

# A Feasibility Study of Host-Level Contention Detection by Guest Virtual Machines

Giuliano Casale  
Department of Computing  
Imperial College London  
London, SW7 2AZ, UK  
g.casale@imperial.ac.uk

Carmelo Ragusa  
TIP Hana Cloud  
Systems Engineering, SAP UK  
Belfast, BT3 9DT, UK  
carmelo.ragusa@sap.com

Panos Parpas  
Department of Computing  
Imperial College London  
London, SW7 2AZ, UK  
p.parpas@imperial.ac.uk

**Abstract**—We investigate the feasibility of detecting host-level CPU contention from inside a guest virtual machine (VM). Our methodology involves running benchmarks with deterministic and randomized execution times inside a guest VM in a private cloud testbed. Simultaneously, using the recently proposed COCOMA tool, we expose the guest VM to host-level CPU stealing events of increasing intensity. This leads us to observe that the use of hyper-threading in the host can hinder detection of CPU contention, which otherwise can be done accurately using the CPU *steal* metric. For systems where hyper-threading is enabled, we investigate the performance of some basic detection algorithms. We find that thresholding often outperforms more sophisticated statistical tests.

## I. INTRODUCTION

Infrastructure-as-a-Service clouds, such as Amazon EC2, are increasingly used to deploy software services. However, a well-known issue in cloud environments is the existence of performance degradations due to multi-tenancy. With multi-tenancy, a single resource, either software or hardware, is used to serve multiple customers, which leads to significant cost savings for the cloud provider, but that can introduce overheads for the final users. Contention events can affect multiple resources (*e.g.* processor, memory, and caches), thus they are quite difficult to predict and model. Furthermore, the problem is not well understood in the literature where characterisation studies are mostly focused on describing hypervisor overhead [5], rather than on detecting contention from inside a guest VM. Thus, innovative measurement and estimation techniques are needed to evaluate contention events in the cloud.

In this paper, we take instead the perspective of a cloud customer and investigate the challenges of detecting CPU contention events from inside a guest VM s/he owns. Due to the inherent complexity of the problem, we focus on CPU contention, leaving other resource types and multi-resource detection to future work. Our analysis considers a pair of VMs, one controlled by the cloud customer we are taking the perspective of, the other by a third party. Our goal is to understand under which conditions the cloud customer can detect capacity contention generated by the third party. This is important since awareness of contention can indicate proper corrective actions; for example, dynamic reconfiguration of a load balancer to suspend for some time the usage of that VM.

In our experimentation, we evaluate different placements of the VMs across virtual cores, different resource consumption patterns, different lengths of the observation period, and the impact of simultaneous multi-threading (*e.g.*, Intel's hyper-threading) in the host machine processor on the contention capability.

## A. Problem Statement

Formally, the problem statement considered in this paper is as follows. We consider a pair of virtual machines,  $C$  (contentious) and  $A$  (user application), running on the same physical host. The two VMs share a subset of resources: either host-level physical cores (CPUs) or, if the server uses simultaneous multi-threading, host-level virtual cores (vCPUs)<sup>1</sup>. We assume to observe the system for a period during which  $C$  continuously consumes core capacity. VMs are always pinned to the same cores for the full duration of the observation period. In this setting, a customer evaluates the QoS offered by the cloud platform to its VM  $A$  through measurement of the execution times of jobs that run inside it. For example, if  $A$  contains a web server, the QoS would be evaluated analysing the execution times of individual requests processed by the web server.

Based on the execution time dataset, we study under what conditions a customer can detect that VM  $A$  suffered CPU contention. While we find that the CPU *steal* metric is a good capacity contention estimator when available, we make the important observation that this may be unavailable when hyper-threading is enabled. This calls for statistical estimation methods to detect capacity contention in this situation. We then investigate the performance of some basic parametric and non-parametric estimators, such as baseline thresholding, the Kolmogorov-Smirnov test, the Student  $t$ -test, the Anderson-Darling test, and others. Our analysis reveals that simple baseline thresholding has overall a better performance for CPU contention than more sophisticated statistical tests, even in presence of statistical randomness in the sample.

The rest of this paper is organized as follows. Section II describes the experiments we have performed. Initial analysis

<sup>1</sup>Recall that under simultaneous multi-threading (SMT), the host maps each vCPU to an hardware thread, and physical CPU can have multiple of such threads. Two hardware threads share a number of physical resources (*e.g.*, L1 and L2 cache, pipeline, memory fetch bandwidth), thus they can contend resource capacities to each other.

TABLE I. SYSTEM CONFIGURATIONS

config	VM <i>A</i> placement			VM <i>C</i> placement			hyper-threading
	socket	CPU	vCPU	socket	CPU	vCPU	
sc1	<b>0</b>	<b>0</b>	<b>0</b>	0	0	0	on
sc2	<b>0</b>	<b>0</b>	8	0	0	0	on
sc3	<b>0</b>	<b>0</b>	<b>0</b>	0	0	0	off
sc4	<b>0</b>	1	2	0	0	0	on
sc5	1	0	1	0	0	0	on

of experimental results is given in Section III. A deeper investigation of the case of a hyper-threaded host is discussed in Section IV. Related work is surveyed in Section V. Conclusions and future work are discussed in Section VI.

## II. EXPERIMENT DESIGN

To provide answers to the above questions, we have developed an experimental campaign as follows. The host machine we consider is a Sun Sunfire X2270, with 2 sockets each with a 2.53Ghz quad-core Intel Xeon 5500 processor; the total available DRAM is 16GB. Each core (CPU) has 2 threads which map to 2 vCPUs when hyper-threading is enabled, or to a single vCPU otherwise. Most of our experiments focus on vCPU-0 and vCPU-8, which map to the same physical CPU-0 on socket 0. The server is part of a private cloud testbed. The testbed utilises OpenNebula 4.0 as cloud management toolkit, configured with KVM version 3.6.0/SMP as hypervisor. Each VM is assigned 7GB RAM and 1 host vCPU; the guest operating system is configured with 1 vCPU as well.

Table I summarises the system configurations we have considered throughout our experiments. VM *C* is always pinned to vCPU-0. Conversely, VM *A* is pinned differently across configurations to evaluate the impact on contention. Resources that VM *A* shares with VM *C* are highlighted in bold. Configurations sc1 and sc3 differ only for the fact that in sc3 we explicitly request the operating system to deactivate vCPU-8, whereas in sc1 vCPU-8 is active but unused. Due to the architecture of this server, in sc4 and sc5 *A* and *C* do not share computational resources, they only share memory bandwidth or part of the caching hierarchy. Since the focus of our investigation is on processor contention, sc4 and sc5 represent effectively cases where there is no contention on *A* because of *C*'s activity; notice that CPU-0 in sc5 is on socket 1, thus it does not share capacity with the CPU-0 on socket 0 used by VM *C*. Thus, sc4 and sc5 are mostly useful to evaluate the sensitivity of the algorithms to background noise due to hypervisor load and environmental randomness.

### A. Workloads and Monitoring

The choice of the workload processed by the application VM is fundamental as it determines the amount of information available for detecting and characterising the capacity contention events. To simplify the interpretation of experimental results, we have considered two workloads, which we call DET and EXP, generated with sysbench, a popular Linux stress testing tool. For both workloads, sysbench tool is cyclically run for 60 successive runs of 1 minute to stress *A*'s virtual core. In workload DET, the execution times of each sysbench call is deterministic and controlled by the primes parameter of sysbench; we set this value to 1000 for all DET experiments. In the case of the EXP experiment, we randomize the value of the primes parameter to be exponential with mean equal to

TABLE II. BASELINE EXECUTION TIMES (1 HOUR RUN)

wkld	mean	95% c.i.	std. dev.	n
DET	0.95388	0.00080	0.01759	1848
EXP	0.97662	0.04076	0.96682	2164

1000. We have observed the execution time of sysbench to grow linearly with the primes value with excellent accuracy; as a result of this, in the EXP workload also the sysbench execution times are exponentially distributed.

We assume that the full trace of DET and EXP execution times is available to the customer, for example through detailed application logging. Furthermore, during the execution of DET and EXP, we collect utilization of the guest vCPU, including the CPU *steal* metric, using the collectl monitoring tool at 1 second resolution. The *steal* metric represents the percentage of time spent waiting the hypervisor while it was servicing another virtual processor. Given that all VMs have a single processor, in experiments sc1 and sc3 this value will effectively correspond to the processor time stolen by VM *C* from VM *A*, as we discuss in Section III. In sc2, sc4, sc5, the *steal* value will be close to zero since *C* and *A* do not share the same core. This is noteworthy in the case of sc2 in particular, since with hyper-threading enabled the VMs running on vCPUs 0 and 8 will share CPU-0, but we have found experimentally that this will not be directly visible from the *steal* metric. In other words, under hyper-threading the *steal* metric describes host vCPU contention, not host physical CPU contention.

### B. Contention Generation

In our experiment, VM *C* generates high contention on its vCPU using the COCOMA tool proposed in the BonFIRE project [1]. COCOMA is a software framework, recently released as open source, that allows creating contentious workload patterns in a controlled fashion [2]. Although COCOMA uses third party tools to create the desired pattern, it abstracts from any of them. It provides a common framework that allows users to focus on specifying the workload characteristics. Furthermore, COCOMA allows creating *reproducible* contention scenarios by combining different patterns of arbitrary complexity (e.g., constant, ramp, trapezoid, etc).

For ease of analysis, in this paper COCOMA produces a constant workload of the specified intensity for the full duration of a sysbench run. COCOMA runs inside VM *C* stressing its guest vCPU with an average load intensity *L* ranging in {*.5*, *.10*, *.25*, *.50*, *.99*}, where *.XX* stands for *XX*% guest vCPU utilization. Note that we jump sharply from *.50* to *.99* since we have found results in sc1 and sc3 to be quite similar in this region due to CPU overloading. That is, when *C* generates 50% contention it already consumes all of its quota of host vCPU capacity since *A* will concurrently run on the same vCPU while continuously executing sysbench.

### C. Montecarlo Experiments

Each combination of system configurations (Table I) and workload type (det, exp) is run for an observation period of *T* = 1 minute and repeated *I* = 60 times. Thus, the experimental dataset consists of 600 experiments each lasting 1 minute. Runs are successive to each other and performed after

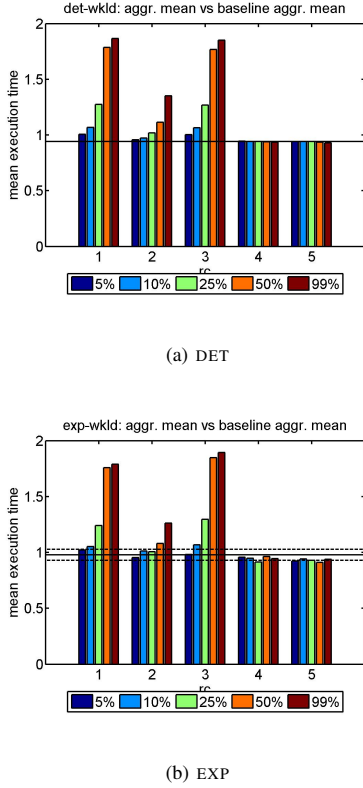


Fig. 1. Long term averages

re-initialization of the tool chain, a task that takes roughly 3 seconds in-between each run.

Additionally, we have run baseline experiments where VM  $C$  is turned off for each workload type; these experiments are used to determine a baseline for the execution time of both workloads in absence of external resource contention. Baseline results are summarised in Table II. Notice that the DET baseline falls within the 95% confidence interval of the mean of EXP, thus the two means are not statistically distinguishable within the cycles of 60 runs we consider.

### III. ANALYSIS

#### A. Detection by Long-Term Averaging

We begin with examining the long-term performance of VM  $A$  across the experiments. Figure 1 illustrates the average execution time of sysbench across all runs. The horizontal lines indicate the baseline and its confidence interval. The legend indicates the load intensity  $L$  produced by COCOMA on the vCPU of VM  $C$ . The results show that SC4 and SC5 perform, as expected, very close to the baseline, since in both cases  $C$  does not contend resources to  $A$ .

Similarly, only minor statistical fluctuations distinguish SC1 and SC3. For these two cases, execution times grow non-linearly with the load intensity  $L$ , but clearly saturate when  $L = 50\%$ , as visible from the fact that for  $L = 99\%$  execution times are only marginally larger. This can be explained by the

fact that when  $L \leq 50\%$ , the host operating system can always provide to VM  $A$  the requested share of vCPU-0; conversely, when  $L > 50\%$ ,  $A$  and  $C$  have to share vCPU-0.

Quite interestingly, we find that SC2 presents significant improvement compared to SC1/SC3. Since vCPU-0 and vCPU-8 run on the same physical CPU, the main difference of SC2 compared to the rest is the use of hyper-threading. Results suggest that hyper-threading remarkably reduces VM CPU contention degradation by more than half. Further, in the case of the EXP workload, contention of up to  $L \leq 25\%$  could pass unnoticed as the mean falls within the confidence intervals of the baseline<sup>2</sup>. While this hints to the fact that a hyper-threaded host provides better CPU performance isolation compared to a non-hyperthreaded host, it also suggest that it is quite difficult for a guest VM to realize existence of host-level CPU contention. In the long run, even a small contention of 2-3% could inflict a cost penalty, roughly equivalent to an extra week expense over a year.

#### B. CPU Stealing Model

We have further investigated the above results by attempting to model the background contention in the case where we can directly measure it. As observed earlier, this is possible in SC1/SC3 using the CPU *steal* metric provided by the monitor. Denote by  $\rho^{st}$  the average utilization stolen to  $A$ 's vCPU and assume to represent VM  $A$  as a processor-sharing queue. Recall that  $(1 - \rho^{st})$  represents the fraction of time, within the observation period, effectively available to  $A$  for processing. Then, a natural conjecture is that dividing the baseline execution time  $E[X]$  of sysbench by  $(1 - \rho^{st})$  could be sufficient to model the incurred overheads. That is, let  $E[Y]$  be the long-term average execution time predicted under an average *steal*  $\rho^{st}$ , we use the model

$$E[Y] = \frac{E[X]}{(1 - \rho^{st})} \quad (1)$$

as estimator of the execution time when  $A$  reads an average of  $\rho^{st}$  from the *steal* metric.

Table III illustrates the outcome of this attempt on the SC1 dataset. In these experiments, we have run additional experiments in which COCOMA generates load only for a fraction of time of the observation period at  $L = 99\%$  load intensity, thus producing a controlled variation of the  $\rho^{st}$  values. We find that our simple model works very well for the DET workload across all intensities of  $\rho^{st}$ . Slightly larger errors are found for the EXP workload, which are expected due to the higher variability and finite size of the sample. We have tested also variations with different values of  $L$ , however we were unable to find major errors.

### IV. DETECTION METHODS

An important observation concerning the SC2-DET and SC2-EXP scenarios in Figure 1 is that they correspond exactly to cases where it is not possible for the VM  $A$  to read the *steal* metric. This is because, with hyper-threading enabled,  $A$

<sup>2</sup>Indeed, more baseline experiments would tighten confidence intervals. Thus, our main claim here is that, for an *equal* number of experiments, the observable contention is comparable in magnitude to the 95th confidence intervals of the mean.

TABLE III. VALIDATION OF CPU STEALING MODEL - SC1 DATASET

wkld	$\rho^{st}$	measured $E[Y]$	model $E[Y]$
DET	0.25	1.0762	1.1086
DET	0.50	1.2580	1.2916
DET	0.75	1.5178	1.5457
DET	0.99	1.8665	1.9017
EXP	0.25	1.0718	1.1973
EXP	0.50	1.2200	1.2851
EXP	0.75	1.5175	1.4348
EXP	0.99	1.7879	1.7455

and  $C$  can contend resources despite being pinned to different host vCPUs. This motivates the investigation, in this section, of methods to detect the presence of host-level contention in the SC2 scenario when hyper-threading is enabled. Our goal is to investigate both accuracy and speed of detection for parametric and non-parametric methods. Below we formally define evaluation metrics and the detection methods we study.

#### A. Metrics

Let  $Y_i$  be a trace of execution times of sysbench in a randomly chosen group of  $i$  1-minute runs,  $1 \leq i \leq I$ ,  $I = 30$ . Thus,  $Y_1$  represents a randomly chosen 1-minute run in our dataset, while  $Y_{30}$  is a half-hour trace. We here focus on the SC2 scenarios only, since SC1 and SC3 can be efficiently handled by the *steal* metric. We also consider here only the EXP workload since in the deterministic case it is much simpler to detect deviations from the baseline. We also use the SC5 dataset to evaluate the algorithms against background noise. Since the number of runs instances in SC2 and in SC5 is identical, our evaluation will contain an equal number of cases where the algorithm should find contention (SC2), albeit small, and cases where it should not (SC5).

We consider a detection algorithm (or hypothesis test)  $\mathcal{D}$  that receives in input  $Y_i$  and tests the hypothesis that there exist contention at the host affecting, at *any* level of intensity, the VM  $A$ . Our goal is to evaluate the performance of  $\mathcal{D}$  under the considered datasets, expecting that it will detect contention in any run coming from the SC2 dataset and not find it for any run in the SC5 dataset.

Since  $\mathcal{D}$  is defined as a classifier, we can use standard performance metrics such as *precision*, *recall*, and  $F_1$ -score to evaluate its performance. Call *true positive* ( $tp$ ) a correct detection of contention, *false positive* ( $fp$ ) a detection of contention when it does not exist, and *false negative* ( $fn$ ) a missed detection of contention when it exists. Precision ( $P$ ), recall ( $R$ ), and the  $F_1$  score are standard metrics to quantify the accuracy of classifiers and are defined as follows

$$P = \frac{tp}{tp + fp}, \quad R = \frac{tp}{tp + fn}, \quad F_1 = 2 \frac{P \cdot R}{P + R}.$$

Precision quantifies the reliability of the algorithm when it reports contention. Recall is maximised when the algorithm identifies all contention events.  $F_1$  is a summary score which is computed as the harmonic mean of precision and recall. The above metrics,  $P$ ,  $R$ , and  $F_1$ , are evaluated for each randomly chosen trace  $Y_i$  and then averaged. We consider averages over 60 random choices of  $Y_i$ . In the special case  $i = 1$ , we forcedly use distinct traces  $Y_i$ 's in order to use the full dataset we have collected, which amounts to exactly 60 1-minute traces for each configuration.

TABLE IV. CONTENTION DETECTION METHODS

method	input 1	input 2	type
STT	$E[X]$	$Y_i$	parametric
KS2	$E[X]$	$Y_i$	parametric
LLF	$E[X]$	$Y_i$	parametric
BL5	$E[X]$	$E[Y_i]$	non-parametric
BL10	$E[X]$	$E[Y_i]$	non-parametric

#### B. Detection methods

We now outline the detection methods we have studied. All methods assume knowledge of the baseline mean  $E[X]$ . Methods based on hypothesis tests are considered at 5% significance level. A summary of the methods is given in Table IV. All implementations are done using MATLAB R2012b.

1) *Baseline gap (BLk)*: The method returns a positive answer (contention detected) for  $Y_i$  if the relative gap between the baseline and the current average exceeds  $k\%$ , e.g., BL5 detects contention if and only if  $E[Y_i] > 1.05E[X]$ .

2) *Student's t-test (STT)*: The  $t$ -test considers the null hypothesis that the data in  $Y_i$  comes from a normal distribution with mean  $E[X]$  and unknown variance. We use the `ttest` implementation in MATLAB.

3) *Kolmogorov-Smirnov test (KS2)*: We apply the Kolmogorov-Smirnov test against the null hypothesis that  $Y_i$  has the same distribution of a sample of size  $n = 10^5$  from a parametric model. For EXP we compare  $Y_i$  with an exponential distribution with mean  $E[X]$  and run the test using MATLAB's `kstest2` function.

4) *Anderson-Darling test (AD)*: We apply the one-sample Anderson-Darling test against the hypothesis that the data comes from an exponential distribution. We use a custom MATLAB script.

5) *Lilliefors test (LLF)*: The Lilliefors test is a goodness-of-fit test that, differently to the Kolmogorov-Smirnov test, does not assume knowledge of the parameters of the distribution to test against. We use the `lillietest` implementation in MATLAB which allows to compare the EXP execution times against an exponential distribution with unknown mean. Thus, the test attempts to detect contention by observing changes in the shape of distribution.

#### C. Results

Results for precision, recall, and  $F_1$  score for the different detection algorithms are given in figures 2-4, respectively. We have not reported results for contention levels  $L > 50\%$  since they are qualitatively similar to  $L = 50\%$ . Also the case  $L = 5\%$  is not reported being quite similar to  $L = 10\%$ .

The horizontal axis displays the size of the sample  $Y_i$  used for evaluating the detection methods, thus from left to right the methods have an increasingly large number of execution times to detect the presence of contention. For example, a typical  $Y_{30}$  trace at  $L = 50\%$  is composed by thousand observations, whereas  $Y_1$  has only a few tens.

The results in the figure indicate that BL5 is most of the times the best technique to detect contention. In the case  $L = 50\%$ , the overhead placed by the contention on the execution times is clearly visible and several methods perform quite well,

with the exception of STT, AD, and LLF that have medium precision but poor recall. As the background load decreases to  $L = 10\%$  and  $L = 25\%$ , all methods suffer a very visible decrease of recall, meaning that they are progressively less able to find contention. Results indicate that BL5 is probably the best method, except perhaps at  $Y_{30}$  where however it has better precision than other methods. In this single point, the AD test seems to perform quite well, however it is in general a poor performer for other values of  $L$ . We have repeated the experiments and found confirmation of the observations for the BL5 method, however we did not find the same result for AD suggesting that the point at  $Y_{30}$  is a statistical fluctuation.

The trends for precision for BL5 follow expectation since, as the sample size increases, averaging becomes more accurate and therefore it is easier to compare accurately against the baseline.

Summarizing, our experiments suggest that thresholding against a baseline can be quite effective and capable of detecting 30%-40% of contention events even at  $L = 10\%$  and  $L = 25\%$  load levels. With hyper-threading, the values of the long term averages fall within the confidence intervals of the baseline, thus it is generally very difficult to perform experimentation in this range. The good results for BL5 precision indicate that the rate of false positive is quite low and thus BL5 do not seem to operate randomly, except perhaps in the case  $Y_1$  where precision is slightly above 50% and thus quite close to the number of true positives for a detection algorithm that responds positively in all cases.

## V. RELATED WORK

Performance isolation is a hot topic in current cloud research. Several works can be found in the literature, which try to address different aspects. In [8] the authors propose various metrics to quantify isolation of performance in cloud applications. In particular they quantify the impact of an increased workload of customer on the QoS of other customers. However, this is based on observations that are external to VMs and can be used by cloud administrators for efficient capacity planning. Georges et al [3] look at the problem from the provider point of view as well, proposing metrics that allow to consolidate VMs to maximize the total infrastructure throughput as well as the single VM one. In [7] the authors study the contention in virtualized environments by characterizing the system-level workload incurred by VMs running various known benchmarks. The characterization was used to create a performance prediction model, which was claimed to succeed with an error of 5%. Huber *et al.* [5] characterized two different virtualization solutions (VMware and Citrix XenServer) in terms of performance and created a model that calculates their overhead compared to bare metal. Also [4] tackled the performance isolation problem by defining a fair system behavior and provided a solution that maximises the system throughput regardless of any different users' SLAs. Another interesting work is [9], where the authors analyse the contention of visible (to the OS such as Memory and IO) and invisible (shared cache space and core pipeline) resources on VMs performance. They apply this analysis to VM models and show their importance. The authors extended their work in [6] further proposing a transition from the VM concept to their virtual platform architecture (VPA), which enables management

of shared resource in a transparent fashion through specific mechanisms at architectural level. All these approaches see the problem from the provider point of view to help cloud administrators to better optimize the resource usage. Although this might have the effect of providing a higher level of service for users applications, the applications themselves do not have visibility on what is happening underneath and this makes it difficult to apply any action to react or prevent performance degradation. The solution under study in this work aims instead to give means to users which will allow applications to understand those critical situations and react as soon as possible, independently of any cloud provider action.

## VI. CONCLUSION

In this paper, we have studied the possibility for a guest VM to detect CPU contention happening at the host-level. We have shown that CPU steal metric readings are very effective in quantifying the contention, but it may not be available when hyper-threading is enabled at the host. In this case, simple thresholding appears quite robust and effective compared to more sophisticated baseline comparisons such as Kolmogorov-Smirnov or Anderson-Darling hypothesis tests.

A message that arises from our analysis is that cloud operators should disclose if their hosts have simultaneous multi-threading enabled (e.g., hyper-threading on). This apparently minor information disclosure can significantly enhance the analysis of CPU consumption. If simultaneous multi-threading is off, the CPU steal metric is a reliable contention estimator. Conversely, if simultaneous multi-threading is on, the user will be aware that the CPU steal metric is insufficient for describing CPU contention from other VMs and could adopt an inference mechanism, such as the baselining approach we have discussed.

Future work should consider the multi-resource case, where resources other than CPU could be contented. These may include, for example, memory, network, I/O channels. This appears a more difficult case since an equivalent of the steal metric will not be likely available. Another interesting direction for future work is in the testing of detection methods in multi-core VMs when the threading level at the resources changes over time, such as in web servers. In this case, it may not be possible to define a static baseline and stochasticity should be taken explicitly into account. Other issues not covered in this paper include the validation of result sensitivities to other CPU architectures, for example AMD processors.

## ACKNOWLEDGEMENT

The work of Giuliano Casale is funded by the European project MODA-Clouds (FP7-318484). Carmelo Ragusa received funding from the European project BonFIRE (FP7-257386). Panos Pappas is funded by a FP7 Marie Curie Career Integration Grant (PCIG11-GA-2012-321698 SOC-MP-ES).

## REFERENCES

- [1] BonFIRE. A multi-site cloud facility for applications, services and systems research and experimentation. <http://www.bonfire-project.eu/>.
- [2] S. S. Carmelo Ragusa, Philip Robinson. A framework for modeling and execution of infrastructure contention experiments. *MERMAT, 2nd International Workshop on Measurement-based Experimental Research, Methodology and Tools, FIA*, May 2013.

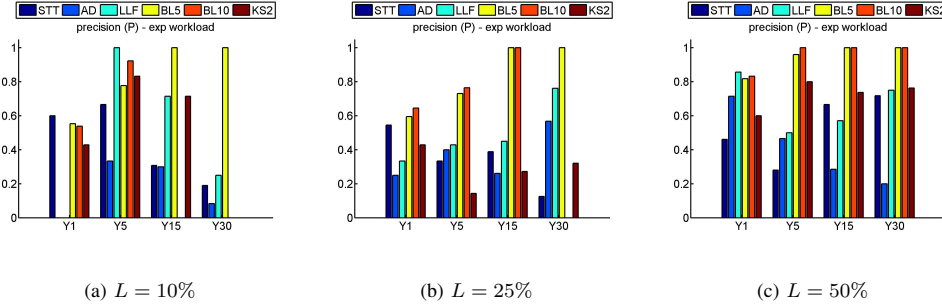


Fig. 2. Precision results for detection when hyper-threading is enabled (sc2 dataset)

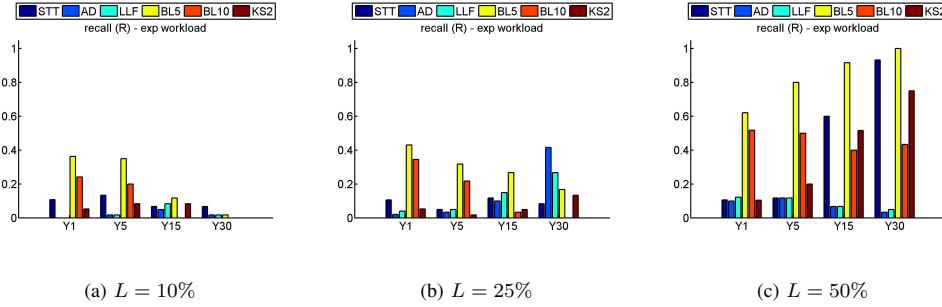


Fig. 3. Recall results for detection when hyper-threading is enabled (sc2 dataset)

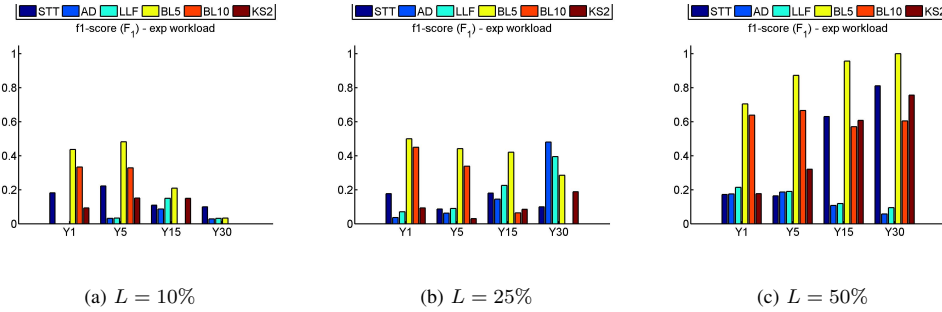


Fig. 4.  $F_1$  score results for detection when hyper-threading is enabled (sc2 dataset)

- [3] A. Georges and L. Eeckhout. Performance metrics for consolidated servers. In *System-level Virtualization for High Performance Computing, 4th Workshop, Proceedings*, page 8. Association for Computing Machinery (ACM), 2010.
- [4] C. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A framework for native multi-tenancy application development and management. In *CEC/EEE'07*, pages 551–558, 2007.
- [5] N. Huber, M. von Quast, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In F. Leymann, I. Ivanov, M. van Sinderen, and B. Shishkov, editors, *CLOSER*, pages 563–573. SciTePress, 2011.
- [6] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell. Vm:::3::: Measuring, modeling and managing vm shared resources. *Computer Networks*, 53(17):2873–2887, 2009.
- [7] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.
- [8] R. Krebs, C. Momm, and S. Kounev. Metrics and techniques for quantifying performance isolation in cloud environments. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, QoSA '12*, pages 91–100, New York, NY, USA, 2012. ACM.
- [9] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell. Modeling virtual machine performance: challenges and approaches. *SIGMETRICS Perform. Eval. Rev.*, 37(3):55–60, Jan. 2010.