# GPaaScaler: Green Energy Aware Platform Scaler for Interactive Cloud Application

MD Sabbir Hasan
INSA Rennes, INRIA, IRISA, UBL
Rennes, France
sabbir.hasan@inria.fr

Frederico Alvares
IMT Atlantique, INRIA, LS2N, UBL
Nantes, France
frederico.alvares@inria.fr

Thomas Ledoux
IMT Atlantique, INRIA, LS2N, UBL
Nantes, France
thomas.ledoux@inria.fr

## ABSTRACT

Recently, smart usage of renewable energy has been a hot topic in the Cloud community. In this vein, we have recently proposed the creation of green energy awareness around *Interactive Cloud Applications*, but in static amount of underlying resources. This paper adds to previous ones as it considers elastic underlying infrastructure, that is, we propose a PaaS solution which efficiently utilize the elasticity nature at both infrastructure and application levels, by leveraging adaptation in facing to changing condition i.e., workload burst, performance degradation, quality of energy, etc. While applications are adapted by dynamically re-configuring their service level based on performance and/or green energy availability, the infrastructure takes care of addition/removal of resources based on application's resource demand. Both adaptive behaviors are implemented in separated modules and are coordinated in a sequential manner. We validate our approach by extensive experiments and results obtained over Grid'5000 test bed. Results show that, application can reduce significant amount of brown energy consumption by 35% and daily instance hour cost by 37% compared to a baseline approach when green energy aware adaptation is considered.

## KEYWORDS

Interactive Cloud application; PaaS; Energy consumption; Autonomic computing; Green IT; Sustainable computing

## 1 INTRODUCTION

The fast growth of internet technology and proliferation of Cloud services have multiplied data centers number in recent years. In 2016, data centers around the world consumed 416.2 TWh of electricity, which is significantly higher than the UK's total consumption of about 300 TWh on the same year[1]. Although, numerous state-of-the-art energy efficient techniques have been adopted by industry and academia, a recent report suggests that energy usage in data centers is expected to increase by 4% until 2020, which will translate to higher carbon emission. Most of today's data centers consume grid tied brown energy, very few are partially powered by renewable energy. Therefore, energy efficient techniques alone is not going to reduce the carbon footprint since energy consumption will continue to grow. On the other hand, the ever increasing enthusiasm and consciousness of reducing energy consumption can lead towards smarter ways to consume energy in cloud data centers. While an efficient energy management technique in data center can reduce unnecessary use of brown energy and better utilize green energy without going to waste [8], [13], smarter ways of consuming energy in the presence/absence of renewable/green energy by an application can further reduce carbon footprint.

Traditionally, data centers host heterogeneous applications, such as *interactive* and *batch* applications/jobs. Goiri et al. [9], [10] first proposed a green energy adaptive framework for batch job oriented tasks *i.e.,* that facilitated the scheduling of these tasks to different times by respecting the deadline when green energy is available. On the contrary, interactive applications possess lesser flexibility, i.e., it should react with little to no latency, otherwise Quality of Service (QoS) can be seriously impacted. To cope up with these limitations, we have proposed a green energy adaptive solution to create green energy awareness inside the application that inherits the capability to smartly use the available green energy having *static* amount of underlying resources [14],[15].

But in a realistic cloud environment, resource requirement might exceed the currently provisioned resources. In contrast, when fewer resources are required, de-provisioning of resources can help to reduce unnecessary energy consumption. Therefore, the capability to detect when resources are required/dispensable and react to it so as to keep performance at a targeted level while energy consumption can be minimized is required. Taking application reconfiguration decision in isolation with resource scaling policies may lead to performance degradation and inconsistency to the system. Hence, coordination between two different types of elasticity (*i.e.,* application reconfiguration vs infrastructure (de)provisioning) is necessary.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*UCC'17, , December 5–8, 2017, Austin, TX, USA.*
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5149-2/17/12. . . $15.00
https://doi.org/10.1145/3147213.3147227

[1] http://www.independent.co.uk/environment/global-warming-data-centres-to\-consume-three-times-as-much-energy-in-next-decade-experts-warn-a6830086.html

Most of the work in the literature propose: (i) multiple autonomic loops in a coordinated manner to control cluster level resources (*i.e.,* one loop for controlling DVFS, another loop for deciding scaling actions)[21], [19]; (ii) per-application local manager which requests to a central autonomic manager to tune the number of cpu core, memory and to change the number of VM's [4], [3]; (iii) adaptive framework to coordinate between system level (DVFS) and application level (degrading quality) adaptation to improve performance and power efficiency [11].

It is clearly visible that - synergy between application and infrastructure based on green energy availability is missing despite having elasticity capability in both layers. We believe that, integrating the autonomic logics of the infrastructure with the one in the applications is a important research direction. Therefore, in response to the existing works, we propose a PaaS (Platform-as-a-Service) solution, named *GPaaScaler* that inherits the capability to adapt both at *application* and at *infrastructure* level in facing to changing condition *i.e.,* workload burst, performance degradation, quality of energy etc. We refer green energy to be better in quality, compared to brown energy. Application adaptation is realized by dynamically reconfiguring application's mode/service level on the fly based on performance and/or green energy availability, whereas infrastructure adaptation takes care of addition/removal of resources based on application's resource demand. During application adaptation, any dynamic change at the software layer that impacts the energy profile can be considered as switching to higher or lower modes/service levels. To this point, we want to study the impact of application adaptation (based on the presence/absence of green energy) on infrastructure to have a global view of energy consumption incurred by the application. Furthermore, both adaptation technique is built in separate modules and coordinated in a sequential manner. For example, when application's performance decreases due to heavy load, the PaaS solution first triggers adaptation to application by downgrading the functionality according to signed SLA (Service Level Agreement) and invokes resource requests to infrastructure module. Followed by the invocation requests, infrastructure adaptation module analyzes and decides whether resources are going to be added or the request is to be ignored. We have tested our proposal at Grid'5000 test bed with a real life application, workload and energy profile to show that, when green energy aware adaptation is adopted, around 35% brown energy consumption can be reduced compared to a baseline approach. By reducing brown energy consumption, ratio of green energy to brown energy can be increased and subsequently carbon footprint can be appreciably reduced. The rest of the paper is organized as follows. Section 2 describes the GPaaScaler architecture. In Section 3, several Application controllers and a generic Infrastructure controller are designed to investigate their impacts on energy consumption and QoS properties and Section 4 validate approach through extensive experiments. Furthermore, in Section 5, we provide discussion based on results and observation. Section 6 describes the related works and we conclude our work in Section 7.

## 2 GPAASCALER ARCHITECTURE

This section presents our auto-scaler architecture named *GPaaScaler*, which continuously listens the instances of events *e.g.,* response time, green energy availability, working modes of application etc., pushed by SaaS(Software-as-a-Service) and IaaS(Infrastructure-as-a-Service) layers in a changing environment. Furthermore, it inherits the capability to actuate both at application and at infrastructure level. We use the most popular self-adaptive design framework: Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) loop [16] for our auto-scaler. Our contribution lies on the *analyze* and *plan* (A-P) block of this autonomic framework.

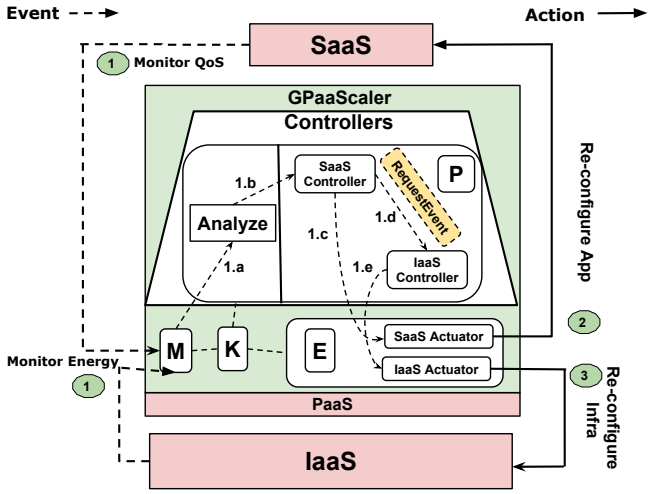


**Figure 1: GPaaScaler architecture**

Figure 1 presents the sequential control flow of the event in an ordered way (from 1.*a* to 1.*e*). Monitoring (M) block pushes listened events to *Analyze* block via (1.*a*) from SaaS layer (*i.e.,* response time, workload, application's working mode, etc.) and from IaaS layer (*i.e.,* quality of energy). Analyze (A) block is responsible for analyzing and decoupling events to extract the pertinent information and feed appropriate event to the event handler at the SaaS controller via 1.*b* flow. Once the events are received, Plan (P) block analyzes and matches to the predefined reactive or proactive rules and creates a configuration plan. The block pushes information through 1.*c* and 1.*e* to Execution (E) block, which consists of two types of actuator *i.e.,* SaaS and IaaS. These actuators act as an API to execute the action at application and at infrastructure layer respectively. Therefore, After the configuration plan, if needed, SaaS controller triggers action through SaaS actuator.

Most of the popular cloud applications provides some extra features (*e.g.,* several product recommendation in an e-commerce application), which enhances user's quality of experience (QoE), but is not the core functionality of the service. These independent application components can be isolated to be activated/deactivated to provide different service levels to the end users. Different service levels can be also adopted

to other Cloud applications, such as: (i) 2D/3D interactive applications over network (*e.g.,* architectural features where the rendering could be customized); (ii) On-line itineraries on maps with different details (*e.g.,* points of interest), etc. For the sake of generality, we propose three user experience levels. Mode High refers to high user experience while Mode Medium and Mode Low indicate to medium and low user experience respectively (see Figure 2). When current application behavior deviates from target system state in terms of SLA, the auto-scaler gracefully downgrades the user experience from higher mode to lower mode and vice-versa through proper actuator value. Once SaaS actuator triggers the adaptation plan, it passes request for addition/removal of resources event as «RequestEvent» to IaaS controller if the former controller decides that application needs more/less resources, which is shown at Figure 1. Following the event, IaaS controller decides to take action via traditional infrastructure API (built in IaaS actuator) that is *scale-in* and *scale-out* or wait/discard the request issued by the SaaS controller. In addition, ①, ② and ③ depict the task flow of our auto-scaler. In summary, IaaS controller only gets activated if SaaS controller issues any «RequestEvent». Since the public IaaS provider's does not expose their resource allocation policies to the upper layers *i.e.,* PaaS or SaaS, we consider infrastructure as a black box. Therefore, our proposed IaaS controller are unaware of resource allocation strategy, for instance, what types of VM is to be added/removed or location of VM's in specific servers etc.

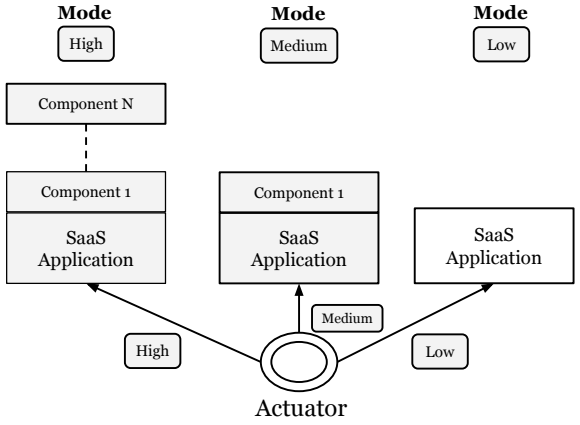


**Figure 2: Applications mode under different service level**

## 3 PROPOSED CONTROLLERS

This section describes several application controllers which have been extended to leverage the underlying elastic infrastructure and a generic infrastructure controller which can be plugged with any application controller to control the elasticity capability of cloud infrastructure.

### 3.1 Application/SaaS controllers

We have designed and validated several single and mutiple metric application controllers which have the capability to reconfigure SaaS application to keep it accessible and performant even at changing conditions at [13],[15]. In this article, we extend the Response time controller (performance aware) and Green Energy Controller (quality of resource aware) with increased capability to request of addition/removal of resources from the underlying elastic infrastructure.

***Response Time controller (RT-C).*** Response time is an essential metric to guarantee cloud based application's performance. Our goal is to keep response time under certain threshold dynamically to maximize the availability of the service in unpredictable and variable workload condition. We closed the managed software system by a feedback loop, where in each control period, the output is forwarded as a Map of response time and workload arrival rate to compare with the target set-point. Afterwards, the information is forwarded to compute a function:

$$f(t) = 1 - \tilde{\lambda}(t) * \tilde{r}(t) \tag{1}$$

where $\tilde{\lambda}(t) = \frac{\lambda(t-1)}{\lambda_{median}}$ and $\tilde{r}(t) = \frac{RT_{95}(t-1)}{RT_{setpoint}}$. Since, unpredictability and burstiness of user requests is common phenomena for Cloud application, we have considered workload arrival rate $\lambda(t)$ as a disturbance to the system. Ignoring the disturbance can lead to dramatic degradation of application performance. For capturing the change in the workload arrival rate, current workload arrival rate in the system is divided by median of previous arrival rates. A median filter is used with window size of four, that provides better estimation about variability of the workload arrival rate. Furthermore, the function $f(t)$ is computed at each $t$ time to analyze how far the multiplication ratio of workload change and response time increment/decrement is from 1. The idea is to keep the function greater than 0 to stabilize the system to operate under target response time. If the function is positive and above a desired/predefined threshold, the controller keeps the highest user experience mode *i.e.,* mode 2. Since the controller is not aware of how much amount of underlying resources are used, it notifies the *vmRemove* event to the IaaS controller (see line 13, Algorithm 1). In case, the condition block falls to $f(t) \leq 0$, a «RequestEvent» of *vmAdd* is notified to IaaS controller (see line 10, Algorithm 1).

***Green Energy Controller (GE-C).*** While *RT-C* is built to avoid performance degradation by keeping response time to a target set point, is not aware of when green energy production is scarce or abundant to actuate application's mode. On the other hand, devising adaptation plan only based on green energy availability can dissatisfy reasonable QoS while workload arrival is higher. Therefore, we intend to build a controller which can make adaptive decision based on the better quality of energy *i.e.,* green energy and application's performance.

**Algorithm 1:** Response time controller (RT-C)

**Input**: $Thr_{rt}, \lambda = [0 \quad 0 \quad 0 \quad 0], setPoint, app$
**Output**: updated $\lambda$, $Curr_{mode}$ = Current application mode.

1 **if** *(handleEvent == responseTime)* **then**
2 $\lambda(t-1) \leftarrow servedRequest$
3 $enqueue(\lambda)$
4 $f(t) \leftarrow 1 - (\lambda(t-1)/\lambda_{median}) * (RT_{95}/setPoint)$
5 **if** $(f(t) > 0) \quad \wedge \quad (function < Thr_{rt})$ **then**
6 $app.mode \leftarrow mode\ 1$
7 $RequestEvent \rightarrow vmAdd$ `/* VM Addition request event sent to IaaS controller along with` $RT_{95}$ `and workload-increment =` $(\lambda(t-1)/\lambda_{median})$ `*/`
8 **else if** $f(t) \leq 0$ **then**
9 $app.mode \leftarrow mode\ 0$
10 $RequestEvent \rightarrow vmAdd$ `/* VM Addition request event sent to IaaS controller along with` $RT_{95}$ `and workload-increment =` $(\lambda(t-1)/\lambda_{median})$ `*/`
11 **else**
12 $app.mode \leftarrow mode\ 2$
13 $RequestEvent \rightarrow vmRemove$ `/* VM Removal request event sent to IaaS controller along with` $RT_{95}$ `and workload-increment =` $(\lambda(t-1)/\lambda_{median})$ `*/`
14 $dequeue(\lambda)$
15 $Curr_{mode} = app.mode$
16 **return** $\lambda$, $Curr_{mode}$

We distinguish between two control periods: long and short. Algorithm 2 presents two «handleEvent» blocks, each associated with specific event *i.e.*, *greenEnergy* and *responseTime*. In *longer control period*, *greenEnergy* block decides application's mode based on green energy availability. Some sources of green energy are only available during certain times. For instance, solar energy is available during the day and the amount produced depends on the weather and the season [10]. Due to the intermittency, we have divided the total green energy production to three different regions *i.e.*, no green energy (at night), few (early morning and late afternoon) and adequate (mid-day). To distinguish between the regions we choose a static threshold $Thr_{max}$, above which the controller activates high user experience mode (mode 2). When green energy production falls between 0 and $Thr_{max}$, the controller chooses an actuator value that triggers the medium user experience mode (mode 1), and in case of current green energy amount is null, mode 0 is activated. In short, the controller activates higher or lower user experience mode based on the energy information pushed by infrastructure in longer control periods. In contrast, the *responseTime* block checks the response time periodically in *shorter control period* to identify overloaded condition in the system. If occurred, the controller downgrades the user experience to lower level. In summary, depending on the event, the specific block gets activated in Algorithm 2.

Afterwards, we try to investigate when performance indicator of an Application can trigger add/remove VM request. Since this controller have two feedback loops activating at two different control periods: long and short, and longer control period's decision depends only on the energy information, hence we focus on the shorter control period loop which is based on response time event. The shorter control loop periodically checks if the targeted response time is violated by application by computing a function at line 15 at Algorithm 2. If the computed function becomes negative (f(t) ≤ 0) meaning, if the current response time is beyond or borderline to set point and/or the tendency of the workload is increasing, the controller downgrades the user experience by subtracting 1 from previous control period's decision value and notify a `vmAdd` event request to the infrastructure controller (see line 18, Algorithm 2). While the function is greater than 0, which suggests that the application is performing well by keeping current 95th percentile response time to the set point, application keeps the user experience as before but notify a `vmRemove` event request to the infrastructure controller (see line 21, Algorithm 2). In both cases, «RequestEvent» notifies the specific event along with application's current 95th percentile response time and workload increment ratio to the IaaS controller.

**Algorithm 2:** Green Energy controller (GE-C)

**Input**: $Thr_{max}$ = Threshold for green energy, $\lambda = [0 \quad 0 \quad 0 \quad 0]$, $setPoint$, $Curr_{GE}$ = Current green energy production.
**Output**: updated $\lambda$, $Curr_{mode}$ = Current application mode.

1 */* Initiates in longer control period */*
2 **if** *(handleEvent == greenEnergy)* **then**
3 **if** $Curr_{GE} == 0$ **then**
4 $app.mode \leftarrow mode\ 0$
5 **else if** $Curr_{GE} > Thr_{max}$ **then**
6 $app.mode \leftarrow mode\ 2$
7 **else**
8 $app.mode \leftarrow mode\ 1$
9 $Curr_{mode} = app.mode$
10 **return** $Curr_{mode}$
11 */* Initiates in shorter control period */*
12 **if** *(handleEvent == responseTime)* **then**
13 $\lambda(t-1) \leftarrow servedRequest$
14 $enqueue(\lambda)$
15 $f(t) \leftarrow 1 - (\lambda(t-1)/\lambda_{median}) * (RT_{95}/setPoint)$
16 **if** $(f(t) \leq 0)$ *and* $(Curr_{mode} \neq 0)$ **then**
17 $app.mode \leftarrow Curr_{mode} - 1$
18 $RequestEvent \rightarrow vmAdd$ `/* VM Addition request event sent to IaaS controller along with` $RT_{95}$ `and workload-increment =` $(\lambda(t-1)/\lambda_{median})$ `*/`
19 **else if** $(f(t) > 0)$ **then**
20 $app.mode \leftarrow Curr_{mode}$
21 $RequestEvent \rightarrow vmRemove$ `/* VM Removal request event sent to IaaS controller along with` $RT_{95}$ `and workload-increment =` $(\lambda(t-1)/\lambda_{median})$ `*/`
22 **else**
23 $app.mode \leftarrow Curr_{mode}$
24 $Curr_{mode} = app.mode$
25 $dequeue(\lambda)$
26 **return** $\lambda$, $Curr_{mode}$

## 3.2 IaaS controller

While under-provisioning of resources can significantly hamper QoS properties by saturating application, over-provisioning of resources can increase energy consumption and other associated costs significantly. Therefore, the scaling decision, for instance, add resources (scale-out) or remove resources (scale-in) should be taken carefully to match with the applications resource demand. To meet *scale-out* condition, a reactive

policy can be easily designed and implemented based on the monitored performance metrics or by listening to predefined appropriate events. A reactive policy is referred to a runtime decision based on current demand and system state - to add resources on the fly. On the contrary, reactive policies can not absorb the non-negligible resource/instance initiation time. In our case, when application starts to face high response time, both the SaaS controllers have the capability to downgrade the user experience level and to invoke an implicit event (`vmAdd`) request to IaaS controller. Therefore, the sequential operation can trigger the application to run at lower mode until the instance is launched and activated. Afterwards, the application reverts back to higher mode if it meets the condition after operation.

In contrast, when *scale-in* event (*i.e.,* fewer resources are required by application) is invoked by SaaS controllers, terminating instance based on reactive policy can have detrimental impact on the system [18]. For example, when application performs better by staying just below or borderline to set point, triggering *scale-in* action can make an application suffering from high response time to saturation. One way to overcome the problem is to VM resizing, that is to reduce the number of cpu cores on the fly by doing fine-grained analysis of resource requirements rather than terminating an entire instance, but popular hypervisors like KVM, VMware, Hyper-V does not allow removing cpu cores of guest VMs at runtime [20]. Additionally, instance termination can cause a sharp rise in response time reaching beyond the set point if workload's behavior or tendency is not taken into consideration. Therefore, devising a plan when to execute *scale-in* event is critical. On the other hand, if the consecutive scaling actions are carried out too quickly without being able to observe the impact of scaling action to the application, undesirable effects such as over and under-provisioning can occur which can leads to performance degradation and/or wastage of energy consumption.

Hence, the idea is to built a generic IaaS controller which is characteristically agnostic to SaaS controllers behavior. Whenever, an implicit event invocation (`vmAdd, vmRemove`) arrives to the controller, it activates the proper module by matching to the event. Since, two non-concurrent events can be invoked by SaaS controllers, our proposed IaaS controller contains two modules to handle each of them. We define a length of period called *coolingLength*, which is composed of instance activation time and the time it requires to impact on the application. Therefore, after triggering any scaling decision, this time period is updated to prevent any scaling decision to be made in between. Hence, when `vmAdd` event arrives to the controller, the *handleEvent == vmAdd* module matches the condition of not being at *coolingPeriod* with an and operator to maximum number of VM's a provider can be assigned to[2]. If it adheres the condition, *scale-out* decision is triggered via IaaS actuator and current number of VM and next *coolingPeriod* is updated (see line 3-5 of Algorithm 3). Otherwise, the module ignores the notification. On the other hand, when `vmRemove` event is invoked by SaaS controller, if the *handleEvent == vmRemove* module is

**Algorithm 3:** Infrastructure controller

**Input**: $[minVm, maxVm]$ = Minimum and maximum number of VM's.
$[RT_{95}, workload_{inc}]$ = Response time and workload increment sent by SaaS controller.
$[rt_{thr}, decWorkPerc]$ = Two tunable parameters.
**Output**: *vmNumber*, *coolingPeriod*

```
1  if (handleEvent == vmAdd) then
2      if (currentTime ∉ coolingPeriod) ∧ (vmNumber < maxVm) then
3          triggerAction → "scale − out"   /* Passing API call through
           cloud infrastructure manager */
4          vmNumber+ = 1
5          coolingPeriod+ = coolingLength
6      else
7          vmNumber = this.vmNumber
8          coolingPeriod = this.coolingPeriod
9      vmNumber = update(vmNumber)
10     coolingPeriod = update(coolingPeriod)
11 return vmNumber, coolingPeriod
12 if (handleEvent == vmRemove) then
13     if (currentTime ∉ coolingPeriod) ∧ (rt_thr > RT_95) ∧ (vmNumber >
       minVm) ∧ ((workload_inc < decWorkPerc) ∨ (Curr_mode = 0)) then
14     triggerAction → "scale − in"   /* Passing API call through cloud
       infrastructure manager */
15     vmNumber− = 1
16     coolingPeriod+ = coolingLength
17     else
18         vmNumber = this.vmNumber
19         coolingPeriod = this.coolingPeriod
20     vmNumber = update(vmNumber)
21     coolingPeriod = update(coolingPeriod)
22 return vmNumber, coolingPeriod
```

not carefully designed, cloud application can face unstable phases *i.e.*, sharp rises of response time to saturate application. Therefore, only looking at *coolingPeriod* and minimum number of VM could be unwise and skeptical.

To overcome this situation, we introduce two key parameters which are tunable to identify when is the good time to release resources *i.e.*, perform scale-in action. The parameters are i) how far the current system's response time should be from set point? For example, x% less than target response time set point, which is denoted by $rt_{thr}$ at Algorithm 3. ii) how much workload rate should decrease from the current trend? For instance, y% decrease in user requests than previous intervals, denoted by *decWorkPerc*. Hence, when *handleEvent == vmRemove* arrives to the IaaS controller, the module checks the cooling period, minimum number of VM, current response time condition with an AND operator. Additionally we put an OR operator between workload decrease parameter and current mode of the application. The rational behind that, in the absence of green energy, *GE-C* controller keeps the application at minimum level. Although, workload may be consistent or increasing, if this application controller invokes `vmRemove` event that matches to be outside of *coolingPeriod*, greater than minimum number of VM and reduced response time than the threshold, it will meet the *scale-in* condition and IaaS controller will trigger the action to release resources. On the other hand, *RT-C* will keep application at the highest mode when resources are slightly to abundantly over-provisioned. Thus, application being at *mode* = 0 and decreasing workload by y% percentage

[2] Amazon EC2 permits maximum 20 on-demand instances per user.

can not happen concurrently if response time is x% less than response time set point for this type of SaaS controller. Apart from *GE-C*, any SaaS controller which invokes `vmRemove` event and satisfies all the conditions mentioned above other than application mode being at lowest, will trigger *scale-in* action by IaaS controller.

## 4 EVALUATION

In this section, we present the evaluation results of proposed SaaS and IaaS controllers and their impact on cloud based application in terms of response time, energy consumption and cost. The goal is to advocate the benefits and limitations of each controller while experimenting with real cloud application and real workload traces.

### 4.1 Experimental setup

***Infrastructure configuration.*** The experiments were conducted in Grid'5000 Lyon site, with 3 physical machines linked by a 10 Gbit/s Ethernet switch and connected to wattmeter. Each machine has two 2.3GHz Xeon processors (6 cores per CPU) and 16GB of RAM, running Linux 2.6. Openstack Liberty was used as platform, which requires one dedicated physical machine for the cloud controller management system. Consequently, the other physical machines were used as compute nodes to host VMs, which in turn, were pre-configured to run Ubuntu 12.04.

***Application Configuration.*** We experimented with RUBiS application [1], an eBay like auction site, which is assumed to be a representative of popular e-commerce application and hence interactive web application. In Brownout [17], authors provided a user-to-user recommendation engine that is not core functionality of the service but can enhance user experience. Along with that, we implemented a fairly simple item-to-item recommendation, to offer another level of user experience, which is showed at Listing 1.

**Listing 1: SQL statement for the recommender system.**

```
SELECT
    items1.id
FROM
    items AS items1.id
    JOIN comments AS c ON items1.id = c.item_id
    JOIN items AS i2 ON items1.category = i2.category
 WHERE
    i2.id = :current_item_id AND
    items1.nb_of_bids >= i2.item_id AND
    items1.id != :current_item_id
ORDER BY rating DESC
LIMIT 10;
```

The simple recommendation engine can be summarized as "Retrieve 10 products from same seller and same product category which have higher or same user bid count with higher customer rating". Although both the recommendation engines lack the sophistication and worldly complexities, they do serve as a reasonable example of providing user experience that a cloud application can isolate from core functionality of the service to activate or deactivate at runtime. The recommendation is added to the item visualisation page and to enable it, we defined a function that reads a file, where actuator value is updated in each control period and execute the associated modes for each user request. For instance, Mode 1 activates the codes of recommendation one, mode 2 activates both recommendations and mode 0 provides no recommendation. Furthermore, the application is deployed with all its tiers *i.e.*, web and database server inside a VM using a LEMP stack[3]. Each application VM and Load-balancer (LB) VM were configured with 4 cores of CPU and 8GB of memory similar to Large flavor VM. Since, we used 2 compute servers, we could use maximum of 6 VMs and minimum at 2 VMs[4].

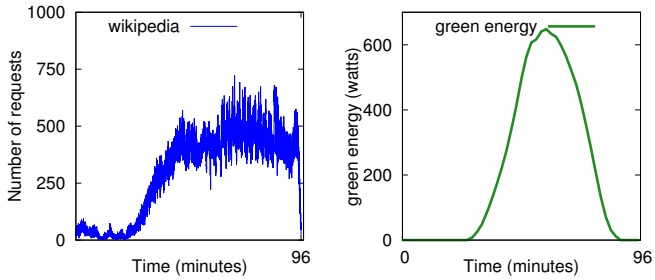


**Figure 3: Workload trace**

***Platform and Workload.*** Our Proposed platform solution is hosted inside the cloud controller machine. It monitors the 95th percentile response time by aggregating the Nginx log of LB in each control period *i.e.*, 20 seconds, whereas green energy information is pushed by the infrastructure through an API in every 60 seconds. We have set $rt_{thr}$=80% and *decWorkPerc*=20% based on our observation at IaaS controller to evaluate our proposal. We took the real traffic pattern of wikipedia german page of one day [6] and scaled the data set to fit with our experiment, which is showed at Figure 3. To generate the workload, we used Gatling as load injector and chose an open system model, where user requests are issued without waiting for other users response from the system. Furthermore, we emulated read-only workload where each user arrives to the homepage, browse any item category from a vast catalog, click on a product to extract its information, view seller rating and his/her reputation related to the product. We traced the solar energy production that was added to the grid for one day (12th April,2016) from EDF, France[5] and scaled the values suited for our experiment. Furthermore, the duration of each experiment was 96min and each was run several times. We considered 96min as 24 hours, i.e., each 4min in our experiments correspond to 1 hour.

### 4.2 Consideration of delaying event

From Section 3.1, we see that, *GE-C* controller has inner and outer loops («handleEvent» blocks) which are activated in different time-scales and push events to the controller to make decision. In our experiments, outer (longer control period) and

[3]https://lemp.io/
[4]1 LB VM and 1 application VM
[5]http://www.rte-france.com/fr/eco2mix/eco2mix

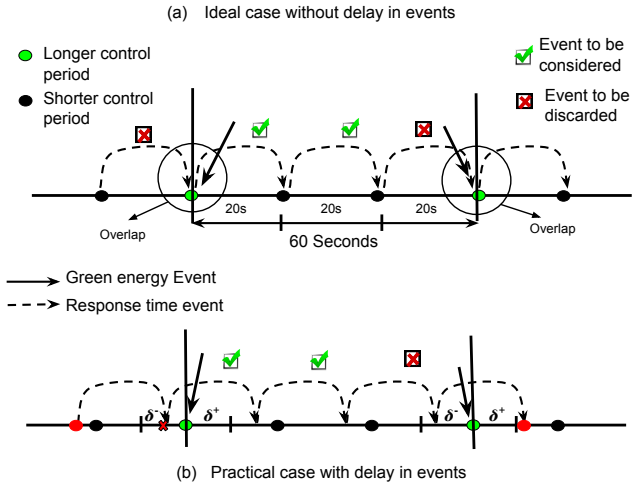


**Figure 4: Algorithm implementation in detail**

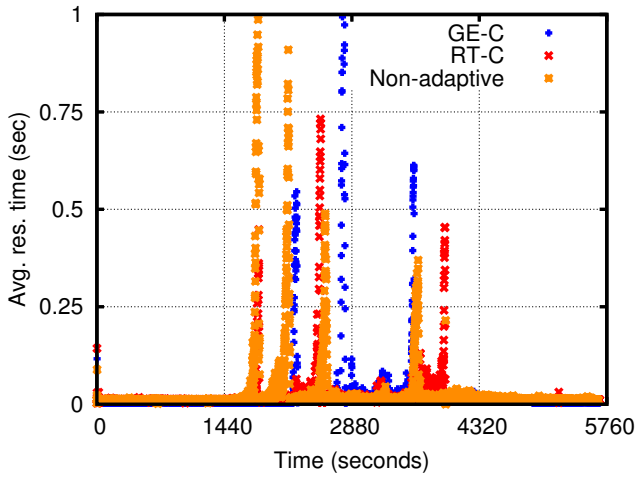


**Figure 5: Response time incurred by application controllers**

inner (shorter control period) events arrived at 60 seconds and 20 seconds respectively to the controller, which is showed in Figure 4. Ideally, if both kinds of events arrive without any delay, two different events will overlap each other at some point. As our motivation is to maximize of green energy usage for *GE-C* controller, we always make primary decision based on the green energy event pushed by IaaS by ignoring the response time event which is activated as inner loop, if both the event arrives concurrently. Concretely, it suggests that, between two big decision events in 60 seconds, we consider only two inner loop events and take actions if it is necessary indicated in Figure 4(a).

But in case of delaying of any event, the scenario will not follow Figure 4(a). As discussed before, the primary decision always depends on green energy event. Even though we receive response time event, no action is taken unless the system's response time is high. Therefore, in case of delaying of response time event by micro to milliseconds, effects to the system remain almost unchangeable. In contrast, if the event delays by couple of seconds, for instance, inner loop event arrives just before or after the primary decision is made, it might affect the system dynamics to achieve the goal. To tackle the problem, we define a safety distance, denoted by $\delta^t$ to ensure that the controller does not take any action if response time event arrives in between "*PrimaryDecision* - $\delta^t$" and "*PrimaryDecision* + $\delta^t$". Figure 4(b) illustrates the phenomena by an example. For our case, we choose safety distance as, $\delta^t$ = Time frequency of inner loop / 2, which is equal to 10 seconds in our experiments.

## 4.3 Results

This section elaborately presents the results obtained during experiment at Grid'5000. We consider a baseline approach *i.e.*, non-adaptive controller (NA), which lacks the capability of application adaptation and rely only on infrastructure adaptation based on response time.

***Response time.*** In Figure 5, we grouped response time by taking average over seconds. We kept response time set point as 1s. Since, our approach allows both level of adaptation depending on the changing environment, both RT-C and GE-C performed very closely by keeping 99th percentile response time around 274ms and 388ms. Figure 6 shows the distribution of response time for all the controllers. Although the baseline approach lacked application adaptation, it kept the 99th percentile response time around 500ms. On average, 3.7 million requests were injected during every experiment and only 7-20 requests failed for RT-C and 70-100 requests failed for GE-C. Therefore, both the controller ensure availability to five 9's (99.999%).

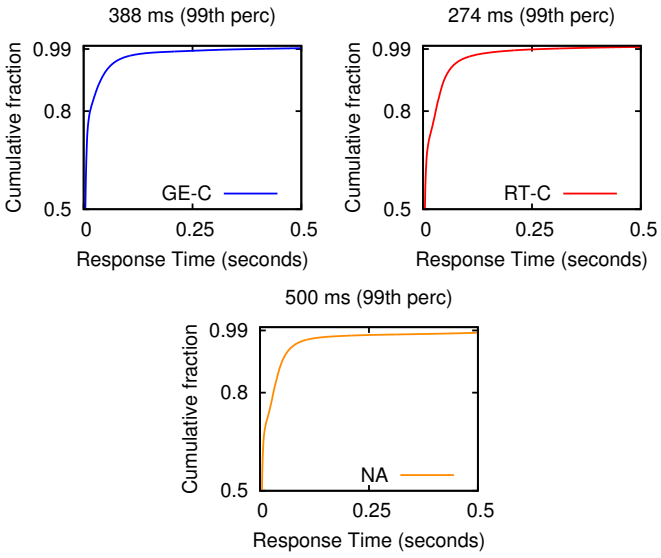


**Figure 6: Response time incurred by application controllers in percentiles**

***Energy Consumption.*** In our experiment, each 4 minutes were considered as an hour, thus we calculated the energy consumption of 24 hours, impacted by each controller, which is presented at Table 1 and 2. Each experiment was run several times and we found the energy consumption difference between each run was 1~2 watts. At first, we scaled down the wikipedia workload to test the application controllers with static infrastructure, that is, in any given time application controllers were unable to use underlying elastic infrastructure. Table 1 summarizes that, GE-C can reduce brown energy consumption by 9.02% and 10.88% compared to NA and RT-C, while total energy consumption was reduced by 6.53% and 6.4% respectively. NA was supposed to consume more energy than RT-C since it lacks the capability to adapt application to lower level. Due to the heavy user requests at some periods, NA approach saturated RUBiS application which was unable to accept any requests which resulted lower availability and lower energy consumption. Although, GE-C controller consumed 1.2% more green energy than NA, the amount was 0.50% less than RT-C controller, (See Table 1)

Afterwards, we proceeded with usual settings with the experiment having infrastructure elasticity capability for all the application controllers. Table 2 shows that, GE-C can reduce significant amount of brown energy by 35.11% and 26.65% compared to NA approach and RT-C respectively. Interestingly, GE-C consumed less green energy (6.83% less than NA, 5.53% less than RT-C) than the other application controllers, although this application controller was designed to consume more green energy than its counterpart application controllers. The reason being that, GE-C activated higher application mode only in the presence of green energy, irrespective to the amount of user requests. Therefore, if the user requests are lower/higher while few green energy is available the controller activates medium service level mode (*i.e.*, mode 1 which activates 1 type of recommendation) and starts activating higher service level (*i.e.*, mode 2 which activates 2 different kinds of recommendation) when the amount of green energy increases. On the other hand, when green energy is scarce, application works at a low service level mode, that is, user can access the system but no recommendation is provided. In this experiment, the user requests started growing when green energy were scarce. Therefore, NA and RT-C both activated more application VMs than GE-C to cope up with the workload, which can be seen at Figure 7. Moreover, Figure 7 shows that, GE-C followed green energy profile very closely while activating application VMs. It was possible due to fact that, we created green energy awareness to the application through *GPaaScaler*. GE-C and RT-C both used maximum of 4 VM's while NA approach used 5 VM's for same workload. The idea behind providing Table 1 and Table 2 is to show that, by exploiting elasticity capability at infrastructure level, application can reduce brown energy consumption even further.

***Cost analysis.*** Since, our proposed GPaaScaler architecture is agnostic to infrastructure provider and type, it can be plugged on top of public cloud infrastructure. Therefore, we wanted to validate how much each application controller would cost with same application and workload profile. Since, we used large flavor VM's, we fixed instance costs to 0.104$/hour. Moreover, we considered 4 minutes of duration to 1 hour in our experiment, we calculated the amount of instances and their duration throughout the experiment. We considered partial time spent in the experiment as full instance hour. Figure 8 shows the cost incurred by all the application controller. GE-C incurred 37.03% and 21.56% lesser cost than NA approach and RT-C.

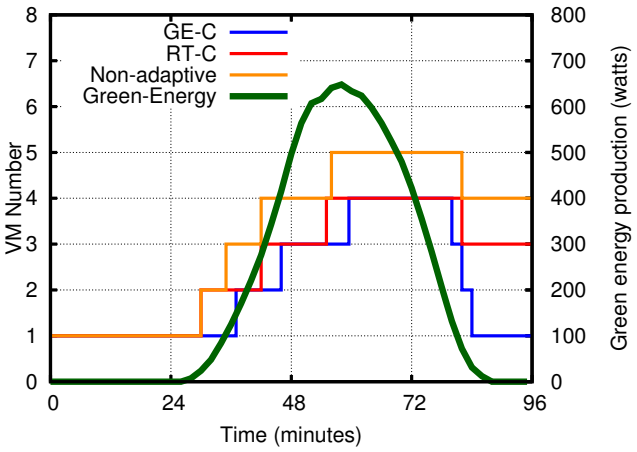


**Figure 7: Number of VM usage during experiment**

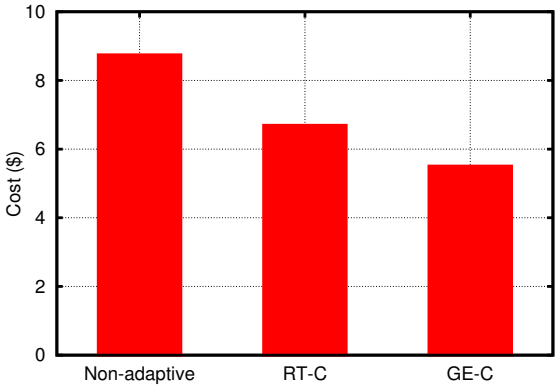


**Figure 8: Cost analysis for VM usage**

## 5 DISCUSSION

After extensively analyzing both the controllers that we have designed, it's evident that the GE-C can outperform RT-C and NA in terms of energy reduction and cost gains. In terms of performance, both the controller performs at the same level. We have investigated that, it takes approximately 5-6 seconds for the application to be stabilized with desired behavior after the application reconfiguration decision is triggered via SaaS actuator. That suggests, if we fix the control time very small,

**Table 1: Energy Consumption in Watts (Application adaptation)**

| Controller | Total Energy Consumption | Brown Energy Consumption | Green Energy Consumption |
|---|---|---|---|
| Non-Adaptive | 3424.20 | 1934.66 | 1484.54 |
| RT-C | 3493.93 | 1975.18 | 1518.75 |
| GE-C | 3270.20 | 1760.09 | 1510.11 |
| | (NA > 6.53%)(RT-C > 6.4%) | (NA > 9.02%) (RT-C > 10.88%) | (NA < 1.2%)( RT-C > 0.50%) |

**Table 2: Energy Consumption in Watts (Application and Infrastructure adaptation)**

| Controller | Total Energy Consumption | Brown Energy Consumption | Green Energy Consumption |
|---|---|---|---|
| Non-Adaptive | 5912.49 | 2516.59 | 3395.9 |
| RT-C | 5574.78 | 2226.16 | 3348.62 |
| GE-C | 4796.17 | 1632.77 | 3163.4 |
| | (NA > 18.88%)(RT-C > 13.96%) | (NA > 35.11%) (RT-C > 26.65%) | (NA > 6.83%)( RT-C > 5.53%) |

the controller can take decisions before the system has been impacted by last control decision. Therefore, the safety time is required for a system to be evolved to a desired state. Furthermore, activating control loop more frequently can increase the overhead of the system by accumulating information from system logs. In case of infrastructure, stabilization occurred after 15-20 seconds. Although GE-C can provide lesser recommendations/optional components which can be "nice-to-have" component, SaaS providers can take advantage of this controller to propose new class of SLA to eco-friendly customers who are willing to involve in reducing energy consumption. Moreover, GE-C controller can help SaaS provider to reduce costs by using fewer instances/hours from Public Cloud. Additionally, IaaS provider will be benefited for hosting these applications, since the applications will be able to consume energy smartly. In the Result Section, Table 2 showed that, Green energy consumption for GE-C was slightly lower than the other controller. In contrast, GE-C reduced significant amount of Brown energy, thus the *Green-to-Brown* energy ratio can be increased, as well as carbon footprint can be reduced. Although, with the change of workload and green energy profile, the values and numbers we have provided might change, but the tendency of the result will remain similar.

## 6 RELATED WORK

***Green Energy aware Application adaptation.*** Goiri et al. [9] proposed *GreenSlot*, a parallel batch job scheduler for a datacenter powered by an on-site renewable plant. Based on the historical data and weather forecast, GreenSlot predicts the amount of solar energy that will likely to be available in the future and subject to its predictions, it schedules the workload by the order of least slack time first (LSTF), to maximize the green energy consumption while meeting the job's deadlines by creating resource reservations into the future. Later, their work evolved into data processing framework by proposing *GreenHadoop* [10]. The idea relies on deferring background computations *e.g.,* data and log analysis, long simulations etc. that can be delayed by a bounded amount of time in a data center to take advantage of green energy availability while minimizing brown electricity cost. The idea is to run fewer servers when brown energy is cheap, and even fewer (if at all necessary) when brown energy is expensive. In conclusion, their proposal leads to operating few hadoop clusters when green energy is scarce. Similar to this work, authors [12] proposed *GreenPar*, a scheduler for parallel high-performance applications to maximize using green energy in a partially powered data center and reduce brown energy consumption, while respecting performance aware SLA. When green energy is available, GreenPar increases the resource allocations to active jobs to reduce runtimes by speeding up the processes while slow down the jobs to a maximum runtime slowdown percentage that is defined in SLA during the scarcity of the green energy. However, all these works focused specifically around batch like applications where job arrives with a deadline, hence can be deferred and scheduled by following the green energy availability. On the other hand, we propose to create green energy awareness around the interactive kind of application to to be self adapted with the presence/absence of green energy. Recently, Klein et al. [17] introduced Brownout paradigm for dynamic adaptation in interactive application through control theory to withstand in unpredictable runtime variations. Content reconfiguration takes into account only the system response time so that to prevent system instability in sudden workload burstiness. While the novelty of the approach is well understood, how the controller should be designed to take the advantage of the presence of green energy and implemented in massively virtualized and distributed cloud environment to exploit the elastic nature of infrastructure, has not been addressed.

To this, in [22], the authors proposed *GreenWare*, a middleware system that maximize the usage of green energy in geo-distributed cloud scale data centers by dynamically dispatching workload requests by following renewable, subject to energy budget constraint. The middleware performs three steps: computes the hourly energy budget and historical behavior of workload, runs an optimization algorithm based on

constrained optimization technique, lastly dispatches requests according to optimization plan. Similar to this, [7] proposed a flow optimization based framework for request-routing considering the trade-off between access latency, carbon footprint and electricity costs to upgrade the plan of choosing data center in specific intervals. Again these works focused more on load-balancing of user requests to different data centers to maximize the usage of green energy rather relying on application adaptation on the fly.

***Platform adaptation.*** Shi et al. [19] proposed two control layer, one responsible for allocating resources to VM's depending on the performance, the other one as a power saving layer to dynamically save energy by tuning voltage and frequency (DVFS) while resource requirement is low. Both layers are designed as autonomic loops in a coordinated manner to control cluster level resources. Later, Hankendi et al. [11] proposed an adaptive framework that jointly utilizes system (DVFS) and application-level adaptation to improve efficiency of multi-core servers and reduction of power consumption. Application-level adaptation has been utilized to meet the performance and accuracy constraints, whereas to meet power constraints, system-level management was adopted. On the other hand, we propose to adopt sequential execution of adaptation technique both at application and infrastructure level that not only met performance but also reduced energy consumption at significant portion.

In contrast to prevoius works, authors at [5] presented an Energy Adaptive Software Controller (*EASC*) to make task and service oriented application adaptive to renewable energy availability. The work was part of by DC4Cities project [6], which aimed at gathering renewable energy related information from energy providers and energy constraint directives from Energy monitoring authority (in context of Smart city) through an interface. Following the information, the PaaS layer is responsible to adapt the application by satisfying energy related constraints to consume more green energy, therefore building more eco-efficient policies for data center. The authors proposed to forward the energy related information to PaaS level via an API, so that an optimization plan can be invoked which involves desired working modes of an application considering energy and SLA constraints. However, service oriented application *i.e.*, interactive application is defined as running web, database and mail servers and higher mode depicts multiple data center site is active with full capacity while lowest mode indicates running a single site with minimum capacity. Apart from that, Moreno et al. [2] proposed how different non-conflicting tactics (*e.g.*, remove one software component and add one server) can be triggered simultaneously so that system can transition from current to desired state. The challenge is to estimate how two types of tactics when applied together reacts to the system. For instance, removing software component can have immediate transition, whereas adding one server can make a delayed transition which is also associated with cost. Depending on the goal, the utility function can be maximized by choosing proper adaptation tactics. Again these works ignored how to adapt and define tactics to leverage green energy availability to either consume more green energy or less brown energy.

[6] An European project on environmentally sustainable data centers for smart cities. Ended on 2016. http://www.dc4cities.eu

## 7 CONCLUSION

In this article, we proposed a green energy aware platform that creates awareness around interactive Cloud application and formulate strategy to understand when to trigger scaling decision based on reactive and pro-active scaling rules. Secondly, we use traditional API such as *scale-in* and *scale-out* to trigger decision based on the strategy we have devised. Later we validated our approach by extensive experiments and results obtained over Grid'5000 test bed. Results showed that, significant amount of brown energy and cost reduction is possible if application can be adapted based on green energy availability. For future work, we want to leverage micro-service architecture of an application to adapt itself in the presence of green energy. It would be interesting to deploy small and decoupled units of the application in a containerized approach (rather than VM) so that, each of the application component if required, can be scaled to guarantee better performance, self healing capabilities. Apart from that, resource can be assigned to specific components where it is required, thus over-provisioning of resource phenomena can be avoided resulting lesser energy consumption. Additionally, this investigation can leads to a decentralized autonomic behavior in modern application.

## 8 ACKNOWLEDGMENT

This work is supported by the EPOC project within the Labex CominLabs (http://www.epoc.cominlabs.ueb.eu/). Experiments presented in this paper were carried out using the Grid5000 testbed, supported by a scientific interest group hosted by INRIA and including CNRS, RENATER, and several Universities as well as other organizations (https://www.grid5000.fr).

## REFERENCES

[1] 2009. RUBiS, Rice University Bidding System. (2009). http://rubis.ow2.org/ (Date last accessed July-2016).

[2] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. 2016. Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation. In *IEEE International Conference on Autonomic Computing, ICAC*. 147–156. https://doi.org/10.1109/ICAC.2016.59

[3] Stefania Costache, Samuel Kortas, Christine Morin, and Nikos Parlavantzas. 2017. Market-based autonomous resource and application management in private clouds. *Journal of Parallel and Distributed Computing* 100 (2017), 85–102. https://doi.org/10.1016/j.jpdc.2016.10.003

[4] Stefania Costache, Nikos Parlavantzas, Christine Morin, and Samuel Kortas. 2011. An Economic Approach for Application QoS Management in Clouds. In *Euro-Par 2011: Parallel Processing Workshops Bordeaux, France, August 29 - September 2, 2011*. 426–435. https://doi.org/10.1007/978-3-642-29740-3_48

[5] Corentin Dupont, Mehdi Sheikhalishahi, Michele Facca Federico, and Fabien Hermenier. 2015. An energy aware application controller for optimizing renewable energy consumption in Cloud computing data centres. In *8th IEEE/ACM Int. Conf. on Utility and Cloud Computing*. Limassol, Cyprus.

[6] Soodeh Farokhi, Pooyan Jamshidi, Drazen Lucanin, and Ivona Brandic. 2015. Performance-Based Vertical Memory Elasticity. In *IEEE Int. Conf. on Autonomic Computing*. 151–152. https://doi.org/10.1109/ICAC.2015.51

[7] Peter Xiang Gao, Andrew R. Curtis, Bernard Wong, and Srinivasan Keshav. 2012. It's not easy being green. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer*

*communication (SIGCOMM '12)*. ACM, 12. https://doi.org/10.1145/2342356.2342398

[8] Íñigo Goiri, William Katsak, Kien Le, Thu D. Nguyen, and Ricardo Bianchini. 2013. Parasol and GreenSwitch: Managing Datacenters Powered by Renewable Energy. ACM.

[9] Íñigo Goiri, Kien Le, Md. E. Haque, Ryan Beauchea, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. 2011. GreenSlot: Scheduling Energy Consumption in Green Datacenters. In *Proc. of Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, Article 20, 11 pages.

[10] Íñigo Goiri, Kien Le, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. 2012. GreenHadoop: Leveraging Green Energy in Data-processing Frameworks. In *Proc. of the 7th ACM European Conf. on Comp. Syst.* ACM, 57–70.

[11] Can Hankendi, Ayse Kivilcim Coskun, and Henry Hoffmann. 2016. Adapt&Cap: Coordinating System- and Application-Level Adaptation for Power-Constrained Systems. *IEEE Design & Test* 33, 1 (2016), 68–76. https://doi.org/10.1109/MDAT.2015.2463275

[12] Md. E. Haque, Íñigo Goiri, Bianchini R., and Thu D. Nguyen. 2015. GreenPar: Scheduling Parallel High Performance Applications in Green Datacenters. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*. 217–227. https://doi.org/10.1145/2751205.2751221

[13] MD Sabbir Hasan, Yousri Kouki, Thomas Ledoux, and Jean-Louis Pazat. 2017. Exploiting Renewable sources : when Green SLA becomes a possible reality in Cloud computing. *IEEE Transactions on Cloud Computing* 5, 2 (April 2017), 1–1.

[14] Sabbir Hasan, Frederico Alvares, Thomas Ledoux, and Jean-Louis Pazat. 2016. Enabling Green Energy awareness in Interactive Cloud Application. In *Int. Conf. on Cloud Computing Technology and Science*. IEEE.

[15] Sabbir Hasan, Frederico Alvares, Thomas Ledoux, and Jean-Louis Pazat. 2017. Investigating Energy consumption and Performance trade-off for Interactive Cloud Application. *IEEE Transactions on Sustainable computing* 2, 2 (June 2017), 113–126.

[16] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (January 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[17] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. 2014. Brownout: Building More Robust Cloud Applications. In *Proc. of the 36th Int. Conf. on Software Engineering*. ACM. https://doi.org/10.1145/2568225.2568227

[18] Gabriele Mencagli, Marco Vanneschi, and Emanuele Vespa. 2014. A Cooperative Predictive Control Approach to Improve the Reconfiguration Stability of Adaptive Distributed Parallel Applications. *ACM Trans. Auton. Adapt. Syst.* 9, 1 (March 2014), 2:1–2:27. http://doi.acm.org/10.1145/2567929

[19] Xiaoyu Shi, Jin Dong, Seddik M. Djouadi, Yong Feng, Xiao Ma, and Yefu Wang. 2016. PAPMSC: Power-Aware Performance Management Approach for Virtualized Web Servers via Stochastic Control. *J. Grid Comput.* 14, 1 (2016), 171–191. https://doi.org/10.1007/s10723-015-9341-z

[20] Marian Turowski and Alexander Lenk. 2014. Vertical Scaling Capability of OpenStack - Survey of Guest Operating Systems, Hypervisors, and the Cloud Management Platform. In *Service-Oriented Computing - ICSOC 2014 Workshops - WESOA; SeMaPS, RMSOC, KASA, ISC, FOR-MOVES, CCSA and Satellite Events, Paris, France*. 351–362. https://doi.org/10.1007/978-3-319-22885-3_30

[21] Xiaorui Wang and Yefu Wang. 2011. Coordinating Power Control and Performance Management for Virtualized Server Clusters. *IEEE Trans. Parallel Distrib. Syst.* 22, 2 (2011), 245–259. https://doi.org/10.1109/TPDS.2010.91

[22] Yanwei Zhang, Yefu Wang, and Xiaorui Wang. 2011. GreenWare: Greening Cloud-Scale Data Centers to Maximize the Use of Renewable Energy.. In *ACM/IFIP/USENIX 12th International Middleware Conference (Lecture Notes in Computer Science)*. Springer, 143–164.