CPU Performance Coefficient (CPU-PC): A Novel Performance Metric Based on Real-time CPU Resource Provisioning in Time-shared Cloud Environments

Toni Mastelić, Ivona Brandić
Institute of Information Systems
Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{*toni, ivona*}*@infosys.tuwien.ac.at*

Jasmina Jašarević
Faculty of Electrical Engineering,
Mechanical Engineering and Naval Architecture
University of Split
R. Boškovića 32, 21 000 Split, Croatia
jjasarev@fesb.hr

***Abstract*—The Cloud represents an emerging paradigm that provides on-demand computing resources, such as CPU. The resources are customized in quantity through various virtual machine (VM) flavours, which are deployed on top of time-shared infrastructure, where a single server can host several VMs. However, their Quality of Service (QoS) is limited and boils down to the VM availability, which does not provide any performance guarantees for the shared underlying resources. Consequently, the providers usually over-provision their resources trying to increase utilization, while the customers can suffer from poor performance due to increased concurrency.**

In this paper, we introduce CPU Performance Coefficient (CPU-PC), a novel performance metric used for measuring the real-time quality of CPU provisioning in virtualized environments. The metric isolates an impact of the provisioned CPU on the performance of the customer's application, hence allowing the provider to measure the quality of provisioned resources and manage them accordingly. Additionally, we provide a measurement of the proposed metric for the customer as well, thus enabling the latter to monitor the quality of rented resources. As evaluation, we utilize three real world applications used in existing Cloud services, and correlate the CPU-PC metric with the response time of the applications. An R-squared correlation of over 0.9557 indicates the applicability of our approach in the real world.

***Keywords*-cpu; performance; resource provisioning; virtualization; kvm; linux; cloud computing;**

I. Introduction

By early adopting a virtualization technology, Cloud Computing has gained rapid increase in utilization by offering virtual machines as part of an Infrastructure-as-a-Service (IaaS) model, such as the one offered by Amazon EC2[1]. The use of virtualization enables on-demand resource provisioning, including CPU cores, memory, storage and network bandwidth, all being serviced to customers under a pay-per-use policy. The policy is defined for each Cloud service in a Service Level Agreement (SLA), a contract between the provider and the customer, which includes the type, quantity and the QoS of requested resources.

While the resource quantity is very well defined, typically through different VM flavours, the QoS is usually limited and is only restricted to the VM availability. Availability defines a percentage of time when the VM was accessible (within a certain time period). However, it does not indicate the availability of the underlying resources during that period, such as CPU, nor its impact on the performance of the customer's applications [1].

Such an approach is feasible for dedicated environments where a resource is utilized by a single user. However, this is not applicable for time-shared Cloud environments, where a single server can host dozens of VMs [2]. Consequently, the performance of the underlying resource can get deteriorated due to its unavailability in a given moment as it is being used by another user [3]. An example is given in [4] for Amazon EC2, where a VM, although having the full amount of resources, exhibits different performance during weekdays due to neighbouring VMs taking their slice of the resources.

The described performance loss occurs due to resource over-provisioning done by the providers, trying to maximize the utilization. However, due to a lack of resource performance metrics, the provider is unable to monitor nor guarantee the performance of the customer's applications that depend on his/her resources. The same is true for the customer as well. On one hand, monitoring the response time of the application cannot provide accurate information on the QoS of underlying resources, as the response time can change due to other factors, such as workload or application updates [5]. On the other hand, benchmarking is considered as waste of resources and thus cannot be done at runtime [7].

Therefore, a metric is required that is able to isolate the impact the resources provisioned by the provider have on the performance of the customer's application [3]. The metric should be measurable both by the provider and the customer, the former for guaranteeing the QoS and the latter for checking if the promised QoS has been delivered [8]. Finally, the metric should be accessible in real world systems.

In this paper, we introduce the CPU Performance Coefficient (CPU-PC), a metric used for measuring the CPU performance based on its real-time provisioning in time-shared Cloud environments. We demonstrate its impact on the response time of customers' applications and provide the measurement of CPU-PC both for the provider and the customer. For calculating the proposed metric, we use

[1] http://aws.amazon.com/ec2/

raw metrics found in the *proc* filesystem, a standard Linux interface for the runtime system information. The raw metrics can be found in all major Linux distributions, which enables a straightforward implementation in existing monitoring systems such as New Relic[2] by simply adding a plugin.

We evaluate our approach by correlating the CPU-PC metric with the response time of the customer's application. We utilize three real world applications running on top of IaaS virtualized using the KVM hypervisor [9], a kernel-based virtualization used by Cloud providers such as CloudSigma[3] and Google[4]. Applications include the FFmpeg video tool [10], MongoDB NoSQL database [11], and the Ruby on Rails web framework [12], all offered as part of real world PaaS systems.

The results show a correlation of CPU-PC with the response time of over 0.9557 measured using R^2, a measure expressing the correlation between data sets. Hence, CPU-PC can be used by real world providers such as CloudSigma for guaranteeing resource QoS, as well as by customers such as Heroku[5] and MongoHQ[6] that host their services on top of IaaS for monitoring the QoS of underlying resources.

The rest of the paper is organized as follows. Related work is given in Section II, while the performance and resource models are described in Section III. Experimental setup and the CPU-PC metric for providers and customers are described in Section V-C. Evaluation is presented in Section VI, and finally Section VII gives the conclusion and the future work.

II. Related work

Tools such as Cloud Crawler [13], CloudBench [6] and Expertus [14] provide automated approaches for evaluating the performance of infrastructure Cloud services. A similar approach is described in [15] for Grid systems. Their focus is on benchmarking the resources before deploying an application to a Cloud or Grid. However, their approaches cannot be applied for monitoring the resource performance at runtime as they provide a static analysis [7]. Other works such as [16] and [17] focus on the performance overhead of virtualization technologies, such as Xen and VMWare.

The CloudClimate project [18] provides the performance monitoring data of several major Cloud providers. Their data clearly shows the performance variability during a day due to resource time-sharing. Dejun et al. [1] and Schad et al. [4] perform a performance analysis of Amazon EC2 during weekdays. They show that EC2 instances suffer from a large performance variance, namely over 24% for the CPU calculated using the coefficient of variation. Although, all papers above give emphasis on performance variance, they use periodical benchmarks to obtain monitoring data, without offering real-time performance metrics.

Wang et al. [19] examine the response time of Cloud deployed applications. They show how different factors, such as user workload burst, thread pool management or even Java garbage collection, can significantly affect the response time of the application. Correlation between the CPU allocation, consumption and the response time is analysed in [20]. The authors provide valuable results showing the significance of CPU contention due to resource time-sharing, and emphasize the importance of measuring this phenomenon.

Work presented in [3] is the closest to ours as the authors try to detect the contention of the customer's VM. However, their analysis is focused only on the customer's perspective without providing an unified quality metric that can be measured and compared both by the provider and the customer. Similar work is done in [21], also from the customer's point of view, however with a focus on SaaS multi-tenancy, where several users utilize the same application instance instead of physical resources. Georges et al. [22] take the provider's perspective trying to consolidate VMs on a single server. However, they again apply a static benchmark approach, while we focus on real-time monitoring of resource quality.

III. Performance and resource model

Here we demonstrate that the performance and resource utilization depend on several factors, rather than just on the amount of available resources. For analysing the performance of an application, we utilize a model shown in Figure 1, similar to the one presented in [5]. The model consists of three inputs and a single output, namely a performance metric such as the response time. The inputs include:

- **Resources** - include the amount of resources such as CPU, memory, storage and network, that are available to the application at the given moment. This input is controlled by the resource provider, i.e., IaaS provider.
- **Implementation** - represents algorithms, programming language, architecture and internal parameters of the application that are not accessible to the end-users. This input is controlled by the IaaS customer, such as PaaS or SaaS provider utilizing the underlying infrastructure for running his/her own applications.
- **Workload** - considers the user interaction with the application, including size of the input files, number of end-users and configuration parameters accessible to them. This input is controlled by end-users, who utilize applications deployed on top of IaaS.

In order to demonstrate the effect of the described inputs on the output, i.e., the response time, we utilize *primes*, a custom made application designed specifically for this purpose. The application uses the *trial division* algorithm [23] for counting prime numbers $|P|$ in a specified interval $[3, n]$, where n is defined by the user. The algorithm divides each number $p \in [3, n]$ with a series of dividers D starting from 2 to

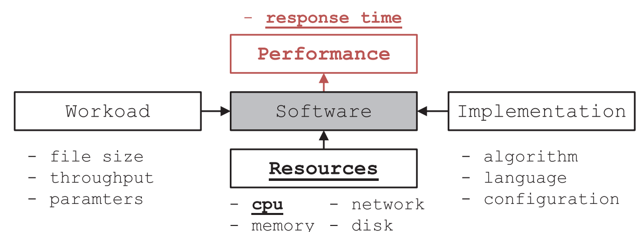


Figure 1: Performance model of an application

[2] http://newrelic.com/
[3] https://www.cloudsigma.com/
[4] https://cloud.google.com/products/compute-engine/
[5] https://www.heroku.com/
[6] https://www.mongohq.com/

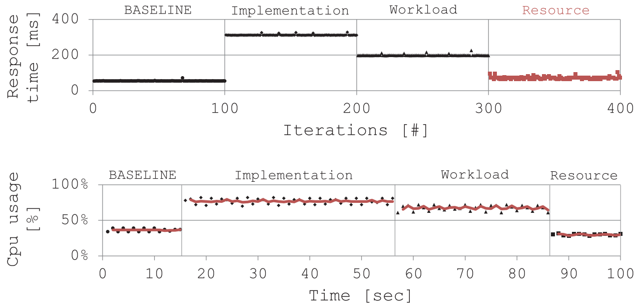


Figure 2: Response time and CPU usage for different cases

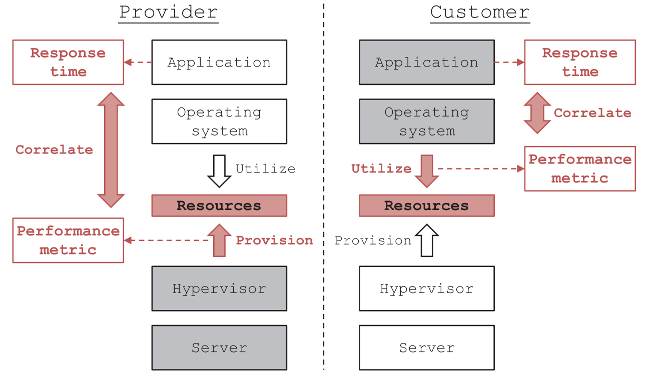


Figure 3: Experiment setup

$n-1$. If $\forall d \in D : p \bmod d \neq 0$ then p is added to P as a prime number, i.e., $|P|{+}{+}$. Finally, the algorithm is iterated 100 times with 100 millisecond gaps between the iterations in order to get averaged results with controllable utilizations. The application is executed within a VM with four different configurations listed in Table I, a baseline and one for each input where a different setup parameter is marked in red:

- Baseline - assumes the basic setup to which the other tests are compared.
- Resource - limits the CPU by sharing it between other applications.
- Implementation - the division sequence is reversed, going from $n-1$ to 2, instead of from 2 to $n-1$.
- Workload - input value n is set to 50000 instead of the baseline value of 10000.

Test	Cpu	Internal loop	n
BASELINE	Max	$2 \rightarrow n-1$	10000
Resource	Limited	$2 \rightarrow n-1$	10000
Implementation	Max	$n-1 \rightarrow 2$	10000
Workload	Max	$2 \rightarrow n-1$	50000

Table I: Metric traces used in the evaluation.

Results of the tests presented in Figure 2 show that the application performs best with the *Baseline* setup, while other tests exhibit higher response times due to the worse implementation, bigger user workload and fewer CPU cycles. Similarly, the resource consumption varies due to the same inputs as shown in the figure. Moreover, it shows that the CPU utilization cannot be correlated with the response time, as the application can utilize more CPU and achieve shorter response time, e.g., the *Implementation* versus the *Workload* test, while the *Baseline* and *Resource* tests exhibit the opposite behaviour.

IV. Use case experiment

In this paper, we focus on IaaS providers and their customers, where the provider is only responsible for the performance of the provisioned *resources*, while the *implementation* and the *workload* are the responsibility of the customer. Consequently, our goal is to isolate and measure the impact of CPU provisioning on the performance of the customer's applications. Moreover, this impact has to be measurable both by the provider and the customer, the former for guaranteeing the QoS and the latter for checking if the guaranteed QoS has been delivered.

On one hand, the IaaS provider has to perform measurements on the hypervisor, as he does not have access to the upper layers, such as the operating system or applications. An example is a Heroku service that hosts the Ruby on Rails platform on top of Amazon IaaS, where the latter does not have access to the platform itself. On the other hand, the customer performs measurements from the operating system within a VM, having no access to the bottom layers, such as the hypervisor and the physical server. From the same example above, Heroku does not have access to the underlying infrastructure managed by Amazon.

For this purpose, we perform a single experiment for these two scenarios as shown in Figure 3, one for the provider and one for the customer. For the experiment we utilize the configuration listed in Table III, where we fix other inputs described in Section III and change only the CPU resource by increasing and decreasing the background CPU load.

Component	Type	Resources
Application	*primes*	VM: 1 core; 1GB RAM
Operating system	Ubuntu 12.04 LTS	
Hypervisor	KVM 3.2.0-65	PM: 4 cores; 8GB RAM
Server	Dell R310	

Table III: Configuration used for the experiments.

For the measurements, we utilize the *proc* filesystem in Linux that provides different system and application runtime metrics, including CPU utilization and scheduling metrics. There, we select a total of 7 CPU metrics, which are listed in Table II. The first six measure resource consumption and are expressed in *user_hz* unit, which is defined in [24] as the number of CPU clock ticks per second. Usually, this frequency is 100Hz for most systems the same as for the one used in our experiments, while the right value can be obtained through the `sysconf(_SC_CLK_TCK)` function.

We derive three composite metrics out of the ones listed in the table, namely

$$cpu_sys_i[\%] = \frac{(u_i + s_i + io_i)\cdot 10}{cpu_cores \cdot t_i[ms]} \cdot 100 \qquad (1)$$

representing the percentage of time the CPUs have been utilized. The multiplication by 10 in the equation is due to the conversion from *user_hz* to milliseconds, while *cpu_cores* is the number of CPU cores. Idle CPU resources are calculated using the equation:

$$cpu_{idle}[\%] = \frac{(i_i)\cdot 10}{cpu_{cores} \cdot t[ms]} \cdot 100 \qquad (2)$$

Metric	Symbol	Unit	Path	Description
`user`	u_i	user_hz	/proc/stat	Total time spent in user mode.
`system`	s_i	user_hz	/proc/stat	Total time spent in kernel mode.
`iowait`	io_i	user_hz	/proc/stat	Time waiting for I/O to complete.
`idle`	i_i	user_hz	/proc/stat	Time spent in the idle task.
`utime`	pu_i	user_hz	/proc/[pid]/stat	Amount of time the process has been scheduled in user mode
`stime`	ps_i	user_hz	/proc/[pid]/stat	Amount of time the process has been scheduled in kernel mode
`wait_sum`	w_i	ms	/proc/[pid]/sched	Amount of time the process was ready to run but had to wait for the CPU

Table II: Metrics collected by the provider. [pid] is replaced with the process ID

The CPU utilization of a single application, which includes all its threads is calculated with:

$$cpu_{app}[\%] = \sum_{p=1}^{P} \frac{(pu(p)_i + ps(p)_i) \cdot 10}{cpu_{cores} \cdot t[ms]} \cdot 100 \qquad (3)$$

where P represents the number of threads. Finally, system and application level metrics, including VM metrics from the hypervisor, are collected using the M4Cloud monitoring tool introduced in [25].

We run the experiment and collect metrics both from the provider's and the customer's perspective. Figure 4 shows the total CPU usage of the server, CPU usage of customer's VM, as well as the response time of customer's *primes* application. At marker 1 only the customer's $VM_{customer}$ is running with the *primes* application configured with the baseline parameters defined in Section III, thus consuming around 40% of the CPU and achieving the response time a little above 50 ms. Another VM_1 starts at marker 2 with a full load, thus consuming 100% of its single CPU core. Same is repeated at markers 3, 4 and 5.

However, unlike marker 2 where additional VM_1 does not effect the performance of $VM_{customer}$, VM_2 and VM_3 increase the response time of $VM_{customer}$ from 50ms to over 60ms, although there is over 150% and 50% of the CPU time available, respectively. This is due to context switching, which is performed by the hypervisor itself, although each VM has a CPU core for itself as there are 4 cores and 4 VMs. Consequently, this clearly shows that a certain amount of free resources does not necessarily mean that every VM is able to achieve its highest performance.

VM_4 starts at marker 5 and additionally increases the response time of $VM_{customer}$ as it now shares a CPU core with other VMs. Unlike in the previous situation, in the current one it is clear that the performance is deteriorated as the CPU is fully utilized. At marker 6 the load on VM_4 is decreased to only 30% of its single CPU core, which now leaves almost 50% CPU resources available on the physical server. However, the response time of $VM_{customer}$ only slightly improves making it still almost double than the initial value.

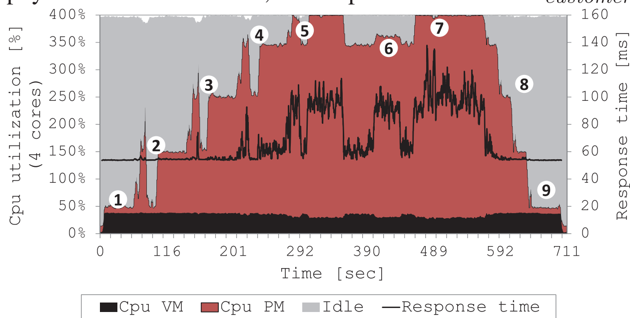


Figure 4: Response time and the CPU utilization.

VM_4 is again fully loaded at marker 7, when an additional VM_5 starts. Consequently, the response time of $VM_{customer}$ increases even beyond the value that is at marker 5, while the CPU usage of the physical machine is the same, i.e., at 400%. Finally, all VMs except $VM_{customer}$ are turned off at marker 8, after which $VM_{customer}$ continues running alone at marker 9, thus having the response time of 50ms as at the beginning.

The experiment shows that VM concurrency has a significant impact on the performance of the customer's application, which cannot be directly detected by monitoring only the CPU utilization.

V. CPU Performance Coefficient (CPU-PC)

In this section we use the measurements obtained in the experiment in order to isolate a CPU related metric that has direct impact on the performance of customer's applications.

A. *Provider - resource provisioning*

Our previous analysis shows that resource utilization cannot provide QoS information of the underlying resources. However, in order to guarantee the QoS of the provisioned resources, the provider has to be able to measure it. Therefore, we analyse the results of the `wait_sum` metric, which measures the amount of time a certain process had a task to execute, but had to wait for the CPU to become available. The results are shown in Figure 5 for the main process of the $VM_{customer}$ and all its threads.

Besides the main process, only two threads exhibit some activity, hence all others are merged into *other threads* in the figure. For executing applications within a VM, a hypervisor creates additional threads, such as threads $T1$ and $T2$, which then perform tasks required by the applications. In our case, thread $T1$ was created for executing the *primes* application. However, the IaaS provider does not have this information, and therefore must monitor the entire VM as a single entity, hence it has to monitor the performance of all threads.

Summing up the obtained results for the `wait_sum` metrics of all threads is not feasible as the threads can wait at the same time, unlike metrics such as `utime` and `stime` where only a single thread or a process can use the CPU.

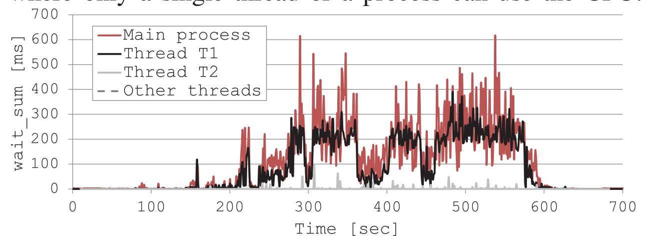


Figure 5: `wait_sum` metric for KVM threads.

Therefore, we calculate a weighted average of the obtained `wait_sum` values at every moment of the measurement, where the value itself is taken as the weight. The reason for this is that the bigger the `wait_sum` value the bigger is its effect on applications running inside the VM. Finally, the actual unavailability time of the CPU is calculated in the following equation:

$$cpu_unavailable_i[ms] = \frac{\sum_{p=1}^{P}(w_i)^2}{\sum_{p=1}^{P}(w_i)} \quad (4)$$

Figure 6 shows both the unavailable time and the response time of the *primes* application. The vertical axes in the figure are intentionally shifted so that the correlation between the metrics is more visible. Figure 7 shows this correlation by plotting the two metrics one against the other. The figure shows a slightly exponential increase of the response time as the unavailable time gets bigger. This is due to additional context switches and the disk and memory usage for storing monitoring data, i.e., the longer unavailability time assumes more concurrency on the CPU, which usually includes other resources as well, such as the disk and memory access in our case.

B. Customer - resource consumption

In the previous section we were able to correlate the metric measured on the hypervisor by the provider with the application performance. However, if the customer himself cannot measure the performance of the rented resources, then he/she cannot know if the promised QoS is actually being delivered or not. Therefore, we introduce the same metric as the one in the previous section, however measurable by the customer.

We select the same metrics listed in Table II and composed in equations 1, 2 and 3. However, we measure them on the operating system running inside a VM, rather then on the hypervisor, as the latter is not accessible to the customer. The results are obtained from the same test that was executed in Section IV in order to be able to compare the results.

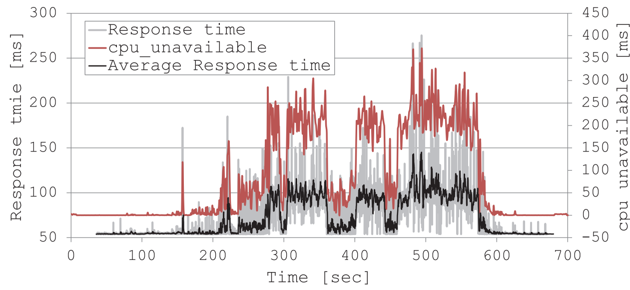


Figure 6: Response time and unavailable CPU time.

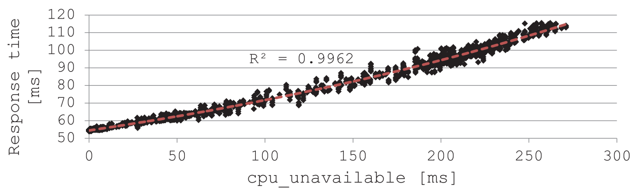


Figure 7: Correlation between the response time and unavailable time.

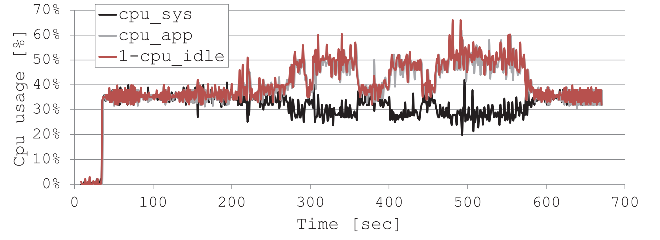


Figure 8: CPU utilization of a customer's VM.

Figure 8 shows the CPU utilization of the *primes* application, which follows the total system utilization calculated by subtracting cpu_{idle} from 1. This shows that the *primes* application is the only active process within a VM. However, the cpu_{sys} metric calculated in Equation 1 shows lower utilization than the previous two. From the system's point of view, the gap between cpu_{sys} and $1 - cpu_{idle}$ is neither utilized nor idle, while the cpu_{app} metric measures that these CPU cycles are consumed by the *primes* application. However, the lost cycles are actually cycles that are assigned to other VMs by the hypervisor while the *primes* application was executing on the CPU. Consequently, the cpu_{app} metric accounts for these cycles as they are consumed by the application, while cpu_{sys} does not.

We calculate the observed gap using the equation:

$$cpu_unavailable_i[ms] = (1 - u_i - s_i - io_i - i_i) \cdot 10 \quad (5)$$

which expresses the amount of time in milliseconds that the CPU was unavailable. The results are plotted against the response time in Figure 9. The figure provides the same results as the ones shown in Figure 6, hence showing that the lost cycles are actually the time that the VM waited for the CPU. We repeat the procedure from the previous section by correlating the two metrics in Figure 10. Finally, having the same measurements from the provider and the customer, we are able to derive a unified performance metric based on the real-time CPU provisioning.

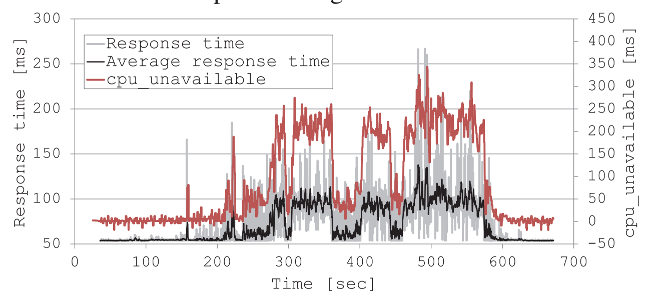


Figure 9: Response time and the performance metric.

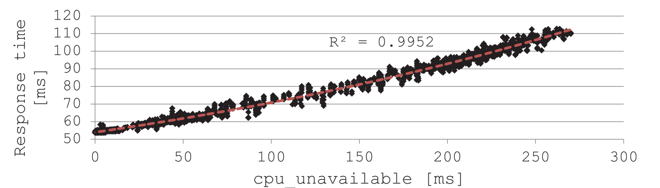


Figure 10: Correlation between the response time and unavailable time.

C. Unified performance metric

In this section we utilize the results obtained in the previous sections in order to derive the CPU Performance Coefficient (CPU-PC). We do this by normalizing the values obtained in milliseconds between 0 and 1 as shown in the following equation:

$$\text{CPU-PC} = \frac{cpu_{unavailable}[ms]}{cpu_{cores} \cdot t_i[ms]} \quad (6)$$

where cpu_{cores} represents the number of CPU cores assigned to the monitored VM. Consequently, metric values closer to 1 represent better CPU performance, while values closer to 0 represent worse CPU performance.

It is important to notice that the proposed metric measures the quality of only those CPU cycles that are actually utilized by the customer. If the customer's VM is 100% idle, the provider is able to use those cycles for other VMs without effecting the performance of the former one. This is due to the raw metrics used for calculating CPU-PC, which are increased only if the customer's VM had something to execute, but was not able due to the other VMs.

The usability of the proposed metric spans from research to industry. As noted in [26] researchers often fail to include the performance variance when Clouds are used in scientific experiments, hence obtaining the results that are influenced by deteriorated performance of the underlying resources. Consequently, Cloud customers require metrics in order to find out "if any occurring performance issue or failure is caused by the provider, network infrastructure, or by the application itself [8]". Finally, "monitoring may allow Cloud providers to formulate more realistic and dynamic SLAs and better pricing models by exploiting the knowledge of user-perceived performance [8]".

VI. Evaluation

We evaluate our approach by running several applications and correlating their response times with CPU-PC, both for the provider and the customer. The evaluation testbed follows the same setup as the one described in Section IV and shown in Figure 3, where the application is executed within a VM. For our evaluation we utilize the following applications and give examples of Cloud services where these applications are deployed on top of IaaS to demonstrate real world usage of our approach:

- FFmpeg [10] - an open-source application for recording, converting and streaming audio and video. FFmpeg is used as part of Monster Muck Mashup[7], a video conversion service, built as a VM image ready for deployment on Amazon EC2. In order to measure the response time of the video conversion, we modify the source code by adding a timer, which measures the response time for rendering a single frame.
- MongoDB [11] - an open-source NoSQL document database, written in C++. An example of MongoDB being deployed on IaaS is MongoHQ that offers MongoDB instances as part of their PaaS, which is deployed on top of Amazon EC2. For measuring the response time we benchmark the database and measure the throughout, based on which we calculate an average response time within the measured interval.
- Ruby on Rails [12] - an open-source web framework. Heroku offers the framework as part of their PaaS, which is also hosted on Amazon EC2. We benchmark the framework by sending *http* requests and measuring their response time.

In order to minimize the impact of the *workload* and *implementation* inputs on the response times, we use the same application's configuration and user workload, such as a video file for FFmpeg, for all our tests. The performance impact of resources other than the CPU are minimized by leaving them unutilized by other applications. For this purpose we run benchmarks for MongoDB and Ruby on Rails directly from the hypervisor, hence avoiding any network traffic as the virtual network between the hypervisor and VMs is used for the communication. Additionally, we use the *cpuset*[8] pseudo-filesystem interface in Linux for dedicating a single CPU core for the benchmark application. Therefore, the background CPU load does not have any impact on the benchmark itself, rather it affects only the monitored application.

A. Evaluation results

Figures 11, 12 and 13 show evaluation results for FFmpeg, MongoDB and Ruby on Rails, respectively. Since the metrics from the figures are measured from different locations and at different frequencies, their timestamps are unsynchronized. Therefore, we resample the traces and use the moving average to get synchronized values and then correlate them on the graphs.

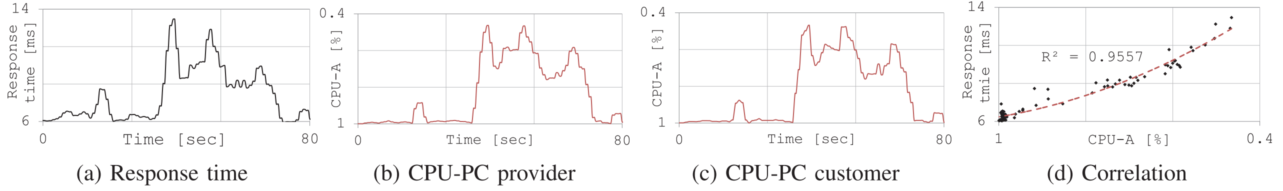
Figures (a) show the response times measured at the application level as the baseline performance metric. Its increase is caused by the background CPU load of additional VMs executed on the same server. CPU-PC metrics measured both by the provider and the customer are shown in Figures (b) and (c), respectively. As it can be seen from the figures, the proposed metric follows the performance of the applications, or more precisely it affects their performance by limiting the access to the CPU at a given moment due to the neighbouring VMs.

CPU-PC measured by the customer in Figures (c) shows slightly higher values at the beginning and at the end of the traces, although the background CPU load is minimal here. This is due to the CPU activity for handling software and hardware interrupts, which is not included in the CPU utilization. Additionally, due to the time gap between the readings of all raw metrics, customer's CPU-PC exhibits more noise than the provider's metric, as the provider's CPU-PC uses a single raw metric, which is already refined using Equation 4.

Figures (d) show the correlation between the response times and CPU-PC metrics, where the correlation is above

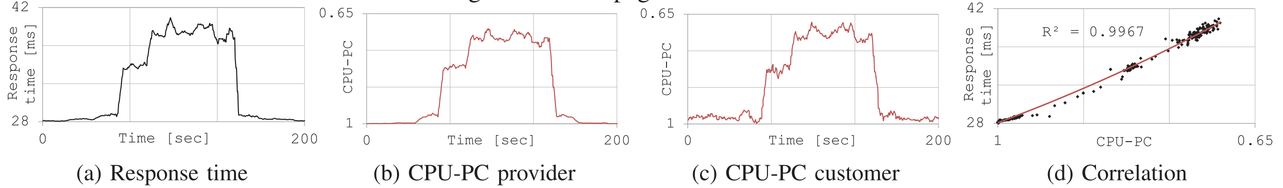
[7] https://aws.amazon.com/articles/Python/691

[8] http://man7.org/linux/man-pages/man7/cpuset.7.html



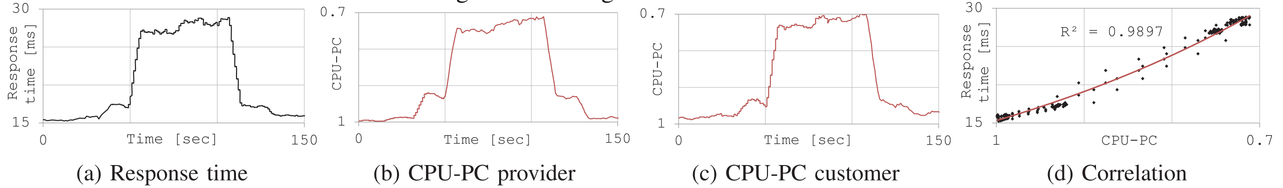
(a) Response time (b) CPU-PC provider (c) CPU-PC customer (d) Correlation

Figure 11: FFmpeg evaluation results



(a) Response time (b) CPU-PC provider (c) CPU-PC customer (d) Correlation

Figure 12: MongoDB evaluation results



(a) Response time (b) CPU-PC provider (c) CPU-PC customer (d) Correlation

Figure 13: Ruby on Rails evaluation results

0.9897 and 0.9967 calculated using the R-squared measure for MongoDB and Ruby on Rails, respectively. Slightly lower correlation is achieved for FFmpeg, namely 0.9557. This is due to rendering of different frames of the input movie, which exhibit different response times regardless of available underlying resources.

To summarize, the results demonstrate the feasibility of the CPU-PC metric in real world systems, where both providers and customers can monitor the quality of delivered resources, specifically the CPU, without benchmarking and monitoring the unreliable response time.

B. Metric availability

Here we analyse the availability of raw metrics used for calculating CPU-PC on different operating systems. We focus on the KVM hypervisor and Linux distributions. Table IV shows the list of tested Linux distributions.

Distro	**Version**	**Kernel**	**CPUPC provider**	**CPUPC customer**
Ubuntu	12.04.4	3.2.0-65	native	calculate
Xubuntu	14.04	3.13.0-24	native	calculate
Kubuntu	14.04	3.13.0-32	native	calculate
Lubuntu	14.04	3.13.0-32	native	calculate
Debian	7.6	3.2.0-4	module	calculate
Mint	17	3.13.0-24	native	calculate
Fedora	20	3.11.10-301	module	native
CentOS	7.0.14	3.10.0-123	module	calculate
Arch	2014.08	3.15.7-1	native	native
OpenSUSE	13.1	3.11.6-4	native	native
Knoppix	7.1	3.9.6	module	native
Gentoo	2.2	3.6.8	module	calculate

Table IV: Availability of CPU-PC raw metrics

All tested distributions support CPU-PC from a customer's point of view, as the customer's CPU-PC uses standard CPU utilization metrics, which is calculated in Equation 5. Additionally, distributions such as Fedora, Arch, OpenSUSE and Knoppix provide a separate raw metric referred to as `stolen_time`, which represents the CPU unavailability within a VM calculated in Equation 5. For the provider, all tested distributions also support the `wait_sum` metric. However, some provide *native* support by including the raw metric within the *proc* filesystem, while others require a Linux module [27] to obtain the metric from the Linux kernel and include it in *proc*.

VII. Conclusion and Future Work

In this paper we have shown that both the response time and the resource utilization depend on several factors, which may be under the management of different entities. Consequently, they cannot be used in a straightforward manner for monitoring only one of these factors, such as resource provisioning. In order to overcome this shortage, we introduced the CPU Performance Coefficient (CPU-PC), a metric that measures the performance of the hosted applications in a virtualized environments. The metric isolates the impact of a single factor on the application performance, namely the CPU resource, hence allowing providers and customers to measure the quality of the CPU resource being provisioned and consumed, respectively.

Correlation results with R^2 above 0.9557 obtained using real world applications show the reliability of the proposed metric in real world systems. Additionally, the wide accessibility of raw metrics required for calculating CPU-PC provides a straightforward approach for implementing the latter in existing Cloud monitoring services such as New Relic, or monitoring tools such as M4Cloud [25], by simply writing a plugin for a composite metric. This makes the CPU-PC metric relatively easy to use for analysing different isolation techniques, such as hypervisors and Linux

containers. Furthermore, the efficiency of consolidation algorithms can also be evaluated using the proposed metric, as they usually exhibit performance trade-offs along with their goal for increasing the utilization.

In our future work, we plan to apply the CPU-PC metric on higher layers of Cloud services, such as time-shared VMs as part of PaaS and SaaS. Additionally, testing and expanding the approach on different virtualization environments, such as Xen and VMware, would provide a more comprehensive metric. Furthermore, resources such as memory, storage and network bandwidth can also play an important role in the application's performance. We plan on developing new performance metrics for each resource, and finally deriving an ultimate performance metric for resource provisioning.

ACKNOWLEDGMENT

The work described in this paper has been funded through the Haley project (Holistic Energy Efficient Hybrid Clouds) as part of the TU Vienna Distinguished Young Scientist Award 2011.

REFERENCES

[1] J. Dejun, G. Pierre, and C.-H. Chi, "Ec2 performance analysis for resource provisioning of service-oriented applications," in *Proceedings of the 2009 International Conference on Service-oriented Computing*, ser. ICSOC/ServiceWave'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 197–207.

[2] T. Mastelic, A. Garcia Garcia, and I. Brandic, "Towards uniform management of cloud services by applying model-driven development," in *COMPSAC*, 2014.

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

[4] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 460–471, Sep. 2010.

[5] F. Benevenuto, C. Fernandes, M. Santos, V. Almeida, J. Almeida, G. J. Janakiraman, and J. R. Santos, "Performance models for virtualized applications," in *Frontiers of High Performance Computing and Networking–ISPA 2006 Workshops*. Springer, 2006, pp. 427–439.

[6] M. B. Chhetri, S. Chichin, Q. B. Vo, and R. Kowalczyk, "Smart cloudbench – automated performance benchmarking of the cloud," *2013 IEEE Sixth International Conference on Cloud Computing*, pp. 414–421, 2013.

[7] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the weather tomorrow?: Towards a benchmark for the cloud," in *Proceedings of the Second International Workshop on Testing Database Systems*, ser. DBTest '09. New York, NY, USA: ACM, 2009, pp. 9:1–9:6.

[8] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, pp. 1389–1286, 2013.

[9] Garnaat, Mitch. (2012) KVM - Kernel-based Virtual Machine. [Online]. Available: http://www.linux-kvm.org/page/Main_Page

[10] FFmpeg Team. (2000) FFmpeg - Multimedia Framework. [Online]. Available: https://www.ffmpeg.org/

[11] MongoDB Inc. (2009) MongoDB - Document-oriented database. [Online]. Available: www.mongodb.org/

[12] Rails Core Team. (2005) Ruby on Rails - Web Application Framework. [Online]. Available: www.rubyonrails.org/

[13] M. Cunha, N. Mendonca, and A. Sampaio, "A declarative environment for automatic performance evaluation in iaas clouds," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, June 2013, pp. 285–292.

[14] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Q. Wang, J. Park, and C. Pu, "Expertus: A generator approach to automate performance testing in iaas clouds," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, June 2012, pp. 115–122.

[15] D. Guo, L. Hu, M. Zhang, and M. Zhang, "Gcpsensor: a cpu performance tool for grid environments," in *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*, Sept 2005, pp. 273–278.

[16] N. Huber, M. von Quast, M. Hauck, and S. Kounev, "Evaluating and modeling virtualization performance overhead for cloud environments." in *CLOSER*. SciTePress, 2011, pp. 563–573.

[17] K. Ye, X. Jiang, S. Chen, D. Huang, and B. Wang, "Analyzing and modeling the performance in xen-based virtual cluster environment," in *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, ser. HPCC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 273–280.

[18] Paessler AG. (2014) CloudClimate - Watching the Clouds. [Online]. Available: http://www.cloudclimate.com/

[19] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, M. Kawaba, and C. Pu, "Response time reliability in cloud environments: An empirical study of n-tier applications at high resource utilization," in *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, Oct 2012, pp. 378–383.

[20] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang, "Probabilistic performance modeling of virtualized resource allocation," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 99–108.

[21] 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*, ser. QoSA '12. New York, NY, USA: ACM, 2012, pp. 91–100.

[22] A. Georges and L. Eeckhout, "Performance metrics for consolidated servers," in *System-level Virtualization for High Performance Computing, 4th Workshop, Proceedings*. Association for Computing Machinery (ACM), 2010, p. 8.

[23] S. Flannery and D. Flannery, *In Code: A Mathematical Journey*. Profile Books Limited, 2001. [Online]. Available: http://books.google.at/books?id=n6C6QgAACAAJ

[24] M. Kerrisk, *Linux Manual Pages*, 2014. [Online]. Available: http://man7.org/linux/man-pages/man5/proc.5.html

[25] T. Mastelic, V. C. Emeakaroha, M. Maurer, and I. Brandic, "M4cloud - generic application level monitoring for resource-shared cloud environments." in *CLOSER*. SciTePress, 2012, pp. 522–532.

[26] M. Schwarzkopf, D. G. Murray, and S. Hand, "The seven deadly sins of cloud computing research," in *Presented as part of the*. Berkeley, CA: USENIX.

[27] P. J. Salzman, M. Burian, and O. Pomerantz. (2007) The linux kernel module programming guide. [Online]. Available: http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html