Making Neighbors Quiet: An Approach to Detect Virtual Resource Contention

Joel Vallone, *Student Member, IEEE,* Robert Birke, *Senior Member, IEEE,* Lydia Y. Chen, *Senior Member, IEEE,*

Abstract—It is imperative for public cloud providers to guarantee performance targets for tenants' virtual machines (VMs) while respecting strict business confidentiality, e.g., having no information on applications nor their performance. A large body of related work addresses the challenges of detecting performance interferences by leveraging client's quality of service (QoS) metrics, e.g., latency, and additional profiling servers. In this paper, we take the perspective of the cloud provider and propose a general black-box approach that detects different resource contentions by throttling neighboring VMs. Specifically, we design a three-phase detection algorithm that includes: (i) an alarm phase to identify statistical outliers using control charts; (ii) a passive clustering phase to match the current sample to historical behaviors; and (iii) an active throttling phase to discern contentions from application phase changes via throttling. The algorithm is specifically designed for scenarios where multiple co-located VMs request detection analysis simultaneously. We implement and evaluate the proposed three-phase algorithm on four latency sensitive applications, i.e., Wikimedia and three benchmarks from Cloudsuite. Our extensive experimental results show that we can reach an average detection accuracy above 90% while limiting the performance degradation experienced by offender workloads to short learning phases.

Index Terms—contention detection, cloud, QoS

1 Introduction

Virtualization is the key technology enabling the cloud computing paradigm on infrastructure, where a large amount of diverse resources are multiplexed together to cater to an ever increasing number of services. Cloud providers boost system productivity, as well as lower their operational costs, by hosting tenants via virtual machines (VMs) [1]. A commonly seen practice is to collocate applications with disparate performance and resource requirements, such as latency sensitive applications and batch applications. VMs have offered solutions to the problem of software heterogeneity, but not necessarily to the problem of performance isolation [2]. As a result, tenant's perceived Quality of Service (QoS), e.g., request throughput and latency, often degrades on clouds [3, 4] due to underlying resource contention, and substantial financial penalties may be incurred by both cloud providers and their tenants. It is of paramount importance for cloud providers to (pro)actively manage resource contention [5].

The challenges of preventing resource contentions in the cloud start right from the very first step – contention detection. To isolate tenants on the same infrastructure via a virtualization layer, providers have a bird's eye perspective on resource consumption of all VMs but lack high-level application performance metrics associated with the QoS perceived by tenant's. Providers are challenged to indirectly infer application performance from simple low-level resource metrics that preserve tenants' business confidentiality. Moreover, as the cloud enables a higher level of resource multiplexing across heterogeneous VMs, the system complexity and the volatility of workload dynamics drastically increase. Contention detection is thus required to be adaptive to fast application dynamics, including examples of load changes or internal application phases changes [6, 7], across collocated VMs.

There is a significant body of prior art [8, 9, 10, 11, 12, 13, 14, 15, 16] addressing the issues of resource contention and performance interferences in Cloud. Their main focus is to detect the QoS drop, either indirectly by inference from low level metrics [8, 12, 14] or directly by measuring the tenant's perceived QoS [9, 10, 15]. The former centers only on significant resource contention that can lead to obvious QoS drop. The latter implicitly requires application information that might intrude on the business confidentiality of tenants. One of the key ideas behind detecting contention is to create an isolated execution environment, e.g., creating a clone VM on a separate and dedicated server [8] or offline profiling [12, 15], so that one can clearly differentiate between resource contention from inherent application dynamics. In summary, the related work may fall short in detecting a wide range of resource contention, some of which might only have a minor impact on QoS, without requiring additional tenant information and profiling servers.

In this paper, we aim to answer the following challenging research question – how can cloud providers detect a wide range of resource contentions of collocated VMs in an on-line and black-box fashion? To such an end, we develop a new algorithm, which automates the processes of monitoring and the proposed three-phase detection framework: (i) **A**larming behavior change, (ii) passive **C**lustering, and (iii) active **T**hrottling diagnosis. Herein, the algorithm will be referred as *ACT* for short. Our main design idea is that we actively throttle neighboring VMs whenever a behavior

• *R. Birke and L. Y. Chen are in the Cloud Server Technologies Group, IBM Research Lab Zurich, Rüschlikon, Switzerland. E-mail: {yic,bir}@zurich.ibm.com.*

Manuscript received September, 2016.

change is confirmed after the second phase of *ACT*, to create an isolated execution environment at a low overhead. *ACT* monitors low-level resource metrics whose representative patterns are stored in a sliding window fashion. To on-line differentiate the application dynamics from contention, we employ a combination of statistical and unsupervised machine learning techniques, i.e., control chart, history look-up, and k-mean clustering, using data from regular monitoring and active throttling. We extensively evaluate the proposed *ACT* on a prototype cloud that collocates different combinations of latency sensitive applications, i.e., Wikimedia and CloudSuite, and batch-like applications, i.e., PARSEC and Cachebench.

Particularly, *ACT* executes on each physical server and monitors metrics related to the shared resources on VMs in discrete windows: CPU, cache, main memory, network and disk. *ACT* consists of three phases, each of which requires different computational complexity. First, an alarm phase (detect) identifies statistical outlier sample units in the victim's VM resource metrics based on control charts. Second, a passive diagnosis phase (remember) tries to match the outlier sample to historical behavior to minimize the detection impact. Third, an active learning phase (learn) throttles neighboring VMs to distinguish contentions from VM behavior changes. The computation overhead increases in phases. *ACT* is able to parallel execute the first (alarm) and second (cluster) phase on multiple VMs which are collocated on the same physical node, whereas the throttle phase can only be executed one at a time. Our evaluation results show an average detection accuracy above 90% when collocating Wikimedia with PARSEC and Cachebench. We also show that the direct throttling overhead on neighboring VMs can be as low as 15% of completion time increase and further amortized across time.

Our scientific contribution is on methodology as well as on experimental validation. We develop a novel and generic three-phase detection algorithm, which can accurately detect different resource contentions for collocated VMs by active throttling. *ACT* preserves clients' confidentiality and incurs low detection overhead. Our evaluation is on a realistic testbed executing representative workloads, i.e., a combination of interactive and batch applications.

The remainder of the paper is organized as follows. Section 2 presents a motivation example, followed by the system overview in Section 3. We detail the monitoring component in Section 4 and the three-phase detection algorithm in Section 5. In Section 6, we present our evaluation results, followed by a discussion on related work in Section 7. Section 9 concludes with a summary of our work.

2 Motivation

Here, we visually illustrate the challenges in detecting resource contention using a case study on the Wikimedia application [17].

When collocating VMs, the applications contend for shared physical resources which translates into changes in key metrics. One such metric is the instructions per cycle (IPC) used to measure program performance and

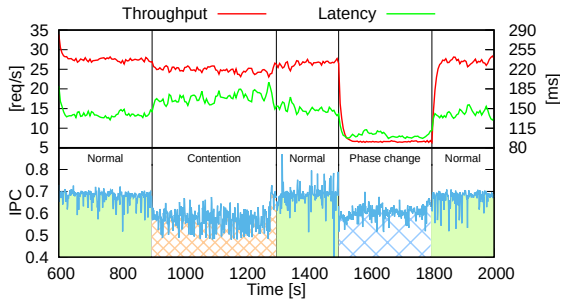


Fig. 1: Ambiguity problem: just observing low level metrics does not help to differentiate contention from phase change.

contention [18, 19]. We deploy Wikimedia on a set of three VMs, each of which is hosted on a separate physical server. The detailed configuration of Wikimedia and servers can be found in Section 6. Table 1 illustrates the QoS metrics of the entire Wikimedia application together with the IPC measured on the Wikimedia frontend VM over time. We mark the entire observation with three labels, namely normal, contention and new phase. The contention period, from 900 to 1300 seconds, corresponds to the activation of a PARSEC benchmark (Blackscholes) [20] inside a VM collocated with the Wikimedia frontend VM. In addition, between 1500 and 1800 seconds, we drop the client request rate to emulate a phase change in the application dynamics.

As the IPC and throughput show similar decreasing patterns both during contention and new phase periods, it is not straightforward to differentiate contention caused by neighboring VMs from a new application phase: With a black-box approach, the challenge is to differentiate contention from application phase change only using hypervisor level metrics. To tackle such difficulty, one of the key elements in related work is to leverage the performance of the victim VM, i.e., Wikimedia, in an isolated execution, either by an initial profiling [12] or by cloning the VM on a dedicated hardware [8]. However, additional hardware and time are thus needed and this argues against the advantages of cloud – efficient and economic resource provisioning. Another observation worth noting is the non-linear relationship between the QoS metrics and contention. QoS may only show a clear difference when there is a high level of resource contention. For example, with the shown Wikimedia frontend, the difference of IPC in normal and contention period is roughly 25%, whereas the throughput and latency of the client requests show differences around 10% and 20%, respectively. As the CPU is rather saturated by the Wikimedia frontend VM and PARSEC VM, there exists a risk of drastic QoS drops when additional VMs appear. Low level resource metrics are more sensitive to potential contention from neighboring VMs, compared to high level QoS. One can view low-level resource contention as a necessary condition for interference. We thus advocate that detecting resource contention can be viewed as a conservative proxy for detecting QoS drops, without leveraging any application information. The immediate challenges of detecting contention arise from the large number of resource metrics, and the distributed nature of applications in the cloud.

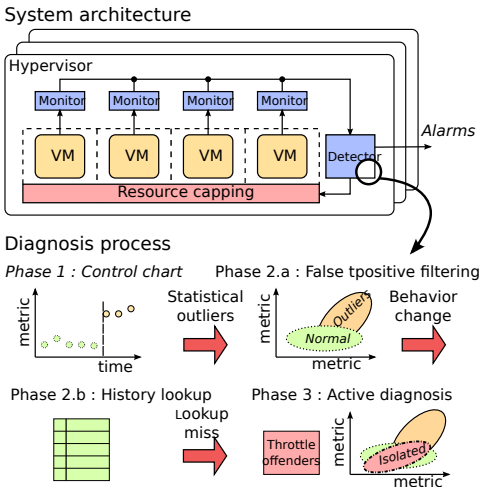


Fig. 2: System architecture: each VM resource usage is monitored and fed into a three-phase detection algorithm, which relies on throttling VMs to discern the ground truth.

3 System Overview

The contention detection system depicted in Fig. 2 operates in parallel to the tenants' VMs inside each hypervisor and is made up of two main component types: monitors, one for each running VM, and detectors, one for each hypervisor. Each monitor periodically collects the resource contention metric data of its coupled VM and feeds it to the detector residing on the same hypervisor. The detector can execute our three-phase detection algorithm for multiple VMs of interest, such as latency-sensitive application VMs. Each phase requires different degrees of computational complexity and results in different levels of detection accuracy. During the first phase (**A**larm), each detector processes the incoming monitoring stream to identify outliers signaling the insurgence of a possible contention. If outlier sample units are found, the detector goes into the second phase (**C**lustering). This passive diagnosis phase tries to infer contention by clustering the current contention sample to known VM behaviors. If the sample can not be classified using historical data, the active diagnosis phase (**T**hrottling) starts. During this phase the detector tries to remove any contention by throttling all neighbouring VMs to emulate an in-isolation run of the victim VM. If the sample before and during throttling can be divided into two distinct groups, the detector can classify the behavior as contention.

On the one hand, scale-out scalability is not an issue since the detection algorithm runs independently in parallel on all hypervisors. On the other hand, the throttle phase limits the number of active diagnosis to one at a time affecting scale-in scalability. In practice, this limitation is eased by the fact that the active throttle phase is necessary only in a few cases.

3.1 Assumptions

The detection algorithm relies on a set of underlying assumptions to work well. The main one is that the victim VM is in a steady state during each phase of the diagnosis process. By steady state we mean constant client load or computation load. In practice, load-balanced client-facing services and big batch workloads appear to be in steady state when the observation window lasts a few tens of seconds. Another assumption is that neighbouring VMs can tolerate sporadic CPU capping. This is true for VMs running batch jobs but might be an issue for latency sensitive applications. Finally, the algorithm assumes to know an initial ground truth on the normal VM behavior. For the moment, we rely on normal samples gathered in isolation conditions. Lifting this last assumption is left for future work.

TABLE 1: Monitored VM metrics.

Metric name	Source	Resource	Context
Instructions retired Per Cycle	`perf`	All	VM PIDs
CPU load buffer stalls cycles ratio	`perf`	Mem. & LLC	VM PIDs
LLC miss per K instructions	`perf`	LLC	VM PIDs
Net. card TX queue len. in Bytes	`tc`	Net. card TX	Virt. iface name
Disk read queue avg. wait time	`iostat`	Disk	Virt. disk name
Disk write queue avg. wait time	`iostat`	Disk	Virt. disk name

4 Monitor

We rely on a monitoring system which consists of many small monitor blocks, one for each running VM. Each VM is monitored directly from the hypervisor level making the whole monitoring infrastructure transparent to the VMs themselves. Each monitor collects different metrics using standard monitoring tools such as `perf`, `tc` and `iostat`. These tools are contextualized to monitor only the specific VM attached to the monitor. This is achieved by monitoring either a specific virtual device attached to the VM or filtering the thread group of the QEMU emulation process. The main focus is on contention metrics used by *ACT*. Each monitor is also able to collect load metrics. Table 1 summarizes the metrics used by the diagnosis tool, the associated monitoring tools and their contextualization.

As seen in Table 1, the focus is on metrics which allow contention on shared resources to be highlighted, since contention metrics are more informative on possible interference cases. To minimize measurement error, we limit the `perf` metrics so as not to exceed the number of multi-purpose hardware performance counters available on Intel[1] i7 CPUs.

The main requirement of the monitor block is to induce the least amount of overhead as possible. Each monitor can easily collect more metrics than the ones listed, including load metrics, but each additional metric increases the measurement overhead. Hence we limit ourself to this list.

Particular care has to be given to the set of metrics collected via `perf`, since `perf` relies on hardware performance counters. If the number of metrics exceed the number of performance counters available within the processor, `perf` time multiplexes the different metrics at the expense of measurement accuracy. Each processor in our testbed is equipped with four performance counters. Thus, we limit the monitored metrics to IPC, Last Level Cache (LLC) misses per K instructions and load buffer stalls. IPC is a good

1. Intel is a trademark or regsitered trademark of Intel Corporation or its subsidiaries in the US or other countries. Linux is a registered trademrk of Linus Torvalds in the US or other countries or both. Other product and service names may be trademarks or service marks of IBM or other companies.

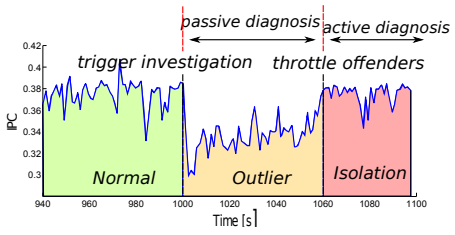


Fig. 3: Three-phase diagnosis cycle. Sample units are separated in three types of intervals: normal, outlier, and isolation, corresponding to alarm, cluster, and throttle phase.

overall measure of CPU stall produced by contention. LLC misses highlight cache conflicts and load buffer stalls increase when the memory system is under heavy contention. We investigated the possibility to monitor other metrics such as L2 D/I misses or arithmetic unit stalls but did not retain them due to the low detection accuracy improvement over the retained metrics.

Limiting the number of low-level metrics collected by `perf` to the number of available hardware performance counters allows a new sample unit to be produced every second without loss of accuracy. Moreover, from preliminary experiments the 1-second sampling frequency results in a good compromise between accuracy, speed and monitoring overhead. All metrics are gathered at the same time and stored in a vector where each element is the measured value of each metric. This vector can be seen as a point in a multidimensional space with one dimension per metric.

5 Detector

The detector block collects the measurement data from all VM monitors collocated under the same hypervisor and runs the three-phase on-line detection algorithm. The detection algorithm can run in parallel on the monitored VMs during the behaviour change detection and the passive diagnosis but only one VM can be actively diagnosed at a time because of neighbouring VMs throttling. The algorithm aims to detect VMs contending on shared resource(s). Contention relates to a bad resource sharing which can mainly be observed as a step change on related contention metric(s).

As an example we present a simplified case where we consider only one metric. Figure 3 depicts the IPC of a victim VM across time. At the beginning, the server is in contention free behavior (normal). We consider the sample as normal when its values match isolation conditions. At 1000 seconds, we have a behavior change in the server due to a neighbouring VM starting a heavy CPU-intensive task (outlier). This creates a change in the IPC trace. The first phase of the algorithm continuously monitors the VM to identify such a change between the normal and outlier parts across any of the metrics. Once in the outlier part, the second phase of the algorithm tries to classify the new behavior by collecting outliers and comparing them to historical data. In the case of miss, the algorithm goes into the third and last phase: throttling of the neighbouring VM(s) (isolation). The 3-phase detection loop is summarized in Algorithm 1, whereas each phase is described in detail in the following subsections.

Algorithm 1 *ACT*– Detector loop

```
while true do
    x = monitorVM()
    updateSample(x, W_normal)
    if detectChange(W_normal) == false then                    ▷ Phase 1: Alarm
        continue
    end if
    for i = 0; i < w; i++ do                                   ▷ Phase 2: Cluster
        x = monitorVM()
        updateSample(x, W_outlier)
    end for
    o = filterFalsePositives(W_normal, W_outlier)              ▷ Phase 2.a
    if o == falseAlarm then
        continue
    else if historyLookup(W_outlier) == match then             ▷ Phase 2.b
        signal outcome of match
    else
        throttleNeighbourVMs()                                 ▷ Phase 3: Throttle
        for i = 0; i < w; i++ do
            x = monitorVM()
            updateSample(x, W_isolation)
        end for
        unthrottleNeighbourVMs()
        if classifyBehavior(W_normal, W_outlier, W_isolation) != failure then
            signal outcome of classification
        end if
    end if
end while
```

The challenge is to differentiate contention from application phase change. An application phase change can be observed when the running code base inside the VM changes creating different measurement patterns due to the program entering a different part of its code. For latency sensitive workloads, the variation of client load is also considered as a phase change. To find the ground truth, we isolate the VM from its neighbors by throttling. If it is an application phase change, the measurements are not affected by the neighbouring VMs and hence should remain unchanged even in isolation. If it is contention, the measurements should improve when in isolation. In Section 6 we show that the throttling cost, even in terms of performance drops of the neighbouring VMs, is limited.

5.1 Alarm phase (I): behavior change detection

The goal of the first phase is to have a computational cheap way of detecting behavior change to minimize the detection overhead. In this phase, fast detection is preferred over accuracy, since the accuracy will be achieved by the subsequent phases.

During this phase, the detector continuously monitors all the metrics of all the VMs independently. We rely on Shewhart control chart [21] to detect operation anomalies on any of the metrics. The basic idea is that during an anomaly the measures will start deviating and the anomaly can be detected by setting a threshold on the deviation. Shewhart control chart is known as a robust statistical quality control which does not make any assumption on the underlying probability distribution. The method uses a moving window $\mathbf{W} = \{x_{t-1}, x_{t-2}, \cdots, x_{t-w}\}$ of size w containing the last sample units to compute the moving sample mean $\overline{x}$ and range mean $\overline{MR}$ as:

$$\overline{x} = \frac{1}{w}\sum_{i=1}^{w} x_i \tag{1}$$

$$\overline{MR} = \frac{1}{w-1}\sum_{i=2}^{w} |x_i - x_{i-1}| \tag{2}$$

When a new measurement x_t arrives, an anomaly is signaled if condition (3) is not satisfied.

$$\overline{x} - 2.66\,\overline{MR} \leq x_t \leq \overline{x} + 2.66\,\overline{MR} \quad (3)$$

where 2.66 is the commonly recommended value given that the moving range is estimated using two points [21]. In practice this method works but is too sensitive to noise. To make the detection resilient to noise, we count the number n of consecutive outliers where sample units are labelled as outliers based on the estimated mean $\overline{x}$, standard deviation σ and the three-sigma rule, i.e., a new value x_t is an outlier if condition (4) is not satisfied.

$$\overline{x} - 3\,\sigma \leq x_t \leq \overline{x} + 3\,\sigma \quad (4)$$

where

$$\overline{x} = \frac{1}{w}\sum_{i=1}^{w} x_i,\ \sigma = \sqrt{\frac{1}{w-1}\sum_{i=1}^{w}(\overline{x} - x_i)^2}$$

Since we monitor sets of metrics the above condition is applied separately to each metric and the first to violate the threshold raises the alarm.

Outlier sample units are counted but not immediately added to the moving window $\mathbf{W}$ of size w containing the last sample units $\mathbf{W} = \{x_{t-1}, x_{t-2}, \cdots, x_{t-w}\}$. If n is greater than a threshold n_t, the next phase starts, otherwise the outliers are added to $\mathbf{W}$. In the following, we set $n_t = 5$ and $|W| = 30$ which proved to be good choices in preliminary experiments.

5.2 Clustering phase (II): passive diagnosis

The goal of the second phase is twofold. The first goal is to classify the outlier sample in the alarm phase as noise or representative of a new system behavior, i.e., false positive filtering. The second goal is to discern contention from phase change based on historical data, i.e., history lookup, to avoid the cost of an active diagnosis (throttling phase). Please note that while phase 1 considers each metric independently, from here on we consider the whole VM measurement vector as a representation of a point in the multidimensional metric space.

False positive filtering: The problem with phase 1 is that it can easily capture noise and result in false alarms as observed from preliminary experiments. To reduce the number of false positives, phase 2 first performs a filtering based on clustering. The algorithm accumulates measures for the outlier sample$\mathbf{W_{outlier}}$ from the new system behavior identified by phase 1 having the same size w as the normal sample $\mathbf{W_{normal}}$ collected in the previous behavior. After that, it runs K-means on the union $\mathbf{W_{normal}} \cup \mathbf{W_{outlier}}$ with $K = 2$. The rationale is that if phase 1 correctly detected a new system behavior, clustering should highlight two very distinct samples of roughly equal size (since $|\mathbf{W_{outlier}}| = |\mathbf{W_{normal}}|$): one representing the previous normal behavior mostly composed of normal sample units and one representing the new behavior mostly composed of outlier sample units. On the contrary, if phase 1 triggered on noise, this clear separation will not be true. Algorithm 2 details all the filtering steps. The *leader()* function, given as input a set of points and a threshold C, returns the identifier of the cluster which is both the biggest cluster in terms of number of points and comprises at least C percent of the total points. If no such cluster is found, a negative value is returned. This ensures that the normal sample clearly defines one cluster while the outlier sample clearly defines the other cluster. More precisely, we never want a negative return value from *leader()* and the return values on the two sets should not be equal. The C parameter can be considered as a characterization threshold over the clustering degree of a subset of points. In practice, we want a high characterization on the normal cluster, hence $C_{normal} = 70\%$, while we want enough outlier sample units in the new cluster to highlight a tendency rather than noise. We investigate the meaning of enough in Section 6.3 by varying $C_{outlier}$.

Algorithm 2 False positive filtering

```
P = W_normal ∪ W_outlier
clusters = kMeans(P, 2)
n = clusters.leader(W_normal, C_normal)
o = clusters.leader(W_outlier, C_outlier)
if n = −1 ∨ o = −1 then
    return falseAlarm
else if n = o then
    return falseAlarm
else
    continue diagnosis
end if
```

History lookup: For each VM the past known normal and contention behaviors are stored in a history table populated by phase 3. For storage compactness and computational speed, we approximate each cluster as a sphere in a normalized multidimensional space of the metrics. This allows us to store each cluster k simply by its outcome, i.e., contention or phase change, its centroid $\mathbf{C_k}$ and its radius r_k, together with normalization factors, i.e., mean $\mu_{i,k}$ and standard deviation $\sigma_{i,k}$, for each metric i. At this point a history lookup of a new VM behavior simply translates into checking if a majority of outlier sample units are within a cluster radius from the VM history table based on the Euclidean distance between the cluster centroid and each of the normalized sample units.

More formally, we first normalize each vector element x_i of sample unit vector $\mathbf{x}$ using $\mu_{i,k}$ and $\sigma_{i,k}$ of the cluster k to obtain a normalized sample unit $\mathbf{x_k^n}$

$$x_i^n = \frac{x_i - \mu_{i,k}}{\sigma_{i,k}} \quad (5)$$

Once normalized, $\mathbf{x_k^n}$ is part of cluster k if:

$$||\mathbf{x_k^n} - \mathbf{C_k}|| < r_k \quad (6)$$

under Euclidean distance. μ_i, σ_i and r are computed from the cluster sample units during phase 3. While μ_i and σ_i are trivially the mean and standard deviation over those values, to avoid ambiguous points the radius r is computed as the 90th percentile of the Euclidean distance distribution of the sample units. Note that normalization is necessary because each metric has its own range of values and thus a direct comparison across different dimensions would be difficult. In practice, we require at least $C_{history}$ percent of outlier sample units to be part of a cluster to signal a lookup hit, where $C_{history}$ is set to 70%. This operation is repeated over all history table entries. In the case of a hit, the diagnosis is stopped and the outcome stored in the history table is

Algorithm 3 Behavior change classification

$\mathbf{P} = \mathbf{W_{normal}} \cup \mathbf{W_{outlier}} \cup \mathbf{W_{isolation}}$
clusters = kMeans($\mathbf{P}$, 2)
n = clusters.leader($\mathbf{W_{normal}}, C_{normal}$)
o = clusters.leader($\mathbf{W_{outlier}}, C_{outlier}$)
i = clusters.leader($\mathbf{W_{isolation}}, C_{isolated}$)
if $n = -1 \vee o = -1 \vee t = -1$ **then**
  return diagnosisFailure
else if $i = n$ **then**
  return contention
else if $i = o$ **then**
  return newPhase
end if
return diagnosisFailure

returned. In the case of multiple hits, the cluster with the highest percent of matching sample units is selected. Finally, in the case of a miss, the active diagnosis phase is invoked.

Over time the size of the history table grows as the detector identifies new VM behaviors. To avoid excessive slowdowns and memory footprints, one can easily limit the number of entries in the history table, but at the cost of extra active diagnosis phases. In practice, this problem is limited due to the fact that only a few new behaviors are discovered, mainly at system startup, while later the algorithm relies on history hits as seen in Section 6. The trade off between the table size and the impact of extra throttling phases is left as interesting future work.

5.3 Throttling phase (III): active diagnosis

The goal of the third phase is to isolate the victim VM from all neighbouring VMs to identify the ground truth. Contrary to the related work, to isolate the VM we rely on throttling which has several advantages. (i) It is easy to deploy: it does not rely on any special feature not readily available in most modern hypervisors. (ii) It is completely application transparent: it does not require any prior knowledge on or cooperation from the VMs. (iii) Throttling by itself bears a minimal cost, especially compared to more complex schemes such as priori off-line VM characterization or VM cloning. The main disadvantage is the performance impact on the applications running inside the throttled VMs. However, in Section 6, we show that the impact, although perceivable, can be maintained within limits both time-wise and performance degradation-wise. Moreover, after some initial learning phases to populate the history table, throttling is mostly avoided.

In more detail, if phase 2 is not able to classify the outliers, phase 3 throttles all the neighbouring VMs capping their CPU usage and accumulating a new isolation sample $\mathbf{W_{isolation}}$ of same size w. After that, the same principles used in phase 2 are applied to discern phase change from contention. Algorithm 3 details the steps. We start with K-means using $K = 2$ on the union $\mathbf{W_{normal}} \cup \mathbf{W_{outlier}} \cup \mathbf{W_{isolation}}$, and identify the main cluster in all three samples via the *leader()* function. For an accurate diagnosis, we want a good characterization on the isolation sample by setting $C_{isolated} = 70\%$ to be noise tolerant but keep the clusters sufficiently reliable.

If the cluster of isolation sample units matches the cluster of normal sample units, the diagnosis classifies the outliers as contention. The rationale is that, by removing/mitigating the impact of the neighbouring VMs, the victim VM behaves as normal because the behavior change was contention induced. On the contrary, if the cluster of isolation sample units matches the cluster of outlier sample units, then we witness a new VM behavior because the neighbouring VMs had no influence on the behavior change of the victim VM. In either case, the diagnosis succeeded and the history table is updated with a new entry. In the case of a contention, this new cluster is based on the outlier sample, whereas in the case of a new phase, the cluster is based on the isolation sample. If neither condition is true, the diagnosis failed. Independently of the outcome, the detection algorithm restarts from phase 1 with its state reinitialized using the last normal sample in the case of diagnosis failure or contention, or using the outlier sample in the case of phase change.

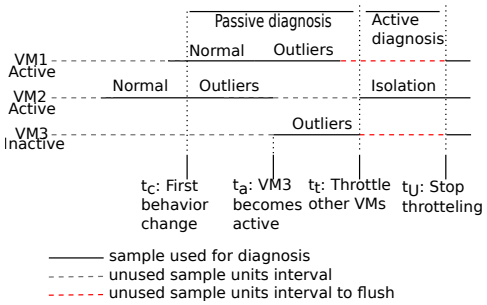


Fig. 4: An illustration of *ACT* parallel analysis on three VMs of interest. The alarm and cluster phases are executed independently for each VM, whereas the active throttle phase is only allowed for VM 2.

5.4 Parallel analysis

We illustrate how to extend *ACT* to the scenario of multiple VMs of interest and conduct a so-called parallel analysis. In contrast to the previous subsection, *ACT* monitors the performance metrics of, and detects contention across multiple VMs concurrently. This introduces two additional complexities to be addressed by the detector. First, while the alarm and cluster phases are completely passive and can easily be performed in parallel on each VM under scrutiny, the throttle phase is active and requires coordination across all the VMs. Second, VMs might switch between active and inactive states during detection which can introduce extra perturbations to the detector. In the following, we first illustrate an example of applying *ACT* on three latency-sensitive VMs hosted on the same physical server and then explain how to coordinate throttling and handle VM state switches in practise.

3-VM example: We apply *ACT* on the three VMs of interest shown in Fig. 4. VM 1 and VM 2 are active during the whole observation period, whereas VM 3 becomes activate at time t_a. We note that there might be other VMs collocated on the same physical server which do not enter in the set of VMs of interest, e.g. VMs running latency-insensitive batch workloads. In our example VM 1 and VM 2 start in a condition classified as normal, until there is a change in the server load due to any of the hosted VMs. This change is first detected by the alarm phase of VM 2 followed by VM 1 which independently switch to collecting outliers. Upon collecting enough outliers for the passive diagnosis

of the cluster phase, *ACT* decides if an active throttle phase is needed for individual VMs. In this example, *ACT* reaches the decision of entering the throttle phase for VM 2 and throttles all its neighbouring VMs on the same server for the purpose of collecting the isolation sample.

Coordination of throttle phase: While the passive diagnosis of the cluster phase of *ACT* can be operated in parallel on all VMs of interest, *ACT* allows only one VM of interest to enter the throttle phase. In other words, *ACT* waits for the cluster phase for all VMs to finish before admitting one VM to the throttle phase. When only one VM is investigated, the passive diagnosis lasts for the duration of collecting the VM outlier sample. However, when multiple VMs are monitored, the passive diagnosis may last longer. Indeed, when a VM experiences a behaviour change on shared resources, it is likely that the other VMs are affected too, see VM 2 and VM 1 in our example. Thus, the cluster phase accepts other VMs for investigation until all have both their normal and outlier samples available, i.e., t_t in our example. Then, based on the outcome of the diagnosis of the cluster phase, if multiple VMs need to execute the active diagnosis, only one is selected, and the others fall out the diagnosis iteration. We prioritize the VM having the least active diagnosis count, i.e., the smallest history.

Handling VM state switches: We consider the VM activity state when running the diagnosis. We define a sample unit as active if at least one of the measured resources (i.e., CPU, disk or network) has a utilization above a given threshold, e.g., 1% in our experiments. During each diagnosis, samples must have a majority (90%) of their sample units marked as active. Otherwise, the diagnosis fails. Similarly we define a VM as inactive after twenty consecutive VM sample units are marked as inactive, and as active otherwise. Once a VM becomes inactive we exclude this data from the diagnosis. In contrast, a newly active VM may already start in a contention scenario (see VM 3 in our example). Hence the newly-active sample is considered as an outlier sample in the cluster phase. If the sample matches a history entry marked as *phase-change*, the sample is considered as normal and the diagnosis finishes. Otherwise, we need to collect an initial (ab)normal sample. We thus need to enter active diagnosis and throttle neighbors for such a VM so as to figure out if it starts with a normal sample.

6 Evaluation

In this section, we present an extensive evaluation of *ACT* on a prototype system that collocates latency sensitive and batch-like applications. We show the effectiveness of *ACT* in detecting resource contention for latency-sensitive applications under various collocation scenarios and parameter settings. The specific metrics of interest are detection accuracy, delay, and overhead that is measured on the throughput degradation of batch-like applications. We first show detection rates and delay of scenarios where one latency-sensitive is collocated with one batch-like application. We then extend to parallel scenarios where multiple *ACT*s monitor latency-sensitive applications in parallel. Moreover, via long-running time-varying experiments, we show that *ACT* can effectively differentiate contention, from internal program change as well as workload dynamics

6.1 Testbed setup

The testbed is composed of four identical physical servers connected via a star topology to a Gigabit switch. Each server runs Ubuntu server 14.04 LTS and is equipped with 16 GB of DDR3 RAM, a 4-core Intel[1] Core i7-3820 processor @ 3.6 GHz with SMT, one 2-TiB Sata III 7200 rpm hard disk, and one Gigabit Ethernet adapter. Three servers host the VMs used to deploy the applications. We run QEMU v2.0 with KVM on top of Linux[1] kernel 3.13 as the hypervisor. Each VM comprises two virtual CPUs and 2 GB of RAM. In most cases, we host two VMs on each server: (i) one VM running latency-sensitive application or its component, and (ii) the other VM running batch workloads to generate contention. The fourth server is used as the experiment orchestrator and load generator.

Latency-sensitive applications: We use Wikimedia as a representative cloud workload [17]. Wikimedia is a latency-sensitive three-tier web application composed of Apache (v2.4.7) plus PHP (v5.5.9) as the application server frontend, Memcached (v1.4.14) as the in-memory key-value store and MySQL (v5.5.40) as the database backend. Each component is deployed into separate VMs which in turn are hosted on separate servers. As a reference, the maximum sustainable throughput of this setup is 38 request/s.

We also use three of the Cloudsuite [22] latency-sensitive workloads to further validate the effectiveness of *ACT*. First, there is data-serving which runs Cassandra (v0.7.3), a distributed scale-out database. The second workload is data-caching, a Memcached (v1.4.14) instance storing a twitter data-set.The last workload, Media-Streaming, is a Darwin Streaming server (v6.0.3) stressed by Faban RTSP clients.

Emulated contention: To create contention, we spawn, in parallel with Wikimedia and/or Cloudsuite workloads, neighboring VMs running two different resource intensive batch workloads resulting in two different types of contention. On the one hand we use a subset of scientific applications from PARSEC 3.0 [20], i.e., Fluidanimate, Blackscholes and Freqmine, characterized by regular but intensive CPU and memory usages. We run these benchmarks using the native input dataset size. On the other hand we use Cachebench [23] which gradually pollutes the whole last-level cache (LLC) producing extremely noisy usage traces spanning from low CPU and cache usage to full CPU and cache usage. Since Cachebench has a very short runtime, we create longer contention periods running it in an infinite loop and killing the loop after a timeout. Throttling is based on the `cgroup` kernel module which allows the CPU usage bounds to be specified based on process identifiers (PIDs). Since each Wikimedia component is placed on a different server and has different resource usages, this setup allows multiple contention mixes to be studied at the same time.

6.2 Contention Analysis

Prior to presenting *ACT* results, we first conduct a detailed contention analysis that provides the ground truth

TABLE 2: EXTENSIVE CONTENTION ANALYSIS MEASURED VIA PERCENTAGE OF DEGRADATION ON INSTRUCTIONS PER CYCLE (IPC), LLC MISSES PER INSTRUCTION (LMI) AND DISK WRITE WAIT TIME (DWW): COMPARING BEFORE AND AFTER BEING COLLOCATED WITH FREQMINIE, FLUIDANIMATE, BLACKSCHOLES, AND CACHEBENCH.

Contender \ Victim	Darwin			Memcached			Cassandra			Wikimedia		
	IPC	LMI	DWW	IPC	LMI	DWW	IPC	LMI	DWW	IPC	LMI	DWW
Freqmine	10.1	58.4	77.7	21.0	29.2	0	9.7	106.8	87.5	9.3	16.1	42361.2
Fluidanimate	10.5	166.8	8.4	21.4	110.8	0	9.7	161.6	-13.2	13.5	17.4	6815.3
Blackscholes	9.8	85.3	28814.9	20.4	46.9	0	-14.6	74.6	76.7	15.6	27.5	76.8
Cachebench	10.2	166.3	105.0	20.5	114.6	0	-17.6	147.0	-5.8	11.7	36.3	17.4

and root cause on how latency-sensitive applications and batch workload contend for different resources. Here, we consider all 16 scenarios of combining four latency-sensitive applications and four batch workloads on a single server. We look into three types of resources: CPU, memory and disk. We analyze the contention patterns first from the perspective of latency-sensitive applications and then of the batch workloads.

Setup and metrics. Specifically, we focus on three metrics: IPC, last-level cache misses per instruction (LMI), and disk write wait time (DWW) which are indicators for contention on three different resource types, i.e., CPU, memory, and disk, respectively. To quantify the contention effect from the neighboring batch workload, we use the degradation of metrics , comparing its values before and after having the neighboring batch workload. We thus use offline profiling to compare these metrics collected on VMs running a latency-sensitive application in isolation against the collocated scenario with a neighboring VM running a batch workload. In addition to Wikimedia, we use three latency-sensitive Cloudsuite benchmarks: media streaming (Darwin), data serving (Cassandra) and data caching (Memcached). As for the batch workloads, we use Blackscholes, Freqmine and Fluidanimate from PARSEC and Cachebench [23].

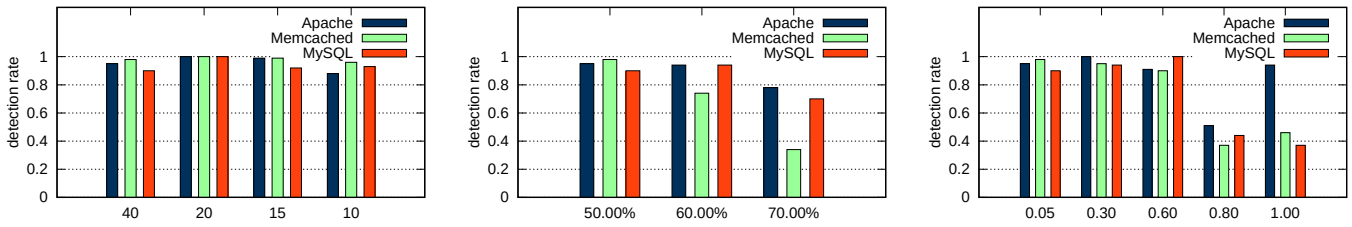
From latency-sensitive applications. Table 2 summarizes the results for media streaming, data serving, data caching and web serving, respectively. Each figure shows the metrics degradation/improvement in percentage after being collocated with batch workloads. The degradation is computed as the performance difference divided by the original performance value. For the IPC, we compute the difference by the value before collocation subtracted by the value after collocation, whereas for LMI and DWW we compute the difference by the value after collocation subtracted by the value before collocation. This is to reflect that a higher IPC value is deemed a better performance but higher LMI and DWW values are deemed inferior performance. Negative values of degradation mean that there is actually a performance improvement after collocating with neighboring VMs.

From the results one can observe that VMs do interfere with each other and the degree/type of interference depends on specific combinations of applications and benchmarks. Shown in Table 2, Memcached has the highest IPC degradation among all four applications. Though Memcached can effectively use the memory space to boost the data store, it is also known to have scaling issues when increasing the number of threads [24], indicating that both CPU and memory can be resource bottlenecks. As for Wikimedia, it is sensitive to disk activities, a counter-intuitive finding. Indeed, Wikimedia frontend consists of Apache passing requests to a stateless middleware script written in PHP without explicitly using disk. However, the default configuration of Apache used in our experiments records all requests to a log file on disk. Table 2 shows that Cassandra's DWW is less affected because the workload used to stress the service is read heavy. Darwin is the most memory sensitive application. This is due to the fact that it uses in-memory caching of the content and that the test videos are of comparable size to the last-level cache.

From batch workloads. Freqmine, Fluidanimate and Blackscholes represent compute-intensive real-world applications and at the same time Freqmine and Fluidanimate (particularly) are also memory-intensive [20]. Moreover all three use the disk to either output their results (Fluidanimate and Blackscholes) or load the input datasets (Freqmine). As for Cachebench, it is a benchmark explicitly written to stress the cache, making it both memory- and compute-intensive. These resource characteristics are clearly visible across all four figures. Let us first focus on the IPC degradation which is rather similar in call cases. With a closed check one can see that Fluidanimate and Cachebench typically cause higher IPC degradation, compared to Freqmine and Blackscholes. This observation demonstrates how all four batch workloads compete for CPU with the latency-sensitive applications. In terms of memory contention, Fluidanimate and Chachebench create the highest LMI degradations as expected. It is worth noting that sometimes the collocation of VMs can bring seemingly beneficial effects. This is the case in Table 2 which shows an improvement of the IPC and DWW for Cassandra, i.e., the negative degradation numbers. We attribute this positive effect to the CPU power governor which increases the clock frequency in response to an overall higher load.

6.3 Effectiveness of *ACT* on Wikimedia

Here, we show a sensitivity analysis on the impact of the *ACT* parameters on detecting resource contention caused by the collocated batch applications for Wikimedia. First, the sample sizes of $\mathbf{W_{outlier}}$ and $\mathbf{W_{isolation}}$ used in the passive and active diagnosis phase, respectively. Intuitively, the bigger the sample, the more time is required to collect it but the lower the risk of considering noise as an actual trend. Note that, on average, one sample unit is produced every 1.15 seconds. Second, the CPU capping limit Th_{CPU} imposed during throttling. The higher the capping the less impact we have on the neighboring VMs. Of course, with less capping the sample of the victim VM will not represent a perfect isolation case and the diagnosis is more likely to fail. Third, the threshold $C_{outlier}$ used to find



(a) Sample size $|\mathbf{W_{o.}}| = |\mathbf{W_{i.}}|$ [#] (b) $C_{outlier}$ [%] (c) CPU limit Th_{CPU} [cores]

Fig. 5: Contention detection rate under different parameter settings of *ACT*.

the outlier cluster. The higher this threshold, the clearer the trend must be to generate alarms, lowering the risk of false positives. However, *ACT* might miss contention cases because the threshold is too restrictive. While tuning one parameter we use the following conservative values for the other two: $|\mathbf{W_{outlier}}| = |\mathbf{W_{isolation}}| = 40$ sample units, $Th_{CPU} = 0.05$ cores and $C_{outlier} = 50\%$.

Experiments consist of Wikimedia running at full load, while we randomly spawn on each server an instance of either PARSEC or Cachebench inside the neighboring VM. 2-minute breaks are spaced out by offender intervals lasting between 3 to 4 minutes. There is an independent 50% probability that the next interval will spawn a batch job on each server. Whenever a batch job is spawned, contention systematically occurs. Each run lasts 3 hours and is repeated 3 times. In the following we present the average results across the runs.

6.3.1 Detection rate

We first evaluate the detection rate defined as the ratio of correctly identified contention intervals (true positives) over all known contention intervals. We consider a contention interval to be correctly identified if one or more alarms were raised in it. An ideal algorithm always correctly identifies contention intervals, i.e., detection rate equal to 1. Fig. 5 shows how the detection rate changes under different tunable parameter values.

We start by varying the number of sample units $|\mathbf{W_{outlier}}|$ and $|\mathbf{W_{isolation}}|$ per diagnosis interval (see Fig. 5a). The best results, detection rates equal or close to 1 across all three Wikimedia components, are obtained with 15-20 sample units. With less units the measurement noise is higher and the diagnosis is more likely to fail. With more units the noise effects are reduced, but since each diagnosis takes longer, less diagnosis cycles can be repeated in the case of diagnosis failures before the contention disappears. Naturally this case is directly related to the expected contention length. Both conditions result in lower detection rates, especially for the Apache and MySQL components.

Moving on to the impact of the outlier threshold $C_{outlier}$, one can observe in Fig. 5b a clear decreasing trend: the higher the threshold, the lower the detection rate. This is true for all three Wikimedia components. Indeed, when we require a clearer clustering, only the VMs under heavy or obvious contention will maintain a high detection rate. Memcached is especially affected with significant drops at both 60% and 70%. This is because this VM has the least resource usage and thus its sensitivity to neighboring VMs contending for shared resources is the lowest.

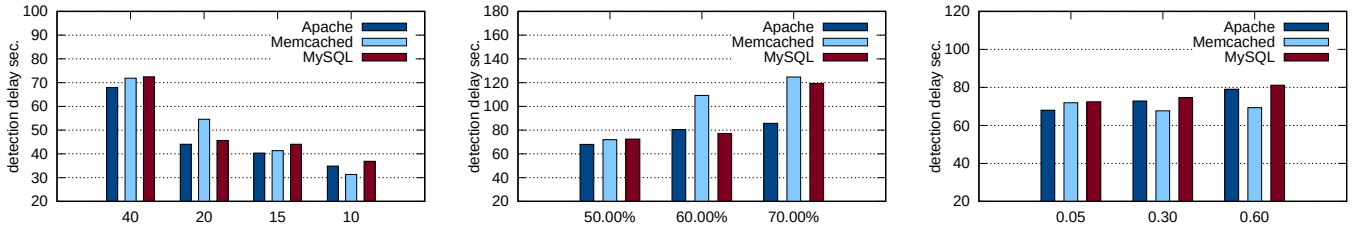
Comparing the detection rate against the CPU capping limit Th_{CPU}, we observe that the active diagnosis phase does not need to fully throttle the neighboring VMs to be able to identify contentions. As highlighted in Fig. 5c, the detection rate remains stable until Th_{CPU} reaches 0.6 cores. Beyond this, it rapidly degrades. However, even if we show the average across both contention scenarios, PARSEC and Cachebench have different influences. Apache is more sensitive to the aggressive CPU usage of PARSEC, while the other two Wikimedia components are more sensitive to Cachebench. Using the more aggressive neighboring VM, contention can be discerned even in low isolation conditions, otherwise the detection collapses. The fact that even low CPU throttling is sufficient to identify contention is desirable because it directly reduces the performance impairment experienced by the throttled VMs.

Parameter Tuning: Overall, the best detection rates are obtained using $15 < |\mathbf{W_{outlier}}| = |\mathbf{W_{isolation}}| < 20$ for the best tradeoff between measurement noise and diagnosis cycle rate. The CPU cap can be rather high as long as $Th_{CPU} < 0.6$ cores. Finally the cluster threshold should not be too strict with $C_{outlier} < 60\%$.

6.3.2 Detection delay

A detection system should not only be accurate, but also fast. We define the contention detection delay as the time elapsed between the start of a contention interval and the first alarm raised by the detector. Fig. 6 shows the average detection delay over all correctly identified contention intervals across the same previously presented parameter sweeps.

Intuitively, the more sample units used for the diagnosis, the longer the detection delay, since new points are sampled periodically. Indeed, the theoretical minimum delay is given by the number of sample units in the passive phase, assuming a history hit is possible, multiplied by the effective sampling interval, i.e., 1.15 seconds in our experiments. Fig. 6a shows both the minimum theoretical delay and the measured ones partly confirming our intuition. However, one can observe that the decreasing tendency quickly reaches a lower bound. If the samples are too small, the clusters are poorly characterized, hence diagnosis failures are more likely. The increase in extra delay with fewer sample units



(a) Sample size $|\mathbf{W_{o.}}| = |\mathbf{W_{i.}}|$ [#] (b) Characterization $C_{outlier}$ [%] (c) CPU limit Th_{CPU} [cores]

Fig. 6: Contention detection delay under different parameter settings of *ACT*.

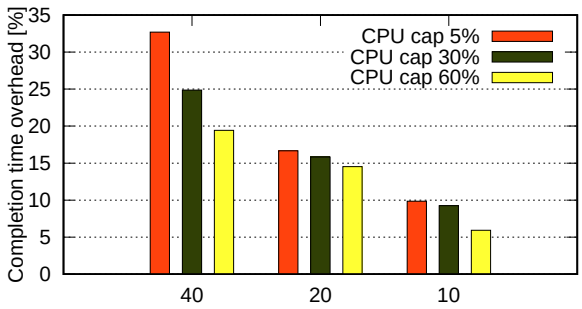


Fig. 7: Average completion time overhead of throttled PARSEC neighboring VM. X-axis: sample size $|\mathbf{W_{o.}}| = |\mathbf{W_{i.}}|$.

introduced by failed and restarted diagnoses can be clearly seen from the increasing relative distance to the theoretical minimum, i.e., overhead, across all three Wikimedia components. Taking Apache as an example, the relative overhead grows from 48%, corresponding to approximately 1 extra diagnosis every 2, to 230%, corresponding to approximately 2 extra diagnoses for each. Similar results hold for the other two components. Combining these results with our previous analysis where a sample size is of 15 or 20 units gives a good trade-off between detection speed and accuracy.

A similar trend is shown in Fig. 6b. Increasing the outlier threshold, increases the probability of failed diagnoses, which in turn increases the detection delay since multiple diagnosis cycles are necessary to raise the first alarm. However, here the influence is lower and the average overhead across different values of $C_{outlier}$ changes from 53% to 86% for Apache and from $\approx$ 60% to $\approx$ 170% for Memcached and MySQL which experience less contention due to lower resource usages.

Finally, considering throttling, the detection delay stays stable as long as the CPU capping limit is low enough for the detection rate not to collapse (see Fig. 6c).

Parameter Tuning: The detection delay is mostly dependent on the number of sample units collected and the success/failure of a diagnosis. The best trade-off is obtained with $|\mathbf{W_{outlier}}| = |\mathbf{W_{isolation}}| = 20$ sample units, $Th_{CPU} = 0.05$ cores and $C_{outlier} = 50\%$. In our tests this translates into a detection delay of 40 seconds and a detection rate never below 90%.

6.3.3 Throttling overhead

We quantify the throttling overhead as the increase in completion time experienced by the neighboring batch

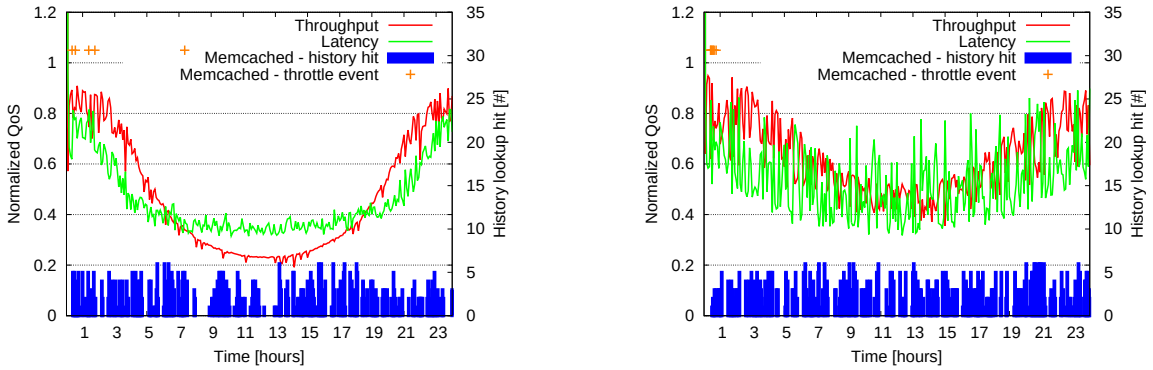
TABLE 3: ACTIVE DIAGNOSIS RATIO.

	$\|\mathbf{W_{o.}}\| = \|\mathbf{W_{i.}}\|$ [#]			Th_{CPU} [cores]			$C_{outlier}$ [%]		
	10	**20**	**40**	**0.05**	**0.3**	**0.6**	**50%**	**60%**	**70%**
Apache	0.05	0.03	0.07	0.07	0.06	0.10	0.07	0.06	0.05
Memc.	0.03	0.04	0.06	0.06	0.03	0.10	0.06	0.09	0.28
MySQL	0.02	0.04	0.08	0.08	0.11	0.07	0.08	0.03	0.07

jobs. More precisely, we concentrate on the longer running PARSEC workloads. For each experiment run, we quantify the relative time loss for each workload as the difference between the mean execution time during throttled and non-throttled intervals divided by the mean non-throttled execution times. Fig. 7 shows the mean results across the three PARSEC workloads while varying the sample size and the CPU capping limit. We see a clear trend with respect to both. A larger isolation sample directly translates into longer throttling intervals to collect the sample units, hence higher overheads, while shrinking the isolation sample size shrinks the throttling time and overhead. In the case of the CPU capping limit, the opposite is true. Higher CPU limits reduce the performance impairment. Finally, the time loss is stable with respect to $C_{outlier}$, since the parameter has no direct influence on the throttling interval (not shown due to space issues).

Even in the worst case, i.e. highest number of sample units and lowest CPU capping limit, the direct overhead is limited to 33%, while for the more reasonable parameter values, i.e., $|\mathbf{W_{outlier}}| = |\mathbf{W_{isolation}}| = 20$ and $Th_{CPU} = 0.3$ cores, the overhead drops to 15%. Moreover one must take into consideration that this overhead is only incurred when *ACT* executes the active throttle phase. These phases are a small fraction of the overall diagnosis cycles as shown in Table 3. In most experiments, less than 10% of detection cycles rely on the active diagnosis phase. Only in the case of high outlier threshold $C_{outlier}$ on Memcached did *ACT* have difficulties. The reason is that Memcached creates a light contention which is difficult to identify with a high threshold. Combining the direct overhead with the fraction of active diagnoses, the impairment across all intervals is even smaller: 1.1% on average and 9.24% in the worst case. Moreover, the active diagnosis phases mostly happen at the system startup, when no history table is yet available (see also Section 6.5). Thus, the longer the experiment runs, the smaller the fraction of active diagnosis phases, and the lower the average impact.

Parameter Tuning: The throttling overhead is strictly tied to the active throttle phase. Hence the important parameters are $|\mathbf{W_{isolation}}|$ and Th_{CPU}. Considering the tradeoff with the best values for detection we opt for $|\mathbf{W_{isolation}}| = 20$



(a) Low contention

(b) High contention

Fig. 8: 24-hour Wikimedia load with random batch jobs. Throughput and latency are normalized to 38 req/s and 330 ms.

TABLE 4: CONTENTION STATISTICS FOR SIX WORKLOADS.

	Det. rate	False alarm	Det. delay [sec]
Data serving	0.94	0.02	40.4
Data caching	0.98	0.00	46.5
Media streaming	0.88	0.02	45.3
Wiki Apache	1.00	0.01	36.9
Wiki Memcached	1.00	0.00	39.5
Wiki MySQL	1.00	0.00	36.2

and $Th_{CPU} = 0.3$ which translates into a throttle overhead of 15%. We note again that this overhead is paid only in the few cases requiring the active diagnosis.

6.4 Effectiveness of *ACT* for Cloudsuite

In addition to Wikimedia, we further evaluate *ACT* on three latency-sensitive benchmarks in Cloudsuite [22]: data-serving , data-caching and media-streaming. Here, we configure *ACT* with the best known parameters taken from Wikimedia experiments: $|\mathbf{W_{outlier}}| = |\mathbf{W_{isolation}}| = 20$ sample units, $Th_{CPU} = 0.3$ cores and $C_{outlier} = 50\%$. The testbed and experimental methodology is identical to the one presented in previous subsections. We summarize the results obtained from Cloudsuite in Table 4.

On the one hand, the false alarm rate is kept way below 10% for all the workloads, again proving the effectiveness of the false positive filtering step. On the other hand, the detection rate drops to 94% for data-serving and 98% for data-caching. Moreover, the detection delay for these workloads is slightly higher than the Wikimedia nodes. This is because the parameters of *ACT* are optimally tuned for Wikimedia. Hence, the diagnosis for Cloudsuite tends to fail to form the clusters in the second phase and thus *ACT* needs to collect more samples from different intervals. Moreover, one can also observe the lowest detection rate of 88% at the media-streaming benchmark. Media-streaming experiences light contention on the memory hierarchy and no contention at all with the network and disk, while being very sensitive to CPU time sharing. If its access to the CPU is impaired, the workload packet transmission latency is affected. The relative lower detection rate indicates that the current metrics collection of *ACT* is not sufficient to capture the CPU time contention.

6.5 Long-running time-varying workload

The object here is to asses how *ACT* performs in realistic scenarios where request rates are time varying. We replace the synthetic constant full load of previous experiments with the load experienced by a real Wikimedia frontend [25] scaled to the capacity of our system. Specifically, the load follows the typical sinusoidal day-night pattern over 24 hours and we scale the intensity such that the offered load oscillates between 90% and 35% of the system capacity. In parallel we spawn random batch workloads (both PARSEC and Cachebench) as in our previous experiments, but also consider two VM sizes to create different degrees of contention: i.e., VMs with 2 virtual cores and 2 GB of memory (low contention), and VMs with 8 virtual cores and 8 GB of memory (high contention). In both, we set $|\mathbf{W_{outlier}}| = |\mathbf{W_{isolation}}| = 20$ sample units, $Th_{CPU} = 0.3$ cores and $C_{outlier} = 50\%$ based on the results of the sensitivity analysis.

Results are shown in Fig. 8. We report the measured QoS of Wikimedia, in terms of normalized throughput and latency, together with the outcome of *ACT* on the Memcached server, in terms of alarm rate and throttling events. We only report the outcome of one server for readability and chose Memcached because it has the lowest resource usage making the contention most ambiguous and thus stressing the detection algorithm the most.

From the QoS metrics one can easily infer the level of contention. In the low contention case, shown in Fig. 8a, the client QoS has only small spikes. Moreover, one can see the influence of different Wikimedia loads. Indeed, the spikes are mainly concentrated in the peak load regions and have larger amplitudes. Similar results hold in the high contention case, shown in Fig. 8b, but with overall larger spikes. The clearer the contention, the more contention alarms are raised within a contention interval. Thus, a high density of events, shown by the histograms at the bottom, means that the contention is more obvious and vice-versa. Indeed in Fig. 8b the contention hit density over time is more regular due to the higher degree of contention.

Looking at the throttling events, shown by the crosses, one can see that few contention events are detected using the active diagnosis phase and mainly at system startup.

TABLE 5: PARALLEL DETECTION ANALYSIS OF *ACT* ON FIVE PAIRS OF COLLOCATED LATENCY-SENSITIVE VMS.

	Det. rate	False alarm	Det. delay [sec]
Wiki Apache + Data serv.	0.98	0.04	50.2
Wiki Memc. + Data serv.	0.98	0.06	61.8
Wiki MySQL + Data serv.	0.83	0.02	56.8
Media stream. + Data serv.	0.96	0.01	60.9
Data caching + Data serv.	0.84	0.11	85.3

Indeed, after an initial learning phase to fill the history lookup table, most detection cycles avoid the throttle phase. Comparing the low and high contention case, the latter allows the table to be filled out faster, i.e., the throttling events are all found within the first hour against the low contention case where the last one is found after 7.3 hours.

6.6 Parallel analysis

We present how *ACT* can detect resource contentions for multiple latency-sensitive VMs in parallel. To such an end, we consider the scenario where multiple latency-sensitive applications are collocated with PARSEC on the same physical server. For each physical node, we first place two VMs from those four applications such that the contention among them is minimized. Via extensive experiments, we observe that a lowly loaded data-serving benchmark (roughly 20% utilization) from Cloudsuite can be collocated with any other applications without causing much contention. Therefore, we place data-serving neighbor VMs to create the five VM pairs shown in Table 5. We then randomly spawn PARSEC on each physical node to inject the emulated contention as in our previous experiments. One *ACT* runs on each physical server and conducts detection analysis for collocated latency-sensitive VMs simultaneously. We summarize how *ACT* can detect resource contention caused by PARSEC across each pair in Table 5, particularly in terms of detection rate, false alarm rate, and detection delay.

The detection rate shown in Table 5 is around 95% for data-serving paired with either wiki-Apache, wiki-memcached or media-streaming, suggesting that *ACT* can handle parallel analysis for multiple VMs well. However, the detection rate drops to around 84% for the pairs of wiki-MySQL or data-caching, as *ACT* tends to incorrectly label contention as phase-change due to the following two observations. First, wiki-MySQL does not reach the steady-state, violating the assumption made in *ACT*. In contrast to experiments in the previous subsections, any of the three Wikimedia VMs can be throttled while *ACT* conducts the active throttling analysis for a collocated VM. As soon as any of the Wikimedia VMs gets throttled, the loads at the other Wikimedia components are also affected and thus the loads fluctuate without staying at steady-state. Secondly, for data-caching under low usage, it is difficult for *ACT* to differentiate contention and isolation samples. These two observations are further amplified by the history lookup. Once a contention cluster is falsely labelled as phase change during the active diagnosis, *ACT* automatically wrongly labels similar contention cases as phase change during the passive diagnosis.

The average value of the false positive rate shown in Table 5 slightly increases, compared to the scenarios where *ACT* only needs to detect contention for a single latency-sensitive VM presented in previous subsections. In the collocated experiments, as both latency-sensitive VMs run at low load to avoid cross contention, the impact of contention caused by PARSEC is thus low. As such, the difference between the contention and normal samples is minimal and diagnosis of *ACT* becomes more ambiguous. This observation is particularly obvious for the pair of data-caching and data-serving.

Compared to the non-collocated case, the time spent to detect the contention, so-called contention delay, is much higher. The main reason is again the low impact of PARSEC inference on lowly utilized latency-sensitive VMs. Thus, *ACT* tends to fail its diagnosis. Another reason is that, with our implementation of the parallel diagnosis, the window used to collect the outlier sample has been purposely increased by 30% (25 sample units instead of 20) with respect to the single VM analysis, to increase the detection accuracy.

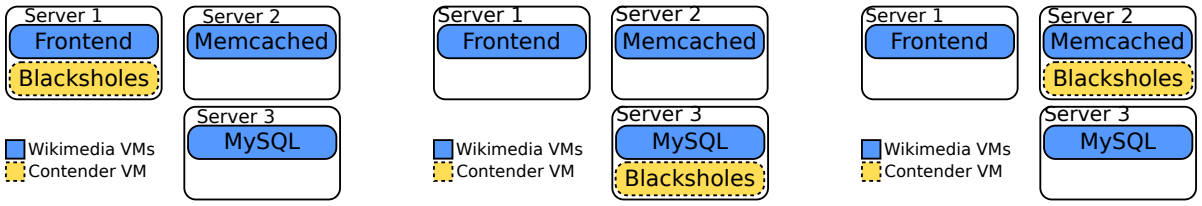
7 RELATED WORK

Most related work focuses on interference characterization. It can be differentiated by the types of input data, diagnosis overhead, and detection strategy.

Input data To detect interference requires both hypervisor and application performance metrics. While hypervisor metrics are readily available to cloud providers, application performance metrics, such as job completion time or request latency, are hidden inside the VM. On the one hand, some prior art [10, 9, 26, 19] explicitly ask such information to be provided. On the other hand, low level metrics such as IPC or CPI can help to predict client side performance [22, 8, 11, 12, 13, 27] or detect phases [6, 7] without the need for explicit feedbacks. Even though IPC can be used for transactional workloads [28], it is not in general applicable [29]. We approach the problem focusing on contention rather than interference, thereby removing the issues of application performance metrics.

Diagnosis overhead Completely passive diagnoses are possible, but require more information than the active ones. [11, 27] rely on a very large amount of statistical information gathered on a per application and per platform type. Alternatively, a human operator manually provides anomalous VM signatures in [14]. Active approaches gather the ground truth about anomalies by running the VMs in isolation on dedicated platforms [8, 12, 13, 15]. We take the approach of isolating the VMs by limiting the resource allocation of neighboring VMs directly on production nodes without the overhead of VM migration/cloning or the need for dedicated platforms. A similar approach is taken by [10], but with the objective of finding the optimal VM resource allocation rather than contention detection.

Detection strategy Proactive approaches either estimate interference between VMs by being aware of their sensitivity profile [12] or use predictive techniques to trigger an action before interference occurs [9, 26]. Reactive solutions [10, 11, 13, 14, 30] try to characterize the workloads in an on-line fashion and react to behavior anomalies. Our approach is both reactive, since we detect contention when it occurs, and proactive, since detected low contention levels anticipate



(a) Before (b) Possibility 1 (c) Possibility 2

Fig. 9: Example of migrating a batch workload to alleviate contention: initial deployment (a) and the two possible deployments after migration (b) and (c).

interference. Another category of related work is simple off-line interference characterization of workloads either running inside VMs or directly on bare metal [27, 31, 32]. We rely on similar metrics to feed our detection system, but strictly in an on-line fashion.

8 Discussion: Resolving Contention

While the focus of *ACT* is to detect the contentions, particularly the transient ones, here we provide a discussion on how to leverage the findings of *ACT* in mitigating the contention. Such a resource contention can be alleviated by various resource management policies, such as changing the VM consolidation plan [5], replicating the same jobs on multiple servers [33] or migrating VMs to different hosts. We specifically present an example of migrating VMs from the contended physical host to the non-contended one. We first deploy Wikimedia together with one Blachscholes benchmark from PARSEC on three servers as in Fig.9a. When Blackscholes activates the contention on server 1, this results in a 16% degradation in the page latency of Wikimedia. Once this contention is detected by *ACT*, we can migrate Blackscholes (to not disrupt the latency-sensitive Wikimedia application) to either server 2 or server 3. Migrating Blackscholes to the server 2 that hosts the MySQL VM reduces the latency degradation to 2.2% (see Fig. 9b). In contrast, migrating Blackscholes to server 3 that hosts the Memcached VM even slightly improves the latency by 2.6% (see Fig. 9c). These results show that migration is an effective means to counter performance degradation from resource contention and reflects well the observations from Section 6.2. However, the performance difference between migrating to server 2 or server 3 also underlines the difficulty in resolving contention. To optimally mitigate the performance degradation, it is critical that one can translate the low level resource contention into high-level application performance for all possible workload consolidation plans. It is our future work to explore performance modeling techniques to predict, resolve, and optimize collocated application VMs in highly virtualized environments.

9 Conclusion

Virtualization in cloud computing aims to increase the overall system utilization but raises concerns over VM performance isolation. In this paper, we propose an on-line algorithm, *ACT*, to monitor and detect the problem of shared resources contention for VMs of interest by only leveraging low-level metrics and thus guaranteeing the tenant's business confidentiality. At the core is *ACT*, a three-phase contention detection algorithm, that consists of an alarm, passive clustering and active throttle phase involving different tradeoffs of computation complexity and diagnosis accuracy. Via throttling neighboring VMs, *ACT* can accurately discern the contention caused by neighbors from internal program changes. We extensively evaluate *ACT* for latency-sensitive applications, namely Wikimedia and Cloudsuite, hosted on a cloud testbed where different batch-like applications are collocated. We exhaustively tune the parameters of *ACT* against different system scenarios, including time-varying workloads and collocated VMs. Our evaluation results show that *ACT* detects contention in 90% of cases in short contention intervals with a direct performance impairment to the collocated VMs as low as 15% which can be easily amortized over time.

References

[1] R. Birke, A. Podzimek, L. Y. Chen, and E. Smirni, "State-of-the-practice in data center virtualization: Toward a better understanding of VM usage," in *IEEE DSN*, 2013, pp. 1–12.

[2] L. Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, and W. Binder, "Achieving application-centric performance targets via consolidation on multicores: myth or reality?" in *ACM HPDC*, 2012, pp. 37–48.

[3] M. Björkqvist, S. Spicuglia, L. Y. Chen, and W. Binder, "Qos-aware service VM provisioning in clouds: Experiences, models, and cost analysis," in *ICSOC*, 2013, pp. 69–83.

[4] M. Björkqvist, L. Y. Chen, and W. Binder, "Opportunistic service provisioning in the cloud," in *IEEE Cloud*, 2012, pp. 237–244.

[5] C. Wang, B. Urgaonkar, A. Gupta, L. Y. Chen, R. Birke, and G. Kesidis, "Effective capacity modulation as an explicit control knob for public cloud profitability," in *ICAC*, 2016, pp. 95–104.

[6] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *IEEE/ACM MICRO-36*, 2003, p. 217.

[7] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan, "Online phase detection algorithms," in *CGO*, 2006, pp. 111–123.

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

[9] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "PREPARE: predictive performance anomaly prevention for virtualized cloud systems," in *IEEE ICDCS*, 2012, pp. 285–294.

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

[11] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI2: CPU performance isolation for shared compute clusters," in *EuroSys*, 2013, pp. 379–391.

[12] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *ASPLOS*, 2013, pp. 77–88.

[13] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "DejaVu: Accelerating resource allocation in virtualized environments," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 423–436, 2012.

[14] B. Sharma, P. Jayachandran, A. Verma, and C. Das, "CloudPD: Problem determination and diagnosis in shared dynamic clouds," in *IEEE DSN*, 2013, pp. 1–12.

[15] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *ACM SOCC*, 2011, pp. 22:1–22:14.

[16] M. Björkqvist, N. Gautam, R. Birke, W. Binder, and L. Y. Chen., "Optimizing for tail sojourn times of cloud clusters," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–14, 2015.

[17] A. Eldin, A. Rezaie, A. Mehta, S. Razroev, S. Sjostedt-de Luna, O. Seleznjev, J. Tordsson, and E. Elmroth, "How will your workload look like in 6 years? analyzing wikimedia's workload," in *IEEE IC2E*, 2014, pp. 349–354.

[18] A. R. Alameldeen and D. A. Wood, "Ipc considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, 2006.

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

[20] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[21] D. C. Montgomery, *Introduction to statistical quality control*. John Wiley & Sons, 2007.

[22] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 37–48, 2012.

[23] P. J. Mucci, K. London, and P. J. Mucci, "The cachebench report," The University of Tennessee, Tech. Rep., 1998.

[24] N. Gunther, S. Subramanyam, and S. Parvu, "Hidden scalability gotchas in memcached and friends," in *VELOCITY*, 2010.

[25] W. project, "Wikimedia Grid Report," http://ganglia.wikimedia.org/latest/, 2015.

[26] D. J. Dean, H. Nguyen, and X. Gu, "UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *IEEE ICAC*, 2012, pp. 191–200.

[27] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *SC*, 2012, pp. 51:1–51:12.

[28] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe, "Simflex: Statistical sampling of computer system simulation," *Micro, IEEE*, vol. 26, no. 4, pp. 18–31, 2006.

[29] A. R. Alameldeen and D. A. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, 2006.

[30] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: Online contention detection and response," in *IEEE/ACM CGO*, 2010, pp. 257–265.

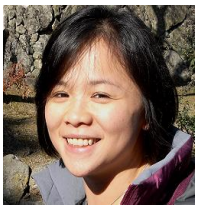
[31] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *ISPASS*, 2007, pp. 200–209.

[32] 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 *MICRO*, 2011, pp. 248–259.

[33] R. Birke, J. Pérez, Q. Zhan, M. Bjoerkqvist, and L. Chen, "Power of redundancy: Designing partial replication for multi-tier applications," in *INFOCOM*, 2017.

Joel Vallone is a software engineer at a Swiss IT consulting firm. He received his master degree from EPFL.

Robert Birke is at IBM Research Zurich Lab. He received his Ph.D. in Electronics and Communications Engineering from the Politecnico di Torino, Italy. His research interests are in the broad area of virtual resource management for large-scale datacenters, including network design, workload characterization and big-data application optimization. He has published more than 50 papers at venues related to communication and system performance, e.g., SIGCOMM, SIGMETRICS, FAST, INFOCOM, and JSAC. He is a IEEE senior member.



Lydia Y. Chen is a research staff member at the IBM Zurich Research Lab, Zurich, Switzerland. She received a Ph.D. in Operations Research and Industrial Engineering from the Pennsylvania State University. Her research interests include big data and cloud systems. She has published more than 50 papers in international conferences and journals. She has lead and participated in Swiss National Science Foundation and European FP7 projects. She is a IEEE senior member.