poor Quality of Service (QoS), leading to frustration for end users of such services. For example, users of a video streaming application hosted on the cloud can experience unexpected drops in video quality due to sudden network contention on PMs [7].

Unfortunately, commercial cloud platforms typically do not support built-in mechanisms to continuously detect and mitigate the adverse impact of interference. Consequently, subscribers of such platforms need to deploy their own mechanisms to ensure a specified end user response time target is continuously met by each of their instances even in the presence of fluctuations in interference as well as service workload. However, developing such mechanisms is challenging since cloud subscribers typically do not have access to data pertaining to how the physical machine âs hardware is used by VMs hosted on that machine. Thus, a subscriber cannot directly determine the extent of interference suffered by their instances from instances belonging to other subscribers.

Due to the challenges in detecting interference and quantifying its impact on performance, subscribers often employ simplistic interference-agnostic performance management techniques that can suffer from many drawbacks. Specifically, performance problems are typically mitigated using techniques such as load balancing and auto scaling. A cloud subscriber can employ load balancing to distribute incoming requests between a set of instances available to the subscriber, collectively called the load balancing group (LBG). Load balancing in public cloud platforms works in conjunction with techniques such as auto scaling that can expand or shrink the LBG as required. Common load balancing algorithms supported by commercial cloud platforms, e.g., round robin and least connections [8], do not explicitly take into account how individual instances within the LBG are impacted by interference at any give point in time. Consequently, the incoming workload can be distributed in an ineffective manner leading to performance degradation. For example, consider a system with 2 identical instances where one instance is currently suffering from interference while the other is not. A round robin policy can incorrectly distribute equal workload to these instances, leading to poor performance in the instance with interference.

Manuscript received February 24, 2019; revised July 14, 2019 and November 23, 2019; accepted December 13, 2019. Date of publication December 23, 2019; date of current version June 10, 2020. The associate editor coordinating the review of this article and approving it for publication was L. Y. Chen. *(Corresponding author: Joydeep Mukherjee.)*

Joydeep Mukherjee is with the School of Information Technology, York University, Toronto, ON, Canada (e-mail: jmukherj@yorku.ca).

Diwakar Krishnamurthy is with the Department of Electrical and Computer Engineering, University of Calgary, Calgary, AB T2N 1N4, Canada (e-mail: dkrishna@ucalgary.ca).

Digital Object Identifier 10.1109/TNSM.2019.2961613

This paper develops an integrated load balancing and auto scaling technique to address such limitations of interference-agnostic techniques. A key challenge in realizing effective interference-aware load balancing and auto scaling strategies is determining the amount of workload an instance can handle given the current extent of interference at that instance and the response time target. As we show later in Section VI-A, the interplay between workload, interference, and response time is typically complex. In particular, the extent of response time degradation accompanying an increase in workload assigned to an instance can depend on the extent of interference at the instance. For example, response time is likely to be more sensitive to an increase in workload when interference, i.e., contention for the shared resource, is severe. Similarly, the impact of increased interference on response time is likely to be more dramatic when the instance is experiencing heavier workloads. These observations motivate the need for an approach where models that capture such non-linear and interacting relationships are used to estimate the amount of workload an instance can handle. Such estimates can then be used to drive load balancing and auto scaling decisions. In this paper we propose such a model-based load balancing and auto scaling approach called performance interference management approach (PRIMA).

PRIMA is an interference-aware integrated load balancing and auto scaling technique. It exploits data-driven models derived by deploying a subscriber-oriented interference estimation system developed in previous work [3]. This system, referred to as the *probe*, reports a metric called the *Severity Factor* (SF) for an instance that represents the severity of response time degradation experienced by that instance due to interference. First, we build a *response time model* that can predict the mean response time of an instance given the workload assigned to the instance and its SF value. We also build an *interference model* that can estimate how the SF value at any given instance changes as a function of the instance's current SF value and the workload assigned to that instance. Next, we implement a runtime PRIMA controller that iterates between these two models to calculate the split of incoming traffic between the current instances in the LBG such that a subscriber-specified mean response time threshold is satisfied at each instance. If the current instances in the LBG are insufficient for accommodating the system workload at their present interference levels, PRIMA can use the models to scale out. Similarly, PRIMA can scale in when instances are not needed.

While others have proposed subscriber-driven interference mitigation systems [4], [9], our work makes several novel contributions. First, unlike these existing solutions PRIMA guarantees that a subscriber-specified response time threshold is satisfied by every instance in an LBG. Second, the existing approaches do not focus on automatically expanding and shrinking the LBG in response to fluctuations in workload and interference. In contrast, PRIMA ensures that a subscriber will avoid unnecessary expenses by automatically determining at runtime the minimum number of instances needed to achieve the response time target given the observed workload and interference conditions. Finally, PRIMA does not require monitoring of hardware counters [10] or service response times [11], which can incur a prohibitive overhead in heavy load and heavy interference scenarios [12], [13].

Using a realistic video streaming service, we validate the effectiveness of PRIMA in our private cloud as well as in the public Amazon Web Service (AWS) Elastic Compute Cloud (EC2). Experimental results demonstrate that PRIMA outperforms load balancing policies commonly supported by public cloud platforms. These results also show that PRIMA provides better performance than competing techniques we developed that do not use models but rather rely solely on monitoring the interference and response time at the instances. Overall, these results indicate that PRIMA is agile in responding to fluctuations in both workload and interference. Specifically, we show that the PRIMA models work dynamically at runtime to ensure that the each instance accepts only as much workload as is possible given the interference it is experiencing while satisfying the response time threshold. We also show that PRIMA can expand and shrink the LBG optimally. Finally, we also show that PRIMA is robust in that its models can be calibrated automatically at runtime to handle any prediction errors.

This paper extends our earlier short conference publication [14] as follows. Section VI-C provides an experimental comparison of PRIMA with load balancing policies typically supported by public cloud platforms. In the same section, we also compare PRIMA with interference-aware policies we developed that do not require the response time and interference models. In Section VI-D, we present new experiments that characterize the behaviour of PRIMA when the system's workload characteristics, e.g., request arrival pattern, differ from those used for building its underlying models. Additionally, we develop and evaluate in Section IV a technique to calibrate the PRIMA models at runtime to improve their predictive accuracy.

II. Related Work

Several past studies have indicated the presence of performance interference in commercial public cloud platforms [7], [15]. Previous studies have devised techniques that providers of public cloud infrastructure can exploit to detect and mitigate the impact of interference [10], [11], [16], [17], [18]. For example, Shen *et al.* propose a prediction driven elastic resource scaling system called CloudScale which works on top of the Xen virtualization platform [18]. CloudScale continuously monitors resource usage metrics such as CPU, memory and I/O utilization of each guest VM from the host PMs and uses them as input to an online resource demand prediction model to drive resource scaling decisions to maintain application service level objectives. Ananthanarayanan *et al.* propose a similar provider-driven interference mitigation approach for data analytics applications [17]. Their approach requires the provider to have full access and control over all tasks running on PMs within a data center to mitigate the deleterious impact of long running tasks on the performance of short, interactive tasks. Subscribers cannot implement such techniques since they do not have access to PM-level metrics, e.g., last level cache misses at a PM's processor, and control over applications run by other subscribers on the cloud platform.

In general, provider-driven techniques require access to PM-level metrics and control over applications run by other subscribers on the cloud platform. This motivates the need for cloud subscriber-driven interference mitigation approaches.

Many studies have proposed subscriber-driven load balancing and auto scaling strategies that allow an application to continuously achieve operator specified performance targets in spite of workload fluctuations [19], [20]. For example, Ren *et al.* propose a weighted least connections algorithm to distribute incoming workload among cloud resource instances [19]. Lakew *et al.* propose a mechanism called SmallScale that can scale vertically to provision virtual cores to VMs and clone requests to achieve the target tail latency at a minimum provisioning cost in the face of workload fluctuations [21]. Similar studies [22], [23] have also considered horizontal scaling as a mechanism to meet QoS requirements under workload fluctuations. While these techniques consider workload fluctuations, they do not consider the impact of interference. Our results in Section VI-C show that both workload fluctuations and interference need to be jointly considered in performance management exercises.

Compared to studies proposing provider-driven solutions, very few studies have focused on subscriber-driven solutions for interference mitigation. Maji *et al.* propose an interference-aware load balancing technique called ICE that is targeted towards public cloud subscribers [9]. ICE limits the workload assigned to instances suffering from interference such that the CPU utilization of these instances is below a certain statically set threshold. In contrast to PRIMA, ICE does not provide an explicit mechanism for maintaining response times below a specified target. One has to tune the CPU utilization thresholds of individual instances by trial and error to achieve a desired response time target. Furthermore, multiple thresholds might be needed to deal with fluctuations in the severity of interference [4]. Finally, since it focuses only on load balancing ICE does not support automated scale in and scale out of an LBG.

Javadi and Gandhi develop a subscriber-driven load balancing technique called DIAL that considers the impact of interference [4]. In contrast to PRIMA, DIAL does not support automatic scale out. Consequently, while it can minimize response time at each instance in an LBG, it cannot guarantee that this minimum value will be below an operator-specified threshold. Furthermore, unlike PRIMA, DIAL does not support scale in to avoid using more instances than necessary to satisfy the desired response time target. Moreover, due to the use of an M/M/1 model to drive load balancing decisions, DIAL is likely to be more effective for systems with exponentially distributed request inter-arrival and service times. Finally, unlike PRIMA, the technique requires service response times to be continuously monitored, which can introduce large overheads when the service is busy or is experiencing heavy interference [13], [24].

III. PRIMA

A. Overview of PRIMA

A system managed by PRIMA consists of a load balancer, an LBG, the probe system [3] deployed on each instance in the LBG, and the PRIMA controller. The controller determines the number of instances in the LBG. It also controls how the load balancer distributes incoming workload to these instances. The LBG scaling and load balancing are controlled such that the mean request response times are maintained below a subscriber-specified threshold R_{th} in all instances while using the least possible number of instances. Although PRIMA can accommodate different types of workloads, we consider network intensive workloads in this paper.

The probe system detects and quantifies interference at an instance in the LBG over a sampling period. As we demonstrate in our earlier work [3], the probe can detect and quantify interference simultaneously for multiple PM resources. However, due to our focus on network intensive services, in this work the probe is configured to estimate contention for a PM's network bandwidth. The probe periodically reports to the controller an SF value SF_m for any given instance *m*. SF_m quantifies the impact of the network interference experienced by the instance over the sampling period.

For the same sampling period, the controller measures the total incoming workload. Specifically, each instance *m* reports to the controller its network utilization U_m. U_m is reported as a percentage of the total network bandwidth available to *m*. The controller aggregates the U_m values reported by the instances in the LBG to obtain the total workload *U* being handled by the system.

Next, the controller uses the SF_m values and *U* within the data-driven models discussed in Section III-B2 to estimate the maximum workload, i.e., network bandwidth, each instance can handle given its current SF value such that the mean response time target R_{th} is not exceeded. We refer to the bandwidth utilization estimated in this manner for instance *m* as its *effective capacity* U_m^{max}. The controller suggests a scale out of the LBG if the total workload *U* exceeds the sum of effective capacities of the existing instances. Similarly, it recommends a scale in if the aggregate of the effective capacities exceeds *U*. Finally, PRIMA instructs the load balancer to distribute the incoming workload to LBG instances in proportion to their effective capacities. Since there is sufficient capacity to handle the incoming workload, this strategy ensures that each instance handles just enough workload to keep its response time equal to or below R_{th}. We note that due to its use of data-driven models, PRIMA can be easily extended to support other performance metrics such as the 90*th* percentile tail latency of service response times.

We note that PRIMA does not require monitoring of instance response times, which can be expensive [13], [24]. However, one can optionally collect the measured mean response time R_m at instance *m* for a short duration to calibrate the models used by PRIMA. We also note that PRIMA's load balancing and scaling decisions are based on the U_m and SF_m values measured over a sampling period. As a result, the controller has to be invoked periodically to handle fluctuations in service workload and interference.

B. Deploying PRIMA

We describe in detail the steps involved in deploying PRIMA. We note that all instances belonging to a subscriber's

LBG have the same specifications, as is typical in well known public cloud platforms such as EC2 and Microsoft Azure.

1) Training the Probe: First, the probe system has to train itself to detect interference for the specific kind of instance it will be monitoring. The probe consists of a low overhead microbenchmark application designed to compete for a PM's network bandwidth. The main objective during training is to characterize the performance of this application under a no interference condition. At runtime, a deviation of the probe microbenchmark's performance from this no interference performance can be used to flag interference. Creating a no interference condition requires a dedicated instance. The dedicated instance has the same characteristics as the instance that needs to be monitored. However, it is executed in isolation on a PM and hence does not suffer from interference. Commercial cloud systems such as EC2 offer such instances. Although such instances cost more, they are only required for a very short duration. For example, the probe training took only 30 minutes in all our case studies.

Performance data under no interference is obtained by concurrently executing on the dedicated instance the probe microbenchmark application and the network service being managed. Specifically, the network service is subjected to a synthetic workload. The workload intensity, e.g., the mean rate of arrival of synthetic requests, is varied to cause a range of network utilizations of interest. The mean execution time of the microbenchmark $R_{iso}(U_{ded})$ is recorded for each utilization level U_{ded} of the dedicated instance to construct a look up table. To obtain a reliable measure of $R_{iso}(U_{ded})$, multiple tests are done so that the width of the 95% confidence interval of $R_{iso}(U_{ded})$ is within 5% of the sample mean. Given a service utilization level U_m for instance *m*, the look up table provides $R_{iso}(U_m)$, the execution time of the microbenchmark at that utilization when there is no interference.

Data obtained in the training phase can be used to quantify the severity of interference at runtime as follows. Consider a case where the mean execution time of the probe microbenchmark recorded at runtime at instance *m* under utilization U_m is $R^P_m(U_m)$. If $R^P_m(U_m)$ statistically exceeds $R_{iso}(U_m)$, then the probe infers interference. The severity factor SF_m is used to provide PRIMA an indication of the impact of interference at instance *m*. SF_m is calculated as shown in Eq. (1). Higher values of SF_m mean that the $R^P_m(U_m)$ is significantly higher than $R_{iso}(U_m)$, which implies the impact of interference is severe.

$$SF_m = \begin{cases} \frac{R^P_m(U_m)-R_{iso}(U_m)}{R_{iso}(U_m)}, & \text{if } R^P_m(U_m) > R_{iso}(U_m) \quad \forall m \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

2) Building the Models: The next step is to develop the response time and interference models, which allow PRIMA to consider how a change in the workload distributed to an instance impacts that instance's response time. Consider a scenario where the utilization and SF of an instance *m* are measured to be U_m and SF_m, respectively. Assume now that PRIMA wants to explore the impact of changing the workload distribution such that the utilization of the instance shifts from U_m to $U_m + \Delta U_m$. This new assignment changes both the response time of the instance as well as the severity of interference perceived by the instance. The response time and interference models together allow PRIMA to predict the response time $\hat{R_m}$ and SF $\hat{SF_m}$ at this new utilization.

To build these models, we conduct automated tests where controlled levels of interference are injected into an instance. Since we need to control interference, we again employ a dedicated instance that has the same characteristics of the production instances managed by PRIMA. We deploy both the service and the probe on this instance. We also execute within the instance a microbenchmark that emulates the load imposed by other instances competing for the PM's network bandwidth, i.e., the *interfering load*. Using the same synthetic workloads employed in Section III-B1, we vary the network utilization of the service U_{ded} to cover a desired operating region. We also vary the interfering load to mimic varying levels of interference. We monitor the mean service response time R_{ded}, the SF_{ded} value from the probe, the service utilization U_{ded}, and the utilization due to the interfering load U_{int} in each test.

We use data gathered from the tests and two dimensional piece-wise linear interpolation to build the response time model RTM. An alternative approach would have been to use a queuing model. However, we choose a data-driven approach since it does not require manual authoring of a model. The model can be directly obtained from the test data. Furthermore, as shown in Section VI-D, this approach simplifies automatic model calibration.

As shown in Eq. (2), RTM predicts the response time $\hat{R_m}$ of the service at instance *m* as a function of the workload at the instance, i.e., U_m, and the severity of interference perceived at the instance, i.e., SF_m. This model can be used to predict whether the mean response time of any given instance is above the operator specified threshold given its current U_m and SF_m values. As described next, PRIMA also uses it in conjunction with the interference model to determine the maximum workload U^{max}_m that can be assigned to instance *m* while still staying below R_{th}.

$$\hat{R_m} = RTM(U_m, SF_m) \quad \forall m \tag{2}$$

As shown in Eq. (3), the interference model IM helps PRIMA ascertain in any instance *m* the relationship between the total utilization of the shared resource, i.e., $U_{total} = U_m + U_{int}$, and its severity factor SF_m. We use one-dimensional piece-wise linear interpolation of the test data to arrive at this model. We note that IM is constructed as a "reversible" model. It can be used to obtain predictions for either U_{total} or SF_m if the other value is known.

$$U_{total} = \mathbf{IM}(SF_m) \quad \forall m \tag{3}$$

PRIMA uses RTM and IM in tandem to capture the dynamics between workload, interference, and response time. At runtime, PRIMA first uses IM to estimate the utilization U_{int} corresponding to the interfering load. We note that U_{int} can only be estimated since it cannot be directly measured by a cloud subscriber in a production deployment. To estimate U_{int}, PRIMA first uses IM to obtain U_{total} at the current measured SF value SF_m. Next, it estimates U_{int} as $U_{total} - U_m$ where U_m is the current measured utilization within the instance.

Next, PRIMA uses both models to estimate the effective capacity U_m^{max} of the instance *m*. Consider a scenario where PRIMA wants to change the workload distribution such that the service utilization changes from the current measured value of U_m to $\hat{U_m} = U_m + \Delta U_m$. Since the total utilization now changes to $\hat{U_m} + U_{int}$, PRIMA needs to estimate a new SF value, i.e., $\hat{SF_m}$, by using the reversible feature of IM. Finally, it can input $\hat{U_m}$ and $\hat{SF_m}$ into RTM to predict whether the new workload assignment violates the response time threshold. PRIMA changes ΔU_m iteratively using this process till it arrives at a utilization U_m^{max} where the predicted response time is just below R_{th}.

3) Deploying the Controller: The dedicated instance used for probe training and model building is now terminated and the PRIMA controller is deployed on the production system. We now describe the controller algorithm. The main objective of the algorithm is to distribute the workload *U* in the system among the LBG's instances such that the response time at each instance does not exceed R_{th}. The algorithm suggests adding a new instance to the LBG only when it is unable to distribute the incoming workload without one or more of the existing instances exceeding R_{th}. Similarly, the algorithm can also suggest removal of instances from the LBG, i.e., it ensures the least amount of instances are used while meeting the R_{th} targets.

The algorithm determines if there is enough effective capacity in the system to handle the system workload. Specifically, it takes as input the U_m and SF_m values for each instance *m* in the LBG to calculate the total workload *U*. Next, the algorithm uses the process described in Section III-B2 to determine the effective capacity U_m^{max} for an instance *m* in the LBG. Finally, $\hat{U}^{max}$ is calculated as the sum of the effective capacities of all instances in the LBG.

The algorithm now considers the scenario where there is insufficient capacity to handle the system workload without violating R_{th}, i.e., when *U* exceeds $\hat{U}^{max}$. PRIMA now spins an additional instance *n* and obtains its SF value SF_n at the next sampling instant. The effective capacity of this instance U_n^{max} is then calculated as outlined previously and added to $\hat{U}^{max}$ to reflect the increased capacity of the LBG. The process of spinning additional instances is continued till $\hat{U}^{max}$ exceeds *U*, i.e., there are enough instances to handle the system workload. Information about the current state of the LBG is maintained in a list denoted as LBG. Each element contains an instance identifier and the estimated effective capacity of that instance.

As a final step, the algorithm determines whether the current LBG, including any newly spun instances, can be scaled in without violating R_{th} at every instance. There are several reasons why the LBG may need to be pruned. For example, the system might be experiencing lower interference and workload than in the previous sampling interval. Furthermore, during the scale out process PRIMA might have added an instance with very low effective capacity before one with a higher effective capacity. In this scenario, there is a chance that PRIMA can relinquish the lower capacity instance without violating R_{th}.

To ensure that the minimum number of instances are used to handle the system workload *U*, PRIMA first sorts LBG in descending order of the effective capacity values. Assuming that LBG has *N* instances, the algorithm selects the first *K* instances in LBG whose aggregate effective capacities exceed *U*. If *K* is less than *N*, then the system has excess capacity. Instances corresponding to elements $K + 1$ and above are marked for deletion. The load balancer weight for any instance *m* in the group of *K* instances not marked for deletion is calculated as the ratio of the effective capacity of that instance and the sum of the effective capacities of all *K* instances in the group. Since there is enough capacity in the LBG, these weights ensure that the workload handled by each instance does not cause violation of R_{th}. Information about the instances to be deleted and the weights of the other instances is communicated by PRIMA to the load balancer.

The controller's behaviour can be fine tuned in a number of ways. First, to avoid reacting to transient transgressions of R_{th}, the controller can be instructed to wait till the problem persists over a specified number of consecutive sampling intervals. A similar restraint can be built in for the scale in process as well. Furthermore, it is also possible to incorporate a "factor of safety" by allocating a specified number of extra instances to the LBG beyond what is required to handle the system workload *U*.

4) PRIMA Overheads: The overheads of PRIMA at runtime consist of the resource consumption of the probe, the resources expended to communicate the resource utilization and interference measures from the instances to the controller, and the computation required to derive the PRIMA controller's actions. The probe executes continuously within every instance. By design [3], it consumes only a very low fraction of the resources of the instance it is monitoring. We verify this in Sections VI-C and VII. Since each instance executes a probe, increasing the number of instances causes a corresponding increase in the number of probes. However, the aggregate network bandwidth consumption of the probes as a fraction of the total bandwidth available to all instances remains the same. We note that the overhead of executing the probe is much smaller than those reported while monitoring metrics such as cache misses and response times [12], [13], [24]. The overhead related to communicating measured quantities, i.e., utilizations and SF values, is typically incurred in all load balancing and auto scaling approaches and is not specific to PRIMA. This overhead is likely to increase with sampling frequency. However, we show in Sections VI-C and VII that even with a high sampling frequency the overhead of transmitting the network utilization and SF values is not significant in our study. Finally, the controller uses simple interpolation using the data-driven models to compute the load balancer weights and determining the LBG scaling. Consequently, determining the control actions require negligible computation as we verify in Sections VI-C and VII.

IV. Calibrating the Models

We propose an automated runtime calibration of the PRIMA models to account for any model prediction errors. Prediction errors can occur due to mismatch between model input parameters during model construction and at runtime. For example, the PRIMA models can incur prediction errors when the incoming Web workload arrival pattern is different from that used to construct the models. If the predicted response time of an instance is lower than its actual response time due to such prediction errors, PRIMA will incorrectly distribute more workload to the instance than is necessary to maintain the response time threshold for that instance. As a result, the response time threshold in the LBG will not be maintained. The proposed calibration policy addresses such cases.

The calibration process is triggered whenever the predicted response time is within a configurable range $\pm\delta\%$ of the response time threshold. This is motivated by our empirical observations that suggest calibration is crucial only when the predicted response time is very close to the threshold. As part of the calibration process, PRIMA enables the collection of actual measured response time from the network service. If the actual response time exceeds the predicted response time, then PRIMA infers that the effective capacity of the instance is lower than that predicted by its models. Consequently, it estimates a revised, lower prediction for the effective capacity by using the measured response times within the models instead of the predicted response times. We note that the calibration process, and hence the monitoring of actual response times, is turned off whenever the predicted response time falls outside the $\pm\delta\%$ range.

We now discuss the details of the calibration policy. We use the response time model RTM to predict the response time of each instance in the LBG after PRIMA takes a mitigation action. If the predicted response time of an instance *m* is within $\pm\delta\%$ of R_{th}, we monitor the response time R_m and utilization U_m of *m* for a short period of time. If R_m is seen to exceed R_{th}, we plug in the *measured* R_m and U_m into RTM to get a revised, higher estimate of $\hat{SF}_m$. We next use the revised value of $\hat{SF}_m$ iteratively in RTM and IM to estimate a revised, lower effective capacity of the instance. Once this is done for all instances in the LBG whose predicted response times are close within $\pm\delta\%$ of R_{th}, the revised values of the effective capacities of these instances are used by PRIMA to recalculate the distribution of load to the LBG instances.

V. Experiment Setup

A. Private Cloud Setup

We use our private cloud setup to compare the gains of PRIMA with respect to the baseline policies, study its sensitivity to workload characteristics, and illustrate the calibration process. We use the EC2 setup described later in Section V-B to validate PRIMA on a larger setup. We limit the number of experiments in the EC2 setup due to cost considerations.

The private cloud setup consists of a dual socket Intel Xeon E5645 server host with 6 cores per socket. Multiple VM instances are consolidated on this server using Kernel-based Virtual Machine (KVM) as the virtual machine monitor. The typical time taken to start up a VM instance in our setup is 30 seconds. The server has two 1 gigabit Network Interface Cards (NICs). Each socket gets access to its own dedicated NIC. Accordingly, instances pinned on the same socket share 1 Gbps network bandwidth. Each instance is configured with 1 virtual CPU (VCPU) and 1 GB of physical memory.

The Web-based network intensive service we consider is hosted on the Apache Web server (version 2.2). Controlled interference is injected by executing the *iperf3* tool on additional Sources of Interference (SoI) VMs hosted on the same socket executing the network service instances. The probe is realized as an application deployed on the lighttpd (version 1.4.35) Web server. The probe Web server has a 1 MB file that serves as its workload. The PRIMA system initiates a download of this file once every sample period $T = 10$ seconds from a separate load generation host. The probe's response time for this download are recorded and used to calculate the SF value using Eq. (1). The probe imposes a network utilization of around 5% of the maximum network bandwidth available to an instance and causes only a modest increase of 2% to 3% in the network service's response time in our tests. We note that communicating the utilization and SF values from the instances to the PRIMA controller did not incur any significant overheads. We also verify that these overheads do not change significantly while scaling up the system, i.e., using a large number of instances.

We use another host, identical to the server host, to generate synthetic workloads. The NIC ports on this load generator host and the server host are connected via a gigabit switch, which eliminates network bottlenecks between the hosts. The *httperf* [25] workload generator is used to generate synthetic requests from the load generator host. We follow the methodology proposed by Mukherjee *et al.* to ensure that there are no bottlenecks in the load generator host [15]. Consequently, response times measured by *httperf* reflect the performance of the network service instances.

We use a modified version of *httperf* that can simultaneously generate workload to multiple instances in an LBG [26]. The PRIMA controller executes on the load generation host and communicates with *httperf* to achieve the desired load distribution across instances. Specifically, the controller collects the utilization and SF values over the sample interval *T* and predicts whether any of the instances in the LBG are violating R_{th}. If so, the controller waits for an additional *W* intervals to check if there are sustained violations. If there are sustained violations, PRIMA determines the instances in the LBG to mitigate this problem and assigns their load balancer weights. This information is passed on to our custom load balancer that uses *httperf*, which then distributes workload to the LBG instances accordingly. For this study, we use $W = 1$. We verify that PRIMA's controller algorithm caused negligible overheads in our experiments. We also verify that it is easy to integrate PRIMA along with a standard load balancer such as HAProxy in our EC2 setup.

B. EC2 Setup

As mentioned previously, we use a EC2 setup to validate PRIMA on a larger setup. For example, the EC2 setup allows us to consider scenarios where PRIMA suggests the addition or deletion of more than one instance from the LBG in one step while responding to fluctuations in interference and load. We configure up to 10 m4.large instances in a LBG for this purpose. Each m4.large instance has 2 VCPU cores, 8 GB of memory and is priced at 0.1 dollars per hour. Each VCPU core of a m4.large instance is equivalent to a 2.4 GHz Intel Xeon E5-2676 v3 processor. Each EC2 instance requires a maximum of 30 seconds to start up, similar to our private cloud setup.

The network intensive service hosted on the EC2 instances is identical to the one used in our private cloud setup. In contrast to our private cloud setup, EC2 does not permit control over the location of SoI VMs. Consequently, we run the *iperf3* tool directly inside an EC2 instance to generate controlled interfering load. Using our private cloud setup, we verified that locating the interfering load within a VM causes similar behaviour on the response time and SF values of the network service as executing the interfering load on external SoI VMs. The probe Web application used in the EC2 instances is similar to the probe described earlier in Section V-A. The probe workload is designed to incur a network utilization of around 5% of the maximum bandwidth available to the EC2 instance and causes a minimal increase of only 2% on the network service's response time.

We use up to 10 identical m4.large instances as the load generating instances. Each load generating instance hosts both the PRIMA controller and the modified version of *httperf* as discussed earlier in Section V-A. The PRIMA controller fetches the utilization and SF values from each instance in the LBG at every sampling interval. These two values are saved in a file of size 0.85 KB and communicated to the controller. Since the measured network bandwidth between a controller instance and an instance in the EC2 LBG is measured as 450 Mbps, the system size can potentially scale up to include thousands of instances without any significant communication overheads. Similar to the private cloud setup, PRIMA's controller algorithm caused negligible computational overheads in EC2.

C. Network Service Workloads

We consider two different synthetic network service workloads. The *file* workload emulates a scenario where the service hosts a single 1 GB file. Multiple concurrent users download this file by issuing HTTP requests. The rate at which HTTP requests are issued by *httperf* is varied to incur network bandwidth utilization in the range of 10% to 90% of 1 Gbps. The inter-arrival time between the HTTP requests is configured to be exponentially distributed. We use the file workload to compare PRIMA against other baseline policies in Section V-D since experiments using the file workload are simpler to design and take lesser time to complete.

We also evaluate PRIMA with a more realistic *video streaming* workload. The characteristics of the workload are summarized in Table I. To create this workload, we follow the methodologies proposed by previous researchers [27], [28] who characterized YouTube video streaming over HTTP. We choose a Zipf distribution with $\alpha = 0.8$ to model the popularity of videos. This is consistent with previous work [27], [29] that characterizes YouTube workloads. Since we have a small scale setup, we restrict the video population, i.e., number of unique videos, to 100. The distributions of video size and video duration are based on past work that characterized YouTube videos [27].

TABLE I
CHARACTERISTICS OF VIDEO WORKLOAD

Parameter	**Value**
Video popularity distribution	Zipf, $\alpha = 0.8$
Video count	100
Mean Video size	11 MB
Video Bit rate	419 Kbps
Session Arrival Process	Poisson
Session count	160
Mean Chunk time	10 seconds
Mean Chunk size	0.5 MB
Chunk pacing delay	10 seconds

We base our video streaming workload generation on a representative HTTP video streaming platform, Apple's HTTP Live streaming [30]. In this platform, videos are segmented into smaller chunks, with each client downloading chunks of the same video file using a technique called *pacing*. In this technique, clients first download chunks at full speed until a video buffer gets filled, upon which subsequent chunks are downloaded only when the buffer gets emptied and needs to be refilled. Consequently, we segment each video file on the service into 10 second chunks with an average size of 0.5 MB, which is the same size used by Apple's Live Streaming [27]. Each video download from the service is denoted as a *session*, which consists of sequential requests of chunks by an user.

We note that although this workload is representative of a realistic video streaming service such as YouTube, a few assumptions have been made for the sake of simplicity. In particular, unlike real video services, we configure our video benchmark to use only one bit rate instead of multiple. We also do not permit clients and servers to adjust video quality based on network conditions.

On the load generator host, we use the *httperf* tool to emulate video downloads through HTTP. We create 160 sessions as input to *httperf* conforming to the Zipf distribution discussed previously. Each session specifies a sequence of requests for the chunks constituting a specific video. Due to the use of the Zipf distribution some videos hosted by the service occur more than once in these sessions while some videos never get accessed. The first 3 chunks of a session are requested without pacing delays, which emulates filling of the video buffer, after which subsequent chunks are requested at the rate of one chunk every 10 seconds. To emulate multiple concurrent streaming sessions, *httperf* generates session arrivals with exponentially distributed time between successive arrivals. Each arrival emulates one of the 160 sessions input to the tool. We note that although session inter-arrival times are exponential, the overall chunk request pattern at an instance can be bursty due to the use of pacing. We use the mean response time of downloading a chunk in a session as our performance metric for this workload.

D. Baseline Policies

We consider two interference-agnostic load balancing policies typically employed in commercial cloud platforms namely, Round Robin (RR) and Least Connections (LC) [8]. The RR policy distributes the incoming requests equally among instances in the LBG. The LC policy forwards an incoming request to the instance that has the least number of active connections.

We compare PRIMA with an interference-aware policy we developed called Least SF (LSF). Similar to PRIMA, LSF uses the probe system. Unlike PRIMA, however, it does not employ the response time and interference models. Furthermore, in contrast to PRIMA, it requires response time measurements to detect violations of R_{th}. Consider a scenario where a violation is triggered due to increased interference at one or more instances. LSF attempts to rectify this problem by directing more workload to instances with lower SF values. Specifically, consider an LBG with *n* instances. Denoting the sum of SF values of these instances as SF_{total}, the policy first calculates a set of weights SF_i/SF_{total} for *i* ranging from 1 to *n*. It then assigns the largest of these values as the load balancing weight for the instance with the lowest SF value, the next largest value as the weight for the instance with the next lowest SF value and so on.

We also study a policy similar to LSF called Least Response time (LR) that uses the measured mean response times instead of the SF values to drive the load balancing. Under the LR policy, the instance with the lowest measured response time is assigned the largest fraction of the incoming workload. The weight calculation is similar to LSF except that the measured mean response times are used instead of the measured SF values.

E. Experiment Process

We first use our private cloud setup to build and validate the PRIMA models, compare PRIMA with other baseline policies, and validate PRIMA's effectiveness. To this end, we use both the file and video streaming workloads. For these 2 workloads, we train the probe to obtain a look up table for a network utilization range of 10% to 90% in steps of 10%. Each run is repeated 5 times. This results in the width of the 95% CI of the measured mean probe response time at any given utilization to be within 5% of the sample mean.

For constructing the models for the file workload, we again cover an operating utilization range of 10% to 90% in steps of 10% by varying the mean request inter-arrival times of the workload. For each step, we vary the interfering load so that the *SF* ranges from 0 to 20. The interfering load is created using the *iperf3* tool. We run 6 interfering loads per step and repeat each experiment 5 times to get a tight bound on the CI of the service response time. Each step takes approximately 200 seconds to complete. We use the test data collected from these experiments to construct the models described in Section III-B2. We follow a similar process for the video streaming workload. We note that running the *iperf3* tool within an instance to generate interfering load incurs similar behaviour as running the tool on a separate collocated instance that competes for the same shared host-level resource. The effect on the application response time and *SF* values in both cases were identical, as verified in our private cloud setup.

TABLE II
RESPONSE TIME MODEL VALIDATION

U (%)	SF	R (ms)	$\hat{R}$(ms)	Error
15	0.1	363.4	349.9	3.8
15	0.8	612.2	586.9	4.1
25	0.3	957.5	918.9	4.2
45	0.3	2207.2	2360.7	6.5
55	2.1	6220.7	6896.6	9.8

We first use the file workload to compare PRIMA with the baseline methods. Next, we validate the ability of PRIMA to maintain mean response times at each instance below R_{th} using the video streaming workload. The R_{th} values for the file and video streaming workloads are 200 ms and 1000 ms, respectively. These targets are approximately the mean response times of the file and video streaming workloads at $U = 35\%$ and $U = 30\%$, respectively when there is no interference. In experiments involving PRIMA, the minimum number of instances in the LBG is set to 1.

Using the video streaming workload, we also study the sensitivity of PRIMA's models to non-exponential session inter-arrival times. Finally, we show an experiment where measured response times can be used to calibrate model prediction errors at runtime.

We next use the EC2 setup for validating PRIMA on a real public cloud platform. We use the video streaming workload for this purpose. For constructing the PRIMA models, we cover an operating utilization range of 10% to 90% in steps of 10% by varying the mean request inter-arrival times of the workload. For each step, we vary the interfering load using *iperf3* so that the *SF* ranges from 0 to 10. The R_{th} value for the video streaming workload is 75 ms, which is approximately the mean response times of this workload at $U = 50\%$, when there is no interference. Similar to our private cloud setup, the minimum number of EC2 instances in the LBG is set to 1.

VI. RESULTS FROM PRIVATE CLOUD

A. Model Validation

We first validate the RTM using the video streaming workload. In addition to the experiments done in order to construct RTM, an additional 10 experiments are conducted by using *SF* and utilization values not covered in the data used to construct the model. Due to space constraints, a subset of results from this experiment are shown in Table II. As seen from the table, the response time of the system is more sensitive to higher values of SF and utilization. Also, note that the response time increases when either the utilization or the *SF* values increase. For example, we notice an increase in response time in the first and second rows of the table where the utilization is the same but the *SF* increases. Similarly, the response time increases when the utilization increases but the *SF* values are the same, as seen in the third and fourth rows. From the table, there is a maximum of 9.8% error in predicting the mean response time when using RTM. Over all 10 experiments conducted, the mean error in actual response time and predicted response time given by RTM is 7.2%

The interference model IM is validated in a similar manner to RTM with a mean error of 6.9% between the actual

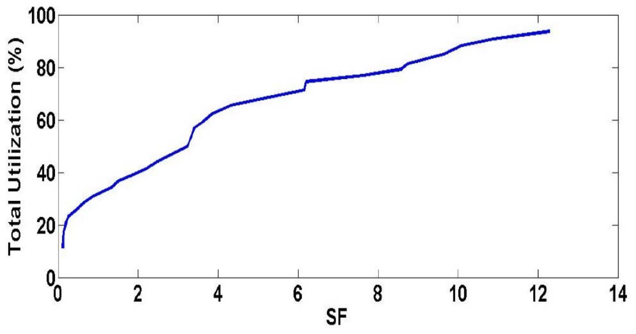

Fig. 1. Interference Model.

TABLE III
RR POLICY

Instance	Throughput	R (ms)	U (%)
1	100	5.9	21.1
2	100	18.4	45.5

TABLE IV
LC POLICY

Instance	Connections	R (ms)	U (%)
1	30	26.1	35.4
2	20	62.3	81.9

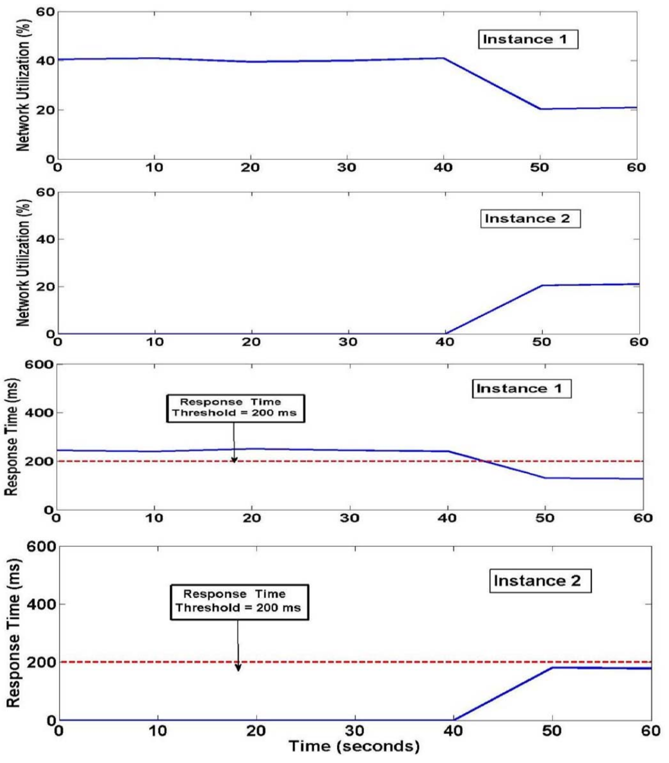

Fig. 2. LSF: Case 1.

and predicted *SF* values. Figure 1 shows IM for the video streaming workload. From the figure, the model captures the non-linear relationship between *SF* and the total utilization U_{total} imposed on the shared resource. The *SF* value of an instance is relatively insensitive at lower utilizations. However, sharp increases in the *SF* value can be observed even for relatively small increases in U_{total} at the higher utilization regions. We obtain similar results for RTM and IM for the file workload. The mean prediction errors reported by RTM and IM for the file workload are 6.8% and 6.1%, respectively.

B. Comparing PRIMA With Baseline Policies

We first focus on the RR policy. We run the network service serving the file workload on two different instances. Both instances run on different sockets of the server host. Network contention is introduced for instance 2 by running an additional SoI instance on its socket. We implement the RR algorithm which distributes the incoming traffic equally between the two instances. Table III shows the results of this experiment. The last column denoted by U in the table refers to the total network utilization of the socket hosting an instance. From the table, the interference injected by the SoI instance on instance 2 is reflected as a higher measured network utilization on this instance's socket. RR causes both instances to have the same throughput as measured in requests completed per second. However, despite the equal workload distribution the interference from the SoI instance causes the response time of instance 2 to be more than 3 times that of instance 1.

Next, we present an experiment to demonstrate the ineffectiveness of the LC policy under interference. We consider the same scenario as RR whereby instance 2 is impacted by interference. By adjusting the idle times between successive requests generated by *httperf* within a session, we create a state where instance 1 and instance 2 are servicing 30 and 20 active connections, respectively as shown in Table IV. Although instance 2 is handling fewer active connections, its mean response time is more than twice that of instance 1 due to interference from the SoI instance. This experiment shows that the number of active connections is not a good predictor of response time in the presence of interference. Hence, the LC policy will not perform well under such conditions.

These experiments demonstrate the problems faced by cloud subscribers using popular load balancing algorithms. These policies could be improved by additionally considering the total network bandwidth per socket while distributing incoming requests. However, cloud subscribers cannot typically monitor this metric directly thereby motivating the need for other policies.

We next consider the interference-aware LSF policy outlined in Section V-D when the network service is subjected to the file workload. We first discuss a scenario where LSF is able to alleviate a response time violation. The results of this experiment are shown in Fig. 2. At $t = 0$ the system has instance 1 active. The utilization of the instance U_1 is 40%. It is also suffering from interference with $SF_1 = 0.75$. The combined effect is that the measured mean response time of the instance $R_1 = 245$ ms exceeds $R_{th} = 200$ ms. LSF requests that instance 2 be spun, which becomes available at $t = 30$. One sampling interval later, it measures SF_2 as 0.75 at $t = 40$. Since the SF values for both instances are the same, LSF divides the system workload equally between both instances such that $U_1 = U_2 = 20\%$ at $t = 50$. This causes the response times of both instances to fall below R_{th}. We note

TABLE V
COMPARISON BETWEEN LR AND PRIMA

Instance	**LR**			**PRIMA**		
	U (%)	**SF**	**R (ms)**	**U (%)**	**SF**	**R (ms)**
1	17	0.3	82	28	0.6	188
2	33	1.9	**365**	22	1.1	192

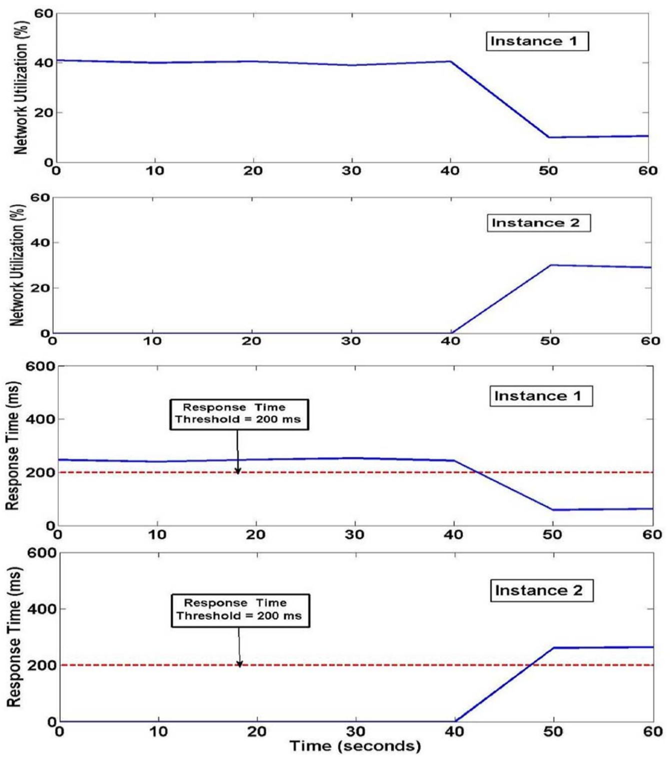

Fig. 3. LSF: Case 2.

that since workload has been removed from instance 1, SF_1 drops from 0.75 to 0.47 at $t = 50$. However, since workload has been added to instance 2 SF_2 increases from 0.75 to 1.10 at $t = 50$. Accordingly, the mean response time of instance 1 and instance 2 are 128 ms and 183 ms, respectively.

We next discuss an experiment depicted in Fig. 3 that illustrates the limitations of LSF. The experiment is similar to the previous experiment except that SF_2 is measured as 0.25 at $t = 40$, i.e., the new instance has 3 times less interference than the existing instance. Consequently, LSF directs 3 times more load to instance 2 causing U_1 to drop from 40% to 10% and U_2 to be 30% at $t = 50$. The workload shift causes SF_1 and SF_2 to be 0.48 and 0.58, respectively at $t = 50$. While the new workload assignment causes R_1 to drop below R_{th} to 59 ms, it causes R_2 to be 262 ms, i.e., above R_{th}. While LSF's decision to allocate more workload to instance 2 is correct, it assigns too much workload to that instance by not taking into account the effective capacity of the instance.

By using RTM and IM to compute the effective capacities of both instances, PRIMA arrives at a better workload distribution decision for the same scenario. PRIMA's workload distributions causes U_1 and U_2 to be 17% and 23%, respectively at $t = 50$. With this assignment, the response times of both instances fall below R_{th}. This experiment confirms the importance of using models that capture the interactions between workload and interference to estimate the effective capacity of an instance.

Finally, an experiment is conducted for demonstrating the limitations of the LR policy. Recalling from before, the LR policy monitors the response time of instances in the LBG and distributes load to each instance in reverse order of the ratio of their response times when the response time of an instance exceeds R_{th}. Two instances, instance 1 and instance 2 are run on two different sockets of the server with $U_1 = 35\%$, $SF_1 = 0.8$, $U_2 = 15\%$ and $SF_2 = 0.8$. Due to interference, the response time of instance 1 is $R_1 = 220$ ms, which exceeds R_{th}. However, due to its lower utilization instance 2 has a response time of $R_2 = 105$ ms, which is below R_{th}. Table V shows how LR and PRIMA handle this scenario. As seen in the table, the LR policy allocates $U_1 = 17\%$ and $U_2 = 33\%$, which results in $R_1 = 82$ ms and $R_2 = 365$ ms, respectively. Thus, the LR policy is unable to maintain the response time threshold in all the instances in the LBG. PRIMA uses its models to better capture the effect of interference in both instances by allocating $U_1 = 28\%$ and $U_2 = 22\%$, thus maintaining the response time threshold in both instances. This experiment further demonstrates the superiority of the PRIMA technique when compared to techniques that do not use models for mitigation.

C. PRIMA With Video Streaming Workload

In this experiment, we evaluate PRIMA's ability to scale out and scale in while facing fluctuating levels of incoming workload and interference. Table VI captures the results of this experiment. From the table, the system initially has instance 1 running on one socket of the server host. Instance 1 is configured to have no contention, i.e., $SF_1 = 0$, and incurs utilization $U_1 = 30\%$, which does not violate the response time target $R_{th} = 1000$ ms. This is seen at the 1 minute mark in Table VI.

After 100 seconds from the beginning of the experiment, we increase the incoming workload to the system so as to incur an utilization of 40% in instance 1 which causes its measured response time R_1 to exceed R_{th}. This violation is detected by PRIMA and verified as a consistent problem, as observed at the 2 minute mark in the table (violations are marked in bold in the table). Since the total incoming load at this time is higher than the effective capacity of instance 1, PRIMA mitigates this problem by requesting a scale out. This results in the addition of a new instance 2 on another socket of the server which does not face any contention. This instance becomes available after 30 seconds and PRIMA monitors $SF_2 = 0$ after an additional sampling interval, i.e., 10 seconds. Since the sum of the effective capacities of instances 1 and 2 exceeds the

TABLE VI
VIDEO STREAMING WORKLOAD

Time(min)	U_1 (%)	SF_1	R_1 (ms)	U_2 (%)	SF_2	R_2 (ms)
1	30	0	978			
2	40	0	**1623**			
3	20	0	475	20	0	481
4	30	0	985			
5	30	0.45	**1520**			
6	18	0.27	535	12	0.37	355

incoming load to the system, PRIMA does not scale out any further and distributes the incoming workload equally between both instances such that $U_1 = U_2 = 20\%$. The response times of both instances are below R_{th}, as seen at the 3 minute mark in the table.

After another 30 seconds, the incoming workload to the system is decreased from 40% to 30%. PRIMA detects this decrease in workload after 10 seconds and confirms that this behaviour is consistent after an additional 10 seconds. Since the effective capacity of instance 1 is enough to accommodate the total incoming traffic to the system at this point, PRIMA scales in by assigning all incoming workload to instance 1 and terminating instance 2. Consequently, the measured response time of instance 1 increases but is still maintained below R_{th}, as seen at the 4 minute mark in the table. These results demonstrate the effectiveness of the PRIMA technique to scale out and scale in to handle fluctuations in the incoming workload to the system.

After 40 seconds, we introduce contention on both sockets of the server by running additional SoI instances on these sockets. This introduces interference in instance 1 such that $SF_1 = 0.45$ and as a result R_1 exceeds R_{th}. PRIMA detects this R_{th} violation after 10 seconds and waits for an additional 10 seconds to confirm this behaviour. This is observed at the 5 minute mark in Table VI. At this point, PRIMA is forced to scale out again, and spins a new instance 2 as detailed earlier. This instance becomes available after 30 seconds and PRIMA monitors $SF_2 = 0.25$ after an additional 10 seconds.

PRIMA verifies that the total effective capacities of instances 1 and 2 exceeds the incoming load to the system and does not scale out any further. PRIMA distributes the incoming workload to instances 1 and 2 such that $U_1 = 18\%$ and $U_2 = 12\%$. We note that since workload is removed from instance 1 and added to instance 2, SF_1 drops from 0.45 to 0.27 and SF_2 increases from 0.25 to 0.37. PRIMA successfully mitigates the interference problem by maintaining the response times of both instances below R_{th}, as observed at the 6 minute mark in the table.

Table VI also illustrates the limitation of techniques that manage interference by imposing statically defined utilization thresholds on instances [9]. Consider a utilization target of 30%. From Table VI, this threshold would have prevented response time violations for the first 4 minutes when the instances do not suffer from interference. However, the threshold would have been ineffective beyond minute 5 when instance 1 suffers from interference. Since the utilization threshold is not exceeded, the system would have failed to scale out thereby causing sustained response time violations.

D. PRIMA Model Calibration

We first conduct a sensitivity analysis experiment to ascertain the effectiveness of PRIMA to see if it works well when the incoming workload arrival pattern changes from the one used to construct the PRIMA models. Recalling, inter-arrival times between successive sessions are exponentially distributed in the experiments used to build the models. We now explore two other arrival patterns for the video streaming workload namely, deterministic and ramp. In the deterministic workload, the inter-arrival time between sessions is fixed. The ramp workload consists of two phases. In the first phase, sessions arrive with an exponentially distributed inter-arrival time. During the second phase, the mean inter-arrival time between sessions is significantly decreased to emulate a sudden surge in incoming sessions as typically experienced by video streaming services. We configure the ramp workload to suddenly switch from a low utilization of 10% in phase 1 to a higher utilization of 60% in phase 2.

We run tests to compare the performance of PRIMA with these 3 different workloads. Table VII shows the results of this experiment. In the table, Exp, Det, and Rmp denote the exponential, deterministic and ramp workloads, respectively. Measured response times above R_{th} are shown in bold. Initially, only instance 1 is run on a socket of the host server with no contention, i.e., $SF = 0$, and incurring utilization $U_1 = 25\%$ for all 3 workloads. The response times of all 3 workloads are maintained below R_{th}, as reflected at time $t = 10$. Contention is introduced in both sockets of the server at this point such that $SF_1 = 4.3$ and $SF_2 = 0.26$. Consequently, the response time for all 3 workloads in instance 1 violates R_{th}, as seen at $t = 20$ in the table. PRIMA detects this violation and the system scales out by running instance 2 on another socket of the server. Finally, PRIMA distributes the incoming traffic between instances 1 and 2 such that $U_1 = 10\%$, $U_2 = 15\%$ and $SF_1 = 3.7$, $SF_2 = 2.3$. As seen at $t = 80$ in the table, PRIMA is successful in mitigating this problem for the exponential and deterministic workloads, but fails to do so for the ramp workload. The response times of both instances are slightly higher than R_{th} for the ramp workload. We note that this happens because the PRIMA models fail to account for the near instantaneous surge in traffic in the ramp workload. However, despite the extreme nature of the surge, PRIMA maintains the response time of the ramp workload within approximately 10% of R_{th}.

We test the automated calibration policy described in Section IV on the experiment conducted for the ramp workload. Calibration is triggered since the predicted response time is within $\delta = 10\%$ of R_{th}. We monitor the response time

TABLE VII
SENSITIVITY ANALYSIS

Time(sec)	Response Time of Instance 1 (ms)			Response time of Instance 2 (ms)		
	Exp	Det	Rmp	Exp	Det	Rmp
10	755	737	834			
20	**3882**	**3669**	**4032**			
80	989	905	**1052**	982	882	**1100**

TABLE VIII
PRIMA IN EC2: INCREASE IN WORKLOAD

Time(s)	VMs	U_1	U_2	U_3	U_4	U_5	U_6
100	1	50					
110	1	**70**					
170	2	35	35				
280	2	**60**	**60**				
340	3	40	40	40			
450	3	**60**	**60**	60			
510	4	45	45	45	45		
620	4	**75**	**75**	**75**	**75**		
670	5	**75**	**75**	**75**	**75**		
720	6	50	50	50	50	50	50

of instance 2 and use it in Eq. (2) to calculate $\hat{SF_2} = 2.7$, which reflects the instance's monitored response time. Note that $\hat{SF_2} = 2.7$ is higher than the $SF_2 = 2.3$ recorded by the probe. We use this higher value to estimate a revised effective capacity of instance 2 from $U_2 = 15\%$ to $U_2 = 13\%$. The response time of instance 2 does not exceed R_{th} when its workload is dropped to this new utilization. A similar result is obtained when applying this policy to instance 1. PRIMA can then request an additional instance to handle the excess workload pruned from these instances.

VII. RESULTS FROM EC2

We now present experiments done on our EC2 setup using the video streaming workload to validate PRIMA on a scalable public cloud platform in face of varying workload and interference conditions. As mentioned earlier, we investigate the behaviour of PRIMA to see if it can handle scenarios where adding just one additional instance to the LBG is insufficient to handle the sudden increase in workload and interference. We also show how PRIMA can scale in and terminate multiple instances from the LBG instantaneously when needed. Prior to running the experiment in EC2, we validated the PRIMA models to confirm that the mean error in prediction across both models does not exceed 6.5%.

A. Handling Sudden Increase in Workload

We first validate PRIMA in EC2 to see if it can handle a sudden increase in incoming workload to the LBG. We do not introduce interference in any instance for this part of the experiment. The results of this experiment are shown in Table VIII. If one or more instances in the LBG exceed the response time threshold, the time and the utilizations of the instances violating the threshold are marked in bold red in the table. We initially start with instance 1 that incurs an utilization $U_1 = 50\%$, which does not violate the response time target $R_{th} = 75$ ms, as seen at the 100 second mark in the table. As we continuously increase the workload to the LBG, PRIMA successfully scales out and adds up to 4 instances to the LBG such that the response time of each instance does not exceed R_{th}, as seen at the 510 second mark. At the 620 seconds mark, we increase the incoming load to the LBG substantially such that when PRIMA spins an additional instance 5 at 670 seconds, the sum of the effective capacities of all 5 instances is insufficient to accommodate the incoming load. PRIMA correctly recognizes this and spins an additional instance 6 before redistributing the incoming load. Only after PRIMA estimates that the sum of the effective capacities of the 6 instances in the LBG exceeds the total incoming load does it distribute the workload equally between all instances. The response time of each instance in the LBG after this distribution does not exceed R_{th}, as seen at the 720 seconds mark. This experiment validates how PRIMA can dynamically add multiple instances to the LBG to accommodate a sudden increase in workload.

B. Handling Heavy Increase in Interference

In the next phase of the experiment, we increase the interference in the LBG by varying the interfering load from the workload generator to the *iperf3* tool running inside each instance. The results of this experiment are shown in Table IX. The values for SF_5, SF_6, SF_7 and SF_8 are not shown in the table since these values are always set to 0. Interference is introduced at the 830 seconds mark such that $SF_1 = 6.3$, $SF_2 = 6.2$ and $SF_3 = 6.4$. Since the effective capacities of instances 1, 2 and 3 are lowered due to contention, the incoming load exceeds the sum of the effective capacities of the 6 instances in the LBG. PRIMA spins an additional instance 7 to mitigate this violation at the 880 second mark. However, since the sum of the effective capacities of the 7 instances is still lower than the total incoming load, PRIMA scales out again and adds instance 8 to the LBG. PRIMA then redistributes the incoming workload such that $U_1 = U_2 = U_3 = 19.5\%$ and $U_4 = U_5 = U_6 = U_7 = U_8 = 48.3\%$. The response time of each instance is maintained below R_{th}, as seen at the 930 seconds mark. We next increase the interference in instance 4 at the 1040 second mark such that $SF_4 = 7.5$. As a result, PRIMA has to scale out again and adds instance 9 to the LBG at the 1090 second mark. However, we introduce contention in instance 9 such that $SF_9 = 1.8$. PRIMA correctly predicts that further scaling out is needed and spins instance 10 where $SF_{10} = 0.5$. PRIMA then redistributes the incoming load such that $U_1 = U_2 = U_3 = 19.5\%$, $U_4 = 9\%$, $U_5 = U_6 = U_7 = U_8 = 48.3\%$, $U_9 = 12\%$, and $U_{10} = 27.3\%$, as seen at the 1140 seconds mark. The response times of all 10 instances in the LBG are just below the R_{th} value at this point. This phase demonstrate how PRIMA successfully scales out

TABLE IX
PRIMA IN EC2: INCREASE IN INTERFERENCE

Time(s)	VMs	$\mathbf{U_1}$	$\mathbf{SF_1}$	$\mathbf{U_2}$	$\mathbf{SF_2}$	$\mathbf{U_3}$	$\mathbf{SF_3}$	$\mathbf{U_4}$	$\mathbf{SF_4}$	$\mathbf{U_5}$	$\mathbf{U_6}$	$\mathbf{U_7}$	$\mathbf{U_8}$	$\mathbf{U_9}$	$\mathbf{SF_9}$	$\mathbf{U_{10}}$	$\mathbf{SF_{10}}$
830	6	**50**	6.3	**50**	6.2	**50**	6.4	50	0	50	50						
880	7	**50**	6.3	**50**	6.2	**50**	6.4	50	0	50	50						
930	8	19.5	2.4	19.5	2.3	19.5	2.3	48.3	0	48.3	48.3	48.3	48.3				
1040	8	19.5	2.4	19.5	2.3	19.5	2.3	**48.3**	7.5	48.3	48.3	48.3	48.3				
1090	9	19.5	2.4	19.5	2.3	19.5	2.3	**48.3**	7.5	48.3	48.3	48.3	48.3		1.8		
1140	10	19.5	2.4	19.5	2.3	19.5	2.3	9	2.7	48.3	48.3	48.3	48.3	12	2.4	27.3	1.7

TABLE X
PRIMA IN EC2: SCALING IN

Time(s)	VMs	$\mathbf{U_1}$	$\mathbf{SF_1}$	$\mathbf{U_2}$	$\mathbf{SF_2}$	$\mathbf{U_3}$	$\mathbf{SF_3}$	$\mathbf{U_4}$	$\mathbf{SF_4}$	$\mathbf{U_5}$	$\mathbf{U_6}$	$\mathbf{U_7}$	$\mathbf{U_8}$	$\mathbf{U_9}$	$\mathbf{SF_9}$	$\mathbf{U_{10}}$	$\mathbf{SF_{10}}$
1250	10	19.5	0.0	19.5	0.0	19.5	2.3	9	2.7	48.3	48.3	48.3	48.3	12	2.4	27.3	1.7
1270	6	50	0.0	50	0.0					50	50	50	50				

in face of increased interference. They also show how PRIMA can dynamically add more instances to the LBG if the newly added instance is itself facing a high level of interference.

C. Scaling in Multiple Instances Concurrently

In the final phase of this experiment, we remove interference in instances 1 and 2 such that $SF_1 = SF_2 = 0$ at the 1250 second mark, as seen in Table X. PRIMA estimates that the sum of the effective capacities of instances 1, 2, 5, 6, 7, 8 are sufficient to accommodate the incoming workload. As a result, PRIMA scales in, i.e., terminates, instances 3, 4, 9, 10 from the LBG. PRIMA then redistributes the incoming load such that $U_1 = U_2 = U_5 = U_6 = U_7 = U_8 = 50\%$ and the response time of each instance in the LBG is maintained below R_{th}, as seen at the 1270 second mark. This phase validates that PRIMA can dynamically scale in and remove multiple instances from the LBG concurrently in one step.

VIII. CONCLUSION AND FUTURE WORK

This paper addresses the challenging problem of subscriber-driven cloud interference mitigation by presenting a novel technique called PRIMA. PRIMA uses data-driven models that consider the joint impact of workload and interference on the response times of resource instances in a load balanced service system. The models are used at runtime to intelligently scale the system and distribute load across all its instances such that the response time of each instance is below an operator specified threshold. To the best of our knowledge, we are not aware of other subscriber-driven interference mitigation techniques that provide an explicit mechanism to meet response time targets while using the least possible number of instances. Furthermore, in contrast to similar techniques, PRIMA does not require hardware counter and service response time measurements, which can impose large overheads.

Results show that PRIMA's model-based approach outperforms approaches such as LSF that do not use models but rather rely solely on monitoring the instances. Using a realistic video streaming workload in private and public cloud environments, we show that PRIMA is able to respond to fluctuations in both workload and interference. Furthermore, results show that PRIMA is robust towards workload assumptions made while building the models. Finally, results show that our modeling approach allows calibration at runtime to effectively rectify the impact of any modeling errors.

Future work will integrate workload prediction techniques into PRIMA to make it more proactive. The probe system used by PRIMA is capable of detecting interference for other PM resources, e.g., processors and memory. Future work can exploit this feature to derive response time and interference models for specific PM resources. This will in turn allow PRIMA to adapt its behaviour depending on the resource that is experiencing the most interference. Finally, we will focus on integrating PRIMA with vertical scaling strategies as part of future work.

REFERENCES

[1] J. Mukherjee, D. Krishnamurthy, J. Rolia, and C. Hyser, "Resource contention detection and management for consolidated workloads," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manag. (IM)*, 2013, pp. 294–302.

[2] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Proc. IEEE INFOCOM*, 2010, pp. 1163–1171.

[3] J. Mukherjee, D. Krishnamurthy, and M. Wang, "Subscriber-driven interference detection for cloud-based Web services," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 1, pp. 48–62, Mar. 2017.

[4] S. A. Javadi and A. Gandhi, "DIAL: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing," in *Proc. IEEE Int. Conf. Auton. Comput. (ICAC)*, Columbus, OH, USA, 2017, pp. 135–144.

[5] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Workload-aware provisioning in public clouds," *IEEE Internet Comput.*, vol. 18, no. 4, pp. 15–21, Jul./Aug. 2014.

[6] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 187–198, 2012.

[7] E. Baik, A. Pande, Z. Zheng, and P. Mohapatra, "VSync: Cloud based video streaming service for mobile devices," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, San Francisco, CA, USA, 2016, pp. 1–9.

[8] K. Al Nuaimi, N. Mohamed, M. Al Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: Challenges and algorithms," in *Proc. Symp. Netw. Cloud Comput. Appl.*, London, U.K., 2012, pp. 137–142.

[9] A. K. Maji, S. Mitra, and S. Bagchi, "ICE: An integrated configuration engine for interference mitigation in cloud services," in *Proc. IEEE Int. Conf. Auton. Comput.*, Grenoble, France, 2015, pp. 91–100.

[10] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "PREPARE: Predictive performance anomaly prevention for virtualized cloud systems," in *Proc. IEEE 32nd Int. Conf. Distrib. Comput. Syst.*, 2012, pp. 285–294.

[11] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: Transparently identifying and managing performance interference in virtualized environments," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*, 2013, pp. 219–230.

[12] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan, "QMON: QoS-and utility-aware monitoring in enterprise systems," in *Proc. IEEE Int. Conf. Auton. Comput.*, 2006, pp. 124–133.

[13] V. Horkỳ, J. Kotrč, P. Libič, and P. Tåma, "Analysis of overhead in dynamic java performance monitoring," in *Proc. 7th ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, 2016, pp. 275–286.

[14] J. Mukherjee and D. Krishnamurthy, "Subscriber-driven cloud interference mitigation for network services," in *Proc. IEEE/ACM 26th Int. Symp. Qual. Service (IWQoS)*, 2018, pp. 1–6.

[15] J. Mukherjee, M. Wang, and D. Krishnamurthy, "Performance testing Web applications on the cloud," in *Proc. IEEE 7th Int. Conf. Softw. Test. Verification Validation Workshops*, 2014., pp. 363–369.

[16] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling scheduling speed and quality in large shared clusters," in *Proc. Symp. Cloud Comput.*, 2015, pp. 97–110.

[17] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 185–198.

[18] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic resource scaling for multi-tenant cloud systems," in *Proc. Symp. Cloud Comput.*, 2011, pp. 1–14.

[19] X. Ren, R. Lin, and H. Zou, "A dynamic load balancing strategy for cloud computing platform based on exponential smoothing forecast," in *Proc. Int. Conf. Cloud Comput. Intell. Syst.*, Beijing, China, 2011, pp. 220–224.

[20] J. F. Pérez, R. Birke, M. Björkqvist, and L. Y. Chen, "Dual scaling VMs and queries: Cost-effective latency curtailment," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Atlanta, GA, USA, 2017, pp. 988–998.

[21] E. B. Lakew, R. Birke, J. F. Perez, E. Elmroth, and L. Y. Chen, "SmallTail: Scaling cores and probabilistic cloning requests for Web systems," in *Proc. IEEE Int. Conf. Auton. Comput. (ICAC)*, 2018, pp. 31–40.

[22] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Proc. IEEE Netw. Oper. Manag. Symp.*, 2012, pp. 204–212.

[23] Z. Gong, X. Gu, and J. Wilkes, "PRESS: Predictive elastic resource scaling for cloud systems," in *Proc. Int. Conf. Netw. Service Manag.*, Niagara Falls, ON, Canada, 2010, pp. 9–16.

[24] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proc. 3rd ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, 2012, pp. 247–248.

[25] D. Mosberger and T. Jin, "httperf—A tool for measuring Web server performance," in *Proc. ACM SIGMETRICS Perform. Eval. Rev.*, 1998, pp. 31–37.

[26] R. Hashemian, D. Krishnamurthy, M. Arlitt, and N. Carlsson, "Improving the scalability of a multi-core Web server," in *Proc. 4th ACM/SPEC Int. Conf. Perform. Eng.*, 2013, pp. 161–172.

[27] J. Summers, T. Brecht, D. Eager, and B. Wong, "Methodologies for generating HTTP streaming video workloads to evaluate Web server performance," in *Proc. 5th Annu. Int. Syst. Storage Conf.*, 2012, pp. 1–12.

[28] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao, "YouTube everywhere: Impact of device and infrastructure synergies on user experience," in *Proc. SIGCOMM*, 2011, pp. 345–360.

[29] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "YouTube traffic characterization: A view from the edge," in *Proc. SIGCOMM*, 2007, pp. 15–28.

[30] A. Begen, T. Akgul, and M. Baugher, "Watching video over the Web: Part 1: Streaming protocols," *IEEE Internet Comput.*, vol. 15, no. 2, pp. 54–63, Mar./Apr. 2011.

Joydeep Mukherjee received the M.Sc. and Ph.D. degrees from the University of Calgary. He is a Post-Doctoral Fellow with York University, Toronto. His research interests include software performance engineering, virtualized systems, cloud computing, and Internet-of-Things.

Diwakar Krishnamurthy is a Professor with the University of Calgary. His research interests are focused on the performance evaluation of software systems. He is currently involved in research projects related to cloud computing, virtualization technologies, big data analytics, and healthcare simulation.