Detecting Performance Interference in Cloud-Based Web Services

Yasaman Amannejad, Diwakar Krishnamurthy, Behrouz Far
Department of Electrical and Computer Engineering, University of Calgary, Canada
{yasaman.amannejad, dkrishna, far}@ucalgary.ca

***Abstract*—Web services have increasingly begun to rely on public cloud platforms. The virtualization technologies employed by public clouds can however trigger contention between virtual machines (VMs) for shared physical machine (PM) resources thereby leading to performance problems for the Web service. Past studies have exploited PM level performance metrics such as Clock Cycles Per Instruction to detect such platform induced performance interference. Unfortunately, public cloud customers do not have access to such metrics. They can typically only access VM-level metrics and application level metrics such as transaction response times and such metrics alone are often not useful for detecting inter-VM contention. This poses a difficult challenge to Web service operators for detecting and managing platform induced performance interference issues inside the cloud. We propose a machine learning based interference detection technique to address this problem. The technique applies collaborative filtering to predict whether a given transaction being processed by a Web service is suffering adversely from interference. The results can then be used by a management controller to trigger remedial actions, e.g., reporting problems to the system manager or switching cloud providers. Results using a realistic Web benchmark show that the approach is effective. The most effective variant of our approach is able to detect about 96% of performance interference events with almost no false alarms.**

I. Introduction

Many Web applications such as Netflix and WordPress have begun to exploit cloud computing due to the promise of unlimited computing resources and the pay-as-you-use model. A cloud system achieves resource virtualization and sharing by executing multiple virtual machine (VM) instances on each physical machine (PM) in data centers. Unfortunately, such virtualized infrastructure can trigger performance degradation when the VM instances contend for shared PM resources, *e.g.*, processor cores, cache, physical memory [1], [2], and other cloud resources, *e.g.*, network and storage. Furthermore, cloud management activities such as VM migration can also cause performance deterioration. Such performance interference issues can be problematic for cloud-based Web services. Long response times resulting from such interference can cause frustration to end users of such Web services. This can in turn lead to financial losses for the businesses operating these services as well as the cloud providers. Cloud customers therefore need management tools to detect performance interference so that they can take mitigative actions such as using a different type of VM instance from the provider or switching to a different provider.

Earlier studies [2], [3], [4], [5], [6] have been focused on exploiting host PM level metrics such as cache misses and Clock Cycles per Instruction for detecting performance interference. However, public cloud subscribers typically have no access to PM level metrics. Cloud users can typically collect only VM level metrics such as network bandwidth utilization, memory consumption and CPU utilization. Although such metrics may be useful in detecting VM level performance degradations, a cloud user is still unaware of the performance degradation due to problems at the PM level [7], [8].

An alternative approach that does not rely on virtual or PM level metrics is to monitor application level metrics such as the response times for handling incoming Web transactions. However, transaction response times alone are not always good indicators of performance interference within a cloud. For example, an increase in transaction response times can occur simply due to a sudden increase in Web service workload instead of a true performance interference issue induced by virtualization.

This paper uses collaborative filtering, a machine learning technique used extensively to build recommendation engines [9], [10], to detect cloud platform induced performance interference. Collaborative filtering is applied on transaction response time data collected by a Web service. Specifically, a Collaborative Response time Estimation (CRE) module is developed to estimate a reference response time for every incoming transaction. This reference response time shows the expected response time for that transaction when the transaction does not suffer from interference. CRE uses the transaction type, the instantaneous Web service load, and a long term history consisting of past response time data for the service's transactions to calculate the reference response time. The historical data can be collected online from the production environment. The reference response time and the actual response time of an incoming transaction are used for interference detection. Specifically, any significant mismatch between these values is used to raise a flag.

Since CRE's estimation takes into account Web service load, it has the ability to ignore response time increases that can be solely attributed to workload surges. Furthermore, by maintaining a long term response time history for each type of transaction, CRE is able to estimate a reference ”no-interference” response time and hence detect any interference that inflates response time of the Web service.

We develop and evaluate three different variants of CRE. Experiments using the RUBiS [11] Web benchmark in a virtualized testbed indicate the effectiveness of CRE. Specifically, all three variants of CRE significantly outperform an alternative technique that ignores Web service load when estimating reference response times. Furthermore, the most effective CRE variant was able to detect about 96% of performance interference events with no false alarms.

This paper is structured as follows. Sec. II reviews related work. Sec. III describes the architecture and the algorithms used in our proposed approach. In Sec. IV, we elaborate details of our experiments, and the results are reported in Sec. V. Sec. VI provides conclusions and future work.

II. Related work

Several studies, e.g., [12], [13], have shown that current VM technologies do not provide complete performance isolation for VMs. VMs can compete for shared micro-architectural resources such as Last Level Caches (LLC), memory channels, storage, and network devices and this can manifest as performance problems within the VMs.

A vast majority of work on mitigating performance interference has focused on using PM level performance metrics to detect inter-VM contention. Blagodurov *et al.* propose an approach that uses PM level metrics such as LLC miss rates to detect and mitigate interference [4]. Similarly, Novakovic *et al.* design a contention mitigation system called DeepDive that uses PM level metrics [5]. Mukherjee *et al.* developed a software probe that executes a micro benchmark program on a PM [2]. To detect contention, execution time of the benchmark is compared with it's execution time when it runs alone on the PM.

All of the above approaches need direct access to the PM and therefore they can only be used by cloud providers. Cloud subscribers typically do not get such access to PMs. In this work, we seek a solution from subscriber's perspective and hence we propose a solution that relies only on information that subscribers would be able to access.

Casale *et al.* [8] propose a subscriber centric solution for detecting CPU contention in public clouds for batch applications. By continuously monitoring the execution times of a set of batch benchmarks within a VM, and using baseline benchmark execution times and the CPU steal metric the authors are able to predict whether the VM is being affected by interference. In contrast to their work, our work focuses on interactive Web applications and generalizes to non-CPU resources as well. Maji *et al.* develop a subscriber centric interference detection technique for Web applications [14]. However, their approach requires 30 hours of offline training. In contrast, as we show in Sec. V-I our approach is robust to the lack of training data.

Our method has some similarities with techniques used to detect outliers in datasets [15], [16], [17], [18] proposed by others. However, the key difference from these methods is that our method is able to differentiate between workload surges and performance interference.

III. Methodology

We first describe the general architecture of our interference detection system in Sec. III-A. Sec. III-B provides more details about the algorithms used in our approach.

A. *Cloud Customer Driven Interference Detection*

We propose a cloud customer driven interference detection approach. As a result, our method does not need either PM level information or information from VM instances not belonging to the customer. Our method only requires transaction response time data from the Web service being monitored. Fig. 1 shows the architecture of the system. The VM instance hosting the Web service is called the monitored instance. The CRE module also resides on the monitored instance. We note that it is also possible to host CRE on a separate instance, e.g., a load balancer instance attached to the Web service. The CRE module intercepts an incoming (resp. outgoing) transaction from an end user (resp. the Web service) and forwards the transaction on to the Web service (resp. the end user). This proxy approach allows us to monitor Web service transaction response times without the costly need to instrument a Web service by modifying its source code. As shown in Fig. 1, the PM hosting the monitored instance may host other instances (VM_{SoI}) that may act as Sources of Interference (SoI) to the monitored instance. The primary aim of CRE is to detect such performance interference.

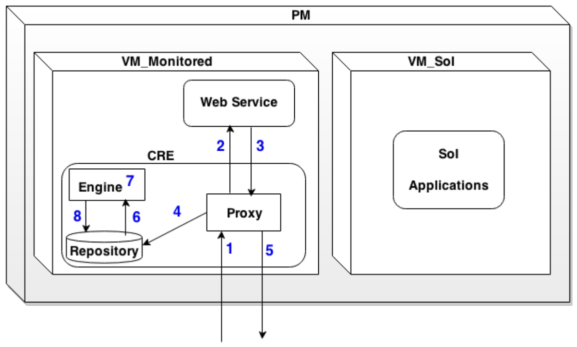

Fig. 1: Cloud customer oriented interference detection

We consider a Web service that supports n distinct types of transactions, e.g., browse, search. Each transaction type is assigned a unique id in the range 1 to n. The CRE module continuously tracks the instantaneous load $\mathbf{L_t}$ of the system at any given time t. We define the load $\mathbf{L_t}$ using the type and number of transactions being processed at time t by the Web service. Specifically, we denote $\mathbf{L_t}$ as an n-dimensional vector where the j^{th} element in $\mathbf{L_t}$, $L_t[j]$, shows the number of transactions of type 'j' executing on the server at time t. The CRE module also relies on a repository $\mathbf{D}$ containing historical response time data for different transaction types when the transactions are not impacted by interference in the cloud. For a given transaction type k and load $\mathbf{L_t}$, a query to $\mathbf{D}$ returns a vector of historical response times $\mathbf{H_k^{L_t}}$. Each element of $\mathbf{H_k^{L_t}}$ is a past response time of a transaction of type k under load $\mathbf{L_t}$.

The repository $\mathbf{D}$ can be obtained in multiple ways. For example, it can be obtained by running a copy of the Web service in isolation on a PM and subjecting the service to synthetic workloads. Many cloud providers such as Amazon Web Services (AWS) allow customers to create instances that have sole access to a PM, e.g., the "dedicated" instances of AWS [19]. While these instances are typically expensive, they can nonetheless be run for a brief period to collect data for the repository. If running such instances is not feasible, then the repository can use historical Web service response times collected over a *long* duration to *estimate* the no interference response times. Specifically, as per the central limit theorem [20] the *mean* response time of a transaction type k is likely to be very close to the true mean if the sample size of response time values is large. A large sample size can ensure that the mean response time does not get biased significantly by transient performance interference problems induced by the cloud platform. We explore both of these scenarios for obtaining $\mathbf{D}$ in Sec. IV.

The CRE module operates as follows. CRE creates a *handler thread* for each incoming transaction. The handler thread intercepts an incoming transaction to the Web service, records the time t that request was intercepted, records the transaction type k, and obtains the current load of the server $\mathbf{L_t}$ (Step 1 in Fig. 1). These inputs are extracted for estimating a mean reference response time for the incoming transaction under no performance interference. Next, the handler thread forwards the incoming request to the Web service (Step 2 in Fig. 1). The Web service completes the transaction and the response to this transaction is intercepted by the handler thread in 3. The handler thread now records the time the response was intercepted and uses this with the timestamp recorded in 1 to obtain the actual response time $r_k^{L_t}$ of the transaction. Data obtained in 1 and the actual response time is then recorded in the repository and a unique identifier id is generated for the new record (Step 4 in Fig. 1). Each record in the repository is hence a tuple of $\langle id, k, r_k^{L_t}, \mathbf{L_t}\rangle$. In parallel to 4, the handler thread forwards the response to the end user in 5. Estimation of the reference response time and interference detection (Steps 6, 7 in Fig. 1) occurs when the new record is added to the repository and will be discussed next.

The mean reference response time $rref_k^{L_t}$ of the incoming transaction is estimated as follows. The inputs extracted by the handler thread in 1 in conjunction with data in $\mathbf{D}$ is used to offer a mean reference response time estimate. Specifically, the CRE engine collects historical response times (Step 6 in Fig. 1) observed for the incoming transaction type under the load $\mathbf{L_t}$ observed in 1. If such information does not exist, then the module collects response times under $\mathbf{L_t}$ from other transaction types that are similar to k. The response times collected from the repository are then used to compute the mean reference response time $rref_k^{L_t}$ for the incoming transaction. We note that the load dependent nature of the reference response time estimate allows CRE to differentiate between performance interference issues and workload surges. Further details of the CRE engine are discussed in Sec. III-B.

Interference detection is done in 7 by comparing the mean reference response time $rref_k^{L_t}$ with actual transaction response time. A significant deviation between the reference and actual response times is used to flag a performance interference. We developed three different variants of CRE that differ in the way the deviation between the actual and reference response times is quantified. These are described in detail later in this section. If the CRE engine raises a flag for a set of transactions, records pertaining to those transactions are removed from the repository in order to not affect future estimations of reference response time (Step 8 in Fig. 1).

B. Collaborative Response time Estimation (CRE)

Algorithm 1 and Algorithm 2 show pseudo code of key functions used by CRE. Details of these functions are presented next.

Algorithm 1: Main methods of CRE

```
Method: Reference Response Time Estimation
Input: $k$, $\mathbf{L_t}$
Output: $rref_k^{L_t}$
$\mathbf{H_k^{L_t}} \leftarrow$ query repository $(k, \mathbf{L_t})$
if ($\mathbf{H_k^{L_t}}$ is not null) then
    $rref_k^{L_t} \leftarrow$ calculate mean response time of $\mathbf{H_k^{L_t}}$
else
    $\mathbf{N_k^{L_t}} \leftarrow$ find similar neighbor transactions for $\mathbf{L_t}$
    foreach $i \in \mathbf{N_k^{L_t}}$ do
        $\mathbf{H_i^{L_t}} \leftarrow$ query repository $(i, \mathbf{L_t})$
        $rref_i^{L_t} \leftarrow$ calculate mean response time of $\mathbf{H_i^{L_t}}$
    end
    $rref_k^{L_t} \leftarrow$ weighted mean of $rref_i^{L_t}$, using Eq.(1)
end
--------------------------------------------------------------
Method: Similarity Calculation
Input: $\mathbf{T}$ (List of transaction types)
Output: $\mathbf{S}$ (Similarity matrix)
foreach $i$ in $1..k$ do
    foreach $j$ in $1..k$ do
        $S[i,j] \leftarrow$ similarity ($T[i]$, $T[j]$), using Eq.(2)
    end
end
--------------------------------------------------------------
Method: Interference Detection
Input: $threshold$, $C$, $M$
//calls one of the variants of the flag interference //interface
Flag Interference ($threshold$, $C$, $M$)
--------------------------------------------------------------
Method: Update Repository
Input: $idList$
Output: $\mathbf{D}$ (updated repository)
Remove rows with $id \in idList$
```

Reference response time estimation - As discussed previously, CRE needs to estimate the mean reference response time of a given transaction k at a given load $\mathbf{L_t}$. As shown in Algorithm 1, if transaction k has experienced load $\mathbf{L_t}$ earlier, the repository would contain historical records for k under $\mathbf{L_t}$. The response time estimation is simply the average of the past response times recorded for k under $\mathbf{L_t}$, i.e., the mean of the elements of $\mathbf{H_k^{L_t}}$.

The estimation is more complicated when transaction k has not encountered load $\mathbf{L_t}$. For this scenario, we identify $\mathbf{N_k^{L_t}}$ as the set of *other* transactions similar to transaction k that have encountered $\mathbf{L_t}$. We then aggregate the response times of each of these transactions at $\mathbf{L_t}$ to estimate the mean reference response time. We note that this is similar to the operation of recommendation engines [9], [10] that predict items that a Web shopper maybe interested in based on the items other shoppers similar to that shopper have purchased in the past. We use the adjusted weighted sum formula proposed by Breese *et al.* [21] for estimation. Eq. (1) shows the aggregation formula we used for calculating the mean reference response time from the response times of the similar transactions.

$$rref_k^{L_t} = rref_k^* + \frac{\sum\limits_{i \in \mathbf{N_k^{L_t}}} (rref_i^{L_t} - rref_i^*) \times S[k,i]}{\sum\limits_{i \in \mathbf{N_k^{L_t}}} |S[k,i]|} \quad (1)$$

As shown in Eq. 1, for each similar transaction i the difference between the mean response time of transaction

Algorithm 2: Flag Interference

```
Method: Instantaneous Approach
Input: $threshold$, $-$, $-$
Output: Detection result: $\mathcal{N}/\mathcal{I}$, $k$
while new record is added to $\mathbf{D}$ do
    //each record is a tuple of $\langle id, k, r_k^{L_t}, \mathbf{L_t}\rangle$
    $\langle id, k, r_k^{L_t}, \mathbf{L_t}\rangle \leftarrow$ fetch $newrecord$
    $rref_k^{L_t} \leftarrow$ Reference Resp. Time Estimation ($k$, $\mathbf{L_t}$)
    if $((r_k^{L_t}$ - $rref_k^{L_t}) > threshold \times rref_k^{L_t}$ ) then
        //$idList$ contains $id$s of suffering records
        append $id$ to $idList$
        Update Repository ($idList$)
        $idList \leftarrow null$ , $result \leftarrow \mathcal{I}$, $k$
    else
        $result \leftarrow \mathcal{N}$
    end
end
```

```
Method: Hysteresis Approach
Input: $threshold$, $C$, $-$
Output: Detection result: $\mathcal{N}/\mathcal{I}$, $k$
while new record is added to $\mathbf{D}$ do
    $\langle id, k, r_k^{L_t}, \mathbf{L_t}\rangle \leftarrow$ fetch $newrecord$
    $rref_k^{L_t} \leftarrow$ Reference Resp. Time Estimation ($k$, $\mathbf{L_t}$)
    if $((r_k^{L_t}$ - $rref_k^{L_t}) > threshold \times rref_k^{L_t}$ ) then
        $c_k \leftarrow c_k + 1$
        //$idList_k$ contains $id$s of records of //transaction $k$ suffering interference
        append $id$ to $idList_k$
        if ($c_k == C$) then
            Update Repository ($idList_k$)
            $result \leftarrow \mathcal{I}$, $k$ , $idList \leftarrow null$
        end
    else
        $result \leftarrow \mathcal{N}$, $idList \leftarrow null$
    end
    $c_k \leftarrow 0$
end
```

```
Method: Mean-based Approach
Input: $threshold$, $-$, $M$
Output: Detection result: $\mathcal{N}/\mathcal{I}$, $k$
while new record is added to $\mathbf{D}$ do
    $\langle id, k, r_k^{L_t}, \mathbf{L_t}\rangle \leftarrow$ fetch $newrecord$
    $rref_k^{L_t} \leftarrow$ Reference Resp. Time Estimation ($k$, $\mathbf{L_t}$)
    update mean$_M(rref_k)$ with $rref_k^{L_t}$
    update mean$_M(r_k)$ with $r_k^{L_t}$
    $m_k \leftarrow m_k + 1$
    append $id$ to $idList_k$
    if ($m_k == M$) then
        if (($mean_M(r_k)$ - $mean_M(rref_k)$) $> threshold$ $\times$ $mean_M(rref_k)$) then
            Update Repository ($idList_k$)
            $m_k \leftarrow 0$
            $idList \leftarrow null$, $result \leftarrow \mathcal{I}$, $k$
        else
            $m_k \leftarrow 0$
            $idList \leftarrow null$ , $result \leftarrow \mathcal{N}$
        end
    end
end
```

i over all system loads, i.e., $rref_i^*$, and its historical mean reference response time at load $\mathbf{L_t}$, i.e., $rref_i^{L_t}$, is weighted by the degree of similarity between the transactions k and i i.e., $S[k, i]$. As shown in Eq. 1, the weighted sum is then used to adjust the load independent mean transaction response time of k, i.e., $rref_k^*$. Any negative values of the weighted sum are set as 0. If no transaction has seen $\mathbf{L_t}$ in the past, we simply use the mean of response times experienced by transaction type k over all recorded values in the repository. This situation can be improved by identifying loads similar to $\mathbf{L_t}$ and using response times for similar transactions under such similar loads. We defer this as future work.

Similarity Calculation - CRE considers two transactions to be similar if they have similar response times under the same loads. Different similarity measures have been used in literature [22], [23], [24]. We use the Pearson correlation coefficient (PCC), which is one of the commonly used similarity measures. Using PCC, the similarity between two transactions k and i is defined by Eq. (2).

$$S[k,i] = \frac{\sum\limits_{e \in \mathbf{E_{ki}}} (rref_k^e - rref_k^*) \times (rref_i^e - rref_i^*)}{\sqrt{\sum\limits_{e \in \mathbf{E_{ki}}} (rref_k^e - rref_k^*)^2 \times \sum\limits_{e \in \mathbf{E_{ki}}} (rref_i^e - rref_i^*)^2}} \tag{2}$$

In this equation, similarity between transactions k and i, is calculated based on the mean response times of these two transactions for all common loads $\mathbf{E_{ki}}$ they have experienced before. Common loads are loads at the Web service that both k and i have experienced in the past. The similarity value between each pair of transaction types is calculated and used within Eq. (1) to estimate the mean reference response time. Similarity values can be in the range of $[-1, 1]$ with 1(resp. -1) indicating perfect positive(resp. negative) correlation between two transactions. Similarity values are updated periodically during the lifetime of CRE to reflect newer response time measurements.

Interference Detection- The interference detection algorithm acts in transaction granularity level and uses $rref_k^{L_t}$ to flag whether it is a normal, i.e, N, event or an interference, i.e., I, event. It takes as input a $threshold$ parameter that quantifies the degree of permissible mismatch between the reference and actual response times. The $threshold$ parameter can take values from 0 to 1. The lower the $threshold$ value, the lesser is CRE's tolerance for a mismatch. This threshold should be set by the Webservice manager. We evaluate three different variants to flag an interference event. These variations are shown in Algorithm 2 and are discussed next.

1) Instantaneous approach - For a given transaction and system load, the mean reference response time $rref_k^{L_t}$ is compared with the actual transaction response time $r_k^{L_t}$. An interference event is flagged if the difference between the reference and actual response times exceeds a tolerance, which is computed as the product of the $threshold$ parameter and the mean reference response time $rref_k^{L_t}$.

2) Hysteresis approach - Similar to the instantaneous approach, an internal flag is raised when the mismatch between the mean reference response time and the actual transaction response time exceeds the tolerance for a given transaction. However, an interference event is raised only when a chain of C continuous flags are raised for that transaction. We note that the hysteresis approach with $C = 1$ corresponds to the instantaneous approach.

3) Mean-based approach - This approach takes as input an aggregation parameter M. For any given transaction k, it computes the mean of the actual response times for the last M records of the transaction (*i.e.* $mean_M(r_k)$). It also computes the mean of the mean reference response times for those M records of the transaction (*i.e.* $mean_M(rref_k)$)[1]. An interference event is flagged if the difference between these two mean values exceeds a tolerance, which is computed as the product of the $threshold$ parameter and the mean of the past M mean reference response times $mean_M(rref_k)$. We note that the instantaneous approach is a specific form of the mean-based approach where $M = 1$.

The above three approaches present different trade offs with respect to the speed and accuracy of decisions. The instantaneous approach enables agile decisions. However, there is a potential for mistaking a routine stochastic variation in a transaction's response time as a performance interference induced by the cloud platform. The other two approaches seek to mitigate this problem by delaying their decisions.

Repository evolution- As mentioned previously, CRE assumes an initial repository containing historical transaction response times in order to estimate no interference mean reference response times for a given transaction under a given load. We refer to the initial dataset in the repository as the *training data*. A good training dataset should be representative of the typical workloads experienced by the Web service. The training phase should be long enough to capture fluctuations in system load. Since it is not practical to have *all* possible system loads in the training data, we evolve the repository by continuously adding response times of completed transactions along with their associated load. As described previously, when CRE finds that a transaction is undergoing performance interference, data corresponding to that transaction is removed from the repository using the model update function of Algorithm 1. This reduces the likelihood of such abnormal response times from biasing future estimates of mean reference response times. Furthermore, the growth in size of the repository is controlled by periodically discarding old entries.

IV. Experiment Setup

In this section, we describe our experiment settings as well as the performance measures for characterizing the effectiveness of CRE.

A. Experiment Testbed

Fig. 2 shows the testbed used for this study. Tab. I lists key hardware and software settings of the testbed.

TABLE I: Hardware and Software Settings

CPU	Two 6-core Intel Xeon 1.6 GHz E5645
Memory	32 GB RAM per socket
Cache	L1: 256 KB, L2: 1MB, L3: 12 MB
OS (Guest and Host)	Ubuntu 12.04
Kernel (Guest and Host)	3.2.0-26-generic shared kernel
Hypervisor	KVM qemu-kvm-1.1.2 shared
Web server	Apache 2, version 2.2
Application Server	PHP version5.3.6
VM monitored	vCPU: 2, Memory: 4GB
SoI VM instances	vCPU: 1, Memory: 1GB

We use a dedicated physical machine (PM1) to host the Web service under study, the CRE module, and the SoI VM instances. PM1 contains 2 sockets, i.e, socket 0 and socket 1, each containing 6 cores. The VM instance hosting all tiers of the Web service, $VM_{Monitored}$, is configured with with 2 virtual CPUs (vCPUs) and 4 GB of RAM. This instance has exclusive use of 2 cores of socket 0. The rest of the 4 cores in this socket are used to host SoI VM instances. An SoI instance is configured with 1 VCPU and 1 GB of RAM. As shown in Fig. 2, each SoI instance has exclusive use of 1 of the cores of socket 0. An SoI instance contends for shared socket resources, e.g., the L3 cache, memory, and the PM's Network Interface Card (NIC) thereby interfering with the Web service instance. From Fig. 2, the CRE module is installed as a separate instance running on socket 1 of PM1.

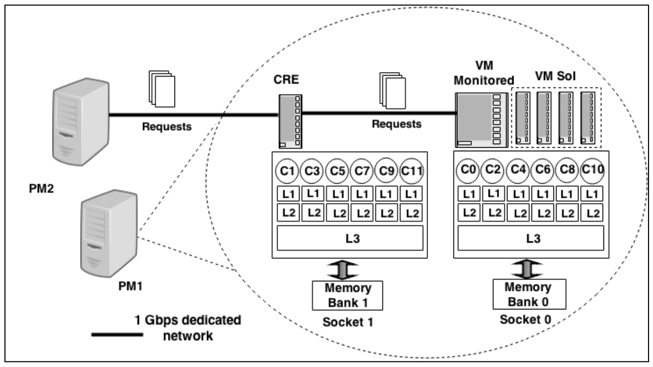

Fig. 2: Experiment testbed

We run the RUBiS benchmark [11] as our Web service. RUBiS emulates the behavior of an auction server that lets users browse and bid for items. We use the RUBiS `browsing` transaction mix in our experiments. This transaction mix supports 14 different transaction types. The CRE module is implemented as an add-in to the open source LittleProxy [25] proxy server available under an Apache 2 license. The CRE $threshold$ parameter is a user input and as an example we set it to 0.05 in our experiments. We use the top 5 similar neighbors for response time estimation (Eq. 1). Similarity values are configured to be updated every 1-hour to reflect newer response time measurements.

As shown in Fig. 2, we use a separate PM called PM2 to generate synthetic workloads to the RUBiS Web service. Both PMs are connected by a Fast Ethernet switch that provides dedicated 1 Gbps connectivity between them. We use the open source `httperf` [26] Web request generator to submit workloads to the server. `httperf` can be configured to issue concurrent user sessions to a server under test. In our tests, we use a non-stationary session arrival process to emulate the bursty workloads observed typically by popular Web services. Specifically, new sessions are submitted such that the session inter-arrival time, i.e, the time elapsed between the arrivals of successive sessions to the Web service, follows an exponential distribution. The mean session inter-arrival time is modulated such that there is a steady increase in the session arrival rate followed by a steady decrease. Fig. 3 shows a snapshot of the per-core utilization of the Web service caused by this workload.

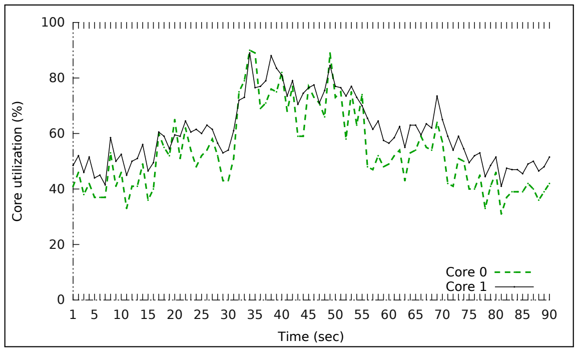

Fig. 3: Core utilization of the monitored VM instance

[1]We drop the $\mathbf{L_t}$ in the notation for these means since the last M records for a transaction may pertain to different loads.

B. *Experiment Factors*

Tab. II lists the various experiment factors and their different levels. Default levels of the factors are shown in bold. The Average Response time Estimation (ARE) technique uses the load independent mean of the historical response times of a transaction type to offer a mean reference response time estimate for a transaction of that type. It is used as a baseline to compare our CRE approach, which considers service load while estimating reference response times. From Tab. II, we consider as a factor the benchmark running within an SoI instance. The *RAMspeed* [27] benchmark places stress on the L3 cache, which is shared between the Web service and SoI instances. The *Iperf* [28] benchmark consumes the network bandwidth of the NIC shared between the Web service and SoI instances. We set the *Iperf* server to run on PM2, and the *Iperf* client to run on an SoI VM instance of PM1. In our setting, *Iperf* utilizes about 90% of the 1 Gbps bandwidth between PM1 and PM2. We also vary the number of SoI instances from 1 to 4 as shown in Tab. II to study the behavior of CRE under increasing levels of interference.

TABLE II: Experiment factors

Factor	**Value**
Estimation technique	{**CRE**, ARE}
Inference detection variant	{**Instantaneous**, Hysteresis, Mean}
Hysteresis factor (C)	{**1**,2,3,4,5}
Mean approach aggregation (M)	[**1**-10]
SoI benchmark	{**RAMspeed**, Iperf}
# of SoI instances (I)	{**1**, 2, 3, 4}
Training data pollution	{**0**, 10, 20, 30, 40, 50}%

Finally, we consider two different ways of initially populating CRE's repository with training data. The first method involves obtaining the data from the Web service when the monitored instance gets exclusive use of the PM. With the other method, the training data is obtained when the monitored instance shares the PM with an SoI instance. Hence, the training data in this case is biased by performance interference. We refer to this as Training data pollution. We consider experiments where 0%, i.e., no interference, to 50% of the response times in the training data are polluted by interference.

C. *Metrics to Evaluate CRE*

CRE is a binary classifier, which receives an input and decides whether there is performance interference or not. Therefore, standard metrics for evaluating the performance of classifiers can be used to evaluate our method. We use True Positive Rate (TPR) and False Positive Rate (FPR) in our evaluations. TPR is the ratio of the number of transactions flagged by CRE as suffering from interference to the actual number of transactions that suffer from interference. FPR is the ratio of the number of transactions CRE wrongly classified as suffering from interference to the total number of transactions not affected by interference.

To calculate TPR, we need a ground truth of the actual number of transactions suffering from interference. We establish the ground truth as follows. As part of each experiment run, we submit the same workload to the Web service twice, once with SoI and once without SoI. We then compare both sets of transaction response time data. We divide each set into batches of 500 requests. For each batch in both sets, we compute the mean response times for the 14 different RUBiS transactions. Next, we check whether the mean response time of a transaction type in any given with SoI batch is greater than the mean response time of that transaction type in the corresponding without SoI batch. If so, all transactions of that type in the with SoI batch are marked as suffering from interference.

We use the without SoI set to calculate FPR. Specifically, we enable CRE when there is no SoI and obtain the number of interference flags raised by CRE. This represents the number of transactions that are wrongly classified by CRE as suffering from contention.

D. *Experiment Process*

As is standard practice, we use 10-fold cross validation [29] while evaluating CRE. The RUBiS workload that we use consists of around $50,000$ requests. We partition the input workload to 10 subsets of around $5,000$ requests each. Each experiment is run 10 times. Each run employs 9 subsets as training data and the remaining subset as validation data over which CRE makes its predictions. We use a different subset for the validation phase across runs. As discussed previously, the validation phase in each run is executed twice, once with SoI and once without SoI, to calculate TPR and FPR. The reported TPR and FPR values are the average of our 10 runs.

V. Results

A. *General observations*

The RUBiS workload that we use is CPU intensive, as previously shown in Fig. 3. The workload causes a very light disk utilization of 6.7%. We verify using the *Iperf* tool that a bandwidth of 1 Gbps can be sustained between PM1 and PM2. The RUBiS workload utilizes only about 1% of this bandwidth. We also check the overhead of CRE. It is important that CRE does not substantially increase the actual response times of the Web service. To check this, we submitted a set of requests with and without the CRE module. On average, the response times increase by 4% because of CRE (Min = 3.1%, Max = 4.6%). Although this is negligible when compared to the extent of interference induced response time degradations that we observe in this system, our future work will focus on optimizing the CRE implementation further to reduce this overhead. The observed results from all experiments show maximum 95% confidence interval width of 0.042 and 0.028 for TPR and FPR, respectively.

B. *Effect of performance interference*

Fig. 4 shows the impact of interference on mean transaction response times. This set of experiments used *RAMSpeed* as the SoI. The number of SoI instances I is set to 0, 2, and 4. To eliminate the impact of load fluctuations for the sake of simplified analysis, these experiments use a workload with a fixed mean session inter-arrival time, in contrast to our other experiments which use non-stationary session arrivals. As a result, the per-core utilization does not fluctuate significantly.

From the figure, the presence of SoI increases the mean response time of all 14 RUBiS transactions. Furthermore, response time degradation increases with increased contention from the SoI instances. These results motivate the need for a technique that detects such interference. Since the load did not fluctuate significantly in these experiments, it is possible to use the long term mean response time of a transaction as a reference to detect transient performance interference effects. However, we show in subsequent sections that CRE's load dependent predictions are needed for more realistic workloads with fluctuating session arrivals.

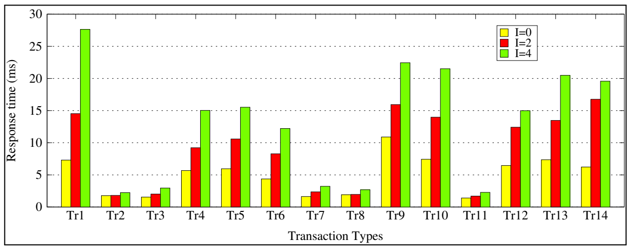

Fig. 4: Negative impact of performance interference

C. *Effectiveness of CRE with default factor levels*

For this experiment, we use the default levels specified in Tab. II. Recalling, CRE is configured to use the instantaneous inference detection method, there is one SoI instance executing *RAMSpeed*, and the training data has 0% pollution. This experiment serves as a baseline for us to quantify other enhancements such as the hysteresis and mean-based inference detection methods. Results show that CRE achieved $TPR = 0.86$, and $FPR = 0.11$. The 95% confidence interval for TPR and FPR is (0.83-0.89) and (0.08-0.13), respectively. Analyzing the results on a per transaction basis, CRE is effective for all transactions with the TPR in the range of ($[0.70 - 0.96]$). Variation in FPR is larger since the values are in the range$[0.0 - 0.44]$. Since there is more historical data to learn about popular transactions, CRE's accuracy is high for highly popular transactions but lower for unpopular transactions.

D. *Impact of estimation method (CRE vs. ARE)*

In this experiment, we compare CRE with ARE. As mentioned earlier, with ARE reference response time for a transaction is estimated as the average of past response times of that type of transaction without considering system load. For this experiment, we used the default levels as defined in Tab. II. Results show that CRE with a $TPR = 0.86$ significantly outperforms ARE with a $TPR = 0.59$. However, ARE has a better FPR of 0.03 when compared to CRE's FPR of 0.11. As we discussed earlier, CRE's use of the instantaneous method can confound routine stochastic variation in a transaction's response time with response time variations due to performance interference thereby leading to false positives. We next explore methods to reduce CRE's FPR.

E. *Impact of the hysteresis parameter C*

For this experiment, we set the detection approach to hysteresis and explore the impact of various values for the hysteresis parameter C. The rest of the settings are at the default levels shown in Tab. II. Fig. 5 shows the results of this experiment. We note that $C = 1$ corresponds to the instantaneous method results shown in the previous subsection. From the figure, an increase in C decreases FPR. FPR decreases from 0.11 for $C = 1$ to 0 for $C = 5$. A high C value allows CRE to better distinguish between usual stochastic variation in transaction response times, e.g., an occasional spike in response time, from sustained variations triggered by performance interference. Unfortunately, a high C value also reduces TPR slightly, as shown in Fig. 5. This shows that there are a small number of scenarios where the hysteresis approach of requiring C continuous threshold violations to raise an interference flag is not effective.

From the results, a reasonable compromise seems to be a setting of $C = 3$, which has a TPR and FPR of 0.82 and 0.02, respectively. Since the choice of C may be workload dependent, a cloud customer may need to conduct controlled tests using synthetic workloads within a non-production environment to select an appropriate value.

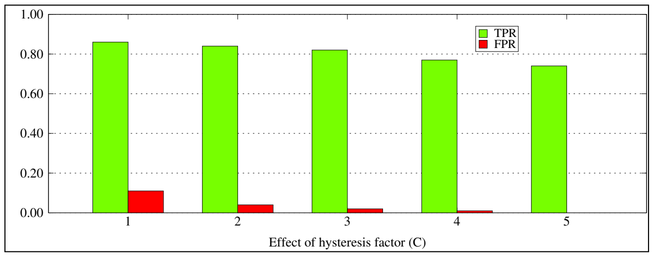

Fig. 5: Performance of CRE with different C values

F. *Impact of the aggregation parameter M*

We now explore the effectiveness of the mean-based detection approach with various values of the aggregation parameter M with other experiment factors at their default levels. Fig. 6 shows results for both CRE and ARE with this approach. From the figure, *both* TPR and FPR improve with increases in M. By changing the value of M from 1 to 10, TPR has improved from 0.86 to 0.96, and FPR has decreased from 0.11 to about 0. Simultaneous improvements to both TPR and FPR with increasing M makes this approach more appealing than the hysteresis approach. The improvement to both FPR and TPR shows that using mean values smooths out any routine stochastic response time fluctuations while preserving the impact of sustained interference. As mentioned previously, the improved performance of the mean-based approach over the instantaneous approach comes at the expense of agility in decision making. A cloud customer needs to pick an appropriate value of M depending on how fast they need to react to potential performance problems.

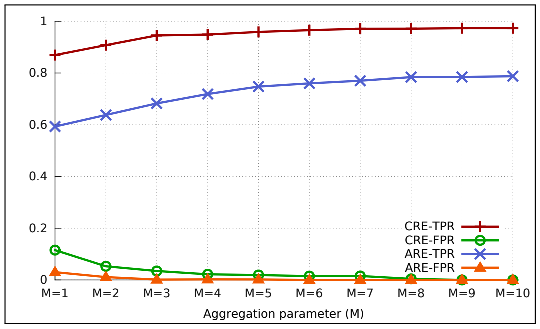

Fig. 6: Impact of the aggregation parameter M

We also repeated this experiment with the ARE method. From Fig. 6, increasing the value of M improves the TPR and FPR of ARE as well. Comparing the TPR results of CRE and ARE shows that CRE is more effective than ARE for all values of M. Furthermore, CRE requires a lower value of M to obtain a given degree of accuracy. For example, the TPR reaches to 0.95 for $M = 5$. In contrast, the TPR for ARE saturates only beyond $M = 8$. These results indicate that CRE is more agile than ARE. The time that elapses between the instant an interference problem manifests itself to the instant an interference flag is generated is lesser for CRE than for ARE.

G. *Impact of multiple SoI instances*

We next study the TPR of CRE under increasing levels of interference, which we trigger by running multiple SoI instances. We increased the number of SoI instances I from 1 to 4, and used the default levels for the other factors. We note that FPR is not impacted by this experiment since it is calculated based on experiment runs without SoI instances. Results show that TPR improves when there is stronger interference. The values of TPR for 1, 2, 3, 4 SoI instances are 0.86, 0.88, 0.89, 0.91, respectively. This suggests that CRE is particularly effective for detecting cloud induced problems that severely impact end user response times. ARE fares poorly in this scenario as well, since it does not consider workload fluctuations.

H. Impact of SoI type

We next characterize the ability of CRE to generalize to interference problems other than contention for the memory hierarchy of the PM. We compare the mean-based method for *RAMSpeed* and *Iperf* as the SoI instance. Other experiment factors are kept at their default levels. As mentioned previously, *Iperf* is a network-intensive tool which places stress on the shared network. In our setting *Iperf* utilizes 90% of the bandwidth between PM1 and PM2. This causes the mean response time of the Web service to increase by 57%. From Fig. 7, CRE is able to detect the performance interference imposed by the bandwidth sharing. *Iperf* inflates the response time less than *RAMSpeed*, therefore, the TPR and FPR are slightly less than those for *RAMSpeed*. The TPR is 0.90 and the FPR is almost 0.0 for $M = 5$. CRE's power stems from its use of response time for interference detection. The technique is likely to detect any type of interference that leads to an increase in response time of the Web service.

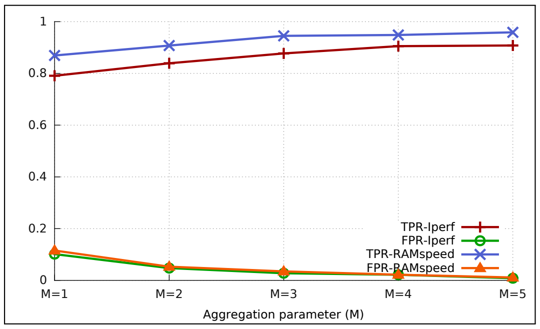

Fig. 7: Performance of CRE for *RAMspeed* and *Iperf*

An interesting observation from this experiment is that all Web service VM instance level utilization metrics are normal even while the Web service's performance is being impacted by *Iperf*. There was no increase in the resource utilizations of the Web instance. This shows that VM level utilization metrics are not always helpful in identifying performance interference thereby motivating the CRE approach.

I. Impact of training data pollution

As discussed in Sec. III-B, CRE needs training data for its predictions. CRE starts predictions assuming that the training data is not impacted by interference, i.e., there is no training data pollution. In real word applications running on the cloud, it might be difficult to gather such interference-free training data. In this experiment, our focus is on the sensitivity of CRE to training data pollution. We run multiple experiments using the default levels and only varying the amount of pollution in the training set. When increasing training data pollution from 0% to 50% in increments of 10%, the TPR values are 0.86, 0.83, 0.79, 0.77, 0.75, and 0.73. We note that having 50% pollution is a stress case for CRE. It is very unlikely that a production cloud system hosting a high volume Web service will be experiencing interference problems half the time. Nonetheless, CRE performs well even for this stress case by detecting 73% of interference events.

We argue that even when the initial training data is impacted by pollution, CRE is able to adapt itself over the course of time. To show this, we start from the point where 50% in the training data. We then continuously measured TPR during the validation phase when the Web service continues serving requests. A *RAMSpeed* SoI instance is also executed on the PM hosting the Web service such that 50% of the transactions in the validation phase are impacted by interference. In the validation phase, periods of interference alternate with periods of no interference.

Fig. 8 shows the result of this experiment. In the figure, we indicate the size of original training data with S and report the TPR whenever the repository grows by $S/2$ records. From the figure, CRE can gradually adapts itself when the system is running. TPR increases from 0.73 to 0.80 even though 50% of transactions in the validation phase are impacted by interference. As described in Sec. III-B, CRE does not update the repository with response times of requests impacted by interference. This ensures that the quality of data in the repository continuously improves thereby resulting in progressively better $TPRs$.

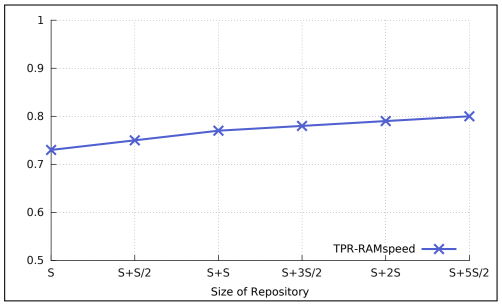

Fig. 8: Evolution of CRE TPR with time

VI. CONCLUSIONS

We propose a machine learning based solution called CRE for detecting cloud induced performance interference problems in a Web service. CRE is a customer oriented approach and relies only on Web transaction response times and does not require access to performance metrics of PM in the cloud. CRE does not require an explicitly authored performance model of the system for its estimations. It adapts itself autonomously using a repository of past historical transaction response times. Results show that CRE is effective in detecting performance interference.

Future work will focus on refining CRE. Specifically, the effectiveness of CRE can be impacted when the number of transaction types and possible system loads grow. For such scenarios, it is possible to use clustering techniques to reduce the number of transaction types and system loads. We will also focus on validating CRE with other types of applications and also in public cloud environments. Finally, we will also work on developing techniques that can work in tandem with CRE to mitigate the impact of interference for cloud-based Web services.

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," *Journal of Systems and Software*, 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 *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pp. 294–302, IEEE, 2013.

[3] S. Fu, "Performance metric selection for autonomic anomaly detection on cloud computing systems," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pp. 1–5, IEEE, 2011.

[4] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, p. 8, 2010.

[5] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," tech. rep., 2013.

[6] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 452–461, IEEE, 2008.

[7] Q. Zhu and T. Tung, "A performance interference model for managing consolidated workloads in qos-aware clouds," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 170–179, IEEE, 2012.

[8] G. Casale, C. Ragusa, and P. Parpas, "A feasibility study of host-level contention detection by guest virtual machines," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 2, pp. 152–157, IEEE, 2013.

[9] F. Ricci, L. Rokach, and B. Shapira, *Introduction to recommender systems handbook*. Springer, 2011.

[10] M. Y. H. Al-Shamri, "Power coefficient as a similarity measure for memory-based collaborative recommender systems," *Expert Systems with Applications*, vol. 41, no. 13, pp. 5680–5688, 2014.

[11] "Rubis rice university bidding system." [Online]. Available: http://www.cs.rice.edu/CS/Systems/DynaServer/rubis.

[12] T. Xu, X. Sui, Z. Yao, J. Ma, Y. Bao, and L. Zhang, "Rethinking virtual machine interference in the era of cloud applications," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pp. 190–197, IEEE, 2013.

[13] Y. Koh, R. C. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments.," in *ISPASS*, pp. 200–209, 2007.

[14] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "Mitigating interference in cloud services by middleware reconfiguration," in *Proceedings of the 15th International Middleware Conference*, pp. 277–288, ACM, 2014.

[15] J. P. Magalhaes and L. M. Silva, "Detection of performance anomalies in web-based applications," in *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*, pp. 60–67, IEEE, 2010.

[16] J. P. Magalhaes and L. M. Silva, "Anomaly detection techniques for web-based applications: An experimental study," in *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, pp. 181–190, IEEE, 2012.

[17] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th international conference on Autonomic computing*, pp. 191–200, ACM, 2012.

[18] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 452–461, IEEE, 2008.

[19] "Amazon Web services." [Online]. Available: http://aws.amazon.com/ec2/purchasing-options/dedicated-instances/.

[20] R. Jain, "The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling," *New York: John Willey*, 1991.

[21] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pp. 43–52, Morgan Kaufmann Publishers Inc., 1998.

[22] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowledge-Based Systems*, vol. 46, pp. 109–132, 2013.

[23] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.

[24] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, no. 6, pp. 734–749, 2005.

[25] "LittleProxy." [Online]. Available: http://www.littleshoot.org/littleproxy/.

[26] D. Mosberger and T. Jin, "httperf—a tool for measuring web server performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[27] "RAMspeed." [Online]. Available: http://alasir.com/software/ramspeed/.

[28] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The tcp/udp bandwidth measurement tool," *[Online]. Available:https://iperf.fr/*, Last access: Sep 11, 2014.

[29] S. Geisser, "The predictive sample reuse method with applications," *Journal of the American Statistical Association*, vol. 70, no. 350, pp. 320–328, 1975.