CloudGC: Recycling Idle Virtual Machines in the Cloud

Bo Zhang, Yahya Al-Dhuraibi, Romain Rouvoy, Fawaz Paraiso, Lionel Seinturier

CLOUDGC: Recycling Idle Virtual Machines in the Cloud

Bo Zhang[1,2], Yahya Al Dhuraibi[1,2,3], Romain Rouvoy[1,2,4], Fawaz Paraiso[1,2], Lionel Seinturier[1,2]
[1]Univ. Lille, France [2]Inria, France [3]Scalair, France [4]IUF, France
firstname.lastname@inria.fr

***Abstract*—Cloud computing conveys the image of a pool of unlimited virtual resources that can be quickly and easily provisioned to accommodate the user requirements. However, this flexibility may require to adjust physical resources at the infrastructure level to keep the pace of user requests. While elasticity can be considered as the *de facto* solution to support this issue, this elasticity can still be broken by budget requirements or physical limitations of a private cloud. In this paper, we therefore explore an alternative, yet complementary, solution to the problem of resource provisioning by adopting the principles of garbage collection in the context of cloud computing. In particular, our approach consists in detecting *idle virtual machines* to recycle their resources when the cloud infrastructure reaches its limits. We implement this approach, named CLOUDGC, as a new middleware service integrated within OPENSTACK and we demonstrate its capacity to stop the waste of cloud resources. CLOUDGC periodically recycles idle VM instances and automatically recovers them whenever needed. Thanks to CLOUDGC, cloud infrastructures can even switch between operational configurations depending on periods of activities.**

I. INTRODUCTION

Cloud computing keeps conveying the image of a pool of unlimited virtual resources that can be quickly and easily provisioned to accommodate the user requirements [1]. However, this flexibility usually comes at the cost of provisioning physical resources at the infrastructure level to keep the pace of user requests. In the specific case of *Infrastructure-as-a-Service* (IaaS), such physical resources refer to compute nodes, which are used to host *Virtual Machines* (VMs). While cloud elasticity [2]–[13] can be broadly considered as the *de facto* solution to cope with this problem, this elasticity can still be limited by budget requirements or physical constraints (*e.g.*, the lack of servers), in the case of a private cloud.

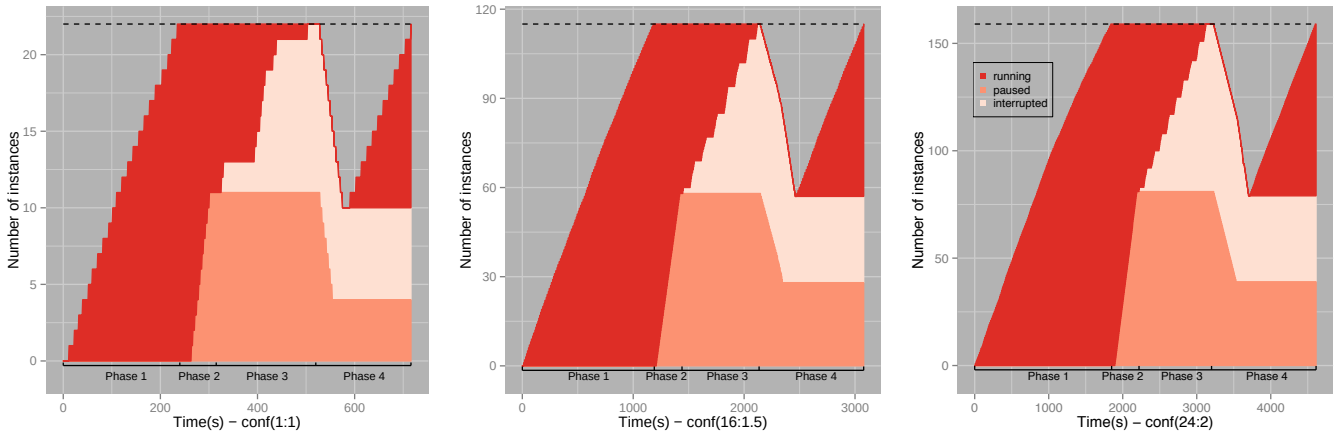
It is generally admitted that cloud data centers operate at very low rates [14], [15]. In particular, a recent study of Amazon's EC2 public cloud reports an average server utilization of 7.3% over a whole week [16]. Furthermore, a cloud provider like Scalair[1] observes that a significant portion of the hosted VMs (from 30 to 40%) are spending much of their time in an *idle* state (ranging from hours to days), which has also been recently assessed by IBM [17], [18]. However, the IaaS model builds on VM *flavours*, which plan and statically provision physical resources *a priori*, independently of the expected VM activity. Therefore, whenever a VM becomes inactive, the associated physical resources keep being provisioning and cannot be recycled for the purpose of another VM. Given the scale of IaaS, the black-box nature of VMs, and the unpredictability of cloud workloads, optimizing the physical resources thus becomes a critical issue for cloud providers, in order to stay competitive.

[1]http://scalair.fr

More generally, we believe that cloud elasticity should not be systematically adopted as the primary solution to scale the cloud according to user requirements. In particular, we claim that neglecting resource management in cloud computing can quickly lead to a waste of resources that can not only induce economic losses for the end-users, but also unnecessary carbon emissions for the cloud providers by over-provisioning the underlying infrastructure. Our objective therefore consists in studying the sources of such resource leaks in order to identify and to define new infrastructure services that can support the cloud by implementing smart resource management heuristics atop of existing services. The integration of these services aims at both improving *i)* the quality of experience for end-users and *ii)* the quality of service for cloud providers.

In this paper, we therefore explore an alternative, yet complementary, solution to the problem of resource provisioning by adopting the principles of garbage collection [19], [20] in the context of cloud computing. In particular, our approach consists in detecting *idle* VMs to dynamically recycle their resources when the cloud infrastructure reaches its limits. We implement this approach, named CLOUDGC, as a dedicated service integrated within OPENSTACK and we demonstrate its capacity to reduce the waste of compute resources in a shared physical infrastructure. CLOUDGC periodically recycles instances of idle VMs and automatically recovers them whenever needed. Thanks to CLOUDGC, cloud infrastructures can even switch between different provisioning configurations depending on periods of activities.

The remainder of the paper is organized as follows. Section II motivates the need of recycling CPU and memory resources in a cloud infrastructure. Section III introduces CLOUDGC, our garbage collector for the cloud and Section IV provides more details on its implementation. Section V reports on empirical experiments we performed to evaluate the benefits of CLOUDGC. Section VI discusses the related work in this area. Finally, Section VII concludes the paper and outlines future work.



(a) vCPU limitation of straight (b) vRAM limitation of standard (c) vRAM limitation of over-commit

Fig. 1. Observation of the IaaS limitations on the number of VM instances that can be provisioned.

II. The Sky Is Not The Limit

Cloud computing provides a model for enabling on-demand access to a shared pool of computational resources, which can be quickly provisioned and released upon needs. In the case of *Infrastructure-as-a-Service* (IaaS), these resources take the form of VMs, which can be created, suspended and deleted by the end-user at any time. Beyond the control of a VM lifecycle, some IaaS solutions, like OpenStack[2], can also control the CPU and memory consumption of VMs though the definition of VM profiles, also known as *flavors*[3] (*e.g.*, tiny, small). Furthermore, the number of VM *instances* can be constrained by the definition of *quotas*[4], which limits the amount of allocable resources for a given user. While quotas can be used to guarantee that an end-user will not allocate more VMs than allowed, the rest of this section will demonstrate that a IaaS may also suffer from internal constraints that limits its scalability, independently of user quotas.

To better understand such constraints, we run an experiment on a vanilla installation of an OpenStack IaaS infrastructure (version 2015.1.4 Kilo). The hardware setup we use is composed of 8 compute nodes, federating 22 CPU cores and 42.2 GB of memory. Each compute node runs Ubuntu (version 15.04) as the operating system, with QEMU (version 2.0.0) as the default hypervisor. Our motivation scenario consists in a synthetic workload that incrementally provisions new VMs for a single user whose quotas is not constrained. Such greedy scenario therefore starts by provisioning new tiny VM instances as long as OpenStack allows it (Phase 1). Once the maximum number of deployed VM instances is reached, our script switches half of the running instances to the *pause* state, before trying to provision some additional VM instances (Phase 2). Once this step is completed, we interrupt the other half of running VM instances and we try to provision some more VM instances, again (Phase 3). Finally, the scenario concludes by deleting all the suspended VM instances and allocates new VMs from there (Phase 4).

We report in Figure 1 on the results of executing this experiment for three configurations of OpenStack (straight (a), standard (b), and over-commit (c), whose details are reported in Table I). The straight (a) configuration reflects a bare-bone configuration that maps physical resources to virtual ones. One can witness that the IaaS saturates when the CPU limit is reached, as expected. The standard (b) configuration is the default configuration of OpenStack and maps 1 CPU core to 16 vCPUs and 1 GB of RAM to 1.5 GB of vRAM. In theory, using the standard configuration, the end-user could therefore expect to provision up to 352 tiny VM instances (as each VM requires 1 vCPU and 0.5 GB of vRAM). But, in practice, we observe that no more than 115 tiny VM instances can effectively be created by OpenStack, due to the limited resources available. Indeed, in the case of the standard configuration hosting exclusively tiny VMs, 352 ($\frac{352}{1}$) vCPUs can be allocated, but the vRAM can only support 118 instances ($\frac{63.3-8\times0.5}{0.5}$ as illustrated in Figure 1b). The maximum number of tiny VMs is therefore limited to 118 ($\min(\frac{352}{1}, \frac{63.3-8\times0.5}{0.5}) = 118$). The subtraction of 8×0.5 in vRAM is due to the virtual memory reserved by compute nodes—*i.e.*, we use 8 compute nodes and each node requires 0.5 GB vRAM for running OpenStack.

TABLE I
OverCommit ratios used as configurations.

configuration	mapping	vCPUs (total)	vRAM (total in GB)
straight	1:1	$22 \times 1 = 22$	$42.2 \times 1 = 42.2$
standard	16:1.5	$22 \times 16 = 352$	$42.2 \times 1.5 = 63.3$
over-commit	24:2	$22 \times 24 = 528$	$42.2 \times 2 = 84.4$

By increasing the ratio of vCPUs and vRAM in the over-commit (c) configuration, one can observe that the number of deployed VM instances can be raised up to 156 (cf. Figure 1c), but over-committing CPU and RAM resources

[2]https://openstack.org
[3]http://docs.openstack.org/openstack-ops/content/flavors.html
[4]http://docs.openstack.org/openstack-ops/content/projects_users.html

to increase the capacity comes at a cost for the cloud. In order to demonstrate the impact of resource overcommitment, we run another experiment, which logs the completion time of a benchmark running continuously in each of the VM instances we provision in our cloud infrastructure. In particular, we use the CPU test provided by the SYSBENCH benchmark[5]. Figure 2 reports on the evolution of this completion time per VM when provisioning from 1 to 10 new VM instances on a single compute node, using the standard configuration of OPENSTACK. One can observe that, once the number of allocated physical core is reached on a compute node (P(CPU)), the benchmark performance becomes linearly impacted by the provisioning of new VMs: the more VM instances, the more time it takes to complete the benchmark. Therefore, although it raises the limits of the cloud infrastructure capacity, resource overcommitment has to be wisely tuned by the cloud provider to limit the performance impact for the end-user.

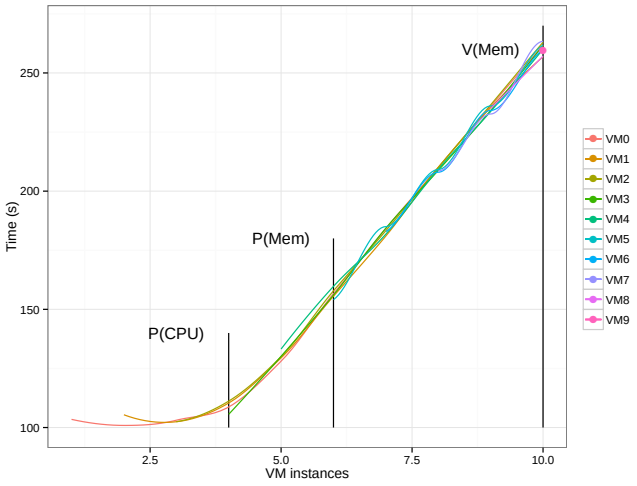


Fig. 2. Impact of resource overcommitment on VM performance.

As initially observed in Figure 1, no matter its level of activities, the only solution to release the resources provisioned by a VM instance consists in deleting these instances before provisioning new VMs. Indeed, neither interrupt nor pause operations allow OPENSTACK to provision additional VM instances. Therefore, by recycling the resources provisioned—but not used—by VM instances, one can expect to raise the capacity of a cloud infrastructure without systematically falling back on cloud elasticity techniques.

The challenge to be tackled in this context therefore consists in automatically recycling *idle* VM instances. The first obstacle is to automatically detect *idle* VM instances—*i.e.*, instances which are provisioned, but not actively used—in order to recycle them. We distinguish between two categories of *idle* VMs: instances which are *explicitly inactive* (*e.g.*, manually paused or interrupted) and instances which are *implicitly inactive* (*e.g.*, no CPU activity for the last 10 days). The second obstacle is to automatically recover the recycled VMs, once end-users require them again.

Based on this assumption, the remainder of this paper introduces our software solution to automatically recycle VM instances. Given that a cloud provider or a cloud administrator cannot manually manage a potentially huge population of VM instances, she requires a supporting service that can take care of resource management duties in order to ensure that resources are always made available to the end-users. Our approach can therefore be integrated upstream of an elasticity service, thus ensuring that these services are only activated when all the cloud resources are allocated and actively used.

III. THE CLOUDGC RECYCLING SERVICE

To seamlessly integrate our VM recycling mechanism with a Cloud infrastructure, we propose to design our solution as a middleware service that can interact with the existing services made available by a IaaS. Our middleware solution, named CLOUDGC, is therefore inspired by the garbage collection mechanism that is embedded in virtual machines like the *Java Virtual Machine* (JVM) and is used to reclaim memory occupied by objects that are no longer in use by the application. During the last three decades, garbage collection has focused a lot of research activities, moving from *"stop-the-world"* algorithms to generational approaches [20], [21]. This form of automatic memory management approach periodically scans the memory of the virtual machines and collects *garbage objects* to free the associate memory in order to facilitate the allocation of new objects.

In this paper, CLOUDGC therefore builds on the results achieved in garbage-collected languages in order to apply the principles of garbage collection on VM and at the scale of the Cloud. However, unlike objects in applications, *idle VM instances* are software artifacts that might still be used in the future. In this context, the challenges of developing a garbage collector for the cloud are threefold, including *i)* to detect idle VM instances, *ii)* to efficiently recycle the VM resources, and *iii)* to support the recovery of recycled VMs.

The following sections address each of these challenges more specifically, as well as their impact of the lifecycle of a VM.

A. Lifecycle of VM Instances

Figure 3 depicts the lifecycle of a VM instance in CLOUDGC. Beyond the standard lifecycle of VM instances, CLOUDGC includes a new state recycled, which corresponds to the transitory state that a VM takes when it is recycled by our middleware service. Whether a VM instance is in the running or suspended stage, CLOUDGC can therefore decide to recycle this VM instance, as we further explain below. A VM can leave this state, either by being manually resumed (in the case of a suspended VM), accessed on-demand (in the case of a running VM), or deleted manually.

The following sections introduce the steps implemented by CLOUDGC to move a VM from a running or suspended state to the recycled state, back and forth.

[5]http://manpages.ubuntu.com/manpages/xenial/man1/sysbench.1.html

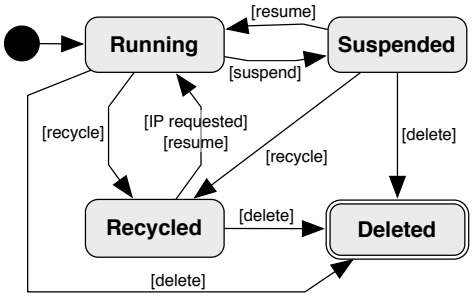


Fig. 3. Lifecycle of a VM instance in CLOUDGC.

B. Detecting Idle VM Instances

CLOUDGC builds on the assumption that not all the VM instances are continuously used in a Cloud infrastructure. Therefore, as part of the VM recycling process, CLOUDGC aims at detecting VM instances that are considered as *idle*. Idle VM instances are either VM instances that have been explicitly suspended by the end-user (*e.g.*, a VM in the paused or interrupted state) or VM instances that have not been active for a long period of time. CLOUDGC distinguishes between *explicit* and *implicit idle* VM instances: the former is not intended to be used by the end-user on a short-term basis, while the latter might be triggered at any time. Nevertheless, inspired by generational garbage collectors [22], we assume that the longer a VM has been flagged as inactive in the past, the longer it will still be in the future.

CLOUDGC therefore maintains two queues of VM instances: the *explicit queue* and the *implicit queue*. To detect and track idle VM instances, CLOUDGC periodically synchronizes the list of deployed VM instances from the IaaS instance manager. The list of suspended VM instances, ordered by interruption date (oldest first), is used to generate the explicit queue. Then, from the list of active VM instances, CLOUDGC queries the IaaS monitoring service to filter out the instances whose CPU activity has not exceeded a given threshold for a given duration (the activity threshold and the duration are two configuration parameters of CLOUDGC we use to tune the level of garbage collection). The items from the implicit queue that are not in this list are first removed, before inserting the items of the list that are not in the queue. The output of this first phase therefore delivers two lists of idle VM instances, ordered by inactivity durations.

C. Recycling Idle VM Instances

As previously mentioned, unlike objects in garbage collected languages, recycled VM instances may be recovered upon request. Therefore, recycling VM instances does not only consist in releasing the Cloud resources that are associated to each of the instances, but it also requires to save the current state of the instances in order to be able to recover them in a similar state, if necessary. In CLOUDGC, the state of idle VM instances is saved as a snapshot in the IaaS storage service. If a snapshot of this VM is already stored in the IaaS, it is automatically overridden by CLOUDGC if some activity has been detected since the last version. CLOUDGC automatically builds a snapshot of explicit idle instances when they are suspended. Given that the activity of implicit idle instances is not frozen, CLOUDGC can only build a snapshot of an implicit idle instance on-demand—*i.e.*, when the VM instances requires to be recycled.

When CLOUDGC is requested to recycle VM instances, it starts by recycling the explicit idle instances, before proceeding with the implicit idle instances, if needed. In both cases, CLOUDGC uses the IaaS instance manager to rebind the IP address of the idle VM instance to a *ghost instance*, which acts as a proxy to recover the VM upon request from a third party. Upon completion of the VM snapshot, the instance is deleted from the IaaS instance manager, thus effectively releasing the associated resources. While this process can be applied to recycle all the detected idle instances, CLOUDGC takes as input the amount of resources to be released, based on the number and the flavors of the new VM instances to be provisioned. Thus, CLOUDGC only recycles the necessary idle instances to allow the IaaS to provision the requested VM instances. If the recycling process fails to release the requested resources, the Cloud infrastructure can either reject the incoming provisioning request, or trigger an elasticity service to provision some additional compute nodes, thus increasing the capacity of the IaaS.

D. Recovering Recycled VM Instances

CLOUDGC recycles idle VM instances to ease the deployment of new VM instances. Nevertheless, recycled VM instances can be triggered at any time, *e.g.* by requesting a resource or a service of the ghost instance. In such a case, CLOUDGC should be able to recover the associated instance in the same state and configuration it was before being recycled, before forwarding the incoming request. As part of this recovery process, one can note that provisioning a recycled VM instance may require CLOUDGC to recycle idle VM instances. Therefore, the recovery process of CLOUDGC follows the same workflow as for provisioning a new VM instance, but loading automatically the snapshot from OPENSTACK Image Service (Glance) and restoring the initial VM configuration (*e.g.*, rebinding the floating IP address).

Both recycling idle instances and recovering recycled instances are not instant processes, taking from seconds to minutes depending on the amount of resources to be recycled and recovered. To prevent CLOUDGC from recycling VM instances that are considered as critical (*e.g.*, expected to react as quickly as possible to incoming requests), a VM can be *pinned* on the Cloud. Pinned VM instances are therefore made invisible from the detection and recycling processes, no matter their activity or their current state.

IV. IMPLEMENTATION DETAILS

This section dives into the details of the integration of CLOUDGC into OPENSTACK. We considered OPENSTACK as it is the *de facto* OSS standard for deploying a IaaS solution

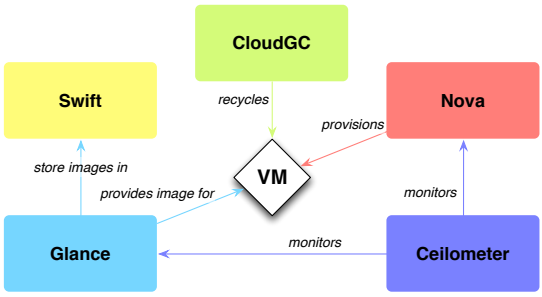


Fig. 4. Integration of CLOUDGC in OPENSTACK.

in a private Cloud, which is representative of the environments we target with CLOUDGC.

A. CloudGC Middleware Overview

Figure 4 depicts our integration of CLOUDGC in the OPENSTACK IaaS. Among all the services deployed by OPENSTACK[6], CLOUDGC interacts more specifically with Nova, Ceilometer, Glance, and Swift. CLOUDGC builds on the standard APIs provided by each of these services to support the VM recycling process. In particular, CLOUDGC uses Nova to recycle idle VM instances and to recover recycled VM instances (and their configuration), while Glance and Swift provide the necessary support to automatically save and restore the snapshots and configurations of recycled VMs, respectively. Finally, the monitoring capability of Ceilometer is used by CLOUDGC to analyze the activity of deployed VM instances.

Our solution is implemented in Python, thus benefiting from the client libraries made available for each of these services. By adopting this service-oriented architecture, CLOUDGC therefore integrates seamlessly with OPENSTACK and the implementation of the recycling process does not impact the API of existing services nor the GUI provided by Horizon, the administration console of OPENSTACK.

The architecture of CLOUDGC is structured in 3 components—Monitoring, Recycling and Recovery—which we detail in the following sections.

B. Monitoring Component

The monitoring component is an active component defined by CLOUDGC to periodically query Nova for the list of deployed VM instances and to update two shared priority queues—*i.e.*, the most idle VMs are enqueued first. Algorithm 1 summarizes the behavior that is periodically executed by this monitoring component (the period can be configured by CLOUDGC). As mentioned in Section III-B, CLOUDGC distinguishes between *explicit* and *implicit idle* VM instances in order to recycle explicit VM instances in priority. The second level of priority in CLOUDGC consists in recycling first the deployed VM instances that have been idle for a while, thus ordering the implicit queue by idleness. Finally, VM instances that are *pinned* are ignored by the monitoring component and therefore not considered as part of the recycling process.

[6]https://www.openstack.org/software

Algorithm 1 Monitoring behavior of CLOUDGC

```
1:  global ExplicitQueue
2:  global ImplicitQueue
3:  procedure MONITORING(duration)
4:      CLEAR(ExplicitQueue)
5:      vms ← LIST(Nova, UNPINNED)
6:      for vm ∈ FILTER(vms, PAUSED|INTERRUPTED)
    do
7:          INSERT(ExplicitQueue, vm)
8:          SNAPSHOT(Glance, vm)
9:      end for
10:     active ← FILTER(vms, RUNNING)
11:     for idle ∈ ImplicitQueue do
12:         if not CONTAINS(active, idle) then
13:             REMOVE(ImplicitQueue, idle)
14:         end if
15:     end for
16:     for vm ∈ running do
17:         if IDLE(Ceilometer, active) ⩾ duration then
18:             if not CONTAINS(ImplicitQueue, vm) then
19:                 INSERT(ImplicitQueue, vm)
20:             end if
21:         end if
22:     end for
23: end procedure
```

C. Recycling Component

The recycling component is a passive component introduced by CLOUDGC and triggered by Nova when it fails to satisfy an incoming provisioning request. In that case, Nova requests the recycling component to recycle some idle VM instances in order to free a sufficient volume of resources to satisfy the provisioning request. Algorithm 2 reports on the implementation of this component, illustrating the recycling priorities we introduced in CLOUDGC. If CLOUDGC succeeds to recycle a sufficient amount of resources, Nova can retry to provision the new VM instances. In case of failure, Nova can reject the request or trigger some horizontal elasticity support of OPENSTACK, which is out of the scope of this paper. To reduce the recycling delay, the STORE, REBIND, and DELETE operations save the instance configurations, rebind the VM instances on the ghost instance, and delete all the selected idle VMs at once. Although the operation SNAPSHOT appears in Algorithm 2, this operation is automatically called by the operation PAUSE as it is the case for explicit idle VM instances.

D. Recovery Component

The recovery component is in charge of handling incoming requests on recycled VM instances. To do so, CLOUDGC binds the floating IP of an idle VM instance to a ghost instance as part of the recycling process (cf. Algorithm 2), so that the recycled VM instances are still perceived as available from outside the IaaS. Therefore, upon receiving an incoming request, the ghost instance triggers the recovery function described in Algorithm 3 and then forwards the incoming request—if

Algorithm 2 Recycling behavior of CLOUDGC

```
1: global ExplicitQueue
2: global ImplicitQueue
3: function RECYCLE(volume)
4:     recycled ← ∅
5:     while AVAILABLE(Nova, recycled) < volume do
6:         if not EMPTY(ExplicitQueue) then
7:             vm ← GET(ExplicitQueue)
8:             ADD(recycled, vm)
9:         else if not EMPTY(ImplicitQueue) then
10:            vm ← GET(ImplicitQueue)
11:            ADD(recycled, vm)
12:            PAUSE(Nova, vm)
13:            SNAPSHOT(Glance, vm)   ▷ called by PAUSE
14:        else
15:            return FAILURE          ▷ lack of idle VMs
16:        end if
17:    end while
18:    STORE(Swift, recycled)
19:    REBIND(Recovery, recycled)
20:    DELETE(Nova, recycled)
21:    return SUCCESS                  ▷ idle VMs recycled
22: end function
```

the provisioning process succeeds—or returns an error to the end-user. Additionally, for the VM instances that need to run periodically, CLOUDGC proposes a timer, which acts as `crontab`, to request a recycled VM periodically. This solution ensures that the periodic VM instances are always being scheduled in the Cloud in order to complete their periodic jobs. As already mentioned, the provisioning process may in turn trigger the recycling process prior to provisioning the requested VM instance, thus introducing an unpredictable delay for processing the incoming request. While the cost of recovering a recycled VM instance is only paid upon the first incoming request, this weakness of CLOUDGC is further mitigated by the support for *pinned* VM instances, which can be kept active to deliver better response time when a VM instance is considered as critical for the end-user.

Algorithm 3 Recovering behavior of CLOUDGC

```
1: function RECOVER(id)
2:     image ← RETRIEVE(Glance, id)
3:     config ← RETRIEVE(Swift, id)
4:     vm ← PROVISION(Nova, image, config)
5:     if vm = NULL then
6:         return FAILURE   ▷ No more resource available
7:     else
8:         REBIND(Nova, vm)          ▷ Disabling the ghost
9:         DELETE(Glance, id)        ▷ Freeing the storage
10:        DELETE(Swift, id)
11:        return SUCCESS
12:    end if
13: end function
```

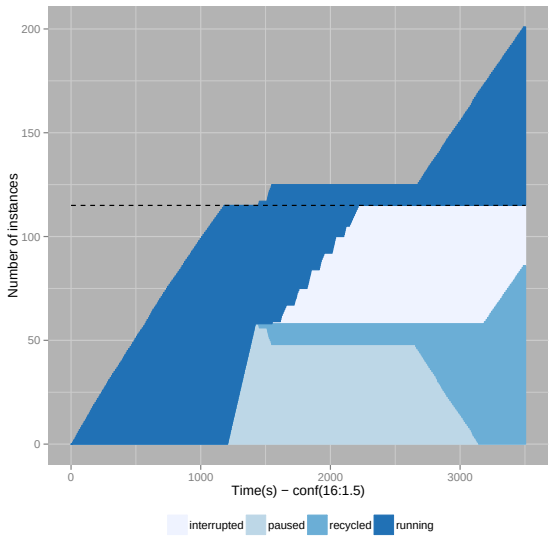


Fig. 5. Provisioning VM instances with CLOUDGC.

In the next section, we demonstrate how the combination of these three components in CLOUDGC performs for different scenarios we considered and we also evaluate the overhead introduced by this new middleware service of OPENSTACK.

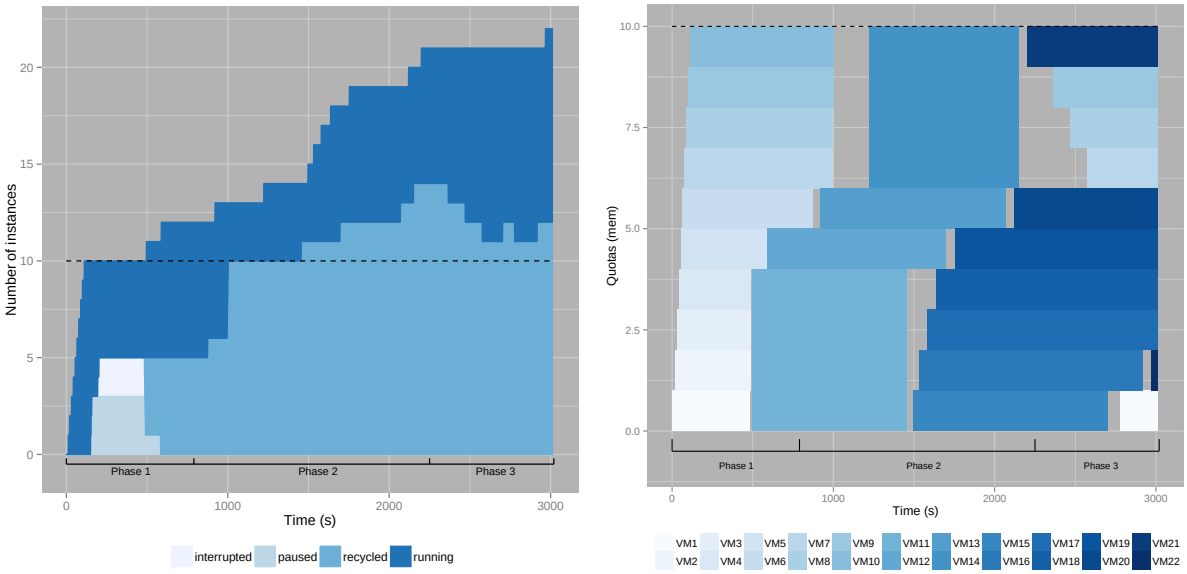
V. EMPIRICAL VALIDATION

This section assesses CLOUDGC with regards to the objectives we defined in Section II—*i.e.*, pushing the limits of a Cloud infrastructure to stop wasting resources by keeping provisioning new compute nodes when new VM instances needs to be deployed. We therefore report on various scenarios we considered to demonstrate the capability of CLOUDGC to better manage the resources of OPENSTACK. In this section, we use the same hardware infrastructure as in Section II and we configure OPENSTACK to run with the standard configuration. We configure CLOUDGC to consider as implicitly idle, the VMs whose activity has not exceeded 7% of CPU share for the past 10 minutes (cf. Section III-B).

A. The Sky Is The Limit

In this first experiment, we run a similar scenario to the one described in Section II—*i.e.*, we first saturated the Cloud infrastructure, then suspend some VM instances before trying to provision some additional instances. Figure 5 depicts the results CLOUDGC achieves on such a scenario. In particular, while the number of VM instances that can be provisioned in a vanilla OPENSTACK is limited, as emphasized in Figure 1b, CLOUDGC demonstrates its capacity to recycle the idle VM instances to accept the provisioning of new VM instances beyond the limits we previously observed. CLOUDGC recycles in priority the VM instances that are explicitly paused or interrupted in order to accommodate the incoming provisioning requests.

For the sake of readability, Figure 6 zooms on a single compute node of OPENSTACK to show how VM instances are



(a) Evolution of recycled VMs.

(b) Scheduling of VMs along time.

Fig. 6. Node-scale scheduling of VM instances using CLOUDGC.

recycled and scheduled by CLOUDGC along time. In particular, Figure 6a demonstrates that once the paused or interrupted VM instances are all recycled (cf. Phase 1), CLOUDGC then focuses on the running instances to identify those which are considered as *idle* (cf. Phase 2). Figure 6a depicts also in Phase 3 that the recycled VM instances can still be requested at any time, and one can observe that CLOUDGC succeeds in recovering these requested VMs. This recycling process keeps working as long as there are enough idle VM instances that can be recycled to satisfy a new provisioning request.

Beyond the results we already reported, Figure 6b demonstrates more specifically the capacity of CLOUDGC to deal with different VM flavors, for example by recycling 4 tiny instances (1 vCPU / 512 MB) to provision a small one (1 vCPU / 2 GB). In Figure 6, one can also observe that the provisioning delay of the two small instances differs. The first small instance is quickly provisioned because CLOUDGC recycles explicitly idle VM instances and does not have to snapshot their state (as the snapshot is operated upon the interruption of the VM instance). However, the provisioning of second small instance requires CLOUDGC to recycle implicitly idle VM instances, which induces an additional overhead for snapshotting the state of 4 tiny instances.

Figure 7 focuses on another OPENSTACK compute node and shows how CLOUDGC behaves when pinning a VM instance. In particular, one can assess that the pinned VM instance is not impacted by the recycling process of CLOUDGC, no matter its current state (running or suspended). CLOUDGC only recycles the VM instances that are considered as *recyclable*.

Among improvements, we are considering the integration of a VM consolidation phase in the recycling process of CLOUDGC. Given the overhead of VM consolidation, we plan to trigger

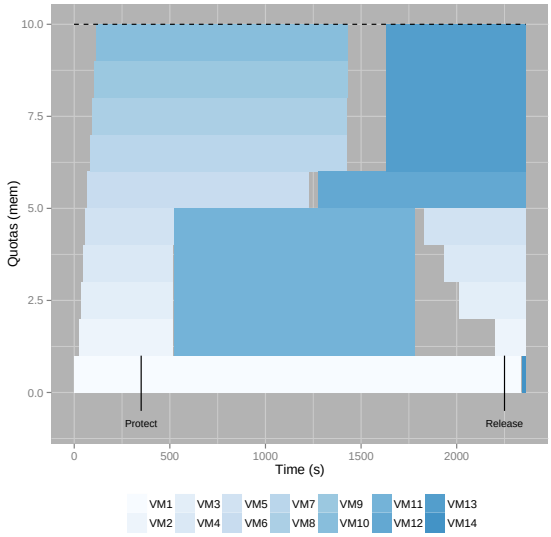


Fig. 7. Pinning VM instances with CLOUDGC.

such a phase only when CLOUDGC requires to compact the resources to ease the provisioning of larger VM instances. In such a case, the VM consolidation phase will aim at grouping *i)* pinned VM instances on a subset of compute nodes and *ii)* the resources made available by CLOUDGC on a single node. Among the possible solutions to implement this phase, we are considering the integration of CLOUDGC with the Watcher service[7], which has been recently released by OPENSTACK.

Alternative *idle* VM detection algorithms could be considered

[7]https://wiki.openstack.org/wiki/Watcher

within CLOUDGC to classify *idle* and *active* VMs, like supervised machine learning approaches based on SVM [17], [18]. However, by adjusting the duration and activity threshold parameters used by CLOUDGC to detect *idle* VMs, an OPENSTACK administrator can implement various VM management policies to urge the Cloud users to utilize the VMs that they provision.

B. CloudGC Performance Analysis

Regarding the delay introduced by the recycling process of CLOUDGC, we profiled the phases of CLOUDGC to identify how it performs depending on the different situations we considered in our scenario (detailed in Section II). Figure 8 therefore reports on the completion times achieved by CLOUDGC to provision new or recycled VM instances in our Cloud infrastructure. As long as enough resources are available in the Cloud infrastructure, one can observe that the monitoring component of CLOUDGC does not include any processing overhead for the system, thus performing equally to a standard configuration of OPENSTACK. When CLOUDGC recycles explicitly idle VM instances, one can assess that the processing overhead of CLOUDGC is rather low compared to a standard provisioning process, adding only 5 seconds to recycle an idle VM that has been explicitly suspended (cf. Table II).

TABLE II
PROCESSING OVERHEAD PER PHASE.

operation	available	explicit	implicit
browse list	-	2 *sec*	2 *sec*
create snapshot	-	-	215 *sec*
delete instance	-	3 *sec*	3 *sec*
create instance	6 *sec*	6 *sec*	6 *sec*
deploy OS	9 *sec*	9 *sec*	9 *sec*
total	15 *sec*	20 *sec*	235 *sec*

The biggest processing penalty introduced by CLOUDGC correspond to the recycling of implicitly idle VM instances that are in a running state. In this specific case, CLOUDGC requires to take a snapshot of the VM instance right before releasing the associated resources, which imposes to wait for the image to be safely persisted on the storage device before completing the provisioning process, and thus explaining the 215 seconds taken by Glance to complete this phase.

Regarding memory consumption, Figure 9 compares the memory consumption of OPENSTACK with and without CLOUDGC. On average, the difference between the two curves represents an overhead of 50 MB for the cloud controller node, on which CLOUDGC is deployed with the other infrastructure services. During the provisioning phases, which are reflected as peaks in Figure 9, one can observe that the memory overhead of CLOUDGC may reach up to 100 MB due to the additional activities performed as part of the recycling process.

Regarding the storage consumption, the storage capacity of Glance is impacted by CLOUDGC as it uses this service to store the snapshots of recycled VMs. The storage overhead of

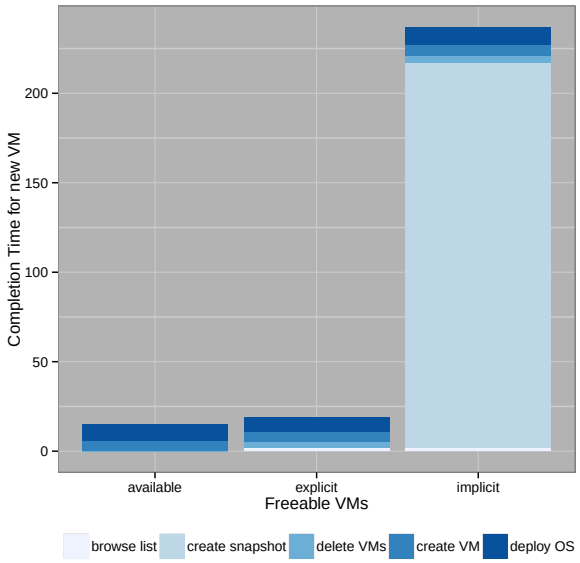


Fig. 8. Recycling delays introduced by CLOUDGC.

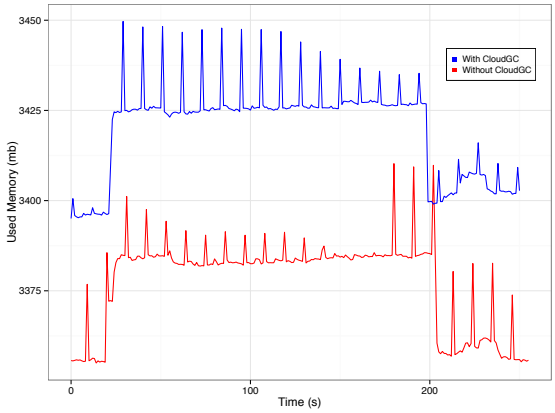


Fig. 9. Memory overhead introduced by CLOUDGC.

CLOUDGC therefore corresponds to the number of currently recycled VMs times the size of a VM, which highly depends on the activity of the Cloud. For example, Figure 5 provides an estimation of the volume of VMs recycled by CLOUDGC and thus subsequent snapshot images it has to store. CLOUDGC therefore trades CPU and memory resources against storage resources, but we assume that the resource limitations of a Cloud are stronger when it comes to CPU and memory resources.

With regards to current limitations, we are therefore exploring solutions to reduce the impact of on-demand snapshotting, which is the major bottleneck of CLOUDGC when recycling implicit idle VM instances. In particular, we are considering the support for incremental snapshots of *idle* VM instances to reduce both the processing and the network overhead imposed by the snapshot operations. By integrating such an incremental snapshot mechanism, CLOUDGC aims at

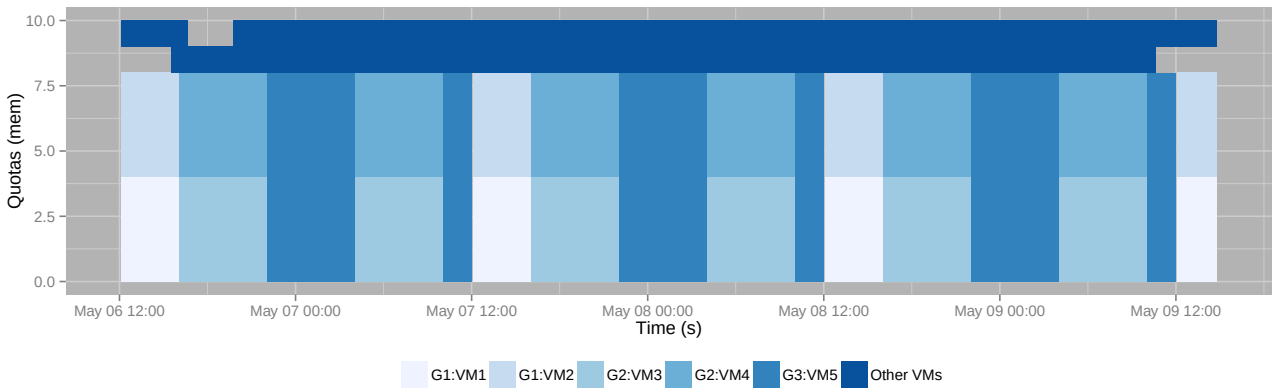


Fig. 10. Supporting periodic VM instances in OpenStack with CloudGC.

completing the snapshot associated to an idle VM instance whenever the monitoring component detects that its internal state has changed.

C. Orchestrating Periodic Usages

Thanks to CloudGC, a single physical infrastructure can be shared by several groups of VM instances that do not operate continuously. OpenStack can therefore periodically and automatically switch between VM instances along periods of variable durations in order to keep delivering the requested services, according to user requirements. For example, a Cloud infrastructure can host a group of services during office hours, then switch to another group of VMs during night before moving to a third profile along week-ends. While supporting this kind of scenario requires carefully handcrafted scripts to orchestrate the groups of VMs in OpenStack, CloudGC delivers this support natively, by exploiting the features we detailed in this paper.

For example, Figure 10 exposes a 4-day experiment we run in our OpenStack infrastructure running CloudGC. In this experiment, we provision 5 VMs, which are operating in 3 groups, next to other standard VM instances, which can be provisioned and used more randomly. The group 1 includes two small VM instances which are used from noon every day. The group 2 contains two other small VM instances, which need to be available twice a day at 4:00 and 16:00, respectively. Finally, the group 3 has a single medium VM instance, which is usually active from 22:00 to 10:00.

From Figure 10, we can observe that, beyond the traditional usage of a cloud infrastructure, CloudGC succeeds to schedule periodic VM instances according to their respective requirements. In order to guarantee the availability of periodic VM instances on time, we also benefit from the timer of CloudGC (cf. Section IV-D) to ensure that the VM instances are provisioned 5 minutes before the expected time. Furthermore, the use of the timer turns out to be also useful for autonomous VM instances, which do not need to be requested by an external user and control their own activity.

Our perspectives regarding this support for periodic VM instances refer to the automatic mining of VM activity patterns in order to configure the timer of CloudGC automatically. By adding such a capability, we believe that CloudGC can evolve to provide a new building block of a Cloud infrastructure saving energy by turning off the compute nodes hosting idle VM instances during periods of inactivity (*e.g.*, nights, week-ends, holidays). Furthermore, the combination of CloudGC with an elasticity service would enable compute nodes to be waken up automatically by the timer or by requesting one of the recycled service.

VI. Related Work

To the best of our knowledge, few approaches have been proposed so far on developing a garbage collector of idle virtual machines in the cloud. The most similar approaches to this work are the Netflix Janitor Monkey[8] and Heroku[9], which operates at the *Platform-as-a-Service* (PaaS) level. In Heroku, cloud services deployed with the free offer automatically fall asleep after 30 minutes of inactivity, and are resumed on-demand, inducing a similar delay to CloudGC for accessing the service again. Our approach generalizes this approach to the IaaS level, like Netflix Janitor, taking into account the constraints of VMs. Yet, at the IaaS level, several server consolidation and elasticity solutions have been developed by public providers and academia. In the remainder of this section, we therefore present and discuss the closest related works that are relevant to our approach.

a) Resource Management: Recycling resources is the process of collecting, processing, and reusing VMs. There are many proposed systems using resource management for various computing areas [23]–[27], but none of these systems focuses on the problem of recycling VMs in order to self-optimize the physical resource utilization. In particular, none of these systems is unable to detect idle VMs and trigger the recycling of VM instances like CloudGC does.

[8]https://github.com/Netflix/SimianArmy/wiki/Janitor-Home
[9]https://www.heroku.com

b) Horizontal Elasticity: Horizontal elasticity consists in adding/removing instances from the user virtual environment. Beernaert et al. [28] propose an horizontal elasticity solutions, Elastack, for cloud infrastructures (IaaS). Elastack can add or remove VM instances according to the workload demand. By monitoring all the hosted VM instances, Elastack can decide to add or remove additional instances from the IaaS. Kaleidoscope [29] and Amazon WS [30] belong to the same category of horizontal elasticity techniques. However, all of these approaches fail to cope with the IaaS limits we covered in this paper with CLOUDGC.

c) Vertical Elasticity: Vertical elasticity consists in adding or removing resources of individual VMs running the applications. OPENSTACK does not support vertical elasticity and, to the best of our knowledge, there is no approach covering such a support in OPENSTACK [31]. cloudScale [32] is an automatic elastic resource scaling system for multi-tenant infrastructures. It provides an online resource demand prediction and an efficient prediction error handling mechanism. It allows VM instances to scale vertically by adjusting the *Dynamic Voltage and Frequency Scaling* (DVFS) mechanism to save energy. In [33], a model-based approach to vertical scaling of vCPUs is proposed. This approach allows to allocate VM resources according to *Service Level Objectives* (SLO). Farokhi et al. [34] designed controllers that dynamically adjust the CPU and memory shares required to meet the performance objective of a given VM. Yet, this approach requires the hypervisor layer to be modified in order to support the dynamic reconfiguration of resources allocation, while CLOUDGC proposes a solution integrated within standard IaaS solution, like OPENSTACK.

d) Server Consolidation: Server consolidation targets a more efficient usage of resources in order to minimize the number of physical servers to operate. OPENSTACK controls instances placement on compute nodes via the component nova-scheduler. For example, Corradi et al. [35] propose a cloud management platform for OPENSTACK to optimize instances consolidation along three dimensions: power consumption, host resources, and networking. They demonstrate that VM consolidation is convenient to reduce power consumption, but can also lead to performance degradations. OPENSTACK Neat [36] is a framework for dynamic consolidation of VMs in OPENSTACK clouds. The objective of this framework is to optimize the power consumption, while avoiding performance degradations. The system detects when a compute host is underloaded, and migrates the hosted VMs before considering to switch the node to sleep mode in order to save energy. OASIS [15] also addresses the issue of idle VMs in Cloud data centers, but instead of recycling such VM instances, OASIS applies *partial VM migration*. Partial VM migration consists in migrating idle VM on a dedicated consolidation host (a low-power memory server) and then letting the migrated VM to fetch, on-demand, only the pages that are accessed.

However, these approaches—like many others [37]–[39]—focuses on minimizing the number of nodes required to host a set of VM instances, while CLOUDGC addresses the challenge of maximizing the number of VM instances to be hosted by a limited number of nodes.

e) Summary: As the rest of the state-of-the-art, CLOUDGC addresses the critical issue of resource optimization in cloud data centers, but it focuses on the detection and recycling of *idle* VM instances in a standard IaaS, instead of scaling up and down the number of required nodes to host the VMs. Yet, CLOUDGC nicely complements these existing techniques and can use, whenever available, *i)* vertical elasticity to replace its default snapshotting strategy, *ii)* horizontal elasticity to provision a new node when its recycling component fails to recycle any further idle VM, and/or *iii)* server consolidation to pack the recycled resources onto the same node. Beyond competing with each of these techniques, CLOUDGC rather paves the way for a win-win composition strategy.

VII. CONCLUSION & PERSPECTIVES

While Cloud computing conveys the image of unlimited computational resources that can be requested on-demand, the state-of-practice shows that a Cloud infrastructure, like an *Infrastructure-as-a-Service* (IaaS), may suffer from physical limitations to satisfy the user requests. In particular, beyond major cloud providers, who manage large datacenters, private clouds are being widely deployed and OSS solutions like OPENSTACK are providing the necessary services to deploy a IaaS atop of clusters of a smaller size. While cloud elasticity is now considered as the *de facto* solution to scale the cloud, we advocate that cloud elasticity should be carefully triggered as it may induce some non-negligible operational costs with budget and carbon emission implications.

In this paper, we therefore propose to identify potential resource leaks in a cloud infrastructure to recycle *idle* VM instances in order to provision new VMs upon requests. *Idle* VMs are instances that are either suspended or inactive for a while. Our middleware service, named CLOUDGC, detects automatically such *idle* VMs and recycles them to free the resources that are necessary to satisfy a VM provisioning request. CLOUDGC is integrated as an OPENSTACK service, thus not only providing a seamless support for recycling *idle* VMs, but also offering the support for the periodic deployment of VMs depending on user requirements. We demonstrate that the recycling overhead introduced by such a service is clearly mitigated by the capability to push the limits of the cloud infrastructure beyond standards.

Short-term perspectives on this work include the integration of additional services of OPENSTACK, such as Neutron and Cinder, in order to provide a full support for more complex VM configurations. In order to further improve the performance of our solution, we are considering extensions of CLOUDGC to include *i)* the automatic consolidation of VMs in order to optimize the utilization of cloud resources, *ii)* the support for incremental VM snapshots in order to reduce the latency of the recycling process, and *iii)* the mining of VM activities in order to automatically characterize and predict recurring activities of VM instances in a Cloud.

REFERENCES

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[2] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *Proceedings of the 10th International Conference on Autonomic Computing*, ser. ICAC'13. USENIX, 2013, pp. 23–27. [Online]. Available: https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst

[3] A. Al-Shishtawy and V. Vlassov, "ElastMan: Elasticity Manager for Elastic Key-value Stores in the Cloud," in *Proceedings of the ACM Cloud and Autonomic Computing Conference*, ser. CAC'13. ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2494621.2494630

[4] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Elastic Management of Cluster-based Services in the Cloud," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ser. ACDC'09. ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1555271.1555277

[5] S. Niu, J. Zhai, X. Ma, X. Tang, and W. Chen, "Cost-effective Cloud HPC Resource Provisioning by Building Semi-elastic Virtual Clusters," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC'13. ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503236

[6] S. Bouchenak, "Automated Control for SLA-aware Elastic Clouds," in *Proceedings of the 5th International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, ser. FeBiD'10. ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1791204.1791210

[7] P. C. Brebner, "Is Your Cloud Elastic Enough?: Performance Modelling the Elasticity of Infrastructure As a Service (IaaS) Cloud Applications," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE'12. ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2188286.2188334

[8] S. Das, D. Agrawal, and A. El Abbadi, "ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud," *ACM Trans. Database Syst.*, vol. 38, no. 1, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2445583.2445588

[9] P. Romano, "Elastic, Scalable and Self-tuning Data Replication in the cloud-TM Platform," in *Proceedings of the 1st European Workshop on Dependable Cloud Computing*, ser. EWDCC'12. ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2365316.2365321

[10] T. Alharkan and P. Martin, "IDSaaS: Intrusion Detection System as a Service in Public Clouds," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid'12, May 2012, pp. 686–687.

[11] L. Yu and D. Thain, "Resource Management for Elastic Cloud Workflows," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid'12, May 2012, pp. 775–780.

[12] X. Zhang, A. Kunjithapatham, S. Jeong, and S. Gibbs, "Towards an Elastic Application Model for Augmenting the Computing Capabilities of Mobile Devices with Cloud Computing," *Mob. Netw. Appl.*, vol. 16, no. 3, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1007/s11036-011-0305-7

[13] D. Chiu, A. Shetty, and G. Agrawal, "Elastic Cloud Caches for Accelerating Service-Oriented Computations," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'10. IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.21

[14] L. A. Barroso and U. Hölzle, "The Case for Energy-Proportional Computing," *IEEE Computer*, vol. 40, no. 12, 2007.

[15] J. Zhi, N. Bila, and E. de Lara, "Oasis: Energy Proportionality with Hybrid Server Consolidation," in *Proceedings of the 11th European Conference on Computer Systems*, ser. EuroSys'16. ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901333

[16] H. Liu, "A Measurement Study of Server Utilization in Public Clouds," in *DASC*. IEEE Computer Society, 2011.

[17] I. K. Kim, S. Zeng, C. Young, J. Hwang, and M. Humphrey, "A Supervised Learning Model for Identifying Inactive VMs in Private Cloud Data Centers," in *Proceedings of the Industrial Track of the 17th International Middleware Conference*, ser. Middleware Industry'16. ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/3007646.3007654

[18] ——, "iCSI: A Cloud Garbage VM Collector for Addressing Inactive VMs with Machine Learning," in *Proceedings of the International Conference on Cloud Engineering*, ser. IC2E'17. IEEE, 2017.

[19] H. Lieberman and C. Hewitt, "A Real-time Garbage Collector Based on the Lifetimes of Objects," *Commun. ACM*, vol. 26, no. 6, Jun. 1983. [Online]. Available: http://doi.acm.org/10.1145/358141.358147

[20] R. Jones and R. D. Lins, "Garbage collection: algorithms for automatic dynamic memory management," 1996.

[21] A. W. Appel, "Simple generational garbage collection and fast allocation," *Software: Practice and Experience*, vol. 19, no. 2, 1989.

[22] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," in *ACM Sigplan Notices*, vol. 19, no. 5. ACM, 1984.

[23] S. Pandey, L. Wu, S. M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Advanced information networking and applications (AINA), 2010 24th IEEE international conference on*. IEEE, 2010.

[24] R. Buyya, A. Beloglazov, and J. Abawajy, "Energy-efficient management of data center resources for cloud computing: a vision, architectural elements, and open challenges," *arXiv preprint arXiv:1006.0308*, 2010.

[25] S. He, L. Guo, M. Ghanem, and Y. Guo, "Improving resource utilisation in the cloud environment using multivariate probabilistic models," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012.

[26] D. M. Quan, R. Basmadjian, H. De Meer, R. Lent, T. Mahmoodi, D. Sannelli, F. Mezza, L. Telesca, and C. Dupont, "Energy efficient resource allocation strategy for cloud data centres," in *Computer and information sciences II*. Springer, 2011.

[27] D. Breitgand, Z. Dubitzky, A. Epstein, O. Feder, A. Glikson, I. Shapira, and G. Toffetti, "An Adaptive Utilisation Accelerator for Virtualized Environments," in *Proceedings of the International Conference on Cloud Engineering*, ser. IC2E'14. IEEE, 2014.

[28] L. Beernaert, M. Matos, R. Vilaça, and R. Oliveira, "Automatic Elasticity in OpenStack," in *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, ser. SDMCMM'12. ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2405186.2405188

[29] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara, "Kaleidoscope: Cloud Micro-elasticity via VM State Coloring," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys'11. ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966471

[30] Amazon WS, Website https://aws.amazon.com.

[31] M. Turowski and A. Lenk, "Vertical Scaling Capability of OpenStack," in *Service-Oriented Computing-ICSOC 2014 Workshops*. Springer, 2015.

[32] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC'11. ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038921

[33] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith, "Runtime Vertical Scaling of Virtualized Applications via Online Model Estimation," in *Proceedings of the IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems*, ser. SASO'14, Sept 2014, pp. 157–166.

[34] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth, "Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints," in *2015 International Conference on Cloud and Autonomic Computing (ICCAC)*, Sept 2015, pp. 69–80.

[35] A. Corradi, M. Fanelli, and L. Foschini, "VM consolidation: A real case based on OpenStack Cloud," *Future Generation Computer Systems*, vol. 32, 2014.

[36] A. Beloglazov and R. Buyya, "OpenStack neat: A framework for dynamic consolidation of virtual machines in OpenStack clouds–A blueprint," *Cloud Computing and Distributed Systems (CLOUDS) Laboratory*, 2012.

[37] H. J. Hong, D. Y. Chen, C. Y. Huang, K. T. Chen, and C. H. Hsu, "Placing Virtual Machines to Optimize Cloud Gaming Experience," *IEEE Transactions on Cloud Computing*, vol. 3, no. 1, pp. 42–53, Jan 2015.

[38] C. Isci, S. McIntosh, J. O. Kephart, R. Das, J. E. Hanson, S. Piper, R. R. Wolford, T. Brey, R. Kantner, A. Ng, J. Norris, A. Traore, and M. Frissora, "Agile, efficient virtualization power management with low-latency server power states," in *ISCA*. ACM, 2013.

[39] Z. Xiao, W. Song, and Q. Chen, "Dynamic Resource Allocation Using Virtual Machines for Cloud Computing Environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, 2013.