Subscriber-Driven Interference Detection for Cloud-Based Web Services

Joydeep Mukherjee, Diwakar Krishnamurthy, and Mea Wang

***Abstract*—Web services are now increasingly being hosted on public cloud infrastructure as a service platforms such as the Amazon Web service elastic compute cloud (EC2). However, previous studies have shown that the virtualized infrastructure used in public clouds can introduce contention among virtual machines (VMs) for shared physical host resources eventually leading to performance problems. Subscribers in a public cloud platform typically do not have access to metrics that can directly quantify the adverse impact of such inter-VM interference on Web service response times. We present a software probe based system to address this limitation. The probe is a lightweight application that runs on each Web service VM that needs to be monitored. We periodically measure the probe's response time on a monitored VM. We then compare this response time with the probe's previously recorded baseline no-interference response time when it executes in isolation on a VM of the same type. Statistically significant increase in the probe's response time from the baseline is used to detect interference. The probe also indicates the type of contention at the physical host that causes the interference. This information can be exploited by a subscriber to mitigate the problem. Results show that our approach is quite effective over two different cloud platforms and a wide variety of workload scenarios. In particular, results indicate that Web service instances hosted on EC2 suffer from interference. Our probe was able to detect 93% of performance degradations triggered by such interference. In all these cases, the probe imposed an average overhead of only 3%–4% on the mean response time of the Web service being monitored.**

***Index Terms*—Cloud computing, virtualization, data center management, software performance engineering.**

I. Introduction

THE NOTION of unlimited computing resources and the pay-as-you-use model of cloud computing has attracted many Web applications, such as Netflix and Pinterest. Resource virtualization and sharing in the cloud is achieved through running multiple virtual machine (VM) instances on each physical machine (PM) in data centers. However, such virtualized infrastructure can cause performance degradation when the VM instances interfere with each other by competing for shared PM resources [1]–[3]. Cloud management activities, such as VM scheduling, startup, and migration, can also cause performance deterioration [4]. Such performance issues raise challenges for cloud-based Web services. Service users expect immediate response when browsing a Web page and long response times can cause frustration. Since most public cloud systems do not typically provide performance guarantees, cloud subscribers need techniques to detect performance interference so that they can take remedial measures, e.g., upgrade their VM instances or switch to a different cloud provider.

Most earlier studies for detecting performance interference have focused on cloud providers and not on cloud subscribers. Specifically, such studies [5], [6] have been exploiting host PM level metrics such as cache misses and Clock Cycles per Instruction (CPI) to detect interference. Unfortunately, most commercial cloud platforms do not implement such methods thereby necessitating subscriber oriented approaches.

Interference detection is a much more challenging problem for a subscriber than it is for a provider. Unlike providers, subscribers generally do not have access to PM level metrics such as CPI that can directly quantify contention between VMs. For example, Amazon Web Services (AWS) offers a monitoring service called Amazon CloudWatch [7] to collect and track a VM's resource usage, e.g., CPU consumption, but nothing at the PM level to track the resource usage of *other* VMs on the PM. Consequently, a subscriber has to rely on measures that can be collected within their Web service VMs to detect interference. In particular, a subscriber oriented approach has to address the difficult problem of selecting an appropriate measure that can distinguish between performance degradations caused by interference from those caused merely due to change in the Web service's workload.

To get around the lack of PM level metrics, public cloud subscribers such as Netflix have used VM level resource usage metrics such as the *CPU steal* metric as a proxy to infer contention for PM resources [8]. However, we find in this work that it is difficult to characterize the *severity* of interference on Web service response times using such metrics alone. An alternative approach is to use thresholds based on Web service level metrics such as request response times and throughput. For example, additional VM instances can be automatically provisioned to a Web service using tools such as AWS's Auto Scaling [9] if the service's response times increase beyond a threshold. However, such a threshold based approach cannot distinguish between performance degradations caused by workload fluctuations, e.g., a change in the mix of requests served by the Web service, from those caused by interference. As a result, a subscriber cannot, for example, quantify the

Manuscript received June 27, 2016; revised December 1, 2016; accepted December 2, 2016. Date of publication December 21, 2016; date of current version March 9, 2017. The associate editor coordinating the review of this paper and approving it for publication was H. Lutfiyya.

The authors are with the University of Calgary, Calgary, AB T2N 1N4, Canada (e-mail: jmukherj@ucalgary.ca; dkrishna@ucalgary.ca; meawang@ucalgary.ca).

Digital Object Identifier 10.1109/TNSM.2016.2642838

1932-4537 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

financial impact of having to provision the additional instances to handle interference.

Others have used machine learning algorithms in conjunction with Web service level metrics to isolate the impact of interference [10], [11]. However, these techniques require a fully instrumented Web service to record response times of every incoming request. This might not be always feasible since such fine-grained instrumentation can impose variable overheads, e.g., increased overheads when system has a higher request throughput. Additionally, while these techniques can detect interference they are not designed to indicate the type of contention at the physical host that causes the interference. For example, a cloud subscriber will not only be interested in detecting contention in the cloud platform but will also want to know which shared resource, e.g., processors, network, or disk, is encountering contention.

We address these issues through an alternative subscriber oriented approach that utilizes a software probe. The technique provides a direct reflection of how contention for a particular PM-level resource can impact the response time of a virtualized Web service that relies on that resource. It requires a subscriber to execute a low overhead application, which we refer to as the *probe*, inside each Web service VM instance to be monitored. The probe is designed to exercise a specific PM-resource, e.g., processors, whose contention the subscriber wants to characterize. The subscriber first runs this probe application in isolation on a separate reference VM instance with the same specifications as the monitored VM instance. By subjecting the probe application in the reference instance to a micro-benchmark workload, the subscriber can estimate for various VM resource utilization levels the baseline response times of the probe application when it is not impacted by performance interference. Next, the probe application is deployed within the monitored instance and periodically subjected to the same micro-benchmark workload. Every time the probe is subjected to this workload, its response time and the resource utilization of the monitored instance are recorded. Any statistically significant degradation in this response time from the previously recorded baseline response time at this utilization is flagged as an indication of contention for the shared PM resource. Additionally, our system allows the subscriber to control the characteristics of the probe application and its micro-benchmark workload to infer the types of resource contention, e.g., processor, network, or disk, occurring on the PM. The subscriber can use information reported by the probe on the type and severity of contention along with knowledge of their application's workload to mitigate the impact of interference.

We experimentally evaluate the approach on our private cloud platform as well as on different types of AWS Elastic Compute Cloud (EC2) instances. In particular, we show that standard resource usage metrics reported by a VM's operating system may alone not be effective in capturing the extent to which contention can impact a Web service's response time thereby motivating our approach. We also show that performance interference is a real phenomenon in public cloud systems and that our approach is very effective in identifying such interference. Specifically, we show that an EC2 Web service can exhibit very different performance for the same workload, indicating the presence of interference. Using two different sets of workloads, we show that our probe is able to detect approximately 93% of EC2 induced performance degradations and predict their severity. The study also shows that the probe is able to differentiate performance degradations caused by interference from those due to workload fluctuations. Furthermore, compared to other approaches based on monitoring PM level hardware counters [12], the probe imposes modest per-core utilization and response time overheads of only 3% - 4% on an average. We also show that our probe design can generalize to identify various types of PM resource contention, e.g., contentions for processors and network. Finally, we show that the probe system will incur only a very minor cost increase to a cloud subscriber.

The rest of the paper is organized as follows. Section II provides an overview of existing work. In Section III, we present the system architecture of the software probe. The experiment setup for validating the probe is described in Section IV. We provide a justification for the probe approach using a private cloud platform in Section V. The probe is evaluated on EC2 in Section VI. We present a sensitivity analysis for the probe system in Section VII. Section VIII concludes the paper.

II. Related Work

Many studies have identified that batch and scientific applications hosted in the cloud suffer from performance variability [13]–[15]. Iosup *et al.* [13] analyze the performance of scientific workloads on EC2. Iosup *et al.* [14] study the performance of the AWS and Google App Engine (GAE) platforms for batch workloads over a period of two months. Both studies conclude that application performance exhibits time-dependent characteristics with daily and monthly patterns.

Existing studies have also focused on performance variability of cloud-based Web services. Dejun *et al.* [16] study the performance variability of three synthetic Web applications on a small EC2 instance. The authors find that the performance of the small instance is relatively stable over the long term. However, they observe high performance variability in small instances across different availability zones in AWS. In contrast, Mukherjee *et al.* [17] observe significant performance variability for Web applications hosted on all types of EC2 instances even within the same availability zone.

A significant number of studies have focused on devising techniques that cloud providers can use for detecting inter-VM performance interference [2], [5], [6], [18]–[25]. Mukherjee *et al.* [2], [12] develop a probe based system to detect performance interference in their private cloud setup. In contrast to our work, their probe is intended to be run by a cloud provider and hence needs to execute directly on the PM. This is not possible for a cloud subscriber in systems such as EC2. Fu [5] proposes a provider oriented approach that uses PM level metrics such as CPU usage, memory and swap utilization, and page faults to detect performance anomaly. Blagodurov *et al.* [6] also propose a similar contention detection approach that uses PM level metrics such as

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

cache miss rates. As discussed previously, subscribers typically do not have access to PM level metrics.

In contrast to the above methods, our focus is on enabling a cloud subscriber to detect performance degradation in a public IaaS cloud in the absence of PM or VM level metrics. As discussed in Section I, subscriber oriented interference detection is a challenging problem. Casale *et al.* [8] propose an approach aimed at enabling subscribers to detect the impact of interference on the performance of batch applications. By continuously monitoring the execution times of a set of batch benchmarks and the average *CPU steal* metric of a VM instance, the authors are able to identify 30% to 40% of the performance interference events affecting the VM. In contrast to their work, our work focuses on interactive Web applications. We find in our study that metrics such as *CPU steal* alone might not be effective in detecting the severity of response time degradations due to interference.

Similar to our work, Maji *et al.* [10] propose an interference detection approach for Web applications that is suited for subscribers of public clouds. The authors employ a supervised decision tree classifier that uses a simultaneous sharp increase in VM CPU utilization, a decrease in the Web application's throughput, and an increase in the application's response time as a signature to infer interference. Amannejad *et al.* [26] apply a collaborative filtering based machine learning technique on the response times collected from a Web application in order to detect interference. The response times required for the method are obtained by intercepting incoming and outgoing requests using a proxy server. Javadi *et al.* [27] propose a subscriber oriented method that can use application metrics such as response time in combination with a queueing model of the application to detect and quantify interference.

In contrast to these three methods, our approach does not rely on monitoring the Web service's response times or throughput. As a result, it does not require fine-grained application instrumentation or proxy servers to record response times. Furthermore, unlike the approaches of Maji *et al.* and Amannejad *et al.*, the probe can be designed to provide insights into the type of PM level contention that is triggering a degradation in the Web application's response times. For example, if the probe indicates frequent contention for the network and the Web application's workload is network-intensive, then a subscriber can ameliorate the problem by upgrading to an instance that provides more network bandwidth. Finally, in contrast to the method of Javadi *et al.*, our approach does not require the construction and parametrization of queueing models, which can be challenging for real-life Web services.

III. Architecture of the Probe

Fig. 1 shows the architecture of our probe based system for detecting performance interference. It has three main components namely, a probe application, a probe workload generator, and a controller. The probe application is installed on the instance hosting the Web application, referred to as the *monitored instance*. The probe application is also installed alone on a separate *reference instance* that has the same specifications as the monitored instance. The probe workload generator

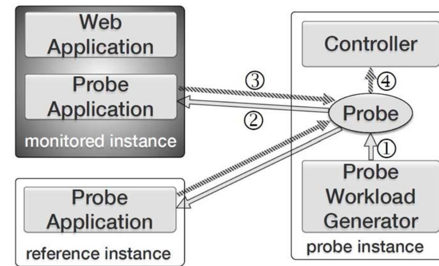

Fig. 1. System architecture of the probe.

and the controller are hosted on a separate *probe instance* that submits a synthetic micro-benchmark workload to the probe applications on the monitored and reference instances. We refer to this workload as the *probe workload*. Interference is detected by analyzing the fluctuations in the performance recorded for the probe workload.

The rationale for the system architecture is as follows. Since the objective is to detect performance degradation due to problems in the cloud platform, the probe system must meet two key design requirements. First, the probe should either ignore the Internet delay or bypass the Internet. We choose to place the probe in the same platform where the Web application is hosted, e.g., EC2 in one of our case studies. Hence, any performance fluctuation recorded by the probe reflects performance problems in the cloud or the stress from the Web application's workload. Second, the probe should distinguish performance interference effects from normal performance degradations due to workload surges or changes in the workload request mix that are experienced by the Web application. We now explain how our design achieves this requirement.

To apply our approach, a subscriber first needs to identify the PM resources whose contention is of interest. For the sake of simplifying this discussion, we assume that the subscriber is interested in monitoring contention for one resource and we refer to this resource as the *contention resource*. For example, if the Web application being monitored is predominantly processor-intensive, then the subscriber might wish to focus only on contention for PM processors since such contention can adversely impact application response times. We note that our approach can generalize to consider multiple contention resources as we show later in Section VII.

Next, the cloud subscriber has to select a standard micro-benchmark probe application specific to the contention resource being monitored and estimate the baseline performance of this probe application when there is no contention for that resource from other VM instances. A reference instance with the same specifications as that of the monitored instance is used for the purpose of obtaining this baseline data. Depending on the cloud platform and the contention resource being monitored, there are several methods to ensure that there is no contention for the resource from other VMs when the baseline performance of the probe application is being recorded. We discuss some of these methods in more detail later. When obtaining the baseline performance, our system automatically varies the utilization of the contention resource by the reference instance by employing

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

a *background workload*. Collecting data at various utilization levels allows our system to differentiate performance degradations due to workload changes experienced by the Web application from cloud platform induced performance interference. The range of utilizations is chosen to progressively go from scenarios where the reference instance lightly loads the contention resource to scenarios where it places heavy load on that resource.

For each utilization level explored, the probe workload is submitted to the probe application on the reference instance and response times are recorded for the requests constituting this workload. Since the probe needs to impose low overhead, the probe workload is designed so as to lightly utilize the contention resource. The results of such tests are then used to construct a lookup table that maps any given utilization level of the contention resource as measured within a VM to a 95% confidence interval (CI) of the probe workload's mean response time at that level. This lookup table is used as an estimate of the baseline performance of the probe under various VM utilization levels when there is no interference, i.e., no competition for the contention resource from other VMs. For this estimate to be reliable, our system automatically repeats each test to obtain tight CIs.

Once the baseline performance has been established, a subscriber can start using the probe application alongside the Web application in the monitored instance to detect interference. The probe instance gathers response time data by submitting the probe workload periodically to the probe application within the monitored instance. For each submission of the probe workload, the workload's response times and the contention resource's utilizations as collected within the monitored instance during this period are recorded and sent to the controller. To detect a performance interference problem, the controller compares this data against the baseline lookup table constructed previously.

The general approach used by the controller to detect interference is as follows. The controller first locates the lookup table entry corresponding to the mean of the resource utilization measurements collected from the monitored instance. It then obtains the CI corresponding to this entry. If the mean probe response time is larger than the upper limit of the CI, the controller infers that the probe response time is abnormally high given the workload experienced by the Web application in the monitored instance. Consequently, it estimates that the increased probe response time is due to interference, i.e., competition from other VMs for the contention resource. This in turn indicates that the Web application's performance might be impacted as well due to this contention.

The controller can also take further measures to mitigate the detected interference. By controlling the PM resources stressed by the probe workload, the system can infer the resources that suffer the most contention. Over a certain period of time, the controller can record the number of performance interference issues along with the type of resource contention encountered in a VM in order to provide an estimate of the quality of the cloud platform to the subscriber. Based on this information, the subscriber can decide to migrate the monitored instance to a different type of VM that is likely to suffer less interference for the contention resources identified. Alternatively, if the interference is very severe and frequent, the provider can decide to move to a different cloud provider. The subscriber can make an informed decision to migrate especially if the timing of the performance interference issues align with that of the performance degradation cases reported by end users. These mitigation algorithms are orthogonal to the work presented in this paper.

The following subsections describe the probe system's components and their interaction in more detail. For the sake of simplicity, we outline our design by considering the processor as the contention resource. We present experiments in Section VII that demonstrate how our approach can be applied to additionally consider a PM's network bandwidth as a contention resource.

A. Probe Workload Generator

The probe workload generator generates a stream of synthetic requests to the probe application, which is a lightweight Web application running on the monitored instance. The workload generator is configured to generate a burst of requests at the rate of C HTTP connections per second (cps) over a period of t seconds. Since we focus on the processor, each request causes a processor-intensive PHP script to be invoked by the probe application. The script causes an exponentially distributed CPU service time with a mean of s seconds. We note that other distributions such as uniform or deterministic distributions can be used as well. We also note that the time between successive bursts sent by the probe workload generator to the probe application is a configurable parameter that can be set by a subscriber. The smaller this value the more frequently the controller checks for interference.

The value of C, t, and s are used to control the overhead imposed by the probe application on the monitored instance. To this end, we use the automated tuning algorithm outlined in [12] to select these values such that the probe application imposes an average per-core VM utilization in the range of 3% to 4% every time it is subjected to its workload. We find that our system is able to detect most instances of interference with these settings and increasing the overhead beyond this level does not significantly increase detection accuracy any further. We note that the probe parameters need to be tuned for each type of instance one wishes to monitor.

B. Probe Application

Previous studies [28]–[30] show that real Web workloads exhibit very good locality, e.g., a significant fraction of requests are for a small set of popular Web objects. This suggests that the demands placed on the processor dominate in such systems. Consequently, we first focus our attention on processor-intensive Web applications handling a large number of HTTP connections per second from multiple concurrent user sessions. Accordingly, our probe application is also a processor-bound Web application. As mentioned in

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

Section III-A, upon receiving each request the probe application invokes a PHP script that incurs an exponentially distributed processor service time with a mean of s seconds.

The probe application is hosted on a separate Web server within the monitored instance. This is done to ensure that any performance degradation observed in the probe is unrelated to problems that might manifest as a result of using the same Web server for both the Web and probe applications.

C. Reference Instance

As mentioned previously, our system requires past performance data collected from a reference instance to detect interference. We now outline in detail the process for obtaining this baseline data.

Since we focus on processors as the contention resource, a processor-intensive synthetic application is used as the background workload to vary the per-core VM utilization of the reference instance from $U_{Min}\%$ to $U_{Max}\%$ in steps of $U_{Step}\%$. These parameters can be configured by the subscriber. As mentioned previously, for each utilization level tests using the probe workload are conducted to record the sample mean probe response time R_{lookup} and the 95% CI of mean probe response time. The upper and lower limits of this CI are denoted as $maxR_{lookup}$ and $minR_{lookup}$, respectively. We note that the tests used to obtain the lookup table should be repeated multiple times to obtain the 95% CI of mean probe response time. To achieve this, our system automatically repeats tests till the width of the CI is within 5% of the sample mean. We also note that the lookup table may need to be refreshed whenever the cloud provider upgrades the PMs used to support the monitored instance. In spite of these requirements, the reference instance will not add significant costs to the subscriber since it does not have to execute continuously. A detailed cost analysis of running the reference instance in a public cloud platform is discussed in Section VI-B.

D. Controller and Interference Detection

The controller is responsible for detecting performance interference. As mentioned previously, the controller uses mean request response time as a performance metric. However, other metrics such as the median and the 95th percentile of response times can also be used. As discussed previously, the probe workload generator periodically invokes the probe application inside the monitored instance with a burst of requests and collects the probe application's request response times for the burst. The probe workload generator then sends this response time data to the controller. Furthermore, per-core VM utilization data for the monitored instance over the period of the burst is also sent to the controller. The controller uses this information to calculate the average response time R_p of the probe workload and the average per-core VM utilization U_p of the monitored instance.

To identify potential cloud induced performance degradations manifested in R_p, the probe system utilizes the lookup table constructed using the reference instance. Given the response time from the monitored instance R_p at an observed per-core VM utilization of U_p, the controller uses linear interpolation on the lookup table data to estimate $maxR_{lookup}$. As we show in Sections V and VI, this interpolation technique is found to be adequate for both our private and public cloud testbeds. If R_p exceeds $maxR_{lookup}$, the controller infers that there is significant performance interference in the monitored instance due to PM-level contention for the processor. Otherwise, the controller infers that the performance of the monitored instance is normal given that the monitored instance's per-core VM utilization is U_p.

Though the 95% CI of R_{lookup} provides a good reference for expected performance when there is no interference, it is still possible that the controller has false detections. A *false negative* report refers to the case where the controller does not detect the interference when the interference presents; a *false positive* report refers to the case where the controller flags a performance interference when there is no interference.

We also use another metric, *severity factor SF*, to provide us more insights on the severity of interference. The SF is calculated as the difference between the actual response time R_p at a per-core VM utilization of U_{lookup} and $maxR_{lookup}$ – the upper limit of the 95% CI of R_{lookup} at U_{lookup}, scaled by the mean response time R_{lookup}, as shown in Eqn. 1.

$$SF = \begin{cases} \frac{(R_p - maxR_{lookup})}{R_{lookup}}, & \text{if } R_p > maxR_{lookup} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Higher values of SF mean that the actual response times are significantly higher than the upper limit of the 95% CI of R_{lookup}, which implies a severe interference. A mitigation algorithm may define a threshold value to determine whether the interference is severe enough to warrant any subsequent action. Proper threshold values prevent costly actions in case of a false positive or minor problems. We use the SF to evaluate the probe approach in Section VI.

IV. Experiment Setup

We use our private cloud setup as well as the EC2 IaaS cloud for evaluating the probe. This section summarizes the experimental setup and the rationale for the setup.

A. Validation Probe Instance

In order to verify that the probe detects interference triggered response time increases experienced by the Web application being monitored, we setup a validation probe instance as shown in Fig. 2. The validation probe instance shares the same design specification with the probe instance. However, as shown in Fig. 2, it interacts with the Web application as opposed to the probe application. This allows us to obtain the ground truth on the Web application's performance.

From Fig. 2, we first use a separate Web application reference instance to obtain baseline no-interference performance of the Web application under study. Similar to the reference instance used by the probe application, the Web application reference instance has the same specifications as the monitored instance. It is used exclusively to execute a copy of the Web application under study. The validation probe instance submits

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

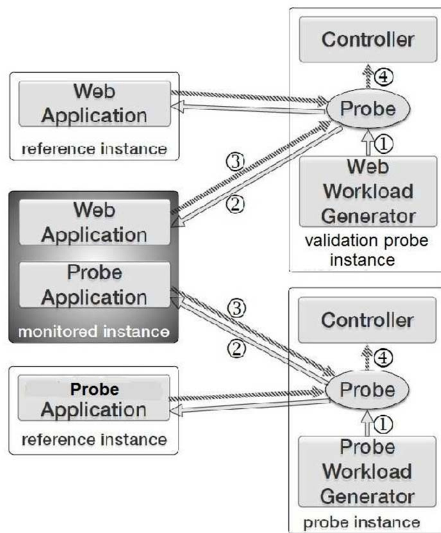

Fig. 2. Experimental Setup.

a specific test workload of interest from the Web workload generator (to be detailed in Section IV-B) to this reference instance. R_w^{ref} represents an estimate of the mean Web application response time under the test workload that is not biased by any performance interference effects.

To obtain baseline response time data of both the probe and the Web applications, we need to make sure that the performance of these applications running inside the two reference instances as shown in Fig. 2 are not affected due to performance interference. For this purpose, we use dedicated instances supported by many public cloud platforms such as EC2 [31] and Rackspace [32] for hosting the reference instances. A dedicated instance in EC2 has the same specification as a general instance but it is physically isolated at the host hardware level from non-dedicated instances of other subscribers. Consequently, a dedicated instance is much more expensive than a non-dedicated instance.[1] However, since the reference instances are not needed continuously the subscriber will not incur significant costs, as we show later in Section VI-B.

To validate the probe system, we run the validation probe instance in parallel with the probe instance. The sequence numbers labeled from 1 to 4 as shown in Fig. 2 for both the validation probe instance and the probe instance indicate how these two instances run in parallel. The validation probe instance first gets synthetic Web requests constituting the test workload from the Web workload generator. It then submits this workload to the Web application on the monitored instance. The validation probe instance calculates the average response times R_w of the Web application on the monitored instance for the test workload and passes it to the controller. Similar to the controller in the probe instance, the controller in the validation probe instance compares R_w with the 95% CI of R_w^{ref} and infers whether there is significant performance degradation in the Web application. We compare reports from the validation probe instance and the probe instance to verify the correctness of the probe system.

It must be noted that a separate validation probe instance as shown in Fig. 2 is used only for the purpose of validating the probe system for this paper. In the real world, the probe deployment will be similar to that shown in Fig. 1. Furthermore, our design allows a single rented instance working as the probe instance to monitor several probes inside several monitored instances. As a result, cloud subscribers have to incur minimal additional cost to use the probe system in a public cloud system. More detailed cost analysis is discussed later in Section VI-B.

B. Workloads

We use `httperf` [33] to generate workloads for both the probe and the Web applications. `httperf` can establish multiple concurrent HTTP connections with a Web server application under test. It can be configured to vary factors such as the load, i.e., connections per second (cps), and the number of requests sent to the server. We use a modified version of `httperf` where one instance of `httperf` can simultaneously send workload to multiple Web servers each having a distinct IP address. The reason for using the modified version is that it allows a single probe workload generator to simultaneously monitor multiple instances, thus keeping the cost of operation low.

We choose two different test workloads for the Web application: micro-benchmark and RUBiS [34]. The micro-benchmark generates requests to invoke a PHP script on the Web application that causes an exponentially distributed synthetic CPU service demand. Due to its simplicity, this workload allows us to quickly evaluate the probe under a wide variety of loads for various cloud instance types. We complement these results with those from the more realistic RUBiS workload [34], which emulates users issuing a series of inter-dependent transactions to an auction server. We use the default read-intensive browsing transaction mix specified by RUBiS.

We also configure the Web workload generator with two different x-second arrival processes: *steady* and *ramp*. The *steady* arrival process is a stationary process with an exponentially distributed mean inter-arrival time between successive requests. We set the mean cps issued by `httperf` such that the mean per-core VM utilization is around 50% for each EC2 instance type. The *ramp* arrival process consists of two equal-length phases. During the first phase, requests arrive with an exponentially distributed inter-arrival time, similar to the *steady* arrival process. However, during the second phase, the mean inter-arrival time between requests is half that of the first phase to emulate a sudden surge of incoming requests that Web applications often experience. Our goal is to show that the probe captures performance interference problems induced by the cloud platform as opposed to expected performance variations caused by the workload surge in the *ramp* workload. We select the mean request inter-arrival times for the two phases such that the per-core utilizations of the instance

[1]It must be noted that providers such as EC2 can still co-locate multiple dedicated instances belonging to the *same* cloud subscriber on the same PM. As a result, using expensive dedicated instances alone may not eliminate interference when a subscriber has multiple such instances.

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

TABLE I
PROBE PARAMETERS FOR PRIVATE CLOUD

C (cps)	s (ms)	t (sec)
25	0.0020	100

for the first and second phase are 50% and 90%, respectively. In our experiments, we set the length of the arrival process x to 100 seconds for the micro-benchmark workload and to 200 seconds for the RUBiS workload.

To obtain the lookup table discussed in the previous section, we execute a synthetic background workload on the reference instance that can incur various levels of per-core VM utilization. This workload consists of successive invocations of a program that randomly selects an integer between 0 and 1000 and calculates all the prime numbers between 0 and that number. The time between successive invocations of this program is automatically controlled to achieve a desired per-core VM utilization on the reference instance.

C. Private Cloud Environment

We first conduct experiments in our private cloud environment to demonstrate the motivation for our probe approach. Running experiments in the private cloud allows for a controlled environment with access to PM level metrics which is not possible in a public cloud environment. Our private cloud consists of a 12-core, dual socket Intel Xeon E5645 server. Multiple VMs are consolidated on this server using Kernel-based Virtual Machine (KVM) as the virtual machine monitor. Details of the server are given in Table II. Each VM is configured with 1 virtual CPU (VCPU) and 1 GB of physical memory. To keep the Web and probe application completely independent of each other, we host the Web and probe application on Apache (version 2.2) and lighttpd (version 1.4.35) Web servers, respectively.

To monitor VM performance under different workloads, we use a workload generator host connected to the server over a dedicated 1 Gbps connection. This host is an 8-core, dual socket AMD Shanghai Opteron 2378 server machine. For these tests, we subject VMs on the server to the RUBiS workload using `httperf`. Using a process similar to that described in Section IV-D, we ensure that the workload generator does not introduce any performance bottlenecks.

Our system automatically tunes the probe parameters C, t, and s for the private cloud such that the probe application imposes a per-VCPU overhead of 3–4% on the server. The resulting probe parameters are given in Table I. We find that these probe parameter settings cause only a modest increase of 3–5% in the Web application's mean response time. The synthetic background workload as described in Section IV-B is used for constructing the lookup table for the server. The values of U_{Min}, U_{Max}, and U_{Step} are chosen as 10, 99 and 10, respectively.

D. Amazon EC2 Environment

We validate the probe system in the well known EC2 IaaS platform. We use three types of EC2 instances in this work: m1.small, m1.medium and m1.large. The characteristics of these instances are summarized in Table III. The CPU power of each instance type is expressed in terms of CPU units. In EC2, one CPU unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 AMD Opteron or 2007 Intel Xeon processor. In our experiments, we used the */proc/cpuinfo* utility of Ubuntu to confirm that EC2 used the same type of physical machine hardware for our instances throughout this study.

TABLE II
HARDWARE AND SOFTWARE SPECIFICATIONS

Parameter	**Values**
Sockets (CPUs)	2
Cores per Socket	6
Processor	Intel Xeon E5645
Memory	64GB
L1 Cache	256 KB
L2 Cache	1 MB
L3 Cache	12 MB (Shared)
NIC	2 port 10 Gbps Broadcom 5771, 2 port 1 Gbps Intel 82576
Operating System	Ubuntu 12.04
Kernel Version	3.2.0-26-generic (Shared)
KVM version	qemu-kvm-1.1.2 (Shared)
Apache	Apache 2 version 2.2
PHP	PHP (version 5.3.6)
MySQL	version 5.1.49-1

TABLE III
AMAZON EC2 INSTANCE TYPES

Instance Type	**CPU Units**	**Cores**	**Memory [GB]**	**Disk [GB]**	**Cost [$/h]**
m1.small	1	1	1.7	160	0.044
m1.medium	2	1	3.7	320	0.087
m1.large	4	2	7.5	850	0.175

The number of cores refers to the number of VCPUs in the instance. All these instances are located in the same region (U.S.-West-Oregon) and their IP addresses belong to the same network subnet. Ubuntu server version 12.04 (kernel version 3.2.0-40-virtual) is installed as the operating system in all three instances. The software stack used for the Web application is the same as that used in our private cloud. Collectl [35] is used in each instance to monitor VCPU utilizations.

The response times measured by `httperf` depend on the performance of the monitored instance, the performance of the probe instance, and the bandwidth available between the two instances. Since we are interested only in the performance of the monitored instance, the setup must eliminate bottlenecks in the probe instance and the network. For this reason, we conduct experiments on different types of EC2 instances to determine the right type of EC2 instance to host the probe instance. Observations from these experiments, which are part of our earlier work [17], are summarized below for the sake of clarity:

1) Choice of Probe Instance: Our results show that choosing a wrong type of instance can lead to incorrect estimation of the response time of the Web application. For example, for

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

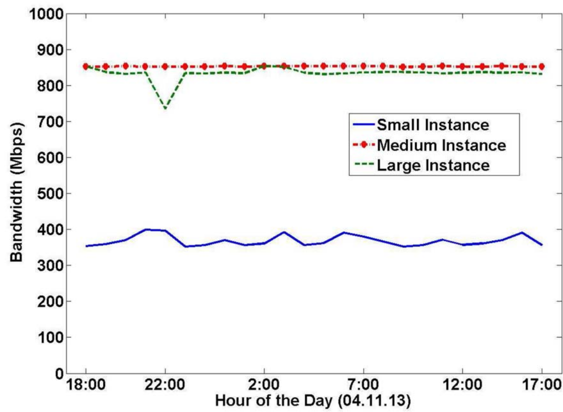

Fig. 3. Hourly available bandwidth inside EC2.

the micro-benchmark workload the response times reported by probe instances running on either a small or a medium instance are three times and two times higher than that from a large instance, respectively. Furthermore, response times obtained from a large probe instance are very close to those obtained by using more powerful instances. Therefore, we use a large EC2 instance as our probe instance. Our results indicate that one large instance can generate workload for up to 50 probe applications thereby ensuring costs are kept low for the subscriber.

2) Bandwidth Between the Probe Instance and the Monitored Instance: Bandwidth variations between the probe instance and the monitored instance can account for huge variations in the response time measured of the probe application. Though bandwidth fluctuation is unavoidable, the impact can be avoided by ensuring the bandwidth available between the probe instance and the monitored instance is ample for the workloads used in this study. We collect hourly network bandwidth in EC2 using the Iperf tool [36] over a period of two weeks. Fig. 3 depicts a representative day in this period. From the figure, the bandwidth available in the EC2 network is quite stable. Furthermore, the network bandwidth available is significantly higher than that required by our workloads.

As a Web service can be hosted on any of the small, medium, or large instance in EC2, we will select one instance of each type to monitor. We test the performance of the Web application subjected to the *steady* micro-benchmark on a small, a medium, and a large instance. As expected, the performance, in terms of response time and throughput, improves as we move from the small instance to the large instance. For example, at 200 cps the mean response time of the small instance is about 17 times higher than that of the large instance. The maximum throughput sustained by a small, medium and large EC2 instance are 300 cps, 600 cps, and 1200 cps, respectively. All three instance types incur a high per-VCPU utilization (in the range of 90%) while delivering their respective maximum throughput.

The probe parameters are selected automatically as described for the private cloud setup. The resulting probe parameters C, s, and t are shown in Table IV. The overheads imposed by the probe are similar to those reported for the private cloud setup. Finally, the values of U_{Min}, U_{Max}, and U_{Step} used for generating the lookup table are the same as those for the private cloud experiments.

TABLE IV
PROBE PARAMETERS FOR EC2

Instance	C (cps)	s (ms)	t (sec)
Small	25	0.0020	100
Medium	30	0.0015	100
Large	40	0.0010	100

TABLE V
SECTION OF THE LOOKUP TABLE

VCPU Utilization in %	95% CI in ms
10	(1.8, 1.9)
20	(1.8, 1.9)
30	(1.8, 1.9)
40	(1.8, 2.0)
50	(1.8, 2.0)
60	(1.8, 2.1)
70	(1.8, 2.2)
80	(1.8, 2.2)
90	(1.8, 2.2)
99	(2.0, 2.8)

V. THE PROBE IN A PRIVATE CLOUD

A. Experiment Methodology

In this section, we motivate the probe approach using the private cloud setup described in Section IV-C. Specifically, we first conduct experiments to construct the baseline lookup table for the probe. We then provide examples of the types of performance interferences that the probe can detect and show how VM level metrics alone are not adequate for detecting those problems.

B. Obtaining Baseline Performance

For constructing the baseline lookup table, we use the probe application, probe workload, and background workload described previously. Experiments follow the configuration settings provided in Section IV-C.

The experiment methodology is as follows. First, we run a reference VM on one socket of the Intel server. This VM is configured such that it can use one physical core in the socket but this assignment is not static, i.e., the VM can execute on any of the 6 cores. We choose this scheduling technique as it outperformed other scheduling techniques. We run the probe application inside this VM along with the background workload for obtaining the lookup table. Each experiment is repeated 10 times to get a 95% CI that is within 5% of the mean. A section of the lookup table obtained experimentally is shown in Table V. We perform linear interpolation on this data to construct a complete lookup table covering all possible per-VCPU utilization values between 10% to 99%.

From Table V, the response time of the probe application does not seem to be very sensitive to the increase in intensity of the background workload. The upper limit of the probe's response time CI increases drastically only after a VCPU

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

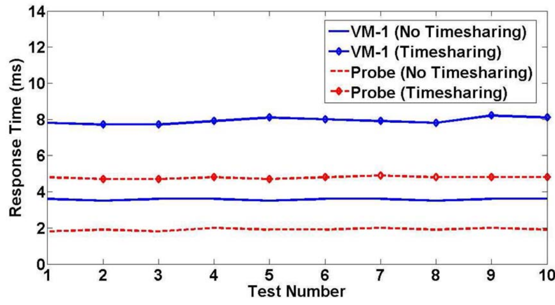

Fig. 4. Probe with CPU time-sharing.

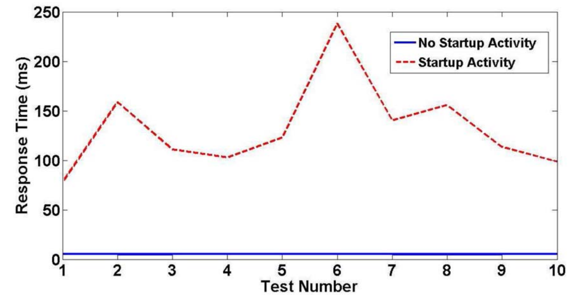

Fig. 5. Performance with startup activity.

utilization of 90%. We use PM level monitoring data to understand the reason behind this behaviour. Analysis shows that even though the reference VM is configured to use one physical core, 2 other cores on the server are utilized by the PM operating system. The combined utilizations of these physical cores are in the range of 5% to 7% in the tests of Table V. This suggests that the use of these relatively idle additional cores allows the probe application's response times to be insensitive to the background workload. To validate this conclusion, we run another test where all cores on the server are kept busy by scheduling processor-intensive workloads on them. In this case, we notice that the probe application's response times are more sensitive to the background workload. The upper limit of the probe's response time CIs starts increasing after the reference VM's VCPU utilization reaches 70%.

C. Impact of Sharing a Processor Core

As shown in past research [3], an EC2 small instance typically receives only a 40% - 50% share of a host's processor core. We demonstrate how the probe can be helpful in detecting performance degradations caused by such timesharing.

We first conduct a series of 10 identical tests when only one VM (VM-1) is active and is pinned to a single core on the server. The RUBiS workload is configured such that the VCPU utilization of the VM is around 27%. Next, we force timesharing by pinning the second VM (VM-2) to the same core. VM-2's workload is identical to that of VM-1. While the VCPU utilization of both VMs are around 27% each, the utilization of the physical core increases from around 27% to 52%.

Figure 4 compares the response time observed at VM-1 in both experiments. The response time of VM-2 in the second experiment is near identical to that of VM-1 and is hence not reported. From Figure 4, the mean response time is nearly doubled due to the timesharing. As expected, the increased contention for the physical core adversely impacts the performance of the VMs.

This experiment also shows that monitoring the VCPU utilization alone may not be sufficient to detect this performance degradation. For example, in the two experiments, the VCPU utilization of VM-1 is 27% both with and without timesharing. Hardware monitoring shows that the L3 cache hit rates of the core with and without timesharing are 0.92 and 0.91, respectively. Since timesharing has no effect on the performance of the L3 cache, the Web request service demand in these two cases does not change significantly, i.e., the VCPU utilization remains unchanged in these two cases.

We next turn our attention to the *CPU steal* metric used by others for inferring interference [8]. In the two experiments, the value of *CPU steal* increases from 3% (without timesharing) to 12% (with timesharing) for both VMs while the response time increases by almost 100% with timesharing. Although this metric's value increases with interference, it does not convey directly the severity of the impact of interference on VM response time. In particular, it would be difficult for a cloud subscriber to ascertain a *CPU steal* threshold beyond which significant response time degradations are likely to occur.

In contrast, the probe can provide a direct reflection of how interference impacts response time. As shown in Figure 4, the mean response time of the probe application more than doubles due to timesharing compared to its baseline response time in Table V that corresponds to a VCPU utilization of 27%. This increase mirrors the similar increase in mean response time of the two VMs.

D. Impact of Management Activities

To provide further motivation for the probe, we study the typical management activity of starting VMs on a host in response to concurrent requests from cloud subscribers. We refer to this management activity as *startup activity*. Previous research [16], [17] attributes performance degradation in virtualized environments to management activities similar to startup activity. We simulate the startup activity by concurrently starting up to 6 VMs on the socket of the Intel server on which one VM is already running the RUBiS application. The RUBiS VM is subjected to an incoming request arrival rate that incurs around 50% VCPU utilization. The performance of the RUBiS VM is first recorded when it is running alone without any startup activity. Next, its performance is recorded when there is startup activity present on the server.

Figure 5 shows the results of 10 identical runs comprising this experiment. As shown in the figure, there is a significant degradation in the response time of the RUBiS application when startup activity is present on the server. On average, the startup activity degrades the response time of the RUBiS VMs by a factor of 20. The average per-core utilization of the 6 cores on the socket increases from 8.3% to nearly 80%. The startup activity causes the L3 cache hit

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

TABLE VI
VALIDATING THE PROBE IN PRIVATE CLOUD

RUBiS-R (No ST)	Probe-R (No ST)	VCPU-U	RUBiS-R (ST)	Probe-R (ST)
4.2	1.8	53	**165.3**	**59.8**
4.3	1.9	54	**188.2**	**79.3**
4.3	1.9	54	**101.5**	**52.1**
4.3	1.8	53	**118.7**	**63.4**
4.4	1.8	55	**163.1**	**78.8**

rate of the socket to decrease from 0.91 to 0.53. Although this experiment represents a stress case, it nevertheless illustrates the unpredictability that can be triggered by management activities.

We next use the probe to see if it can detect the performance degradation due to the startup activity. Using the lookup table, we compare the probe's observed response time with the upper limit of the 95% CI of its baseline response time at the appropriate VCPU utilization value. Table VI shows results from 5 representative runs. In the table, ST refers to startup activity, VCPU-U refers to the VCPU utilization in percentage, and R represents the response time in milliseconds. From the table, in all the runs the mean response time of the probe increases significantly with startup activity mirroring a similar increase in the RUBiS response time. The probe's observed response time far exceeds the upper limit of the 95% response time CI recorded in the appropriate lookup table entry.

The experiments in this section show that the probe can detect problems that arise due to contention for the processing cores of a host in a private cloud environment. In the ensuing section, we provide further validation for the probe approach using the EC2 setup.

VI. The Probe in EC2

A. Experiment Methodology

We conducted experiments on EC2 over a 2 month period spanning April 2014 to May 2014 for this study. All the workloads considered are processor intensive. As a result, their performance is likely to suffer when there is contention for PM processors.

Before starting the experiments, we construct a lookup table of baseline response time data for each of the three types of instances shown in Table III. For each type of instance, we request a dedicated instance of that type from EC2 and use that as the reference instance. The lookup table is then obtained using the procedure described in Section IV.

For each experiment, we create an EC2 instance of the desired type running the Web as well as the probe applications and monitor their performance throughout the day. For each test, we first create a synthetic trace that realizes the desired arrival process and workload. We conduct 10 identical back to back runs using this trace at the start of every hour and report the mean of these runs. The 95% CIs of mean response time for the Web application are within 5% of their corresponding mean values. We also record at the start of every hour the available network bandwidth between the workload generator and the instances. The network bandwidth is stable throughout the day for every day in the week, similar to what is shown in Fig. 3. Due to cost limitations, we did not conduct a full factorial experimentation over all factors and levels. However, sample results obtained from other factor-level subsets do not suggest any change from the observed probe behaviour.

The remainder of this section is organized as follows. We present a cost analysis for running the probe system in Section VI-B. Results for the *steady* arrival process are presented first in Section VI-C. The ability of the probe to distinguish between EC2 interference and workload surges is demonstrated through the use of the *ramp* arrival process in Section VI-D.

TABLE VII
SAMPLE SIZE FOR EC2 INSTANCES

Instance	Duration (hours)	Cost ($)
Small	2.78	5.69
Medium	2.78	5.83
Large	2.78	6.01

B. Cost Analysis

The probe is designed to be cost effective for the cloud subscribers. We first evaluate the cost of obtaining the baseline performance of the probe for the three different types of EC2 instances that we use in our experiments. As mentioned earlier, we use dedicated EC2 instances as the reference instances for this purpose, the per-hourly costs of which are considerably higher than general instances. However, using dedicated instances allows for estimating the baseline response time values of the probe application unbiased by performance interference.

Table VII provides the costs incurred for constructing the baseline lookup tables. Based on our choice of experiment parameters and our desire for CIs that are within 5% of their means, we only need 2.78 hours each to obtain the baselines for small, medium and large instances. Consequently, the additional costs of running these tests are minimal, with the maximum cost of 6.01 being incurred by the large dedicated instance.

Next, we discuss the cost of running the probe instance that executes the probe workload generator. As mentioned earlier, we use a large instance as the workload generator for the probe application. Our experiments show that one large instance can generate workload for up to 50 probes. Ideally, this could be implemented by the cloud provider or a 3rd party vendor selling the probe service, from whom companies hosting Web applications can buy and pay only for the number of probes they need. Based on such a scheme and EC2's current pricing, we estimate that the cost of hosting and generating workload for a probe would be roughly equal to 0.084 per day. For larger companies hosting more than 50 instances inside EC2, the probe approach can be integrated into their infrastructure by buying an additional large instance for every 50 instances monitored. For these companies, adopting the probe technique would increase their budget by merely 2%.

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

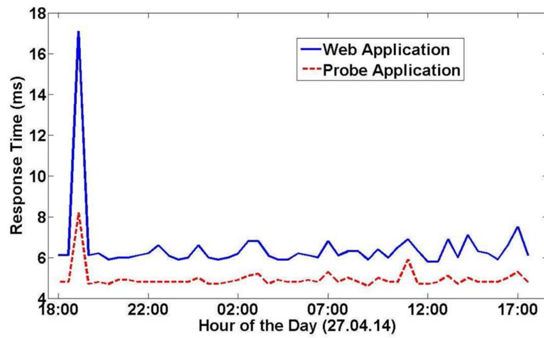

Fig. 6. *Steady* micro-benchmark - medium instance.

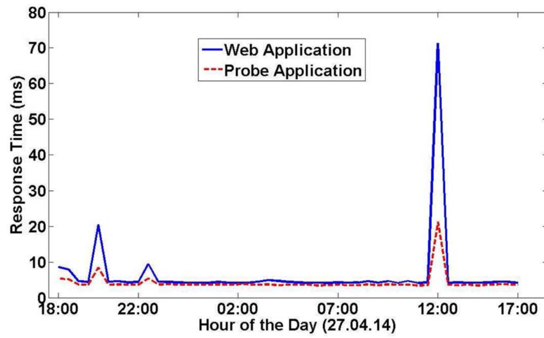

Fig. 7. *Steady* micro-benchmark - large instance.

C. Steady Workload

We first present results from the *steady* micro-benchmark workload. In all these cases, the arrival process causes a 50% mean utilization of each core in the instance.

Fig. 6 shows the performance of this workload on the medium instance over a representative day. From the figure, in spite of subjecting the Web application to the exact same synthetic workload trace, its performance varies throughout the day. In the worst case, the mean response time increases by a factor of almost 3 with respect to the baseline mean response time of the Web application. The mean response times are outside their baseline 95% CI for 6 hourly recorded results out of 24, i.e., 25% of the tests.

Fig. 7 shows the performance of the large instance over the same day. Although the performance of the large instance is more stable, one can still observe peaks where the mean response time increases significantly. In the worst case, the mean response time increases by a factor of 25 from the baseline. The mean response times fall outside their baseline 95% CI for 4 tests out of 24. We note that the performance problems are not due to overloading of the Web application since the application operates at a modest VCPU utilization of 50%. These findings validate previous results [17] that report significant Web application performance variability in EC2.

We next focus on detecting the observed variabilities. The resource utilizations measured within the instance did not show any deviations. In contrast, from Fig. 6 and Fig. 7, the probe tracks the deviations in the Web application's performance behaviour. The probe's performance degrades simultaneously when degradations are observed in the Web application's performance. Probe response times corresponding to the degradations are outside the relevant CI entry in the lookup table. For these cases, the probe did not have any false positives or false negatives. Sample results for the small instance also show that the probe is effective in detecting interference.

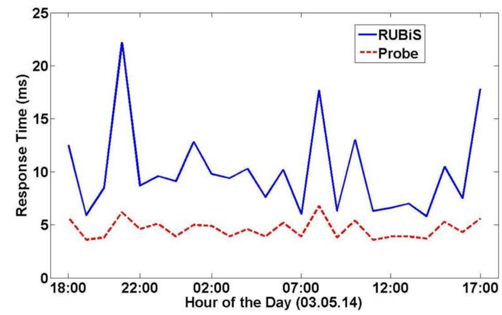

Fig. 8. *Steady* RUBiS workload - small instance.

We next focus on the *steady* RUBiS workload over a period of a week. As with the micro-benchmark workload, the *steady* RUBiS workload is designed to cause around 50% per-core utilization in each instance. Fig. 8 shows the performance of a small instance for a representative day in the week. The Web application's performance deviates from the baseline CI in 11 out of the 24 tests shown in the figure. The probe is able to identify 10 out of these 11 cases. Considering the entire week's data, the probe is able to identify 61 of the 68 performance deviations encountered by the Web application, i.e., nearly 90% of the total deviations. There are 2 false positives and 5 false negatives.

Tests show that the medium and large instances are very stable for this workload. Over a single day that we consider, the medium instance encountered no performance problems. On the same day, the large instance had one deviation, which is successfully detected by the probe. Since these instances are stable, we did not conduct tests for additional days.

D. Ramp Workload

We next use the *ramp* micro-benchmark workload on the medium and large instances to understand whether the probe distinguishes between performance interference effects and performance changes due to a workload surge. Table VIII shows the performance of the probe in a medium instance over a week. Column **W** indicates the number of hourly test results where the Web application showed performance degradation. Similarly, the column **P** indicates the number of times the probe detects the performance degradation reported in column **W**. Column **U** shows the average per-VCPU utilization of the instance during the test. The columns **False +ve** and **False -ve** indicate the number of false positives and false negatives reported by the probe respectively.

From the table, 60 out of the 168 tests, *i.e.,*, 36% of tests, in this period experienced performance interference. The percentage of tests impacted by interference varies from 25% per-day to 46% per-day over the week. The VCPU utilizations of the instance remains unchanged in spite of the interference.

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

TABLE VIII
Ramp MICRO-BENCHMARK (MEDIUM INSTANCE)

Date	W	P	U	False +ve	False -ve
27.04.14	8	7	0.78	0	1
28.04.14	7	5	0.78	1	1
29.04.14	11	10	0.80	0	1
30.04.14	7	7	0.79	0	0
01.05.14	6	6	0.77	0	0
02.05.14	10	9	0.81	0	1
03.05.14	11	9	0.80	1	1

TABLE IX
Ramp MICRO-BENCHMARK (LARGE INSTANCE)

Date	W	P	U	False +ve	False -ve
27.04.14	3	3	0.82	0	0
28.04.14	0	0	0.81	0	0
29.04.14	4	3	0.81	0	1
30.04.14	2	2	0.82	0	0
01.05.14	3	3	0.82	0	0
02.05.14	0	0	0.80	0	0
03.05.14	6	5	0.81	0	1

The probe is able to detect 53 of the 60 performance degradations, i.e., 90% of degradations, in the Web application during the course of that week.

Note that if the probe mistakes the workload surge in the second phase of the ramp workload as interference, then it would yield 100% false positive reports. However, from Table VIII the probe only reports 2 false positives over the week. Furthermore, the probe yields only 5 false negatives in this period.

Table IX shows the performance of the probe in a large instance for the same week. The performance of the Web application in the large instance is quite stable, with just 18 hourly results out of 168, i.e., 11% of tests, reporting performance degradation in a week as opposed to 60 in case of a medium instance. On two days of the week, there is no performance degradation in either the Web or the probe application. The probe is successful in detecting 16 out of 18 performance degradations recorded by the Web application in the large instance over the week. There are no false positives but there are two false negatives in this week. These results indicate that similar to the medium instance, the probe is able to detect nearly 90% of the performance problems manifesting inside a large instance over the course of a week.

We now use the *SF* metric introduced in Section III to analyze the severity of the performance degradations reported in Table VIII. Fig. 9 shows a scatter plot of the probe and Web application *SF* values for the 53 cases where the probe is successful. From the plot, the probe's *SF* estimates are effective in capturing the actual severity of interference, as embodied by the Web application's *SF* values. The Pearson correlation coefficient between the probe and Web application *SF*s is +0.97. This suggests that the controller can use the probe's *SF* to deduce the severity of the performance degradation in the Web application.

We next take a closer look at the 7 *ramp* micro-benchmark workload cases with the medium instance where the probe was

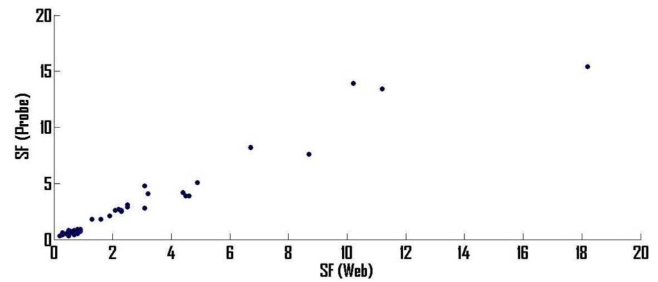

Fig. 9. Scatter Plot of SF.

TABLE X
WEEKLY *SF* VALUES OF MEDIUM INSTANCE

Date	False +ve	$SF(\mathbf{P})$	False -ve	$SF(\mathbf{W})$
27.04.14	0	0.00	1	0.09
28.04.14	1	0.12	1	0.14
29.04.14	0	0.00	1	0.06
02.05.14	0	0.00	1	0.21
03.05.14	1	0.13	1	0.11

not successful. Table X shows the *SF* values for the false positive and false negative cases recorded by a medium instance and its probe in the week as described earlier in Table VIII. The second and fourth columns in Table X indicate the number of false positives and false negatives recorded in a day, respectively. The 3rd column *SF*(**P**) and the 5th column *SF*(**W**) represent the *SF* values of the probe application and the Web application, respectively. Consider the 5th row in Table X, which shows a day with 1 false positive and 1 false negative. For the false positive case, the probe reports an *SF* of 0.13 when the Web application's *SF* is 0. Similarly, consider the 4th row in the table. The probe has a false negative with its *SF* being 0 when the Web application's *SF* is 0.21. Considering that the Web application's *SF* values range from 0 to 19 and probe application's *SF* values range from 0 to 23 in our tests, we can conclude that these cases represent relatively minor performance infractions. Similar trends are observed for other workloads and instances in our tests. These results suggest that the controller needs to consider the probe's *SF* estimates while making decisions.

Finally, we present results to show that the probe is also effective for the RUBiS *ramp* workload. We run the RUBiS *ramp* workload for a day only on a small instance for cost related issues. The *ramp* workload designed for RUBiS caused around 75% VCPU utilization in a small instance. As seen in Fig. 10, the performance of the probe closely tracks that of the RUBiS application. Performance degradation was observed in 10 hourly test results out of 24. The probe was able to successfully detect all 10 cases of degradation.

The results in this section show that the probe is effective in detecting performance interference issues in the cloud instances running both the micro-benchmark and RUBiS workloads. The probe is also able to distinguish between performance interference from those that occur due to workload surges. Furthermore, the probe's *SF* value provides a good estimate of the severity of the performance issues impacting the Web application. The probe is also lightweight and

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

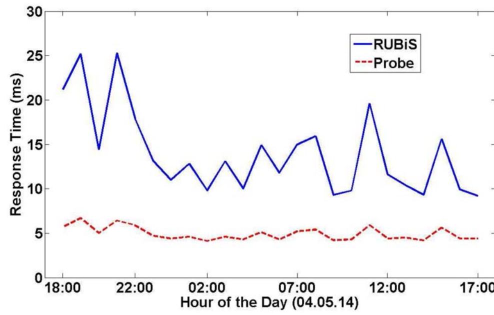

Fig. 10. *Ramp* RUBiS workload - small instance.

TABLE XI
PROBE WITH A DIFFERENT WEB SERVER

Date	W	P	False +ve	False -ve
07.05.14	11	10	0	1
08.05.14	10	10	0	0

imposes very low overhead on the response times of the Web application.

VII. SENSITIVITY ANALYSIS

A. Robustness

In this section, we run experiments to ascertain the effectiveness of the probe for other Web servers and workloads. Specifically, we run experiments to see if the probe application works well when hosted on a different Web server. We also study the probe when it is used to monitor a Web application hosted on a different Web server. Finally, we investigate a scenario with a more database-intensive RUBiS workload.

The first set of experiments involves running different sets of Web servers for the RUBiS application and probe application. All our previous tests are conducted using Apache as the Web server for the Web application being monitored and lighttpd as the Web server for the probe application. To test the generality of our approach, we run a set of tests in which we reversed the earlier order- lighttpd is used for hosting RUBiS and Apache is used for hosting the probe. We run the same *ramp* RUBiS workload as previously seen in Fig. 10. The test was conducted at the start of every hour over two days beginning at 18:00 on 7th May, 2014.

As seen in Table XI, the probe is very effective at detecting the performance problems even when it is being hosted by a different Web server than in the previous tests. Furthermore, this result also shows that the probe is able to monitor applications hosted on both Apache as well as lighttpd.

The next set of experiments involves running a mix of RUBiS workload other than the default browsing mix which we use in all earlier tests. We run the database-intensive ordering transaction mix of RUBiS to emulate a Web application handling a large number of transactions involving writes. We run this mix inside a small instance for 24 hours. A *ramp* style workload is created for this transaction mix that utilizes roughly 80% of the VCPU in the instance. We run the test at the start of every hour for a day beginning at 18:00 on 12th May, 2014.

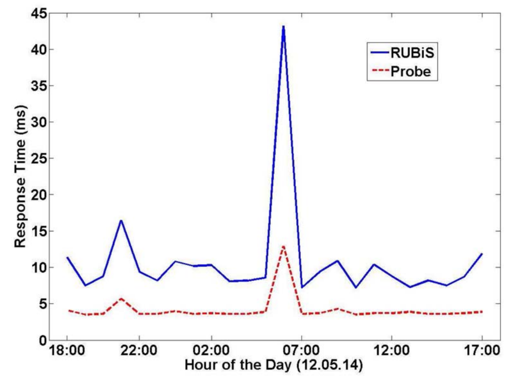

Fig. 11. RUBiS ordering mix - small instance.

As seen in Fig. 11, the performance of the probe closely emulates that of the RUBiS application running the ordering mix. The results are more stable compared to the earlier results obtained from running the browsing transaction mix. There are only 2 hourly tests that show performance degradation in RUBiS out of 24, both of which are successfully detected by the probe.

B. Detecting Contention for Shared Network

As a final step, we demonstrate how the probe system can be extended to monitor multiple contention resources. We consider as an additional source of contention the network shared between VMs hosted on a PM.

Performance interference due to network contention is hard to detect using only the standard tools provided by a cloud platform to subscribers. For example, assume that a VM's network bandwidth consumption drops during a given time interval. The drop could either be due to interference, i.e., activities of other network intensive VMs hosted on the PM, or due to a decrease in the application workload. This necessitates the need for the probe approach.

Contention for the shared network can be detected by introducing a new phase to the probe. In this network phase, the probe executes code designed to explicitly sense network contention. We design the network phase of the probe as follows. The probe's network phase generates a probe workload where HTTP connections are initiated at the rate of X cps to download a small file of size S MB from the probe application. The values of parameters S and X need to be tuned for each monitored instance such that the probe application consumes only a small fraction of the network bandwidth available to the host.

As before, our system first constructs a lookup table for the network phase by using a dedicated reference instance. The lookup table maintains the 95% CI of the response time of the probe's network phase at various VM network bandwidth values ranging from low to high.

Next, the network phase of the probe periodically submits its probe workload to the probe application on the monitored instance. Response times for the requests in the workload and the network bandwidth consumed by the monitored instance over this period are sent to the controller. The controller then

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

TABLE XII
SECTION OF THE NETWORK LOOKUP TABLE

Network B/W	95% CI in ms
100	(22.8, 24.9)
300	(22.7, 25.0)
500	(22.8, 25.2)
900	(23.5, 26.8)
950	(144.7, 149.5)

consults the network phase lookup table to detect the presence and severity of a network interference.

We experimentally evaluate the network phase of the probe on our private cloud setup. We use the Intel server described in Table II. We use only one socket of the server. All VMs running on this host are configured to share a single Ethernet network interface on the physical host. This interface is connected to a load generation host through a dedicated 1 Gbps connection. The load generation machine hosts the probe and reference probe instances shown in Fig. 2.

We first run a single VM on the socket on which we run the probe application. We tune the parameters S and X for the probe application to be 1 MB and 5 cps, respectively. The tuned probe imposes a network bandwidth utilization of around 5%. We run this application along with the Iperf tool [36], which generates the background workload for constructing the lookup table. We configure Iperf to incur a wide range of network bandwidth utilizations to build the lookup table shown in Table XII (B/W refers to the bandwidth utilized by the VM in Mbps).

Next, the monitored VM executes a network-intensive Web application alongside the probe application. The Web application's workload consists of downloads of a 1 GB file. The rate at which HTTP connections are issued to download this file is selected such that the Web application consumes around 75% of the total available network bandwidth when executing alone in its VM. Up to 3 additional VMs executing workloads statistically similar to those of the monitored VM are progressively consolidated on the socket.

Table XIII shows that the probe can detect the network resource contention triggered by this consolidation. Values marked in bold represent cases where the Web application or the probe application experience performance degradations. When there is only a single VM on the socket, the Web application on the VM does not suffer from interference. The probe's mean response time is within the 95% CI estimated from the lookup table for a VM bandwidth of 792.4 Mbps. As seen in the table, when more network intensive VMs are consolidated on the socket, the Web application's response time increases and the monitored VM's bandwidth consumption drops as a result of network contention. The network phase of the probe is able to detect these performance degradations, as indicated by the increases in the probe application's mean response times with respect to its baseline response time CIs recorded in the lookup table. If the probe reports frequent network contention, then a subscriber can mitigate the problem by migrating to an instance that guarantees more dedicated bandwidth.

TABLE XIII
EFFECTIVENESS OF THE NETWORK PHASE

No. of VMs	Web R (s)	Network B/W	Probe R (ms)
1	22.1	792.4	24.2
2	**35.4**	503.4	**38.1**
3	**53.1**	343.8	**51.3**
4	**70.9**	274.3	**68.5**

VIII. CONCLUSION

This paper focuses on the performance of Web services hosted on public IaaS cloud platforms such as Amazon EC2. We show that such platforms can induce significant performance degradations due to contention for shared resources, which can adversely impact the cloud subscriber owning the service. Such behaviour has also been corroborated by others [13]–[16] as well as in our private cloud setup. Unfortunately, most existing contention management techniques are targeted for use by cloud providers and cannot be used by cloud subscribers. The few subscriber oriented approaches proposed in literature require fine-grained Web service instrumentation. Furthermore, while many of these techniques can detect contention they cannot provide information about the specific cloud resources experiencing the contention.

To address these issues, we develop a subscriber oriented contention detection system that employs a software probe. The probe executes alongside each Web service instance to be monitored. Response times emitted by the probe provide a direct reflection of how contention for any given shared cloud resource impacts the response time of the monitored Web service.

We show that the probe system outperforms methods that solely rely on common performance metrics made available to subscribers on public cloud systems. We also show through experiments spanning a variety of workloads, Web servers, and cloud systems that the probe is very effective in detecting the existence and quantifying the severity of performance interference problems triggered by contention. The probe system imposes very little overheads on the monitored instance. Furthermore, it does not require any Web service level instrumentation to record response times. The probe can also offer insights on the specific cloud resources, e.g., processor and network, suffering from contention. Subscribers can use this information to decide on an appropriate method to mitigate the effect of that contention. Finally, our study shows that the probe system imposes only modest additional costs for the subscriber.

In this work, we exploit dedicated reference instances to get an estimate of baseline probe performance without interference. The probe system can be adapted in a straightforward manner to handle platforms that do not support dedicated instances. Considering the processor contention example, baseline tests could be conducted using a non-dedicated reference instance. We can then use the VM-level *CPU steal* metric collected during these tests to isolate those tests where the reference instance is not impacted by interference, i.e., tests during which the value of *CPU steal* is 0.

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.

Future work will focus on interference mitigation techniques that can exploit the probe. Specifically, we will devise algorithms that can use the probe's severity factor to drive techniques such as load balancing. Due to cost constraints, we were only able to focus on EC2 and limit ourselves to a period of 2 months. Future work will consider a longer period and include other public cloud platforms to further corroborate this study.

References

[1] 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.*, vol. 84, no. 8, pp. 1270–1291, 2011.

[2] J. Mukherjee, D. Krishnamurthy, J. Rolia, and C. Hyser, "Resource contention detection and management for consolidated workloads," in *Proc. IEEE IM*, Ghent, Belgium, 2013, pp. 294–302.

[3] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Proc. INFOCOM*, San Diego, CA, USA, 2010, pp. 1–9.

[4] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *Proc. IEEE INFOCOM*, Toronto, ON, Canada, 2014, pp. 1285–1293.

[5] S. Fu, "Performance metric selection for autonomic anomaly detection on cloud computing systems," in *Proc. GLOBECOM*, Houston, TX, USA, 2011, pp. 1–5.

[6] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, p. 8, 2010.

[7] *AWS CloudWatch*. Accessed on Dec. 2016. [Online]. Available: http://aws.amazon.com/cloudwatch/

[8] G. Casale, C. Ragusa, and P. Parpas, "A feasibility study of host-level contention detection by guest virtual machines," in *Proc. CloudCom*, Bristol, U.K., 2013, pp. 152–157.

[9] *Auto Scaling*. Accessed on Dec. 2016. [Online]. Available: http://aws.amazon.com/autoscaling/

[10] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "Mitigating interference in cloud services by middleware reconfiguration," in *Proc. Middleware*, Bordeaux, France, 2014, pp. 277–288.

[11] Y. Amannejad, D. Krishnamurthy, and B. Far, "Managing performance interference in cloud-based Web services," *IEEE Trans. Netw. Service Manag.*, vol. 12, no. 3, pp. 320–333, Sep. 2015.

[12] J. Mukherjee, D. Krishnamurthy, and J. Rolia, "Resource contention detection in virtualized environments," *IEEE Trans. Netw. Service Manag.*, vol. 12, no. 2, pp. 217–231, Jun. 2015.

[13] A. Iosup *et al.*, "An early performance analysis of cloud computing services for scientific computing," Faculty Eng., Math. Comput. Sci., Delft Univ. Technol., Delft, The Netherlands, Tech. Rep. PDS-2008-006, Dec. 2008.

[14] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *Proc. CCGrid*, Newport Beach, CA, USA, 2011, pp. 104–113.

[15] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, Porto Alegre, Brazil, 2011, pp. 248–259.

[16] J. Dejun, G. Pierre, and C.-H. Chi, "EC2 performance analysis for resource provisioning of service-oriented applications," in *Proc. ICSOC/ServiceWave*, Stockholm, Sweden, 2009, pp. 197–207.

[17] J. Mukherjee, M. Wang, and D. Krishnamurthy, "Performance testing Web applications on the cloud," in *Proc. ICSTW*, Cleveland, OH, USA, 2014, pp. 363–369.

[18] Y. Tan *et al.*, "PREPARE: Predictive performance anomaly prevention for virtualized cloud systems," in *Proc. ICDCS*, 2012, pp. 285–294.

[19] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: Transparently identifying and managing performance interference in virtualized environments," in *Proc. USENIX ATC*, San Jose, CA, USA, 2013, pp. 219–230.

[20] Q. Guan, C.-C. Chiu, Z. Zhang, and S. Fu, "Efficient and accurate anomaly identification using reduced metric space in utility clouds," in *Proc. IEEE NAS*, Xiamen, China, 2012, pp. 207–216.

[21] Q. Guan and S. Fu, "Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures," in *Proc. IEEE SRDS*, Braga, Portugal, 2013, pp. 205–214.

[22] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 77–88, 2013.

[23] A. Roytman, S. Govindan, J. Liu, A. Kansal, and S. Nath, "Algorithm design for performance aware VM consolidation," Microsoft Res., Bengaluru, India, Tech. Rep. MSR-TR-2013-28, 2013.

[24] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for QoS-aware clouds," in *Proc. EuroSys*, Paris, France, 2010, pp. 237–250.

[25] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov, "Stay-away, protecting sensitive applications from performance interference," in *Proc. Middleware*, Bordeaux, France, 2014, pp. 301–312.

[26] Y. Amannejad, D. Krishnamurthy, and B. Far, "Detecting performance interference in cloud-based Web services," in *Proc. IEEE IM*, Ottawa, ON, Canada, 2015, pp. 423–431.

[27] S. A. Javadi, S. Mehra, B. K. R. Vangoor, and A. Gandhi, "UIE: User-centric interference estimation for cloud applications," in *Proc. IEEE IC2E*, Berlin, Germany, 2016, pp. 119–122.

[28] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," *SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 126–137, 1996.

[29] A. M. Faber, M. Gupta, and C. H. Viecco, "Revisiting Web server workload invariants in the context of scientific Web sites," in *Proc. ACM/IEEE SC*, Tampa, FL, USA, 2006, p. 25.

[30] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 53–64, 2012.

[31] *Dedicated Instance*. Accessed on Dec. 2016. [Online]. Available: https://aws.amazon.com/ec2/purchasing-options/dedicated-instances/

[32] *Collectl*. Accessed on Dec. 2016. [Online]. Available: https://www.rackspace.com/

[33] httperf. *HTTP Performance Measurement Tool*. Accessed on Dec. 2016. [Online]. Available: http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.pdf

[34] *Rubis Rice University Bidding System*. Accessed on Dec. 2016. [Online]. Available: http://rubis.ow2.org/

[35] *Collectl—Linux Man Page*. Accessed on Dec. 2016. [Online]. Available: http://linux.die.net/man/1/collectl

[36] *IPERF—The TCP/UDP Bandwidth Measurement Tool*. Accessed Dec. 2016. [Online]. Available: http://iperf.fr/

Joydeep Mukherjee received the M.Sc. degree from the University of Calgary and the B.E. degree from the National Institute of Technology, India. He is currently pursuing the Ph.D. degree with the University of Calgary, Canada. His research interests include software performance engineering, virtualized systems, cloud computing, and computer networks.

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

Mea Wang is currently an Associate Professor with the Department of Computer Science, University of Calgary. Her research interests include peer-to-peer networking, multimedia networking, cloud computing, as well as networking system design and development.

Authorized licensed use limited to: California Polytechnic State University San Luis Obispo. Downloaded on September 08,2024 at 20:53:59 UTC from IEEE Xplore. Restrictions apply.