Enabling Dynamic Virtual Frequency Scaling for Virtual Machines in the Cloud

Emile Cadorel, Romain Rouvoy

Enabling Dynamic Virtual Frequency Scaling for Virtual Machines in the Cloud

Emile CADOREL
Spirals, INRIA, Univ. Lille, CRIStAL
Lille, France
emile.cadorel@inria.fr

Romain ROUVOY
Spirals, Univ. Lille, INRIA, CRIStAL
Lille, France
romain.rouvoy@inria.fr

***Abstract*—With the democratization of the Cloud paradigm, many applications are developed to be executed inside virtual machines hosted by remote data centers providing an Infrastructure-as-a-Service (IaaS). These applications, developed by different users with different goals, tend to have different behaviors, hence a similar treatment on the Cloud provider side seems to be sub-optimal. Indeed, VM are black boxes to which are attached vCPUs, whose frequency are all the same, and are mainly indicative. In our opinion, an important limitation can be noted here. Because the Cloud provider is unaware of the applications that are executed inside the VMs, it has little insight on the behavior of the applications, and how to manage the VMs. For these reasons, Cloud provider can assign too much or too few resources to a VM, and might rely on migration mechanism to cope with that problem.**

In this paper, we propose to attach a virtual frequency to the VM template, which can be configured by the customer to better describe her expected application requirements, and the associated quality of service. Then, to enforce this virtual frequency, we designed a controller that leverages the Linux cgroup system to dynamically adjust the configuration on the host machine. We evaluate our new controller on a real infrastructure with real CPU-intensive applications executed by VM with different frequencies. We also discuss the benefits of our virtual frequency capping for VM placement.

***Index Terms*—Virtual Frequency, Cloud computing, Virtual Machines, Linux cgroup.**

I. INTRODUCTION

In the past decade, Cloud computing has become one the mainstream infrastructure used to deploy applications online. In 2021, 41% of EU companies adopted Cloud technologies, with 73% of them using advanced Cloud offers for databases or computing platforms [1]. With an increase of 5% over one year (2020–2021), one can expect that percentage to increase rapidly in the next years. Applications hosted in a Cloud infrastructure may have completely different purposes and behaviors [2], [3], from Big Data, heavy computing applications, to personal websites. Because of this heterogeneity, computing requirements can diverge from one application to another, leading to a wide diversity of Cloud platforms (FaaS, IaaS, PaaS, etc.). In this paper, we mostly focus on applications that are executed as *Virtual Machines* (VM) in an *Infrastructure-as-a-Service* (IaaS) solution.

In the literature, VMs are units of computations leveraging vCPUs, where the effective mapping of a virtual CPU on a physical CPU is given by a consolidation factor (also known as overcommitment ratio). This consolidation factor is statically fixed by the Cloud provider based on *Service-Level Objectives* (SLO) they want to guarantee, eventually relying on performance mitigation techniques, like VM migration, whenever SLO fails to be satisfied [4], [5]. On the customer side, depending on the application hosted by VM, computing power may be of varying degrees of importance. For example, personal website and big data applications have completely different requirements. In current public IaaS offers (*e.g.*, Amazon EC2, Microsoft AZURE, etc.), the customer has to pick a VM template and configure a number of vCPUs and a quantity of memory to allocate *a priori*, while the frequency of the aforementioned vCPUs is mostly indicative. In practice, the frequency of a vCPU is allocated on a best effort basis, mostly depending on the current frequency of the physical machine hosting the associated VM, which may vary from one machine to another. However, this frequency variability resulting from the hardware configuration and the associated scheduler (*e.g.*, CFS for the default configuration of KVM) cannot guarantee any fairness in the allocation of vCPU cycles. This lack of guarantees leads Cloud provider to limit the VM consolidation factor, hence contributing to a waste of provisioned, yet unused, resources (DRAM and CPU in most of the cases). Given that data centers are often criticized for their environmental impact, we believe that novel contributions are required to optimize both performance guarantees and resources utilization.

In this paper, we therefore introduce a novel approach for vCPU management, by including the *virtual frequency* as a new configuration settings in the VM template. This virtual frequency is defined by the customer, based on her application requirements and the quality of service she requires. The introduction of virtual frequency brings new challenges, namely: *i)* how to guarantee the frequency requested by the customers, when the frequency is different from one VM to another and *ii)* How to leverage this new dimension to improve the VM consolidation and reduce the number of physical machines required to host a given VM workload?

The remainder of this paper is organized as follows. First Section II presents the related work on VM consolidation and frequency management, and highlights the limitations of the existing solutions. Second, Section III presents the contribution, a controller for the virtual frequency of vCPUs.

Third, Section IV evaluates the proposed controller. Finally, Section V concludes this work and opens some perspectives.

II. State of the art

Beyond the wide diversity of application workloads hosted by IaaS—from Big Data imposing heavy I/O, to HPC with intensive CPU and memory requirements—Cloud provider are operating VM as black boxes with limited observations delivered by their monitoring capabilities. In this context, this lack of visibility leads to delivering the same access to resources to every kind of applications. However, supporting resource-intensive applications (CPU, I/O, Memory, etc.) is required to limit resources under-utilizations and associated wastes [6]–[8]. To address this challenge, many works have been conducted on VM consolidation. VM consolidation aims to deliver a fair access to resources, *e.g.*, by emulating two vCPUs atop a CPU core, and assuming these vCPUs can share CPU cycles. When this assumption fails to be met, a consolidation algorithm can be triggered to better place VM on set of *Physical Machines* (PM).

In [9], [10], a consolidation algorithm is used to place a set of VMs with different workloads on a set of PMs. In this paper, a model is proposed to predict the VM workloads during the placement phase, aiming to reduce the number of allocated nodes, and therefore the energy consumption of the infrastructure. In [9], the prediction phase is also used to control the frequency of the host machine and reduce the energy consumption by using DVFS techniques: when the observed workload is not heavy enough to use all the resources of the host nodes. While a plethora of consolidation algorithms exists in the state of the art [11], legacy techniques mostly rely on VM migrations, PM transition to low power mode, and DVFS. This means that, to the best of our knowledge, none of them is applying performance capping on VMs. Practically, all VMs are free to use all the provisioned resources, typically the max frequency of a vCPU is the one of the physical CPU. One can argue that sharing two vCPUs on a single CPU core is a form a performance capping, but two VMs share the resources, with the CPU time being fairly divided between the two vCPUs by the scheduler. This approach can be sub-optimal if a VM owner places less importance on performance than the other.

As previously mentioned, VM are black boxes grouping a fixed number of vCPUs that are emulated on a physical machine. These vCPUs may be running at variable speeds, depending on the allocated CPU time and the frequency of the physical CPU. CPU cores can dynamically adjust their frequency and the energy required to run applications, depending on the current workload. This technique, known as DVFS, can be controlled by the operating system to cap the power consumption of the physical machine. However, such a capping is not available for VMs, and controlling the maximum frequency of a VM is not always ideal. For example, when a VM is running alone on a host—or neighboring VMs have little or no workload—capping its performance would only result in resource wastes at the scale of the node. Furthermore, as CPUs are more energy efficient at high frequency, wasting compute power may actually lead to consume more energy to complete the same workload [12].

Recently, public Cloud infrastructures have introduced a new type of VM template called *Burst VM* [13], [14]. Unlike classical VM, Burst VM adopt vCPU with a capped frequency, which cannot be exceeded. The user gain credits, when the VM is running idle, and accumulated credits can be used to remove the frequency capping for a brief period of time. The VM can be set in unlimited mode, meaning that when the VM has no credit, going over the base frequency is possible, but increases the hourly price of the VM instance. However, this system of binary toggle has some limitations. Firstly, the low frequency is not really chosen by the customer, but is a part of the VM template, where the limitation is about 10% of the vCPU max utilization, which is abstract and relatively low—hence being more suitable for applications with low utilization and periodic peaks of activity (*e.g.*, low traffic websites). Secondly, when the VM frequency is uncapped, there is no limitation of the vCPU utilization of the CPU time, hence raising the problem of classical VM consolidation. Thirdly, when a Burst VM has no credit but a heavy workload, and is running on a node that has a low workload, its frequency is kept capped and thus resources are wasted even if a burst could have been engaged without degrading the performance of neighboring VMs.

In [15]–[17], a new kind of VM is presented: the *spot instances* [18]–[20]. Spot instances are VMs that have an unpredictable lifespan, as they can be killed by the Cloud provider when their provisioned resources are requested to host classical VMs. In [15]–[17], the authors propose to shrink the size of the VMs when some of their resources are claimed by the Cloud provider, and restore them when those resources are recovered. This kind of VMs are more suited for applications with two properties: being bound to a limited period of time and being replayable without loss of information (*e.g.*, batch applications).

In [21], authors propose a model for predicting the CPU utilization of a given VM. This prediction is then used to control the CPU share of the aforementioned VM to reduce the energy consumption of the host. The authors claim that their prediction algorithm reduces SLA violations and energy consumption. However, their proposed approach does not deliver differentiated frequencies to the hosted VMs, assuming they share the same priority. Moreover, the evaluation was not tested on a real infrastructure, which may raise some challenges in terms of monitoring latency over controlling. In [22] a pretty similar approach is presented, where a predictive algorithm is used to estimate the amount of CPU time that is required by any hosted VM, and apply a scaling of the frequency of the host node in order to reduce the energy consumption. In both these previous works, one can observe a key limitation: in case of high demand of CPU consumption, VM will compete for resources at the frequency imposed by the hardware and, if it is not possible because the consolidation is too high, migrations would be performed.

To better address the challenge of optimization and fair

access to resource utilization in IaaS, in this paper we propose to control the frequency of vCPUs by taking into account the current state of the host operating system, in order to avoid resource wasting. The idea of the proposed controller is to control the CPU time allocation of hosted VM, and guarantee the minimal frequency requested by the customer, while performing boost of frequency when there are some remaining resources to spend. The minimal frequency to enforce captures the minimal *Quality of Service* (QoS) expected by the customer. To the best of our knowledge, there is no work in the state of the art that proposed such an approach.

III. Virtual Frequency Controller

In this section, we start by introducing some useful definitions before presenting our control algorithm of the vCPU frequency and, finally, we briefly present how frequency can improve VM consolidation to reduce the number of required migrations.

A. Problem Statement

Before explaining how to control the frequency of vCPUs, we need to define some elements. Every symbols that are defined in this section are summarized in Table I. Firstly, each VM template $v \in \mathcal{V}$ has some capacities, specifying the amount of provisioned resources, denoted k_v. These capacities are similar to state-of-the-art VM templates (number of vCPUs, quantity of RAM, etc.). The frequency of a vCPU, expressed in MHz, is denoted F_v.

Let $n \in \mathcal{N}$ be a IaaS node (physical machine) that can host VM instances. Each node has capacities denoted k_n, and a frequency F_n^{MAX}, which is the maximal frequency that the CPU cores of the node can achieve, when all the CPU cores are fully loaded. In the infrastructure, multiple nodes with heterogeneous hardware can be found, meaning that the F_n^{MAX} can differ from one node to another.

To properly control the frequency of the VMs, and to guarantee the QoS of hosted VM instances, we leverage the *cgroup* control system. *cgroup* is a core feature of Linux operating system that can be used to set the amount of CPU time to allocate to a given set of threads. CPU time slice is different from frequency, as it is defined as the number of micro-seconds allocated to a given thread for a given period of time, offering a generic approach for CPU time allocation. In this paper, we call a time slice (a micro-second) as a *cycle*, and we denote C_m^{MAX} as the maximum number of *cycles* that a node can allocate to threads. Its value is defined by Equation 1, where p is the duration of the allocation period, in micro-seconds, and k_m^{CPU} refers to the number of CPU cores available on the node m.

$$C_m^{\text{MAX}} = p \times k_m^{\text{CPU}} \tag{1}$$

To illustrate the concept of *cycle*, let a, b, c be 3 threads, p be equal to 1 second, and k_m^{CPU} be equal to 1 core, with 10^3 *cycles* to be distributed among the 3 threads. If we assume that thread a has an higher priority than b and c, and has twice the CPU time than b and c allocated to it. To enforce such a priority using *cgroup* is pretty straightforward: we allocate 0.5 mega-cycles to a, and 0.25 to b, and another 0.25 to process c. Figure 1 illustrate this example. Note that the default Linux scheduler, *Completely Fair Scheduler* (CFS), divides the allocated times into shorter time slices, to simulate parallel execution. One can note, that frequency is completely abstracted thanks to this system, and that time allocation is relatively similar from one node to another (only depending on the amount of CPU cores).

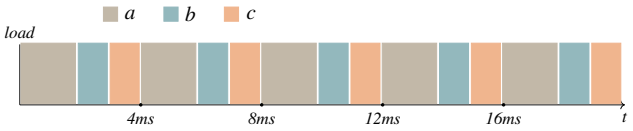


Fig. 1: Example of *cgroup* capabilities on 3 processes and 1 CPU core

Let $i \in \mathcal{I}$ be an instance of a VM template $v \in \mathcal{V}$, and $V(i) \in \mathcal{V}$ be the template used for the instance i, and $N(i) \in \mathcal{N}$ be the node hosting the instance i. Let $c_{i,j,t}$ be the number of *cycles* allocated to the vCPU j of the VM instance i at instant t for $t \in \mathbb{N}$. And let $u_{i,j,t}$ be the number of *cycles* consumed by the vCPU j of the VM instance i at instant t, such that $u_{i,j,t} \leq c_{i,j,t}$.

Name	Definition
$\mathcal{V}$	Set of virtual machine (VM) templates
$\mathcal{N}$	Set of physical machines (PM)
$\mathcal{I}$	Set of hosted VM instances
$V(i)$	The VM template of the instance i
$N(i)$	The PM hosting the instance i
k_r^{X}	Amount of X capacities on the resource r
C_m^{MAX}	The maximum amount of *cycles* available on machine m
F_m^{MAX}	The maximum frequency of the machine m, in MHz
F_i	The frequency of the VM instance i, in MHz
C_i	The amount of *cycles* to guarantee for the vCPUs of instance i
$c_{i,j,t}$	The number of *cycles* allocated to j of i at instant t
$u_{i,j,t}$	The number of *cycles* consumed by j of i at instant t

TABLE I: Summary of symbols used for frequency controlling

Let C_i be the number of *cycles* to target in order to guarantee the QoS of a VM instance i, such that it translates the frequency $F_{V(i)}$ into *cycles* on node $N(i)$. Its value is defined by Equation 2. This value depends on the location of the VM instance, as it depends on the maximal frequency of the machine that is hosting the instance, where $F_{V(i)} \leq F_{N(i)}^{\text{MAX}}$. We will see in Section IV, that there is a strict relation between *cycles* target and frequency target. Note that C_i is the number of vCPU *cycles* to enforce for the VM instance i, but not a capping limit, thus $c_{i,j,t}$ can be higher than $C_{V(i),N(i)}$. It can also be lower depending on the consumption of the vCPU $u_{i,j,t}$.

$$C_i = \frac{p \times F_{V(i)}}{F_{N(i)}^{\text{MAX}}} \tag{2}$$

B. Controlling the Frequency of vCPUs

This section reports on the implementation of the vCPU frequency controller we designed. This controller is based on

a feedback control loop, which is composed of 6 different stages described in Figure 2. The controller is using the Linux *cgroups*, and VMs are provisioned and managed using KVM. There are two versions of *cgroup* in Linux, however the version is not important as our controller works on both versions, but for sake of simplicity the explanation in the following subsections is assuming that *cgroup* v2 is used. The controller is triggered every p seconds, to adapt to the dynamic nature of the context.

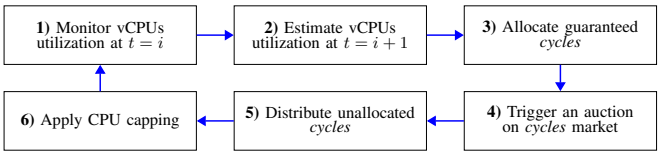


Fig. 2: Block diagram of the vCPU Controller

1) Monitoring the vCPU resource consumption: When a new VM is provisioned, a *cgroup* is created by KVM. The *cgroup* created is divided in multiple sub *cgroups* with a sub *cgroup* for each vCPUs, making vCPU monitoring possible. Each *cgroup* sub-directory includes the descriptors `cpu.stat`, `cpu.max` and `cgroup.threads`. The `cpu.stat` file gives access to the number of *cycles* consumed by the *cgroup* (here a vCPU) since its creation as the sum of the consumption of the threads it is controlling. From this information, the consumption of the vCPU can be computed, and used by the following control stage. The vCPU consumption is computed as the number of *cycles* consumed at instant t minus the number of *cycles* consumed at instant $t-1$. The reading of the consumption is made at every iteration of the controller, providing us the value of $u_{i,j,t}$, for a VM instance i, and a vCPU j, at instant t.

The file `cgroup.threads` lists the identifiers of the threads controlled by the vCPU *cgroup*—only one identifier when using KVM virtual machines. This thread identifier is useful to retrieve more information about the consumption of the vCPU. Indeed, the file `/proc/{tid}/stat` delivers more information about the execution of the thread, such as the identifier of the last CPU core that was executing the thread. From this information, a realistic estimation of the vCPU frequency can be computed. In addition the files located in `/sys/devices/system/cpu/cpu(`i`)/cpufreq/scaling_cur_freq` share the current frequency in Hertz of the CPU core whose identifier is i.

However, because there can be many vCPUs running on the node, and because the scheduler can move vCPUs from core to core relatively often depending on how stress the threads are (can be a matter of milliseconds), the file `/proc/{tid}/stat` should be read quite often to accurately trace the location of the vCPU threads, imposing high CPU usage for the monitoring of the vCPUs, which is not acceptable in our context. For that reason, the following assumption is made, vCPU threads with high workload are moved less often than vCPU threads with low workload. As low loaded vCPUs have a low impact on the frequency of the CPU cores, their virtual frequency is less dependent on their location, and is in any cases relatively low. In such situations, accurate information is not critical. On the other hand, because highly loaded vCPUs are moved less often, read the location information frequently is useless. In addition, because the Linux scheduler increases the speed of the CPU cores that are running this kind of vCPUs—making all the CPU cores running at approximately the same speed—the allocation of the vCPUs becomes of little importance in estimating the virtual frequency. Moreover the frequency is mostly used for monitoring and evaluation purpose, and not for control, and we will see in Section IV that the estimation obtained when reading the location of the vCPU only once by iteration (every seconds) delivers accurate results.

2) Estimating the upcoming vCPU utilization: There is a limited number of *cycles* to distribute every iteration—C_m^{MAX}—depending on the number of CPU cores on the node hosting the VM instances. Furthermore, there are more vCPUs running on the node than CPU cores and, thus, allocating all a CPU core for the execution of one vCPU is impossible. In this context, there is a competition among the vCPUs to use the available CPU time, this competition is called the market of *cycles*.

The number of *cycles* consumed by a vCPU $u_{i,j,t}$ depends on its workload. The distribution of *cycles* among vCPUs relies on the estimation of the number of upcoming *cycles* to be consumed by the vCPUs during the next period. Indeed, the objective is to deliver a fair distribution of the *cycles*—weighted to their respective frequency—and avoid as much as possible to allocate *cycles* to a vCPU that will not use them, and thus waste resources that could have proved to be useful if allocated to another vCPU.

Because the virtual frequency controller is running on the same node as the vCPUs, it also consumes *cycles*. On top of that, it must implement fast allocations to quickly adapt the dynamic context imposed by changing behaviors of vCPU requests. For these reasons, it must consume as little as possible CPU time. This is why we opted for a trigger-based system for the estimation stage. For each vCPU, an history of consumption is computed, storing the consumptions of the last n iterations. This history is used to compute the vCPU consumption trend, hence guessing the behavior of the vCPU and its demand. Equation 3 describes this trend, where $\overline{u_{i,j}}^{t,t+n}$ is the average of consumption from instant t to instant $t+n$, and $S_n = \frac{n(n+1)}{2}$.

$$trend_j = \frac{\sum_{x=1}^{n} ((x - S_n) \times (u_{i,j,x} - \overline{u_{i,j}}^{t,t+n}))}{\sum_{x=1}^{n} (x - S_n)^2} \tag{3}$$

The goal of this computation is to avoid brief peaks or drops of consumption to constantly change the capping $c_{i,j,t}$ and thus generating an oscillation around the proper capping, increasing the amount of resource wasting and consequently degrading

the overall performances. There are three different cases to consider:

a) The vCPU is increasing its *cycles* consumption. In such a case, $trend_j$ is strictly positive and $u_{i,j,t}$ exceeds the increase trigger, as it can be seen in Figure 3. The increase trigger is configurable. When exceeded, the capping is increased by a percentage that is configurable as well. For example, in Figure 3, increase trigger is set to 0.9 of the capping, and the increase factor is 1.3. The higher the increase factor, the faster the convergence to the correct capping will be reached, but also the higher the resource wastage will be.

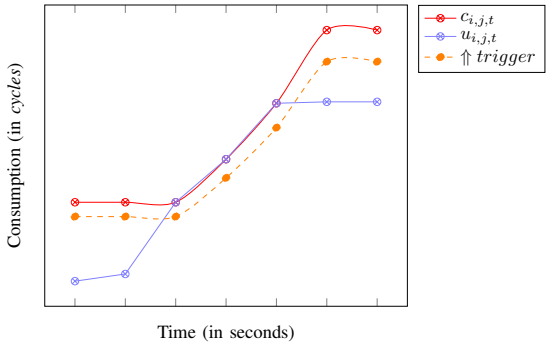


Fig. 3: Increasing *cycles* consumption of a vCPU

b) The vCPU is decreasing its *cycles* consumption. In this case, the trend is strictly negative, and the consumption $u_{i,j,t}$ is lower than the decrease trigger. As for increase trigger and increase factor, decrease trigger and decrease factor are configurable. Figure 4 shows an example where decrease trigger is 0.5 and decrease factor is 0.8. The decrease factor should not be too big, indeed in case of short decrease of resource usage for a few seconds (or a big decrease during just a few iteration), the capping would have to go up again with the limitation of the increase factor. This would result in some sort of oscillation which is counter productive for performance and resource wasting.

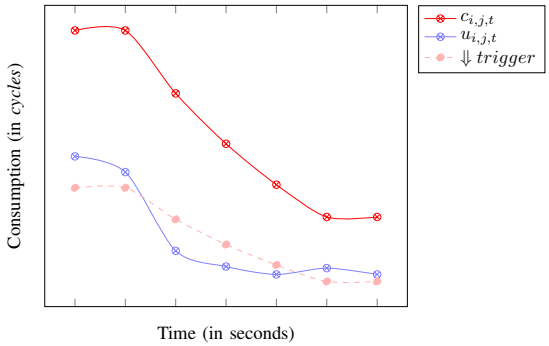


Fig. 4: Decreasing *cycles* consumption of a vCPU

c) The consumption is stable. In this last case, the trend equals to 0 (or almost equal to 0 with a margin of error). Because the consumption is stable it does not exceed any of the increase of decrease triggers. However, it can still be wasting some *cycles*, for example when decreasing really slowly without triggering the decrease mechanism. In that case, capping is set to avoid the triggering of the increase mechanism in the next iteration, but close enough to the consumption to avoid *cycle* wastes. An example of this calibration is depicted in Figure 5.

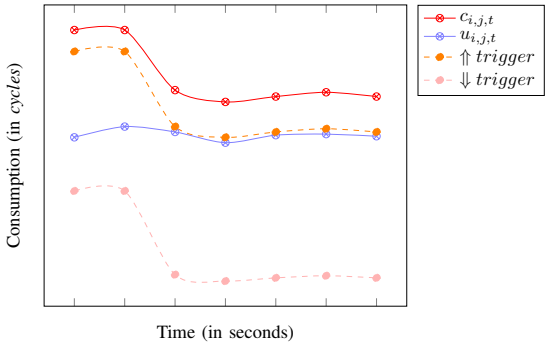


Fig. 5: Stable *cycles* consumption of a vCPU

Once the estimation of the number of *cycles* that will be consumed by the vCPUs in the next iteration is made, the controller enters the third stage.

3) Enforcing the vCPU cycles: Based on the usage of the vCPUs during the last iteration, the controller allocate credits to VM associated to the vCPUs, and keeps track of each VM wallet. These credits are used in the next stage of the controller to prioritize VMs that are often consuming less *cycles* than expected, over the more greedy VMs, when there are some unallocated *cycles* to distribute. Whenever a VM instance has vCPUs that are consuming fewer *cycles* than the number of *cycles* associated with their base frequency, it earns credits as presented in Equation 4, where $k_{V(i)}^{vCPUs}$ is the number of vCPUs of the VM instance i.

$$\forall i \in \mathcal{I},$$
$$credit_i+ = \sum_{j=0}^{k_{V(i)}^{\text{vCPU}}} \begin{cases} C_i - u_{i,j,t-1}, & \text{if } C_i > u_{i,j,t-1} \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

Then, the controller sets the base capping of each vCPUs, where the base frequency is to be guaranteed if the estimated vCPUs consumption at next iteration is higher than the guaranteed *cycles*, as presented in Equation 5, where $e_{i,j,t}$ is the estimation of the usage of the vCPU j of VM instance i at instant t, as computed by the previous stage of the controller.

$$\forall i \in \mathcal{I},\ \forall j \in [0, k_{v(i)}^{vCPUs}],$$
$$c_{i,j,t} = \begin{cases} e_{i,j,t}, & \text{if } e_{i,j,t} < C_i \\ C_i, & \text{otherwise} \end{cases} \quad (5)$$

4) Triggering an auction for vCPU cycles: The estimation of the number of *cycles* that will be consumed in the next iteration are not taking into account the finite number of resources a node can provide. In the previous stage, the base frequency of the vCPUs was taken into account by making

sure that vCPUs requiring access to their guaranteed resources can access them (cf. Equation 5). From the previous stage, one can also note that some vCPUs might not use their guaranteed *cycles* due to a period of low intensity in their workload. The *cycles* that are unallocated would be wasted if left as such, as they could be allocated to other vCPUs that have higher demand than their base frequency.

This is the goal of this fourth stage: to launch an auction in order to sell the *cycles* that are allocated to no vCPUs, in order to avoid resource wasting while giving priority to VMs that used this possibility of allocation burst less often. Let $buyers$ be the set of vCPUs whose current allocation $c_{i,j,t}$ are lower than the estimation $e_{i,j,t}$. Let $market_n$ be the number of *cycles* that are not allocated to any vCPUs as presented in Equation 6 on node $n \in \mathcal{N}$, where $\mathcal{I}_n$ is the set of VM instances hosted on node n, and p is the duration of a control loop iteration.

$$market_n = (k_n^{\text{CPU}} * p) - \sum_{i \in \mathcal{I}_n} \sum_{j=0}^{k_{V(i)}^{\text{vCPU}}} c_{i,j,t} \tag{6}$$

The *cycles* included in the $market_n$ are distributed among the vCPUs owned by the $buyers$ set as presented in Algorithm 1. One can note that a *window* is used when allocating the remaining *cycles* of the $market_n$. This window is used to avoid that a rich VM steals all the *cycles* included in the $market_n$, and to distribute the *cycles* fairly among vCPUs attached to VM instances having credits to spend.

Algorithm 1 *Cycles* auction algorithm

```
function TRIGGERCYCLEAUCTION(market_n, buyers, window)
    while |buyers| ≠ 0 and market_n ≠ 0 do
        for i, j ∈ buyers do
            tobuy ← e_{i,j,t} − c_{i,j,t}
            buying ← min(tobuy, market_n, credit_i, window)
            if buying = 0 then ▷ no credit left, or the vCPU needs no more cycle
                Remove i, j from buyers
            else                                          ▷ Buy some more cycles
                c_{i,j,t} = c_{i,j,t} + buying
                market_n = market_n − buying
                credit_i = credit_i − buying
```

5) Distributing the unallocated cycles: As it can be noticed in the above algorithm, the auction can stop if no VM instance has enough credits to buy the $cycles$ sold on the market. This means that at the end of Algorithm 1, the market may still contain unallocated $cycles$. These $cycles$ are freely distributed among the vCPUs whose allocation $c_{i,j,t}$ is still lower than their estimate $e_{i,j,t}$. This allocation is proportional to the demand of each vCPU to the overall demand.

6) Applying vCPU capping: At this final stage, every vCPU is allocated $c_{i,j,t}$ *cycles*. This *cycles* allocation can be translated into *cgroup* capping in a straightforward way.

After the capping of every vCPU is done, the controller is put in sleep for the remaining time $p - spent$, where p is the triggering period of the controller and $spent$ is the time taken by the different stages presented in Figure 2.

C. Improving VM placement

While VM placement and consolidation is not the primary focus of this paper, we believe that virtual frequency capping can contribute to improve state-of-the-art VM placement algorithms. In particular, our contribution can enable a CPU core to host multiple vCPUs without enforcing overcommitment—*e.g.*, a CPU core running at 3 GHz could host 3 vCPUs with a frequency guaranteed at 1 GHz. We therefore believe that future works could explore the extension of state-of-the-art algorithms, such as First-Fit or Best-Fit, to include the virtual frequency as a new consolidation dimension. To do so, the CPU core constraint (stating that the number of vCPUs must be lower or equal to the number of CPU cores) is replaced by the core splitting constraint defined by Equation 7.

$$\sum_{i \in \mathcal{I}_n} (k_i^{\text{vCPU}} \times F_i) \leq k_n^{\text{CPU}} \times F_n^{\text{MAX}} \tag{7}$$

As it is already done for the the CPU core constraint, a consolidation factor can be added (*e.g.*, multiple by 1.2 the number of available cores on the node), but this could lead in the loss of the guarantee of the vCPU frequency, or would impose migrations to keep the guarantee.

IV. Experimental Evaluation

In this section, we first evaluate the benefits of the virtual frequency controller, before highlighting the energy savings that can be achieved thanks to our contribution.

A. Evaluating the Virtual Frequency Controller

We provide a C++ implementation of the Virtual Frequency Controller in a public git repository.[1]

1) Experimental protocol: In this first evaluation, we consider different templates of VM, which are co-hosted on a single node, and we execute two workloads provided by Phoronix test suite: the `compress-7zip` and `openssl` benchmarks.[2] These benchmarks are commonly used when evaluating the performances of a compute node [23]–[25]. The objective of this evaluation is to assess the capability of the controller to adapt to the workload, while guaranteeing the base vCPU frequency on two different nodes with different CPUs. The description of the nodes we used is reported in Table IV, these nodes are part of the Grid'5000 [3] experimental platform. Each workload was executed 4 times (and twice on each node): twice without the controller—labelled as A—and twice with the controller enabled–labelled as B.

VM	vCPUs	Frequency	Instances	Workload
small	2	500 MHz	20	`compress-7zip`
large	4	1800 MHz	10	`compress-7zip`

TABLE II: Description of the workload on *chetemi*

Tables III and II report on the VM configurations that were deployed on the two different nodes. Note that the number of

[1] https://gitlab.inria.fr/ecadorel/cgroup-monitor
[2] https://www.phoronix-test-suite.com/
[3] https://www.grid5000.fr

VM	vCPUs	Frequency	Instances	Workload
small	2	500 MHz	32	`compress-7zip`
large	4	1800 MHz	16	`compress-7zip`

TABLE III: Description of the workload on *chiclet*

VM instances are different from one node to another, because *chiclet* has more CPU cores than *chetemi*, but both nodes are equally loaded (cf. Equation 7). VM instances are running Ubuntu 20.04 operating system. The workload of *large* instances are started after the workload of the *small* instances at $t = 200$ seconds. The objective is to evaluate the impact of a variable workload where, at the beginning—for 200 seconds—the nodes are underpovisioned, before becoming fully loaded from instant 200 seconds.

In this experiment, the controller is configured with the following settings: the increase trigger and increase factor are 95 % and 100 %, while decrease trigger and decrease factor are 50% and 5%, respectively. This variables may seem arbitrary, but were set experimentally, and offer a good tradeoff between stable capping and fast convergence. To compare the executions A and B, the monitoring stage of the controller is executed in all scenarios, while only the control part of the controller is disabled in execution A. The controller period p is set to 1 second.

2) Experimental results: Figure 6 depicts the average frequency of the vCPUs of the different instances of the two set of VMs during the execution A. All vCPUs belonging to a VM type—*i.e.*, *small* or *large*—have a relatively similar behavior, so reporting on average values loses little or no information, while making the results much easier to read. The frequency are computed using the method presented in Section III, by reading the location of the vCPUs on the CPU cores at each control iteration—*i.e.*, every seconds. Thus, this is only an approximation of the real speed of the vCPUs, as they can be moved from CPU core to other CPU core at any moment. However, because the frequency of the CPU core was also read during the experiment, we can observe that they are all running at approximately the same speed, with an average variance of 16 MHz for execution A and of 37 MHz for execution B on *chetemi*. Therefore, we can conclude that the location of a vCPU does not have a big impact on its frequency.

Because no capping of resources is applied, the scheduler splits the available resources between the different *cgroups* equally. The results obtained without the controller may seem surprising, but can be explained by the fact that the *small* instances represent $\frac{2}{3}$ of the number of VM instances, $\frac{2}{3}$ of the CPU resources are provided to them, however in term of pure resource demand *large* instances represent half the total (with 40 vCPUs). To confirm this hypothesis, we ran two experiments: *a)* an experiment with 20 VMs with 4 vCPUs each and *b)* an experiment with 40 VMs with only one vCPU, and 10 VMs with 4 vCPUs. In the experiment *a)* all the vCPUs were running at the same speed, while in the experiment *b)* $\frac{4}{5}$ of the resources were allocated to VMs with only 1 vCPU. These experiments demonstrated that the Linux CFS scheduler assumes the VMs as a whole, and not directly the vCPUs.

Figure 7 depicts the average frequency of the vCPUs of the different instances of the two set of VMs during the execution B. One can observe from this Figure 7 that vCPUs of the *small* instances are using all the resources of the CPU cores (running at the maximum frequency of the cores, *i.e.*, 2.4GHz) until the second workload established by the *large* instances starts. At this instant ($t = 200$ seconds), a difference between the scenarios A and B can be observed. When the controller is running, because the *large* instances have a higher base frequency than the *small* ones, they are prioritized by the controller. One can note that the vCPUs of the *small* instances are running at approximately 500 MHz, and that the vCPUs of the *large* instances are running at approximately 1800 MHz. This is the behavior expected for the execution with the control loop enabled.

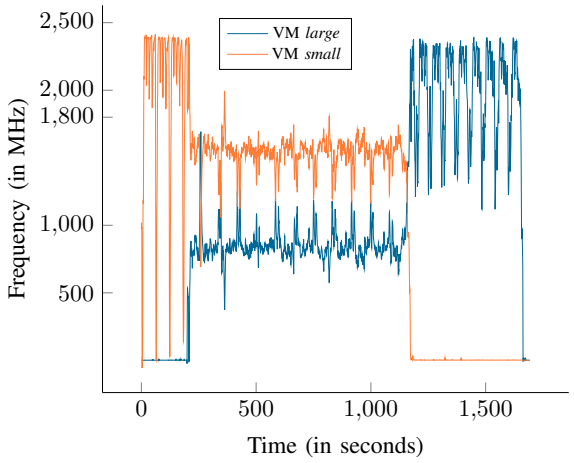


Fig. 6: Average frequency of vCPUs on *chetemi* for configuration A

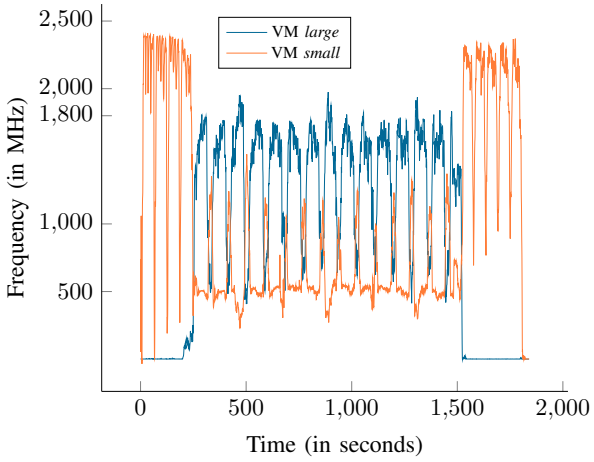


Fig. 7: Average frequency of vCPUs on *chetemi* for configuration B

Some picks in the frequency for the vCPUs of the *small* instances can be observed, when the frequency of the *large* instances is reduced due to some decrease of the workload of the compression benchmark. This demonstrates the capability

Name	CPU	Cores	Frequency	Memory	Disk
chetemi	2x Intel Xeon E5-2630 v4	10 cores/CPU	2,400 MHz	256 GB	2x 300 GB HDD
chiclet	2x AMD EPYC 7301	16 cores/CPU	2,400 MHz	128 GB	480 GB SSD

TABLE IV: Nodes used for the experimentations

of the controller to better allocate resources that would have been wasted otherwise. During the execution type B on *chetemi*, the controller was executed every second, and took 5 ms on average, including 4ms taken by the monitoring stage. Note that these 4 ms are also present during the execution A, as we need it for result comparison. The controller is running on only one core, thus it is consuming very few resources.

Figures 8 and 9 shows the scenarios A and B on the *chiclet*. As for the execution B on *chetemi*, during the execution B on *chiclet*, the vCPUs of the *small* instances are running approximately at 500 MHz, and the vCPUs of the *large* instances are running approximately at 1800 MHz. On the other hand, scenario A is again giving more priority to *small* instances than *large* instances, but this time it is less obvious, this can be explained by many side effects related to the architecture of the host node. These differences could also explain why the decrease of consumption during synchronization of the `compress-7zip` benchmark seems to be less marked in comparison to *chetemi* executions. As for *chetemi*, all the CPU cores are running at approximately the same speed, with an average variance of 88 MHz for execution A and of 150 MHz for execution B.

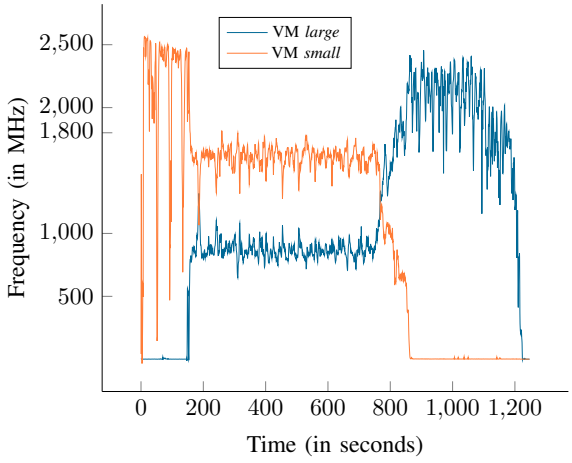


Fig. 8: Average frequency of vCPUs on *chiclet* for configuration A

Figures 10 and 11 depicts the results of the compression benchmark for the *small* instances and the *large* instances, for the four scenarios A and B on both nodes. The compression benchmark was ran 15 times in each VM instances, that is represented by the x-axis. The results are the average of the results of each VM instances. The first observation is that execution on *large* instances is more stable in scenario B than in A. In scenario A, the Linux scheduler gives the same priority to *small* and *large* instances, resulting in the same compression

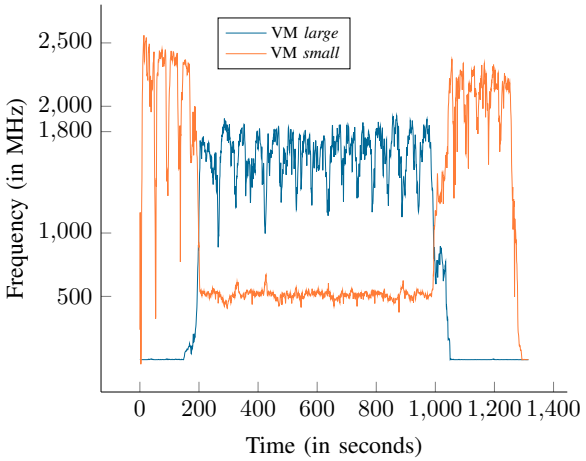
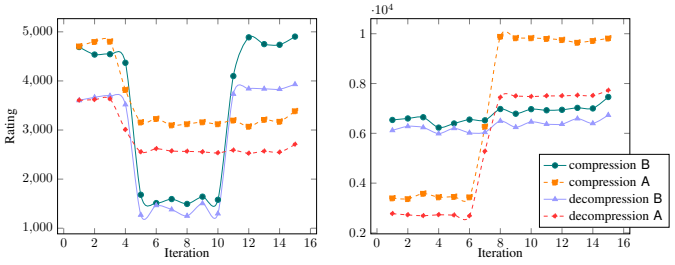


Fig. 9: Average frequency of vCPUs on *chiclet* for configuration B

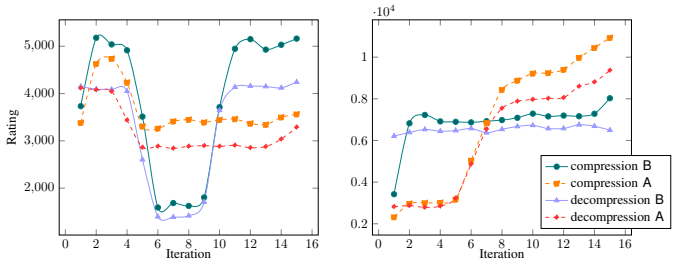
and decompression rates, when running at the same time. On the other hand, in scenario B, *small* instances have a lesser priority and thus, when *large* instances starts their workload, the perfomance boost of *small* instances drops, as expected. One can also note that when no capping is needed, because there is no collision in *cycle* demand, scenarios A and B have similar results (first 3 iterations of the benchmark in *small* instances). In addition, it can also be noted from this result that our explanation about resource sharing in scenario A can be acknowledged, as *small* and *large* instances have the same results when running along, even if *large* instances have more vCPUs than *small* instances.



(a) VM *small* (b) VM *large*

Fig. 10: Compression and decompression efficiency of *small* VM instances on *chetemi*

The executions of scenario B on *chetemi* and *chiclet* report on the same behavior, and give almost identical performances for both *small* and *large* instances, with slightly better performances on *chiclet* than *chetemi*. Therefore, the performances are much more predictable from a customer perspective when using the controller, than without it, as they are less dependent to VM collisions.



(a) VM *small* (b) VM *large*

Fig. 11: Compression and decompression efficiency of *small* instances on *chiclet*

VM	vCPUs	Frequency	Instances	workload
small	2	500 MHz	14	`compress-7zip`
medium	4	1,200 MHz	8	`openssl`
large	4	1,800 MHz	6	`compress-7zip`

TABLE V: Description of the workloads executed on *chetemi*

B. Evaluating the Impact of Heterogeneous Workloads

1) Experimental protocol: In this second evaluation, we added *medium* VM instances. These *medium* instances are running the *openssl* benchmark of the Phoronix test suite, and have 4 vCPUs that are running at 1,200 MHz. Table V summarizes the VM configurations that were executed on the *chetemi*. As for the first evaluation, there are two configurations: A and B, but this evaluation was only executed on *chetemi* (for the sake of space). The workload of *medium* instances is started at instant $t = 100s$, and the workload of *large* instances at instant $t = 200s$.

2) Experimental results: Figures 12 and 13 report on the frequency of the VM instances for configurations A and B, respectively. One can note that once again, during the execution of configuration A, *small* vCPUs are running faster that the others, and that *medium* and *large* vCPUs are running at the same speed, for the reason we explained.

On the other hand, *large* instances are running at approximately $1,800\,MHz$, *medium* instances at approximately $1,200\,MHz$ and *small* instances at approximately $500\,MHz$, with configuration B, when all the instances are running a workload concurrently. There is a small decrease of the performances of *large* instances in comparison to the first evaluation, that is translated into a small decrease of the performances of *compress-7zip* benchmark as depicted in Figure 14. However, this decrease is really small, and can be due to other factor than CPU *cycle* allocation (*e.g.*, cache allocation) that could be subject of future work. Results of *small* instances are similar to those obtained during the first evaluation, when running at the guaranteed frequency.

When the workload on *medium* instances completes, there are unallocated *cycles* that are distributed among *large* and *small* instances, increasing their frequencies, and thus the performances of their applications.

C. Evaluating the Impact on VM Placement

In this last evaluation, we observe the impact of the performance capping on the placement phase of the VMs. This paper

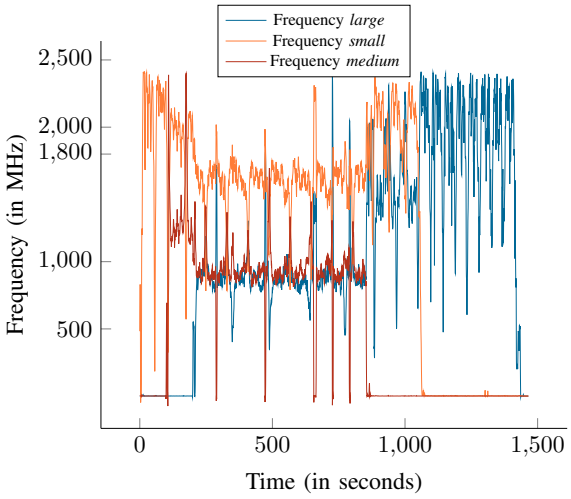


Fig. 12: Average frequency of vCPUs on *chetemi* execution type A of second evaluation

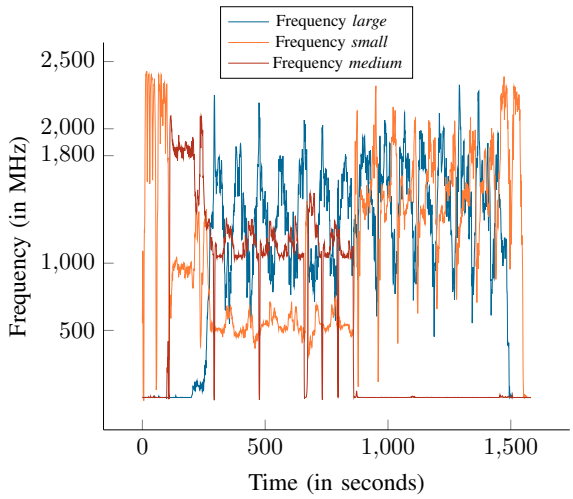
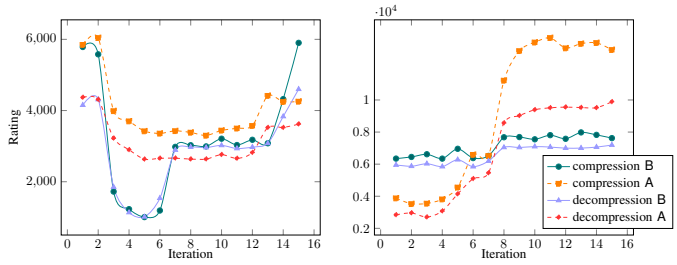


Fig. 13: Average frequency of vCPUs on *chetemi* execution type B of second evaluation

is more focused on the performance capping, and its goal is to present a proof of concept about frequency capping for virtual machines. This last evaluation delivers an example of what could be achieved in term of infrastructure management and energy gain thanks to the performance capping at VM level. A more advanced placement algorithm and evaluation remains an interesting perspective for future works.

In this evaluation, the placement algorithm BestFit was implemented in two different versions: the first version without taking into account the frequency of the VMs, and the second version working as presented in Section III. Let the cluster be composed of 12 *chetemi*, and 10 *chiclet*, where the specifications of the cluster is presented in Table IV. Let the workload be composed of 250 *small* VMs with 2 vCPUs running at 500MHz, 50 *medium* VMs with 4 vCPUs running at 1200MHz, and 100 *large* VMs with 4 vCPUs running at 1800MHz.

With the frequency capping taken into account by the place-



(a) *small* instances (b) *large* instances

Fig. 14: Compression and decompression efficiency of *small* instances on *chetemi*

ment algorithm, only 15 out of 22 nodes are used, meaning that the 7 other nodes can be reused for additional workload, or shutdown in order to reduce the energy consumption. One can argue that the number of used nodes could be reduced without the frequency capping by adding a consolidation factor. However, this would reduce the performances of the VM instances (or trigger migrations, and thus use more nodes in the end). To obtain the same result (15 used nodes) with a consolidation factor, the number of CPU cores on the nodes has to be multiplied by 1.8. Nevertheless, in that case two *chiclet* are highly loaded with 28 *large* VM instances, against 21 when frequency capping is enabled (with constraint 7). On the other hand, 36 *small* VMs are placed on a *chetemi*, against 48 when frequency capping is enabled. By adding a frequency capping to the VMs, a form a consolidation factor is added to the placement problem. However, this consolidation factor is based on customer requirements instead of an arbitrary value, and is supported by the frequency controller, instead of migration mechanism.

V. Conclusion

This paper introduced a new system to control the frequency of the vCPUs of virtual machines. Unlike the state of the art, this controller takes into account the various requirement of the customers based on a performance guarantee. Indeed, because Cloud infrastructure are used for a wide variety of applications, customer requirements and performance expectations are quite diverse. For this reasons, our controller was designed in order to guarantee a minimal frequency for the vCPUs of the VMs—a frequency that has been chosen by the customers– by using the Linux cgroup system, while adapting the performances to the context of the execution.

The controller has been evaluated on a real infrastructure, using two different nodes with CPU from different constructor, VMs were provisioned using KVM, and tested against two different CPU-intensive benchmarks. This evaluation has shown that the controller is able to guarantee the frequency of the vCPUs, leading to more consistent and predictable performances for the customers, in comparison to standard execution models. This frequency capping also brings promising results in term of VM placement and in number of nodes used, that could lead to energy savings. This placement part is not the main focus of the paper, and could be the subject of interesting future works.

In this paper, we have pointed out some elements that can be subject of promising perspectives. First, we plan to take into account cache access for the different vCPUs, in order to give priority to faster vCPUs, and reduce the contention during highly demanding workload execution. Second, we plan to investigate the problem of memory allocation, in this paper we assumed that there was enough memory on the host nodes for all the VMs. This assumption could be false as the number of VMs per node increases due to better consolidation. Finally, we plan to work on the placement algorithm in order to evaluate more deeply the impact of frequency capping in a context of a cluster management, and observe the impact on energy consumption.

Acknowlegments

This work has received funding from the French National Research Agency (ANR) and the French Region of Hauts-de-France (under grant ANR-21-HDF1-0006).

References

[1] "Eurostat - cloud computing - statistics on the use by enterprises." [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises

[2] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan, "The rise of "big data" on cloud computing: Review and open research issues," *Information Systems*, vol. 47, pp. 98–115, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437914001288

[3] P. Verma and S. K. Sood, "Cloud-centric iot based disease diagnosis healthcare framework," *Journal of Parallel and Distributed Computing*, vol. 116, pp. 27–38, 2018, towards the Internet of Data: Applications, Opportunities and Future Challenges. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517303301

[4] F. D. Rossi, M. G. Xavier, C. A. De Rose, R. N. Calheiros, and R. Buyya, "E-eco: Performance-aware energy-efficient cloud data center orchestration," *Journal of Network and Computer Applications*, vol. 78, pp. 83–96, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804516302569

[5] M. Hosseini Shirvani, A. M. Rahmani, and A. Sahafi, "A survey study on virtual machine migration and server consolidation techniques in dvfs-enabled cloud datacenter: Taxonomy and challenges," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 3, pp. 267–286, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157818302842

[6] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 153–167. [Online]. Available: https://doi.org/10.1145/3132747.3132772

[7] Q. Zhang, P. A. Bernstein, D. S. Berger, B. Chandramouli, V. Liu, and B. T. Loo, "Compucache: Remote computable caching using spot vms," in *12th Conference on Innovative Data Systems Research, CIDR*, 2022, pp. 9–12.

[8] A. Fuerst, S. Novaković, I. n. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini, "Memory-harvesting vms in cloud platforms," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 583–594. [Online]. Available: https://doi.org/10.1145/3503222.3507725

[9] M. Chehelgerdi-Samani and F. Safi-Esfahani, "Pcvm.arima: predictive consolidation of virtual machines applying arima method," *The Journal of Supercomputing*, vol. 77, no. 3, pp. 2172–2206, Mar 2021. [Online]. Available: https://doi.org/10.1007/s11227-020-03354-3

[10] S. Mazumdar and M. Pranzo, "Power efficient server consolidation for cloud data center," *Future Generation Computer Systems*, vol. 70, pp. 4–16, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X16308093

[11] N. Hamdi and W. Chainbi, "A survey on energy aware vm consolidation strategies," *Sustainable Computing: Informatics and Systems*, vol. 23, pp. 80–87, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537918300507

[12] W. Wu, W. Lin, and Z. Peng, "An intelligent power consumption model for virtual machines under cpu-intensive workload in cloud environment," *Soft Computing*, vol. 21, no. 19, pp. 5755–5764, Oct 2017. [Online]. Available: https://doi.org/10.1007/s00500-016-2154-6

[13] "Amazon elastic compute cloud. 2022. amazon ec2 burstable performance instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/#Burstable_Performance_Instances

[14] "Microsoft azure. 2022. azure burstable virtual machine." [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable

[15] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini, "Providing SLOs for Resource-Harvesting VMs in cloud platforms," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 735–751. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ambati

[16] P. Sharma, A. Ali-Eldin, and P. Shenoy, "Resource deflation: A new approach for transient resource reclamation," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303945

[17] Y. Wang, K. Arya, M. Kogias, M. Vanga, A. Bhandari, N. J. Yadwadkar, S. Sen, S. Elnikety, C. Kozyrakis, and R. Bianchini, "Smartharvest: Harvesting idle cpus safely and efficiently in the cloud," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–16. [Online]. Available: https://doi.org/10.1145/3447786.3456225

[18] "Amazon elastic compute cloud. 2022. amazon ec2 spot instances." [Online]. Available: https://aws.amazon.com/ec2/spot/

[19] "Google cloud. 2020. preemptible vm instances." [Online]. Available: https://cloud.google.com/compute/docs/instances/preemptible.

[20] "Microsoft azure. 2020. azure spot virtual machines." [Online]. Available: https://azure.microsoft.com/en-us/pricing/spot

[21] K. Shojaei, F. Safi-Esfahani, and S. Ayat, "Vmdfs: virtual machine dynamic frequency scaling framework in cloud computing," *The Journal of Supercomputing*, vol. 74, no. 11, pp. 5944–5979, Nov 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2508-1

[22] C.-M. Wu, R.-S. Chang, and H.-Y. Chan, "A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters," *Future Generation Computer Systems*, vol. 37, pp. 141–147, 2014, special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X13001234

[23] A. Kaur, G. Raj, S. Yadav, and T. Choudhury, "Performance evaluation of aws and ibm cloud platforms for security mechanism," in *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*, 2018, pp. 516–520.

[24] T. Deshane, Z. Shepherd, J. Matthews, M. Ben-Yehuda, A. Shah, and B. Rao, "Quantitative comparison of xen and kvm," *Xen Summit, Boston, MA, USA*, pp. 1–2, 2008.

[25] K. M. Derdus, V. O. Omwenga, and P. J. Ogao, "The effect of cloud workload consolidation on cloud energy consumption and performance in multi-tenant cloud infrastructure," *International Journal of Computer Applications*, vol. 181, no. 37, pp. 47–52, 2019.