Automatic Memory-based Vertical Elasticity and Oversubscription on Cloud Platforms

Germán Moltó[a,*], Miguel Caballer[a], Carlos de Alfonso[a]

[a] *Instituto de Instrumentación para Imagen Molecular (I3M). Centro mixto CSIC Universitat Politècnica de València CIEMAT, camino de Vera s/n, 46022 Valencia, Espaa*

Abstract

Hypervisors and Operating Systems support vertical elasticity techniques such as memory ballooning to dynamically assign the memory of Virtual Machines (VMs). However, current Cloud Management Platforms (CMPs), such as OpenNebula or OpenStack, do not currently support dynamic vertical elasticity. This paper describes a system that integrates with the CMP to provide automatic vertical elasticity to adapt the memory size of the VMs to their current memory consumption, featuring live migration to prevent overload scenarios, without downtime for the VMs. This enables an enhanced VM per host consolidation ratio while maintaining the Quality of Service for VMs, since their memory is dynamically increased as necessary. The feasibility of the development is assessed via two case studies based on OpenNebula featuring i) horizontal and vertical elastic virtual clusters on a production Grid infrastructure and ii) elastic multi-tenant VMs that run Docker containers coupled with live migration techniques. The results show that memory oversubscription can be integrated on CMPs to deliver automatic memory management without severely impacting the performance of the VMs. This results in a memory management framework for on-premises Clouds that features live migration to safely enable transient oversubscription of physical resources in a CMP.

Keywords: Cloud computing, Cloud Management Platform, Virtualisation, Vertical Elasticity, Memory Overcommitment, Oversubscription

*Corresponding author
Email address: `gmolto@dsic.upv.es` (Germán Moltó)

1. Introduction

Elasticity [2], or the ability to rapidly provision and release resources, is one of the integral characteristics of Cloud Computing. Horizontal elasticity is commonly employed to provision additional computational nodes in order to sustain the quality of service delivered by an architecture deployed on a Cloud platform, specially after an increase in the number of users or workload. Horizontal elasticity has been extensively studied in the past, with services already available for public Clouds, such as Auto Scaling[1] for Amazon Web Services (AWS), and Heat[2] for OpenStack.

Instead, vertical elasticity enables to increase and decrease the number of resources allocated to a single Virtual Machine (VM). The increased support to techniques such as *memory ballooning* [3] and *CPU hot plugging* by popular hypervisors such as KVM, Xen or VMware paves the way for vertical elasticity to be adopted by Cloud platforms. However, popular open source CMPs such as OpenNebula and OpenStack do not currently support vertical elasticity without stopping the VMs. As an example, the KVM hypervisor fully supports memory ballooning in order to dynamically modify the allocated memory to a given VM without any downtime, and the main Operating Systems (OSs) support this feature. However, CMPs require to stop the VM in order to change its allocated memory.

In our previous work [4] we demonstrated the benefits of introducing vertical elasticity to dynamically adjust the allocated memory of VMs to their current memory consumption, specially for applications with dynamic memory requirements during their execution. In fact, the number of VMs that one physical machine can support is typically limited by its memory size. Besides, users tend to overestimate the amount of memory required by their applications resulting in unused memory that could be dedicated to additional VMs running on the same physical machine [5]. In addition, CMPs typically provide templates, such as the *flavors* in OpenStack, which enforce a certain amount of memory size regardless of the actual memory requirements of the application. Just as airlines sell more tickets than available seats (i.e. oversubscribe the plane) in the hope that some passengers do not show up, Cloud providers can oversubscribe their resources by

[1]Auto Scaling: `http://aws.amazon.com/autoscaling`

[2]Heat: `https://wiki.openstack.org/wiki/Heat`

deploying additional VMs in a host, in the hope that VMs will actually use less memory than initially requested.

However, this situation might incur in memory overload for a host, where the sum of used memory of its VMs exceeds the physical memory of the host. Therefore, oversubscription [6] is a technique that can lead to an increase in the number of VMs per physical host though it can have an impact on the Quality of Service and probably violate the Service Level Agreement established by the Cloud provider. However, oversubscription can enable Cloud providers to better use the available memory in their physical systems if the appropriate countermeasures are introduced. As Williams et al. [7] state, in well-provisioned datacenters, overload is unpredictable, relatively rare, uncorrelated, and transient, indicating that an opportunity exists for memory oversubscription in those facilities.

In this paper we introduce CloudVAMP (*Cloud Virtual machine Automatic Memory Procurement*) a memory oversubscription framework that can be integrated in an on-premises CMP to automatically monitor the VMs and to dynamically adjust their allocated memory to adapt to the current memory requirements of their running applications. Without any user intervention, the system automatically manages the memory of the VMs (or a subset of VMs) irrespective of the memory initially allocated by the user. This introduces enhanced VM consolidation per physical node while live migration is employed to prevent overload of the physical machines.

The remainder of the paper is structured as follows. First, section 2, describes the related works in the area of vertical elasticity and memory oversubscription. Next, section 3 briefly describes the problem addressed and the underlying technologies employed. Later, section 4 describes the architecture of CloudVAMP, in order to manage vertical elasticity in an on-premises Cloud. Then, section 5 describes two case studies carried out to assess the behaviour and benefits of the developed platform. Finally, section 6 summarises the paper and points to future work.

2. Related work

There can be found other works in the literature that have focused on vertical elasticity and memory oversubscription (also called in the literature memory overcommitment), though most of them are just focused on virtualisation platforms and, thus, not covering the intricacies of CMPs. In [8], the authors propose an Elastic VM architecture that scales the number of cores, CPU capacity and memory using the Xen hypervisor. They study the adaptation of the VM capacities to the requirements of a web application. However, their case study does not address memory scaling but only increasing the virtual CPU allocation.

In [9], a system to provide proactive dynamic memory allocation based on the Bayesian predictions is introduced to increase server consolidation. In [10], the Ginkgo memory overcommitting framework is introduced, which dynamically estimates VM memory requirements for applications and automates the distribution of memory across VMs through ballooning techniques. It uses performance profiles of the applications to characterise incoming load. The case study focuses on VMs running on a single physical host. These two works focus on a set of virtual machines running in a single hypervisor, while our work focuses at the whole infrastructure provided by the CMP, involving memory management across multiple physical hosts. In [11] an extension of ballooning techniques is applied to applications, using as example a database engine and the Java runtime, to reallocate memory between memory managers of different applications. However, these requires modifications of the Xen Balloon Driver and does not address the overcommitment problems that arise in CMPs.

Overdriver [7] is a system to mitigate the problems that arise in oversubscribed virtualised hosts, by automatically deciding when to use network memory, using a cooperative swap approach, or live migration depending on whether the workload is considered to be transient or sustained, respectively. However, they do not consider memory ballooning as a mitigation strategy for oversubscription. This is the case of the work by Hwang et al. [12] where a system to opportunistically use memory during periods of light loads is introduced. For that, they allow the hypervisor to dynamically allocate memory at fine granularity, focusing on disk and application level caches. The work by Baset et al [13] describes the different techniques employed to alleviate oversubscription and mitigate overload. They designed an event-driven simulator to develop an understanding of oversubscription. However, they focus exclusively on offline and live migration but ballooning techniques are discarded.

Regarding memory ballooning, the KVM hypervisor has a project called Automatic Ballooning [14] where the management of the balloon is automatic. When the host is under pressure, it asks guests to relinquish memory. When a guest detects memory pressure, it gets some memory back from the host. This requires Linux kernel 3.10+ and a specific version of QEMU. However, this approach focuses exclusively on the VMs running on a single physical machine and, thus, it does not solve the problems that arise when the host is overloaded, specially within an on-premises Cloud, where VMs could be live migrated across other physical hosts to restore the level of service.

The most similar work to our proposal is the one carried out by Litke [15], where the Memory Overcommitment Manager (MOM) is introduced. This system requires a daemon to be installed in the VMs to gather information regarding the memory usage from the VMs and a policy actuator that runs on the host's OS to decide when to increase or decrease memory though memory ballooning techniques. While this approach is of interest for a virtualisation platform where VMs have dynamic memory requirements, it does not introduce countermeasures for overloaded hosts.

As the authors of [6] state, much of the research conducted thus far has focused on managing oversubscription of a single physical machine, though this narrow focus is rather limiting. While other projects successfully manage memory overcommitment at a host level, we have not found any previous work that automatically manages oversubscription in an on-premises Cloud. Therefore, building on previous works in the area we introduce CloudVAMP, a memory management framework for on-premises Clouds that features live migration to safely enable transient oversubscription of physical resources in a CMP.

As opposed to previous work, our approach considers memory management not at a single physical host but at the whole infrastructure level in an on-premises Cloud. In addition, CloudVAMP is responsible to safely reduce the allocated memory to the VMs in order to enable transient oversubscription of the memory of the physical hosts. The fact that CloudVAMP is integrated with a CMP enables the latter to deploy additional VMs to the same physical host according to the stablished scheduling policies within the Cloud infrastructure. Therefore, CloudVAMP not only manages but also enables oversubscription at the Cloud infrastructure level, which is a feature not included in

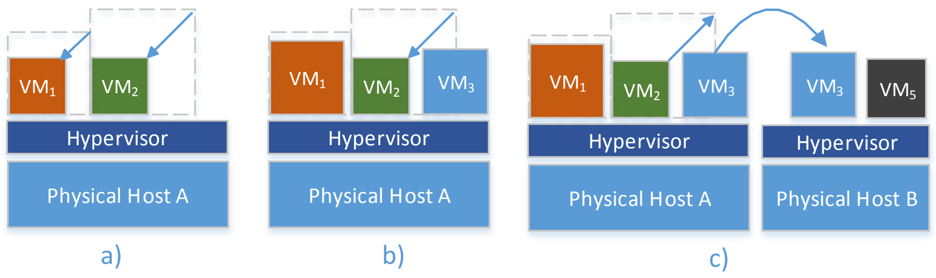


Figure 1: Depiction of an on-premises Cloud with support for dynamic memory management. a) the allocated memory of the VMs has been reduced because there is enough free memory, b) a third VM is deployed on the same physical host and c) live migration is employed to prevent memory overload in the physical host.

previous aforementioned related works.

In addition, we introduce a proof-of-concept open source implementation based on OpenNebula, which can be easily adapted to other CMPs (such as OpenStack). Therefore, this introduces unattended efficient memory management for on-premises Clouds.

3. Problem, Methods & Materials

This paper is based on the following underlying technologies. First, KVM [16], a popular open source hypervisor that fully supports memory ballooning. Second, OpenNebula [1], an open-source Cloud Management Platform that manages the life cycle of VMs on a physical infrastructure.

According to [13], there are different mechanisms to mitigate the problems that arise with oversubscription: i) stealing, which allows a hypervisor to steal (actually borrow) resources from underloaded VMs running on the same physical host; ii) quiescing VMs, so that a VM is shut down and migrated offline to an underloaded physical machine; iii) live migration, to hot migrate VMs from an overloaded physical machine to an underloaded one; iv) streaming disks, to transfer the minimum portion of a VM's local disk to allow the VM to be started on another physical machine, and v) network memory, to use memory of another machine as a swap space over the network.

In this paper we focus both on memory ballooning and live migration techniques together with its integration in a CMP. We rely on these techniques because they are fully supported on most hypervisors and by the main OSs (including Linux and Windows). Therefore, this enables to create a system that can be easily integrated in current on-premises Cloud deployments to seamlessly leverage these techniques.

Figure 1 summarises the main problem that aims to be addressed. In *a*), two VMs (VM_1 and VM_2) have been deployed by a CMP on the same physical host (A). Depending on the scheduling configuration of the CMP this situation can be very frequent. For example, OpenNebula can be configured to use a packing scheduler and so, the VMs tend to be allocated to the same physical machine if there is enough memory available. In KVM, a deployed VM has both a *memorysize* and a *maxmemorysize* attribute. A VM cannot grow beyond the *maxmemorysize*, which corresponds to the memory initially allocated when the VM was created. However, its *memorysize* (the memory currently allocated to the VM) can range from the minimum amount of memory to support the OS, typically in the order of 200-300 MB for a Linux VM [4], to its *maxmemorysize*. Notice that in *a*) the memory size of both VMs has been shrunk due via memory ballooning, because the applications running on the VM were not using that amount of memory. Then, in *b*) since there is enough available memory to host an additional VM (because the used memory by VM_1 and VM_2 is less than the original amount requested), the CMP's scheduler has decided to allocate a new VM to that physical host.

Later, in *c*), VM_2 requires more memory because the application (or applications) running inside has requested so and, thus, the physical host might become overloaded. Therefore, one or more VMs (in this case, only VM_3) have to be relocated to another physical host to maintain the quality of service across the infrastructure managed by the CMP. In our case, this involves live migration, according to a certain policy, so that no downtime is introduced for the migrated VM.

4. Architecture

The architecture of CloudVAMP consists of three components:

- Cloud Vertical Elasticity Manager (CVEM). An agent that analyses the amount of memory actually needed by the VMs and dynamically updates the memory

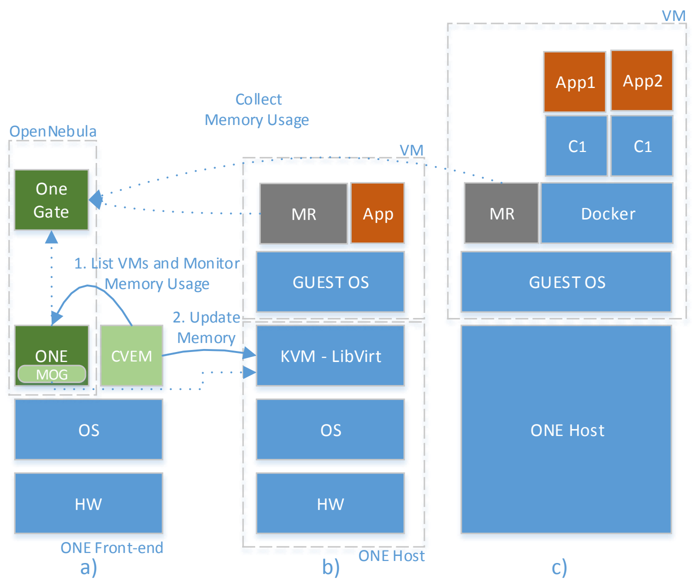


Figure 2: An on-premises Cloud with CloudVAMP. a) the OpenNebula (ONE) frontend host, b) a ONE host that executes VMs and c) configuration employed for the second case study in this paper.

allocated to each of them, according to a set of customisable rules. It is an agent that queries the monitoring system of the CMP, and has access to the hypervisors (e.g. ssh access to the physical nodes of the on-premises Cloud). It can decide to live migrate VMs in order to restore the level of service under memory overload situations.

- Memory Reporter (MR). An agent that runs in the VMs and reports to a monitoring system the free, used memory and usage of the swap space, by the applications in the VM. This information must be available for CVEM, so it should be integrated within the CMP's monitoring system (as it has been currently implemented) or by relying on a third-party monitoring system (e.g. Ganglia).

- Memory Oversubscription Granter (MOG). A system that informs the CMP about the amount of memory that can be oversubscribed on the hosts, to be taken into account by the scheduler of the CMP.

Figure 2 depicts the architecture of the proposed system and how it fits in an on-premises Cloud. The proof-of-concept implementation is based on OpenNebula (ONE) and it is seamlessly integrated using the components that it offers. OpenNebula requires a cluster-based installation in which the main services are installed in the front-end node (ONE Front-end in Figure 2.a) whereas the VMs are deployed on the internal working nodes (ONE Host in Figure 2.b), where the KVM hypervisor (other hypervisors are supported as well) has to be installed.

The architecture of CloudVAMP has been implemented via lightweight Python-based agents. For example, CVEM runs alongside ONE to obtain the monitoring information regarding the actual memory usage of all the VMs in the infrastructure. For that, we rely on the MR, which runs in the VM. The MR agent periodically (by default every five seconds although it can be configured on a per-VM basis) reports the memory usage to OneGate[3] by properly querying $/proc/meminfo$ to obtain both the total and free memory in the VM as well as the usage of the swap space. We rely on the contextualisation mechanisms provided by OpenNebula to dynamically stage in the running VM the agent that periodically monitors the memory consumption and the memory available reporting back to OneGate. This enables CVEM to access centralised monitoring information about the memory usage of all the VMs deployed in the on-premises Cloud (by default also every five seconds). Notice that MR and CVEM are decoupled systems which can work at different frequencies. In addition, the MR only reports significative memory changes so it can run very frequently.

The usage of contextualisation avoids the need to have pre-packaged Virtual Machine Images (VMIs) with the MR agent pre-installed. Instead, by using the contextualisation mechanisms offered by OpenNebula, our solution is independent of the VMI chosen by the user (though our proof-of-concept is based on GNU/Linux-based VMIs), since the agent is installed on-the-fly when the VM is deployed. Notice that, for other CMPs, DevOps tools such as Puppet or Ansible could also be used to dynamically deploy the MR agent right after the VM has booted.

Finally, to inform the ONE scheduler about the amount of memory from the hosts that

[3]OneGate: `http://docs.opennebula.org/4.12/advanced_administration/application_insight/onegate_overview.html`

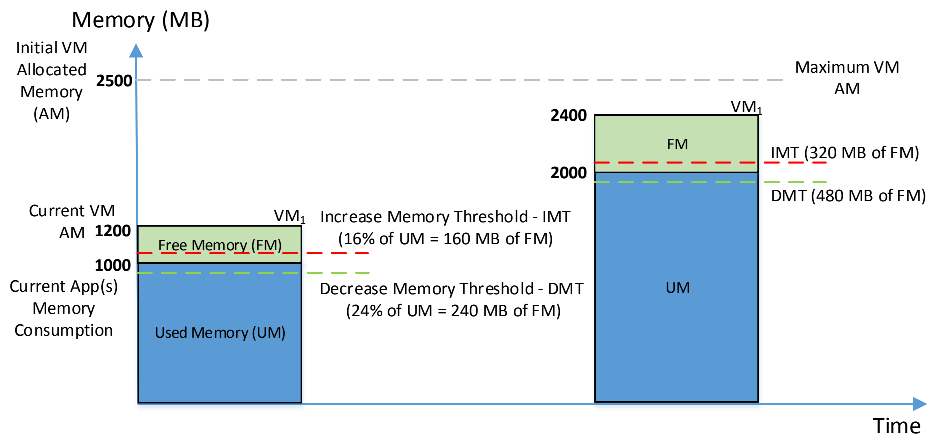
can be oversubscripted, we have created a modified version of the KVM Virtual Machine Manager (VMM) monitoring driver component that is shipped with OpenNebula. This version calculates the amount of *stolen memory* from each host and instructs the ONE scheduler to use part of it to allocated additional VMs in that host. We define the *stolen memory* for a given physical host as the total amount of memory that CVEM has been able to freed from the different VMs running on that physical host, in our case, through the use of memory ballooning via KVM. CVEM decides to enlarge or shrink the VM's allocated memory depending on the actual memory usage reported by the MR to OneGate.

Notice that the current Allocated Memory (AM) to a VM is divided between the current Used Memory (UM) by the applications running inside and the Free Memory (FM) and, therefore, $AM = UM + FM$. The vertical elasticity rules implemented in CloudVAMP build on our previous work [4] to maintain a Memory Oversubscription Percentage (MOP) of an additional 20% of the current UM. The goal is to keep that extra amount of free memory in case the application running in the VM starts requesting more memory. In our previous work we assessed the behaviour of different values of MOP, in particular 10% and 30% to understand the tradeoff between reducing the free memory in a VM at the expense of increasing the chances of an application to start thrashing due to lack of free memory in case the application requires a memory increase [4].

However, the vertical elasticity rules are only triggered if the percentage of free memory of the VM is smaller than 80% or greater than 120% of the MOP. This enables the system to only react when substantial changes in the used memory of the VM occur, thus removing unnecessary oscillatory memory changes. In these circumstances, CloudVAMP dynamically adapts the VM memory size using (1),

$$AM = UM \times (1 + MOP) \tag{1}$$

where AM is the newly allocated memory to the VM by the hypervisor and UM is the current used memory by the applications in the VM. As an example, a MOP of 20% means that the elasticity rule will only be triggered when the free memory of the VM is lower than 16% (80% of 20%) or greater than 24% (120% of 20%) of the used memory of the VM. As an example, if a VM has 1000 MB of AM and the application starts using



Memory (MB)
Initial VM Allocated Memory (AM)
2500
Maximum VM AM
2400
VM1
FM
IMT (320 MB of FM)
2000
DMT (480 MB of FM)
Current VM AM
1200
VM1
Increase Memory Threshold - IMT (16% of UM = 160 MB of FM)
Free Memory (FM)
1000
Current App(s) Memory Consumption
Decrease Memory Threshold - DMT (24% of UM = 240 MB of FM)
Used Memory (UM)
UM
Time

Figure 3: Memory thresholds that trigger the vertical elasticity rules in two example VM configurations. In the left, a VM with 1000 MB of Used Memory (UM) and, in the left, a VM with 2000 MB of UM. MOP=20% of UM.

900 MB, then the new AM will be 1080 MB (900×1.2).

For the sake of clarity, Figure 3 shows the memory thresholds that trigger the vertical elasticity rules in an example VM (left part of the figure) that was initially deployed with 2500 MB and was subsequently downsized to 1200 MB (AM), of which 1000 MB are being used by the application (UM) and 200 MB are the free memory (FM) provided by the MOP (20% of the UM). Whenever the UM exceeds the Increase Memory Threshold (IMT) or the Decrease Memory Threshold (DMT) the vertical elasticity rule (1) is applied in order to maintain that extra 20% free memory. Any memory consumption changes between those thresholds will not trigger the elasticity rules to avoid unnecessary oscillations. In case the application starts demanding additional memory, the system allocates extra memory resulting in the case shown in the right part of the Figure 3, which corresponds to the VM with, for example, 2000 MB of UM.

The elasticity rule has been complemented with a fail-safe mechanism when thrashing has already occurred within a VM. In that case, the memory size increase should be much larger to rapidly counteract the devastating effects that thrashing has in application performance [17]. For that we use a mechanism that greatly inspires in exponential backoff [18]. If there is no available free memory in the VM, an additional 50% of the difference between the maximum memory and the current allocated memory is assigned. This enables to rapidly increase the allocated memory to the VM, attempting to scape from thrashing as fast as possible. If there is still shortage of memory, the same additional allocation of memory is performed. Finally, if the third monitoring interval still reports a shortage of memory in the VM (probably because the application running in the VM is requesting memory faster than the rate at which CloudVAMP is increasing the allocated memory to the VM), the VM is allocated its maximum memory size. Notice that any excess of allocated memory will be corrected in subsequent steps by CloudVAMP by reducing the allocated memory according to the rule in (1), leading to a self-regulatory system.

4.1. Oversubscription via Stolen Memory

The KVM VMM monitor shipped with ONE has been modified in order to instruct the ONE scheduler to oversubscribe the memory of the physical hosts. The actual amount of memory available in the host that is reported to the ONE monitoring system is the amount of physical memory obtained by the actual monitoring system plus a percentage O from the amount of memory that could be *stolen* from the free memory available in the VMs. The scheduler shipped in ONE is unaware of the memory reduction of the VMs, and calculates the amount of memory available for virtual machines in one host as the memory available in the host minus the memory requested by the VMs when they were deployed, as shown in (2).

$$HostVMs_{mem} = Host_{mem} - \sum_{VM\ in\ Host} VM_{mem} \tag{2}$$

Using this approach, the ONE scheduler will act as if the hosts had more memory available for the VMs and will try to deploy new VMs in the physical host even if the total amount of memory requested by the VMs is greater than the physical memory available at the destination host.

The value of O can be configured for the on-premises Cloud in order to increase the degree of memory oversubscription. It is a percentage so a value of 0% means that no memory oversubscription will be introduced by CloudVAMP. This means that the sum of allocated memory of all the VMs of a host in the on-premises Cloud will never exceed the available memory of that host. A value of 100% for O means that CloudVAMP will try perform as much oversubscription as possible. This means to reclaim all the free memory from the VMs to enable maximum oversubscription, since the CMP scheduler will allocate additional VMs to the underlying hosts. Notice that this may require to migrate VMs more frequently if applications start demanding additional memory. Also, under no circumstances CloudVAMP will reclaim memory being used from the VMs since that would have a dramatic impact on the performance of applications. In the end, this parameter should be properly fine-tuned depending on the requirements of the on-premises Cloud.

4.1.1. Live Migration in On-premises Clouds

KVM fully supports live migration among physical hosts without any downtime provided that i) the Virtual Machine Image is located on a shared storage among the source and destination physical machines, and ii) both physical machines reside in the same subnet. These assumptions are commonly (and easily) met in an on-premises Cloud deployment.

Migration involves copying the memory pages from source to destination machines. The time involved in the live migration depends on the memory size of the VM but it is much more dependent on the rate at which dirty pages are created, which depends on the application usage of memory. As Clark et al. [19] noted, if the VM continuously dirty pages faster than the rate of copying, then the copy of pages work will be in vain. In particular, we have detected stalled live migrations for VMs executing memory-intensive applications, in which the memory is being frequently modified, thus creating new dirty pages at a faster rate than the ability of KVM to transfer those pages to destination.

This behaviour of live migration affects the policy employed to select the VM that should be live migrated under memory overload scenarios. Notice that the VM whose allocated memory is being increased, as happens in Figure 1.c, is expected to later use that memory, thus being a candidate to produce more dirty pages. Therefore, CloudVAMP will try to avoid choosing that VM when considering which VM should be migrated. In particular, CloudVAMP uses the following approach: First, it selects the VM with the least amount of allocated memory, running on the same machine that hosts the VM whose memory is growing, in a host that might become overloaded. This policy tries to minimise the migration time. Then it selects the destination host, selecting the one with the largest amount of free memory. In case that none of the available hosts has enough free memory to receive the VM, the migration is not performed. Notice that enhanced live migration strategies can be addressed although they lie out of the scope of this paper.

5. Assessment via Case Studies

This section assesses the usefulness of the developed system in a standard production environment based on OpenNebula 4.8 that consists of three dual 4 core Xeon E5620@2.40GHz with 16 GB RAM and three quad 4 core Xeon E7520@1.86GHz with 64 GB RAM, for a total of 72 cores and 240 GB RAM. The operating system for the platform is Ubuntu 12.04.5 LTS, using KVM version 1.0. Any piece of software is installed from the official repositories of Ubuntu and OpenNebula, except for the implementations made in this paper. In our tests, the value of O is set to 100% to gain the maximum amount of memory for other VMs, thus fostering maximum oversubscription.

For that, two case studies are executed. The first one integrates this technique in a production elastic virtual cluster of the es-NGI[4] infrastructure, the Spanish National Grid Initiative, to seamlessly accommodate workload of different sizes within a virtual cluster that features both horizontal and vertical elasticity. The second one focuses on the deployment of Docker containers running on a multi-tenant vertical elastic VM to adapt its memory size to the varying workload.

5.1. Fully Elastic Virtual Clusters for Grid Infrastructures

The es-NGI infrastructure is the spanish national Grid initiative that contributes computing and storage resources to the European Grid Initiative (EGI[5]) which is a global Grid infrastructure (also supporting federated Clouds) that supports scientific activities. More than 320 organisations across 43 countries offer a computing capacity that exceeds 480.000 cores where more than 1.4M jobs per day are executed[6].

[4]es-NGI: `http://www.es-ngi.es`
[5]EGI: European Grid Initiative
[6]`http://www.egi.eu/infrastructure/operations/egi_in_numbers/`

Our research group contributes with computing capacity in the shape of elastic virtual clusters in which VMs are dynamically provisioned to support the execution of incoming jobs in a sandboxed environment created by EC3 (Elastic Cloud Computing Cluster)[7] [20] an open-source tool to create elastic virtual clusters on hybrid Cloud infrastructures. This virtual cluster is based on a front-end node managed by CLUES [21] that monitors the LRMS (Local Resource Management System) and decides when to scale out (provision additional working nodes) and scale in (terminate working nodes) according to a set of configurable rules. However, the nodes of the cluster, which are VMs, are deployed with a fixed amount of memory, regardless of the amount of memory actually consumed by the applications being executed in them. The Workload Management System (WMS) of the Grid infrastructure is responsible for allocating the applications to resources with at least as much free memory as the application requests. However, the running applications typically use less memory than the one actually available in the VMs. That memory could be employed to allocate new VMs for the execution of other jobs thus increasing both the job throughput and the usage of our on-premises Cloud platform.

In this case study we wanted to assess the effectivity of introducing vertical elasticity in the shape of dynamic memory management within this production platform. We introduced CloudVAMP into the platform, and recorded data from a representative workload that arose from different real jobs in a period of 12 hours. The case study involves three physical hosts with 16 GB of RAM (*niebla02*, *niebla03* and *niebla04*) and one physical host with 64 GB of RAM (*niebla13*).

Figure 4 represents a summary of the evolution of the stolen memory for the physical hosts that hosted the VMs that where part of the elastic virtual cluster deployed to support the execution of jobs coming from the es-NGI. Remember that the stolen memory is the memory that has reclaimed in a physical host by CloudVAMP. The figure also depicts the sum of stolen memory across the physical hosts. The larger the amount of stolen memory per host, the higher the chances are that the CMP deploys additional VMs in that host. The figure shows that under real workload scenarios, CloudVAMP is able to free memory from the VMs by adjusting their allocated memory to the real memory requirements of the VM.

[7]EC3: `http://www.grycap.upv.es/ec3`

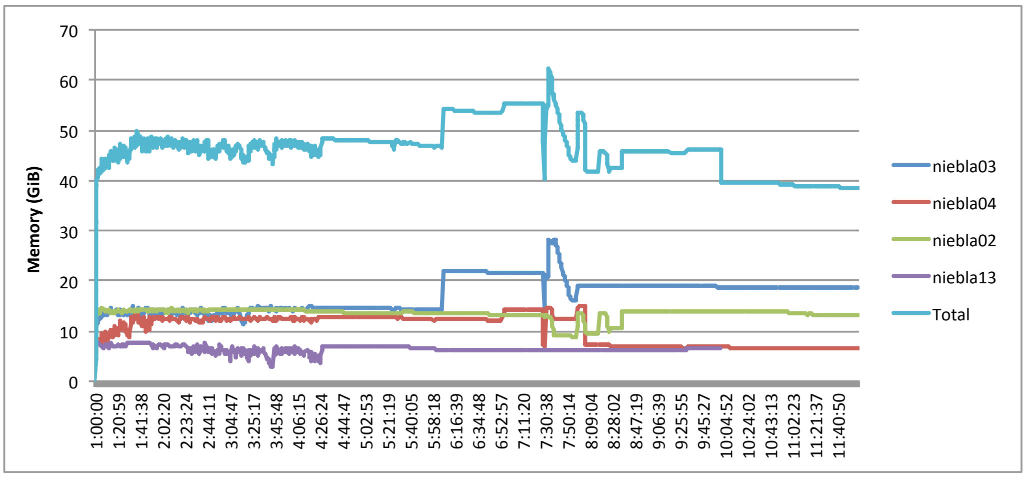


Figure 4: Evolution of the stolen memory of the hosts that execute the VMs that support the virtual elastic cluster that executes the jobs from the es-NGI Grid infrastructure.

Within the same time frame, Figure 5 describes the evolution of a representative subset of nine VMs of that virtual cluster. Take into account that the number of nodes of the virtual cluster dynamically changes depending on the number of jobs currently received by our Grid site. Notice that all the VMs are initially deployed with 8 GB and they are almost instantly downsized depending on the actual memory consumption of the application. A VM can host the execution of different simultaneous Grid applications, depending on the number of virtual CPUs, which for this study is set to 4 vCPUs. Subsequent executions of different applications in the same VM is also possible.

The oscillatory memory allocation patterns that can be seen at the beginning of the execution of some VMs, as is the case of VM1 and VM7, can be both due to the highly dynamic memory consumption patterns of a single application or the concurrent execution of different applications and, henceforth, with different memory consumption.

Notice that VM8 and VM9 are deployed at around 7 hours and 30 minutes since the study was started. These represents the horizontal elasticity of the virtual cluster in action, where two additional nodes are deployed because new incoming jobs are requested to be executed in the virtual cluster. Then, VM8 is terminated approximately 20 minutes after its deployment. These depends on the horizontal elasticity rules provided by CLUES, where new VMs are dynamically deployed to host the execution of incoming

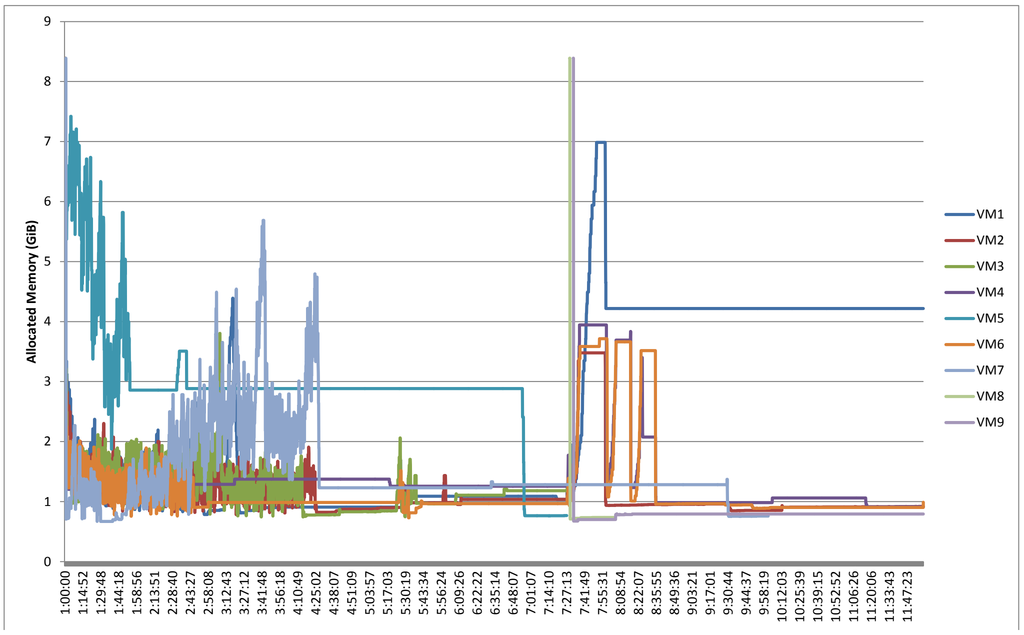


Figure 5: Evolution of the allocated memory of some of the VMs that compose the virtual elastic cluster.

Host	niebla13	niebla02	niebla04	niebla03
Phys. Mem. (GiB)	64	16	16	16
Avg. Stolen Mem. (GiB)	4.97	12.82	9.91	16.38

Table 1: Comparison of the physical memory of the hosts vs the stolen memory (the freed memory per node when using CloudVAMP) .

jobs and they are terminated when no longer required.

Table 1 compares the physical memory of the hosts with the average stolen memory obtained as a result of the application of CloudVAMP during the 12-hour case study. Notice that in the case of the host *niebla03* the average stolen memory exceeds its physical memory. To understand this, consider a scenario in which two 8 GB VMs are deployed on a physical host with 17 GB of RAM and, with the help of CloudVAMP, the VMs are reduced to 1 GB to fit the memory consumption of the applications running in them. Then, an additional 8 GB VM is deployed on the same host and later shrank to 1 GB. These VMs fit in the physical host and you are saving 7 GB per VM, which represents a total save of 21 GB, an amount greater than the physical host's memory. Therefore, CloudVAMP enables memory oversubscription to take place, by allowing the CMP to schedule the deployment of additional VMs in the physical hosts. However, when applications running in the VMs start using more memory, and CloudVAMP increases their allocated memory, the physical host might incur in memory overload. This is why live migration techniques can be used to maintain the quality of service delivered by the on-premises Cloud. This is the topic addressed in the next case study.

5.2. Preventing Memory Overload via Live Migration

This section introduces an approach to prevent memory overload by using live migration techniques available in the KVM hypervisor. For that, we are going to introduce a multi-tenant scenario based on Docker containers.

Docker [22] introduces the ability to package applications and their dependences into lightweight containers tailored for specific distributions which, as opposed to VMs, can be spun up very fast. This technology is of special interest for multi-tenant scenarios in which a set of physical resources has to be shared among different users, by leveraging process isolation and without the overhead introduced by a hypervisor layer. In fact containerization is one of the underlying technologies among popular open-source PaaS tools such as CloudFoundry[8] and OpenShift[9].

This case study features the deployment of a VM with Docker that supports the deployment of containers to host different applications within the same VM. This approach separates infrastructure provision (from the Cloud) and application deployment (using Docker containers) which introduces significant benefits to deploy applications on multiple back-ends. In multi-tenant scenarios, where a single VM can be used to deploy multiple containers from multiple users, it is expected a larger variation in the memory consumption patterns, when compared to a single application running on a single VM. This is why we believe that CloudVAMP can be beneficial by automatically managing the allocated memory to the VM (or a set of VMs) according to the memory used by its active containers.

Figure 6 describes the scenario employed, along with the following events:

[8]CloudFoundry: `http://www.cloudfoundry.org`

[9]OpenShift: `http://www.openshift.com`

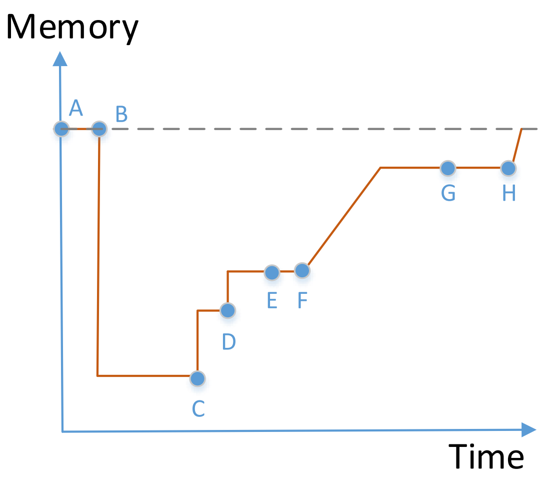


Figure 6: Sequence of events that introduce memory overload in a multi-tenant scenario based on Docker containers.

A A VM (VM_1) is provisioned with a certain amount of RAM on a given physical host of the on-premises Cloud.

B CloudVAMP reduces the allocated RAM of VM_1 since the memory consumption of the VM after its boot is very low (no application is being ran yet).

C Docker is installed and the first container is deployed based on an image with the Apache Tomcat application server. This will result in an increase of the allocation of memory to VM_1, as requested by CloudVAMP.

D A second container is deployed based on the same image. We expect a memory increase, although slightly lower due to the sharing of some pages between the two containers.

E Since VM_1 memory was reduced, there is enough free available memory in the physical machine to host another VM (VM_2), as decided by the OpenNebula scheduler, which will be running in the same physical host.

F A memory-intensive application is executed on a third container which introduces memory pressure for VM_1. We will use a synthetic benchmark application that

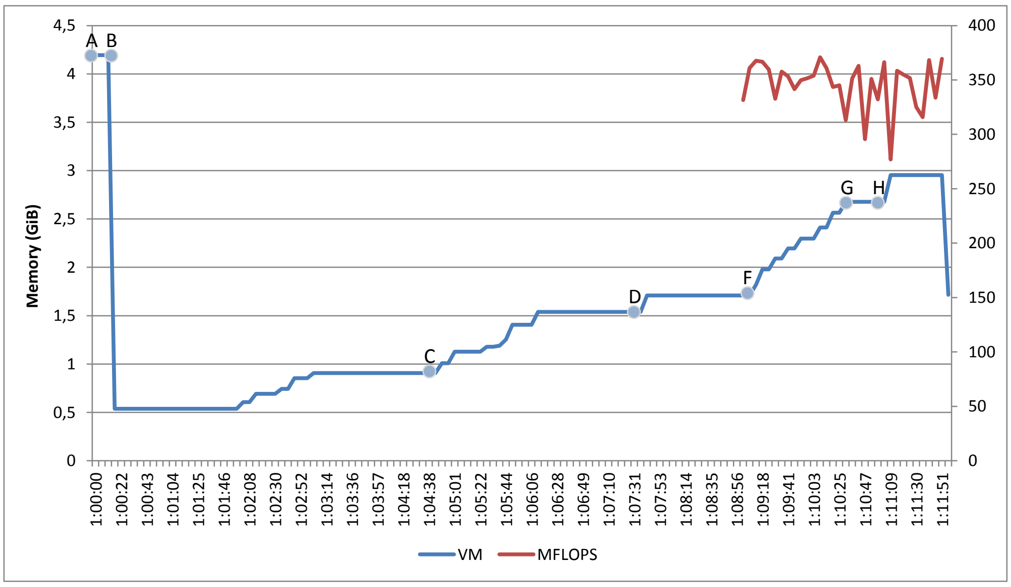


Figure 7: Memory consumption of a VM

enables us to control the memory allocation pattern and to obtain the performance of the application (in MFLOPS) as described in [4].

G CloudVAMP will try to increase the memory of VM_1 but since this would result in memory overload, it will use a live migration strategy as a contingency plan. This involves migrating a VM from the physical host to restore the capability of VM memory without overload.

H When enough memory has been freed from the physical host, VM_1 can be allocated more memory. Remember that the VM memory size will not be able to grow beyond the amount of memory specified when initially created the VM.

The aforementioned sequence of events has been carried out in the same on-premises Cloud. Figure 7 shows the real memory allocation of the VM that hosted the different Docker containers.

First of all, the VM is deployed with 4 GB of RAM at time instant 1:00:00 (which corresponds to event A in Figure 6). A few seconds later, CloudVAMP detects that the VM has enough free memory and decides to reduce its allocated memory to slightly over than 500 MB (event B). At 1:01:57 we perform the installation of required packages to use Docker, what demands additional memory and results in a periodic increase in the allocated memory to the VM.

At 1:04:44, the first Docker container is deployed (C) what increments the memory requirements for the VM, thus resulting in an increase of its allocated memory. The plateau of allocated memory to the VM that can be noticed from 1:06 until 1:07 is due to the steady state of the VM, since once the Docker containers are started, no activity is really performed with those containers, for the sake of clarity in this case study.

At 1:07:36, the second container is deployed (D) what introduces memory pressure in the VM resulting in a periodic increase in the allocated memory, according to the increasing memory requirements for the containers, which host Tomcat, a memory-intensive Java application server. Within this period we have purposely deployed other VMs within the same physical host to introduce memory pressure in the host when the analysed VM starts demanding more memory.

At instant 1:09:01 the memory-intensive application is deployed in the VM to force a steady memory consumption from 0 to 1000 MB in two minutes and maintain that memory consumption for other 60 seconds (F). This results in CloudVAMP to periodically increase the allocated memory of the VM at relatively similar memory chunks according to the periodic memory consumption increase of the VM. At instant 1:10:36, there is so much memory pressure in the physical host that no additional memory can be allocated to the VM. Although the application is constantly demanding more memory, CloudVAMP cannot allocate additional memory to the VM because the host is starting to become overloaded, in terms of memory. In this situation, the application might incur in thrashing because it has to rely on swap memory. Since this situation would affect the performance of application, it is important to prevent the memory overload in the physical host. This requires live migration techniques to move a VM away from the physical host so that the available free memory can later be allocated to additional VMs running on the physical host.

In this case CloudVAMP was configured to live migrate the VM with the least amount of allocated memory to prevent as fast as possible the memory overload situation. Remember that the time invested in live migration is typically related to the memory size, though it is much dependent on the applications running inside, in particular the rate at which dirty pages are created.

Therefore, at around instant 1:10:36 (G) a VM other than the one considered in this case study is migrated away from the physical host in a process that lasted less than a minute. This way, at instant 1:11:09 (H) the VM can now be allocated more memory to comply with the increasing memory requirements of the application. Shortly after, the application is stopped and the case study is finished.

It is important to point out that the usage of CloudVAMP in an on-premises Cloud has enabled to dynamically manage the memory allocated to the VMs and to alleviate the memory pressure that arises due to the oversubscription via live migration techniques without any VM downtime.

Notice that Figure 7 also shows the MFLOPS that delivers the application, to evaluate the impact of the memory oversubscription scenario and the live migration of the VM on the performance of the application being executed. You can notice a reduction of up to 15% in the MFLOPS delivered by the application which can be attributed mainly to eventual thrashing and secondarily to live migration. However, this reduction is very transient and, for long running applications, might be negligible. In addition, CloudVAMP can be fine tuned in order to try to prevent the applications from thrashing at the expense of wasting additional memory by increasing, for example, the value of MOP or reducing the value of O at the infrastructure level.

As a final remark, notice that certain type of applications that require low latency responses may prefer not to be live migrated to other hosts, which might have an impact (although relatively small, as shown on the case study) on its performance and the level of service expected by the client. If a Cloud provider needs to run applications that are very sensitive to performance, this can be supported in our system by allocating the VMs that run those applications to a subset of hosts that will not be monitored by CloudVAMP. This way, the allocated memory to those VMs will not be reduced and applications will run on the requested resources without being migrated to other hosts.

Also, notice that the goal of CloudVAMP is not to allocate more resources to increase the performance of an application but to reclaim the unused resources (in particular we focus on the memory because hypervisors support their dynamic management) without affecting the performance of the application. It is possible to reduce the allocated memory of a VM that is currently not being used by an application for other VMs to use it. Of course, depending on the memory consumption patterns, the application might require the extra memory back and this might introduce a performance penalty. In the end, these techniques can be further customised for a specific on-premises Cloud depending on the workload and application characteristics.

6. Conclusion and Future Works

This paper has introduced CloudVAMP, a customisable system to safely enable transient memory oversubscription in on-premises Clouds via vertical elasticity without VM downtime and featuring live migration to prevent oversubscription scenarios. By leveraging the memory ballooning techniques and live migration capabilities available in the KVM hypervisor, CloudVAMP integrates with Cloud Management Platforms to dynamically reduce and increase the allocated memory to the VMs so that they fit the memory requirements of the applications running in the VMs.

We have introduced a generic architecture that can be deployed for different CMPs, and we have implemented a fully functional open-source proof-of-concept based on OpenNebula which is currently being used in production at our research center[10]. The benefits of CloudVAMP have been assessed via a case study that uses horizontal and vertical elastic virtual clusters that run jobs from a production Grid infrastructure and a multi-tenant scenario based on Docker containers. The ability of CloudVAMP to reclaim unused memory from the VMs to enable temporary oversubscription for the CMPs has resulted in increased VM-per-host consolidation ratio with a reduced impact for the running applications. The usage of live migration has been beneficial to restore the level of service in memory overload scenarios.

Future works includes adjusting the O percentage on a per-VM level considering the stability of each VM. For example, CloudVAMP could reclaim different percentages of free memory depending on the amount of time in which a VM's memory consumption has remained among a certain range. For VMs with long periods of stable memory

[10]CloudVAMP, available at `https://github.com/grycap/cloudvamp`

consumption it might be safe to assume that the unused memory will not be used, and a greater percentage can be reclaimed by CloudVAMP to be used for additional VMs to be hosted on the same physical node. Also, we plan to evolve the CVEM to consider historical information of the memory consumption of the VMs to avoid unnecessary transient memory changes while maintaining the ability to rapidly react when memory consumption spikes are detected.

In addition, we plan to explore memory bursting, where a VM could temporarily allocate more memory than the one initially requested, much in the same way as CPU bursting is available for certain instance types (e.g. *t2.micro*) in Amazon EC2. This can be easily implemented by increasing the VM deployment memory request by a certain percentage, which would depend on the policies of the on-premises Cloud, and letting CloudVAMP to dynamically manage the memory consumption, which could temporarily exceed the amount of memory initially requested.

Finally, we plan to generalise our development to other CMPs (e.g. OpenStack). For that, one can use Ganglia as the memory reporting system and modify the monitoring system of OpenStack to integrate CloudVAMP.

Acknowledgments

The authors would like to thank the Spanish "Ministerio de Economía y Competitividad" for the project CLUVIEM (TIN2013-44390-R) and the European Comission for the project INDIGO-DataCloud with grant number 653549.

References

[1] R. Moreno-Vozmediano I. M. Llorente, IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures, in: IEEE Computer (Long. Beach. Calif)., vol. 45, no. 12, pp. 6572, Dec. 2012.
URL `http://doi.ieeecomputersociety.org/10.1109/MC.2012.76`

[2] G. Galante, L. C. E. de Bona, A Survey on Cloud Computing Elasticity, in: 2012 IEEE Fifth International Conference on Utility and Cloud Computing, IEEE, 2012, pp. 263–270.
URL `http://dl.acm.org/citation.cfm?id=2415689.2415736`

[3] C. A. Waldspurger, Memory resource management in VMware ESX server, ACM SIGOPS Operating Systems Review 36 (SI) (2002) 181.
URL `http://dl.acm.org/citation.cfm?id=844128.844146`

[4] G. Moltó, M. Caballer, E. Romero, C. de Alfonso, Elastic Memory Management of Virtualized Infrastructures for Applications with Dynamic Memory Requirements, in: Proceedings of the International Conference on Computational Science (ICCS 2013), Elsevier, 2013, pp. 159–168.

[5] L. Tomás, J. Tordsson, Improving cloud infrastructure utilization through overbooking, in: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference on - CAC '13, ACM Press, New York, New York, USA, 2013, p. 1.
URL `http://dl.acm.org/citation.cfm?id=2494621.2494627`

[6] R. Householder, S. Arnold, R. Green, On Cloud-based Oversubscription, International Journal of Engineering Trends and Technology 8 (8) (2014) 425–431.

[7] D. Williams, H. Jamjoom, Y.-H. Liu, H. Weatherspoon, Overdriver: handling memory overload in an oversubscribed cloud, ACM SIGPLAN Notices 46 (7) (2011) 205.
URL `http://dl.acm.org/citation.cfm?id=2007477.1952709`

[8] W. Dawoud, I. Takouna, C. Meinel, Elastic VM for Cloud Resources Provisioning Optimization, in: Advances in Computing and Communications. First International Conference, ACC 2011, Kochi, India, July 22-24, 2011. Proceedings, Part I, Vol. 190, 2011, pp. 431–445.
URL `http://www.springerlink.com/index/K75M2705443R2402.pdf`

[9] E. Tasoulas, H. r. Haugerund, K. Begnum, Bayllocator: a proactive system to predict server utilization and dynamically allocate memory resources using Bayesian networks and ballooning, in: Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques, USENIX Association, 2012, pp. 111–122.
URL `http://dl.acm.org/citation.cfm?id=2432523.2432532`

[10] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, M. Ben-Yehuda, Applications Know Best: Performance-Driven Memory Overcommit with Ginkgo, in: 2011 IEEE Third International Conference on Cloud Computing Technology and Science, IEEE, 2011, pp. 130–137.
URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6133136`

[11] T.-I. Salomie, G. Alonso, T. Roscoe, K. Elphinstone, Application level ballooning for efficient server consolidation, in: Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13, ACM Press, New York, New York, USA, 2013, p. 337.
URL `http://dl.acm.org/citation.cfm?id=2465351.2465384`

[12] J. Hwang, A. Uppal, T. Wood, H. Huang, Mortar: Filling the Gaps in Data Center Memory, ACM SIGPLAN Notices 49 (7) (2014) 53–64.
URL `http://dl.acm.org/citation.cfm?id=2674025.2576203`

[13] S. A. Baset, L. Wang, C. Tang, Towards an understanding of oversubscription in cloud (2012) 7.
URL `http://dl.acm.org/citation.cfm?id=2228283.2228293`

[14] KVM, Automatic Ballooning.
URL `http://www.linux-kvm.org/page/Projects/auto-ballooning`

[15] A. Litke, Manage resources on overcommitted KVM hosts, Tech. rep. (2011).
URL `http://www.ibm.com/developerworks/library/l-overcommit-kvm-resources/`

[16] A. Kivity, Y. Kamay, D. Laor, KVM: the Linux virtual machine monitor, Proceedings of the Linux

Symposium (2007) 225–230.
URL http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf

[17] P. J. Denning, Thrashing: Its causes and prevention, in: Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I), ACM Press, New York, New York, USA, 1968, p. 915.
URL http://dl.acm.org/citation.cfm?id=1476589.1476705

[18] L. Miller, Performance analysis of exponential backoff, IEEE/ACM Transactions on Networking 13 (2) (2005) 343–355.
URL http://dl.acm.org/citation.cfm?id=1066626.1066636

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, Live migration of virtual machines, in: NSDI'05 Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation, USENIX Association, 2005, pp. 273–286.
URL http://dl.acm.org/citation.cfm?id=1251203.1251223

[20] M. Caballer, C. de Alfonso, F. Alvarruiz, G. Moltó, EC3: Elastic Cloud Computing Cluster, Journal of Computer and System Sciences 79 (2013) 1341–1351.
URL http://authors.elsevier.com/sd/article/S0022000013001141

[21] C. de Alfonso, M. Caballer, F. Alvarruiz, V. Hernández, An energy management system for cluster infrastructures, Computers & Electrical Engineering 39 (8) (2013) 2579–2590.
URL http://www.sciencedirect.com/science/article/pii/S0045790613001365

[22] D. Merkel, Docker: lightweight Linux containers for consistent development and deployment, Linux Journal 2014 (239) (2014) 2.
URL http://dl.acm.org/ft_gateway.cfm?id=2600241&type=html