Subscriber-Driven Cloud Interference Mitigation for Network Services

Joydeep Mukherjee
University of Calgary
Email: jmukherj@ucalgary.ca

Diwakar Krishnamurthy
University of Calgary
Email: dkrishna@ucalgary.ca

***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 the 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. This paper outlines a solution called PRIMA that subscribers of such platforms, i.e., network service operators, can deploy to ensure a specified end user response time target is met even in the face of fluctuations in workload and interference. 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. PRIMA 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 provides an explicit mechanism to guarantee that the specified response time threshold is met at every resource instance assigned to a service. Furthermore, in contrast to these approaches PRIMA can help an operator avoid using more instances than necessary for handling the observed workload and interference.**

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]. 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 [6].

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 a cloud 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 [7], do not explicitly take into account how individual instances within the LBG are impacted by interference at any given 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.

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. We present a model-based technique called Performance Interference Management Approach (PRIMA) that addresses this challenge. PRIMA exploits data-driven models derived by deploying a subscriber-oriented interference estimation system developed in our previous work [4]. 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

978-1-5386-2542-2/18/31.00$B)2018IEEE$

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 optimal split of incoming traffic between the current instances in the LBG such that an operator-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.

The key novel contributions of our work with respect to other subscriber-oriented interference mitigation solutions proposed in literature [8], [9] are as follows. First, unlike these solutions PRIMA provides an explicit mechanism to guarantee that the target response time requirement is met by every instance in an LBG. Second, existing approaches do not focus on automatically expanding and shrinking the LBG in response to fluctuations in workload and interference. In contrast, PRIMA's models ensure that the system will use only 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. Experiment results obtained in our private cloud setup indicate that PRIMA is agile in responding to fluctuations in both workload and interference.

II. Related Work

Previous studies have devised techniques that cloud providers can exploit to detect and mitigate the impact of interference [11], [12]. However these techniques rely on collecting detailed PM-level metrics which cloud subscribers typically do not have access to. 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 [8]. 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. Also, since it focuses only on load balancing ICE does not support automated scale in and scale out of an LBG. Javadi *et al.* develop a subscriber-driven load balancing technique called DIAL that considers the impact of interference [9]. 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.

III. PRIMA

A. Overview of PRIMA

A system managed by PRIMA consists of a load balancer, an LBG, the probe system [4] 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 an operator-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. Due to our focus on network intensive services, 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 Sec. 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.

We note that PRIMA does not require monitoring of instance response times, which can be expensive. 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 public cloud platforms such as Amazon EC2.

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 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 $R_m(U_m)$. If $R_m(U_m)$ 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 $R_m(U_m)$ is significantly higher than $R_{iso}(U_m)$, which implies the impact of interference is severe.

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

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 and SF. 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 Sec. 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**. 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_m^{max} that can be assigned to instance m while still staying below R_{th}.

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

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 increase the workload distribution such that the service utilization increases from the current measured value of U_m to $\hat{U_m} = U_m + \Delta U_m$. The total utilization now increases to $\hat{U_m} + U_{int}$ and this increased utilization makes the instance more sensitive to interference, i.e., its SF value will likely increase. Consequently, PRIMA needs to estimate a new SF value, i.e., $\hat{SF_m}$, by using this new value of the total utilization and exploiting the reversible feature of **IM**. Finally, it can input $\hat{U_m}$ and $\hat{SF_m}$ estimated in this manner 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 algorithm first 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 Sec. 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 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 in the list contains an instance identifier and the estimated effective capacity of that instance.

As a final step, the algorithm determines whether there is excess capacity in the system. Specifically, it checks whether the current **LBG**, including any newly spun instances, can be scaled in without violating R_{th} at any 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, i.e., an instance suffering from heavy interference, 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 equal or 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. 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.

IV. Experiment Methodology

A. Experiment Setup

Our 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 VM 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% 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 PRIMA did not incur any significant overheads.

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* [13] workload generator is used to generate synthetic requests from the load generator host. We follow the methodology proposed in our previous work to ensure that there are no bottlenecks in the load generator host [14]. 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 [15]. 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 *httperf*, which modifies its workload generation accordingly. For this study, we use $W = 1$. We note that PRIMA's controller algorithm caused negligible overheads in our experiments.

B. Network Service Workload

We evaluate PRIMA with a 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 [16], [17] who characterized YouTube video streaming over HTTP. We use freely available stock videos [18] to create the workload. We choose a Zipf distribution with $\alpha = 0.8$ to model the popularity of videos. This is consistent with previous work [16], [19] 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 [16]. Finally, following previous work on YouTube [16], [19] we choose a fixed bitrate of 419 Kbps for all videos, which represents 10 seconds of video consuming 0.5 MB of storage.

Using these settings, we are able to achieve up to 66 concurrent video sessions in our experiments. By varying the mean session inter-arrival time, we are able to utilize the network bandwidth in the range of 10% to 90%. We note

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

that the mean response times to download video segments for utilizations beyond 60% are greater than 30 seconds, which is considered excessive in realistic video streaming platforms. Hence we assume a range of operation of 10% to 60% for the video streaming workload.

V. Results

In this experiment, we evaluate PRIMA's ability to scale out and scale in while facing fluctuating levels of incoming workload and interference. Table II 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 II.

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 interference 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 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

TABLE II
Video Streaming Workload

Time(min)	$\mathbf{U_1}$ (%)	$\mathbf{SF_1}$	$\mathbf{R_1}$ (ms)	$\mathbf{U_2}$ (%)	$\mathbf{SF_2}$	$\mathbf{R_2}$ (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

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 II. 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.

VI. Conclusions 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. Using a realistic video streaming workload, we show that PRIMA is able to respond to fluctuations in both workload and interference.

Future work will consider optimizing the cost of load balancing by exploiting the simultaneous use of different types of instances. We will also look into integrating workload prediction techniques into PRIMA to make it more proactive.

References

[1] J. Mukherjee, D. Krishnamurthy, J. Rolia, and C. Hyser, "Resource contention detection and management for consolidated workloads," in *IM, 2013*.
[2] G. Kousiouris, T. Cucinotta, and T. Varvarigou, "The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks," *J. Syst. Softw.*
[3] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," ser. INFOCOM, 2010.
[4] J. Mukherjee, D. Krishnamurthy, and M. Wang, "Subscriber-driven interference detection for cloud-based web services," *IEEE TNSM*, 2016.
[5] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *INFOCOM, 2014 Proceedings IEEE*.
[6] E. Baik, A. Pande, Z. Zheng, and P. Mohapatra, "Vsync: Cloud based video streaming service for mobile devices," in *IEEE INFOCOM 2016*, 2016.
[7] K. A. Nuaimi, N. Mohamed, M. A. Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: Challenges and algorithms," in *2012 Second Symposium on Network Cloud Computing and Applications*.
[8] A. K. Maji, S. Mitra, and S. Bagchi, "Ice: An integrated configuration engine for interference mitigation in cloud services," in *ICAC, 2015*.
[9] S. A. Javadi and A. Gandhi, "Dial: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing," in *ICAC, 2017*.
[10] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," ser. ICDCS '12.
[11] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," ser. USENIX ATC'13.
[12] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: reconciling scheduling speed and quality in large shared clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing, 2015*.
[13] httperf, "Http performance measurement tool,."
[14] J. Mukherjee, M. Wang, and D. Krishnamurthy, "Performance testing web applications on the cloud," in *ICSTW, 2014*.
[15] R. Hashemian, D. Krishnamurthy, M. Arlitt, and N. Carlsson, "Improving the scalability of a multi-core web server," ser. ICPE '13.
[16] J. Summers, T. Brecht, D. Eager, and B. Wong, "Methodologies for generating http streaming video workloads to evaluate web server performance," ser. SYSTOR '12.
[17] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao, "Youtube everywhere: Impact of device and infrastructure synergies on user experience," ser. IMC '11.
[18] "Pexel videos:free stock videos."
[19] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: A view from the edge," ser. IMC '07.