UNIVERSIDADE FEDERAL FLUMINENSE

Roberto Kendy Sawamura

Improving Application Throughput by Exploiting Vertical Memory Elasticity in Virtualized Environments

NITERÓI

2015

UNIVERSIDADE FEDERAL FLUMINENSE

Roberto Kendy Sawamura

Improving Application Throughput by Exploiting Vertical Memory Elasticity in Virtualized Environments

Dissertation presented to the Computing Graduate program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic area: Computer Systems

Advisor:
Cristina Boeres and Vinod Rebello

NITERÓI

2015

Roberto Kendy Sawamura

Improving Application Throughput by Exploiting Vertical Memory Elasticity in Virtualized Environments

Dissertation presented to the Computing Graduate program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic area: Computer Systems

Approved in December 2015 by:

Prof. Maria Cristina Silva Boeres, Ph.D. - Advisor, UFF

Prof. Eugene Francis Vinod Rebello, Ph.D. - Advisor, UFF

Prof. Claudio Luis de Amorim, Ph.D., UFRJ

Prof. Orlando Gomes Loques Filho, Ph.D., UFF

Niterói

2015

Resumo

A natureza dinâmica das aplicações e serviços tem levado a comunidade da área de nuvens a investir no estudo e desenvolvimento de funcionalidades elásticas que permitem o redimensionamento dinâmico de recursos. Até os dias atuais, grande parte da área científica tem focado em abordar a elasticidade horizontal, a fim de fornecer a quantidade apropriada de máquinas virtuais (VMs) para manter a qualidade de serviço. Mais recentemente, a *elasticidade vertical* tem recebido maior atenção, onde o processamento, memória e armazenamento de uma VM é ajustada de acordo com as necessidades da aplicação. Tanto aplicações online com cargas imprevisíveis e workflows científicos com diferentes conjuntos de dados necessitam de redimensionamento autônomo, a fim de evitar perda de desempenho ou pagar por recursos subutilizados e possivelmente desnecessários. Dado o crescente impacto da disponibilidade de memória no desempenho, este trabalho apresenta as principais características do Memory Elasticity Controller (MEC), um escalonador de VMs que visa maximizar a vazão de jobs ou workflows, calibrando criteriosamente a quantidade de memória alocada para cada VM em execução, de acordo com os requisitos dinâmicos das aplicações, evitando, ao mesmo tempo, comprometer seus desempenhos. Este trabalho apresenta a arquitetura da ferramenta e avalia uma variedade de aspectos através de experimentos que destacam os benefícios, tanto para os provedores de recursos quanto para as aplicações, com maior eficiência, vazão e desempenho.

Palavras-chave: Elasticidade vertical, Escalonamento de máquinas virtuais, Alocação dinâmica de memória.

Abstract

The dynamic nature of application or service requirements has lead the cloud community to invest in the study and development of elasticity features that have the ability to re-dimension resource capacities dynamically. To date, the dominant share of the research literature has focused on tackling horizontal elasticity in order to provision the appropriate number of virtual machines (VMs) to meet given quality of service criteria. More recently, work has begun to investigate *vertical elasticity* where the processing, memory or storage capacity of a single VM is adjusted in accordance with the application's needs. Both online applications under unpredictable workloads or scientific workflows with different datasets require autonomic scaling in order to avoid performance degradation or paying for additional, sub-utilized and possibly unnecessary, resources. Given the increasing influence of memory availability on performance, this work presents the principal features of the Memory Elasticity Controller (MEC), a VM scheduling tool that aims to maximize the throughput of jobs or workflow tasks by judiciously calibrating the amount of host memory allocated to each running VM in accordance with that VM's respective job's changing run time requirements while, at the same time, trying to avoid compromising the job's performance. This work presents the tool's architecture and evaluates a variety aspects through experiments that highlight the benefits of the approach for both resource providers and applications with improved efficiency, throughput and performance.

Keywords: Vertical elasticity, Virtual machines scheduling, Dynamic memory allocation.

List of Figures

3.1 Relationship between $mo(vm_i)$ and $maxml(vm_i)$ 11

3.2 Swap Activation Threshold in relation to $ma()$ and $maxml()$) configurations. 12

3.3 Free memory and swap consumption during the execution J_1 13

3.4 Relative job performance on a VM with different amounts of allocated memory 14

3.5 Memory utilisation and swap consumption for J_1, J_2 and J_3 15

3.6 Swap-out and swap-in for J_1, J_2 and J_3 16

3.7 The $siso(vm_i, t)$ indicator and swap consumption for $m = 70\%$ 17

3.8 Concurrently VMs with and without memory constraints 18

3.9 Sequential and concurrent execution of VMs, when using swap and not (ideal execution) 19

4.1 The Memory Elasticity Controller (MEC) Architecture 21

4.2 Communication between the component and their threads of the layers of MEC . 22

4.3 VM status and actions 23

4.4 Calculating positive memory shaping $ms(vm_i, t)$ for $\Delta_{cm}(vm_i, t) > 0$. The current free memory $fm(vm_i, t)$ is shown at (1). The estimated free memory $efm(vm_i, t + I_{host})$ is calculated at (2). The memory shaping $ms(vm_i, t)$ is then calculated at (3), to avoid letting the free memory fall below the $sat(vm_i)$ value before the next scheduling event at $t + I_{host}$ (4). 27

4.5 Calculating memory shaping $ms(vm_i, t)$ when $\Delta_{cm}(vm_i, t) > 0$. The current free memory $fm(vm_i, t)$ is shown at (1) and the estimated free memory $efm(vm_i, t + I_{host})$ at (2). The memory shaping $ms(vm_i, t)$ is calculated at (3) to avoid leaving "idle" unused memory but still keeping free memory above $sat(vm_i)$ limit until next scheduling time $t + I_{host}$ at (4). 27

4.6 Calculating memory shaping $ms(vm_i, t)$ when $\Delta_{cm}(vm_i, t) < 0$. The current free memory $fm(vm_i, t)$ is shown at (1) and the estimated free memory $efm(vm_i, t + I_{host})$ at (2). If the $ms(vm_i, t)$ calculation where to consider the negative $\Delta_{cm}(vm_i, t)$ and keep the free memory equals to the $sat(vm_i)$ value at the time $t+I_{host}$ (in the future (4)), then the current memory allocation $ma(vm_i, t)$ would be decreased by $ms(vm_i, t)$ in such a way that would leave $fm(vm_i, t)$ below $sat(vm_i)$ (at (3)) at the current time t. 28

4.7 Memory State transitions diagram 33

4.8 The *CloudManager* at CL layer 41

5.1 Consumed memory from the host point of view 45

5.2 Memory and swap consumption during the execution of J_1 using the function "= 1" and $I_{host} = 5$s under MEC management 46

5.3 The effects on the memory allocation during J_3 execution on vm_i due to memory extraction 49

5.4 The effects on the memory allocation during J_1 execution on vm_i due to memory extraction 49

5.5 Scenarios SC1 and SC2 with different priorities for choosing active VMs 51

5.6 Scenario SC3 where VMs face the same *siso*(), breaking ties with the "Distribute", "High ms" and "Low ms" criteria 52

5.7 Pausing mechanism evaluation when executing two VMs concurrently in different situations 53

5.8 Execution time ratio of n VMs under MEC management versus static memory allocation 55

5.9 Execution time ratio: Static versus MEC 56

5.10 Parsec with MEC vs STATIC 57

5.11 MEC vc STATIC-V 57

5.12 MEC vc MOP 59

List of Tables

3.1 Corresponding total memory $tm(vm_i)$ and memory overhead $mo(vm_i)$, for each memory allocation $ma(vm_i, t)$, with a fixed $maxml(vm_i)$ value of 12288, in MB 10

4.1 Virtual machines states $vstate(vm_i, t)$ and descriptions 24

4.2 Memory State transitions 33

4.3 Actions and state transitions of virtual machines 40

5.1 Jobs specifications 44

5.2 Memory Consumption rates (GB/s) and Required Extra Memory (GB) for different functions 46

5.3 MEC overhead 47

5.4 Jobs specifications for memory extraction 48

5.5 Jobs specifications for memory extraction 49

5.6 Comparison on the Pausing Mechanism (Memory Unit in GB) 53

List of Acronyms and Abbreviations

CL	:	Cloud layer;
$cm(vm_i, t)$	:	Consumed memory reported by guest OS of vm_i at time t;
$fm(vm_i, t)$	:	Free memory reported by guest OS of vm_i at time t;
$hfm(pm_j, t)$	:	Host free memory of pm_j at time t;
I_{host}	:	Interval between scheduling events on the host machine;
$I_{monitor}$	:	Interval at which information is collected from the OS in each VM;
$ma(vm_i, t)$	:	Memory allocation for VM vm_i at time t;
$maxml(vm_i)$	:	Maximum memory allocation for VM vm_i;
MEC	:	Memory Elasticity Controller;
$mo(vm_i)$	:	Memory overhead of vm_i;
$ms(vm_i, t)$	:	Memory shaping for vm_i at time t;
$pdr(W_k)$	:	Peak dynamic range of W_k;
PM	:	Physical machine;
pm_j	:	Physical machine j;
PML	:	Physical machine layer;
$sat(vm_i)$	:	Swap activation threshold of vm_i;
$sc(vm_i, t)$	:	Swap consumption of vm_i at time t;
$si(vm_i, t)$	:	Amount of swap-in since last boot of vm_i at time t;
$siso(vm_i, t)$	:	Swap-in/Swap-out indicator of vm_i at time t;
$so(vm_i, t)$	:	Amount of swap-out since last boot of vm_i at time t;
$tm(vm_i, t)$	:	Total memory reported by guest OS of vm_i at time t;
VM	:	Virtual machine;
vm_i	:	Virtual machine i;
VML	:	Virtual machine layer;
W_k	:	Workflow k;
$ws(W_k, t)$	:	Working set of W_k at time t

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Objectives 3

1.3 Contributions 4

1.4 Structure of the dissertation 4

2 Related work **5**

3 An Analysis of Memory Parameters **8**

3.1 Definitions 8

3.2 Identification of relevant characteristics 9

3.2.1 Memory Overhead 9

3.2.2 Swap Activation Threshold 11

3.2.3 Performance impact of distinct working sets 13

3.2.4 Evaluating Swap Usage 15

3.2.5 Collective Impact of Swap Usage in Concurrent VMs 16

3.2.6 Summary 18

4 MEC - A Memory Elasticity Controller **20**

4.1 MEC layers 20

4.2 The Host Manager 22

4.2.1 Virtual Machine State Transitions 22

4.2.2 The Host Manager Algorithm 24

4.2.3 The Host Manager Parameters 24
4.2.3.1 Memory Shaping 25
4.2.3.2 The swap-in and swap-out indicator 30
4.2.4 Memory State Transitions 31
4.2.4.1 Managing Active VMs 33
4.2.4.2 Managing Inactive VMs 38
4.2.4.3 Committing changes 39
4.3 The Cloud Manager 41

5 Experimental Results **43**
5.1 Overview 43
5.2 Evaluating Specific Characteristics 43
5.2.1 Host memory consumption pattern under MEC supervision 44
5.2.2 Management frequency 45
5.2.3 Memory extraction mechanism 47
5.2.4 Prioritizing VMs 50
5.2.4.1 Breaking ties with lowest $ms(vm_i, t)$ 51
5.2.5 Pausing VMs 51
5.3 MEC overall evaluation 54
5.3.1 Synthetic Jobs 54
5.3.2 Parsec Benchmark 55

6 Conclusion and future work **60**
6.1 Conclusion 60
6.2 Future work 62

References **64**

Appendix A – Parsec Benchmark Applications **66**

Appendix B – DMSS implementation algorithm **68**

Chapter 1

Introduction

With the continuous advances in multi-core technology and consolidation of virtualization, systems such as KVM [2], Xen [6] and VMware [5] have become the foundation of cloud computing, allowing IaaS services to run several virtual machines (VMs) on a single physical node. While virtualization provides environments isolated from one another, it brings a new challenge from the host's point of view – how to distribute the available physical resources among the VMs. A conservative approach to allocate VMs can be applied, based on the host server's ability to meet the VM's maximum CPU, I/O and memory requirements. However, given that the requirements of applications within the VM may vary during execution, it might be more efficient to also vary over time the amount of resources dedicated to the VM.

According to the National Institute of Standards and Technology (NIST) [18], one essential characteristic of cloud computing is rapid elasticity, where "*capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand*". The cloud system should be able to scale resources horizontally or vertically [12].

Horizontal elasticity is concerned with the rapid provisioning and releasing of nodes in order to efficiently deal with the change on the workload. For example, if necessary and depending on the application demands, increasing (or decreasing) the number of virtual machines might be necessary in order to deal with changes in workload. In the Amazon Elastic Compute Cloud (Amazon EC2) [7], the "Auto Scaling" [8] feature works in this way, allowing EC2 instances (virtual machines) to be initiated/terminated automatically, according to user defined conditions.

On the other hand, vertical elasticity is related with the ability to modify the capacity

in terms of vCPU cores or memory allocated to a single VM, already in execution, to best attend the application(s) it hosts. Specifically for RAM memory, elasticity can be achieved through the virtual memory ballooning [22], a technique used by the hypervisor to change the amount of memory allocated to the VM. The process of allocating/deallocating memory is called inflating/deflating the balloon.

Vertical memory elasticity is advantageous given that application working sets change in size [19] and distinct applications have differing memory demands. This presents Infrastructure Providers with an opportunity to improve the effectiveness of their server consolidation plans and thus, further reduce energy consumption, capital expenses and administration costs.

The technique of over-provisioning or over-committing VMs, where more resources are allocated than physically exist, is becoming common practice. Memory over-provisioning is motivated by the fact that there can be a substantial difference between the amount of memory allocated to a VM and the amount that is actually used. The memory pages requested by a process are usually only allocated when they are first used. Furthermore, sharing pages between multiple processes can also obviously save memory. While not perfect, as available memory becomes scarce, the system can free up space by swapping out less frequently accessed pages to disk. This discrepancy can help optimize the use of limited resources. Over-provisioning however is not an ideal solution for all memory issues. Although increasing the degree of over-provisioning might take better advantage of resource capacity by fully utilizing it, this can also cause a dramatic loss in performance and, since VMs will require resources for longer, will in turn adversely affect both efficiency and throughput.

Rather than adopt memory over-provisioning, this work proposes to adjust the amount of memory allocated to each VM while it executes its jobs. The allocated memory will be elastically increased when required and reduced when there is a surplus and additional memory is required by another VM on the same server. By developing a hypervisor agnostic tool to allocate memory to VMs on demand in order to improve VM scheduling, it is possible, for example, in an IaaS cloud system or data centre to consolidate servers, without sacrificing performance.

To do this, this dissertation investigates the impact of memory allocation and swap usage on VM performance and through experimental evaluations, a hypervisor independent metrics and policies are identified. Based on the experiment results, this work goes on to present the Memory Elasticity Controller (MEC), a framework of a tool to dy-

namically manage memory allocations and states of VMs. Results with the proposed tool highlight some of the benefits to both resource providers and applications through improved efficiency, throughput and performance.

1.1 Motivation

The difficulty in estimating the dynamic memory usage of applications as well as absence of an automatic memory management on cloud infrastructures creates a challenge in VM memory allocation and vertical elasticity. Usually the allocation is static and is not changed automatically during VM execution. Overestimating or underestimating the memory allocation can have serious impact on applications as well as for the cloud system.

Overestimating the amount of memory can result in poor resource utilization. Typically, if a VM of 10GB is allocated to a physical host with 10GB of available memory, the host machine will not allocated any other VM. If this VM uses all of the whole memory, 100% of the time, but no other resource just as much, such as CPU, computational capacity is being wasted. The situation gets worse in common situations, where the VM uses its entire memory for only a small percentage of the time. In this case, RAM memory and CPU are wasted.

On the other hand, underestimating the amount of memory can result in performance degradation since, in a memory shortage, swap space is used, which is slower than memory access and thus delays the application's execution. This degradation will occur even if there is free memory in the host machine, since the VM is not able to access memory that is not allocated to it.

Managing memory efficiently according to the workload can result in better resource utilization without losses in performance. To do so, it is necessary to analyze and understand how applications in VMs behave and how they are impacted when memory elasticity is applied in a cloud system.

1.2 Objectives

This work proposes a tool to increase the throughput of a cloud system environment, based on analysis of behaviours of a set of applications towards memory and swap usage. This multi-tier tool uses the vertical memory elasticity capability of VMs to maintain performance and to avoid memory over-provisioning. In a cloud environment, with data

centers incorporating increasing numbers of physical machines and even higher numbers of virtual machines, an efficient manager focusing on better resource utilization and avoiding performance drops can result in higher throughputs.

1.3 Contributions

The main contributions of this work are listed below:

- A study of parameters and behaviors of VMs to identify metrics and policies for a vertical memory elasticity tool;
- Based on the study, the design and implementation of a guest OS independent tool and platform to dynamically manage VM's memory and running state, detailing its main components and features;
- Validation of the tool, which is able to increase the throughput of cloud systems when possible without any previous knowledge of the applications running in the VMs.

1.4 Structure of the dissertation

This dissertation is organized as follows: Chapter 2 summarizes some related work and describes other approaches that also tackle vertical memory elasticity; Chapter 3 presents experiments to identify and monitor controllable memory related factors that influence the performance of an application, in order to define metrics and policies to manage VMs; Chapter 4 explains in detail the mechanics of the Memory Elasticity Controller (MEC), a tool to manage VMs aiming to increase the system's throughput; Chapter 5 present some experiments with MEC and results with synthetic applications that represents different memory patterns and also, with the Parsec benchmark [4]; and finally, Chapter 6 draws a few conclusions and indicates some future work directions.

Chapter 2

Related work

Over the last few years, a growing interest has been paid to elasticity in cloud systems. The survey in [11] covers many works on cloud computing elasticity, and addresses different aspects such as definitions, metrics, tools, and existing solutions. It classifies the solutions for elasticity in methods and models. Methods (or actions), can be Horizontal, where instances, such as virtual machines, applications or containers, are added/removed from the environment, or Vertical, where resources, such as processing, memory or storage, are added/removed from a single virtual machine. Migration is also mentioned as a method, which consists of moving a VM or application from a physical server to another. The Horizontal scaling is reported as the most widely used approach to provide elasticity.

As for methods, the article points out that the literature tackles the problem in reactive or in proactive manner. The reactive approach responds to the current load, and does not anticipate them, while the proactive approach uses forecasting techniques to adapt to the future workload, usually based on the history of the load.

The article concludes by reporting open issues, such as elastic approach, stating that most of the researches has been done in the use of static threshold, where unsuitable values may lead to instability of the system. Instead, it proposes dynamic thresholds that adapts based on changes of the application's behavior.

Baruchi and Midorikawa present a concise survey on vertical memory elasticity [10] and compares two techniques, one based on the Exponential Moving Average and the other based on Page Faults. Although their work advocates that with Exponential Moving Average, memory could be used more efficiently, the Page Faults technique as the main criteria to allocate or remove memory led to better performance.

The Vertical Elasticity Manager (VEM) [13] implemented an elasticity rule with the

aim of maintaining an user-defined percentage of free memory, called the Memory Over-provisioning Percentage (MOP), available in the VM. The goal is to avoid thrashing in the virtual memory subsystem (swapping pages from memory to disk) by scaling up or down the VM's memory so as to maintain a statically pre-defined but sufficient amount of memory free in the VM. If the percentage of free memory is lower than 80% or greater than 120% of the MOP, the VM's memory allocation is increased or decreased accordingly. As an example, when using a MOP of 10%, the elasticity rule will only be triggered when the free memory of the VM is lower than 8% or greater than 12% of the memory allocated to it. Then the new VM memory size is calculated with the used (or consumed) memory and the MOP percentage. Not only can MOP be troublesome for the user to define, but a high MOP leads to wasted memory while a low MOP could hurt performance. Although the aim is to avoid swap, the proposal fails to cope should this actually occur (since the rule for free memory does not apply). An additional mechanism is needed to rapidly increase the VM's memory – and VEM assumes that sufficient memory is available on the host for this allocation.

The dynamic memory scheduler system (DMSS) [16], captures the actual memory assigned to each VM, the amount of physical memory being accessed, the free memory and number of page faults in order to identify the memory activeness and performance of each VM. Again this approach is based on swap avoidance through the maintenance of a free memory cushion. A free memory threshold is used to classify VMs in terms of memory abundance and shortage, so that the DMSS can reclaim memory from abundant VMs to give to VMs with a shortage. The reclamation process is gradual, verifying page faults as memory is removed, and may require several scheduling periods. This process may have a significant adverse impact on performance of the VMs with a memory shortage. Concerning memory distribution when it is not possible to satisfy all VM's requirements, memory is provided to the VM with the highest requirement until this amount is not the largest. This criteria can lead to even higher degradation. Furthermore, when there is no memory available for redistribution, the authors suggest only to migrate VMs.

The Ginkgo framework presented in [14] models application behavior under different loads and memory sizes, and uses this information to calculate the desirable memory assignment for the VMs. The main difficulty with their approach is the need to profile the applications to be executed so that performance indicators (e.g. transactions or requests per second, processing time, etc.) are identified. It also assumes that there is enough memory available to attend the VMs.

While the above mentioned works tackle vertical memory elasticity, [21] considers the number of vCPUs in a VM as a mean to implement vertical elasticity. Their work specifies a Layered Performance Model in order to estimate demands of online applications, which is limited to the number of physical cores of each machine. Still to be done, resources like memory and I/O are not tackled by their work.

In [17], a holistic and hierarchical performance management tool that scales VMs vertically by adjusting either processing (vCPUs) or memory resources. It translates the application's Service Level Objectives (SLO) into resource requirements, by reading performance metrics (such as response time and throughput) and resource utilization (such as CPU usage and consumed memory). Those inputs are used to build and refine a model that associates the application SLO with the VM resource allocation. The relationship between application performance and resource allocation are simplified, using a liner model to estimate the underlying nonlinear relationship. Although their work considered vertical memory scaling, no study on the limited amount of memory was discussed. Also, when building a model, the solution requires a period for learning, which can be troublesome and generate a high performance degradation.

Chapter 3

An Analysis of Memory Parameters

To help define the design and implementation of the memory manager, this chapter presents a series of experiments that aim to identify the relevant parameters that should be considered as well as charaterize their behaviours in order to decide how they should be used within the memory management framework.

3.1 Definitions

Let $\{pm_1, pm_2, \ldots pm_n\}$ be the set of physical machines that compose the cloud and let $\{vm_1^j, vm_2^j, \ldots vm_m^j\}$ be the set of virtual machines to be instantiated in a physical machine pm_j. As a matter of simplification, when not required, the notation vm_i^j representing the virtual machine i on the physical machine j is denoted as vm_i.

Before instantiating a virtual machine vm_i, a series of parameter values must be defined, for example, in relation to the memory, the maximum amount of memory and the initial memory allocation. Let $ma(vm_i, t)$ be the memory allocation of vm_i at time t, configured through the hypervisor. This value can be changed dynamically while vm_i is running with the Libvirt setmem option of the virsh command [3]. In the case of KVM, for example, $ma(vm_i, t)$ has an upper limit defined by the maximum memory allocation limit, $maxml(vm_i)$ for vm_i, i.e. $ma(vm_i, t) \leq maxml(vm_i), \forall t$. In order to change this limit, the VM must first be shutdown.

This work assumes that the collection of applications executing concurrently or sequentially in a VM can be represented as a workflow W_k. Associated with a workflow is the concept of its *peak dynamic range*, $pdr(W_k)$, which represents the maximum amount of data space required to load and execute W_k. For the sake of clarity in Section 3.2, only

one application or job is executed per workflow, i.e. $W_k = J_k$, but this is not a restriction. Furthermore, the initial experimental analysis was based on a synthetic test application J_k that carries out the same function, but models different memory access patterns. In the experiments, $pdr(J_k)$ is modelled by the test application allocating a vector of $ne(J_k)$ floating point numbers so that $pdr(J_k) \approx ne(J_k) \times sizeof(float)$. Thus, the difference between jobs is in the manner in which this vector is accessed, i.e. the jobs have different sized working sets $ws(J_k, t)$, with each element in the working set being accessed and operated on a number of times before the application's execution moves on to the next working set.

3.2 Identification of relevant characteristics

Our goal is to create a hypervisor-independent solution to minimise the quantity of memory allocated to each VM without adversely affecting the performance of its executing applications. The first step is to identify and monitor controllable memory related factors that influence the performance of an application, taking into consideration an environment composed of a single server running multiple VMs with limited memory and CPU resources.

These experiments were carried out on a server with two Intel Xeon X5650 2.67Ghz CPUs, with a total of 12 physical cores (hyper-threading was disabled), and 24GB of RAM. The host and guest operating systems were CentOS 6.5, kernel version 2.6.32. The hypervisor was Kernel-based Virtual Machine (KVM)[9] and libvirt was used as an API to manage the virtual infrastructure without over-committing CPUs. The results presented are based on an average of 10 executions.

3.2.1 Memory Overhead

Although vm_i is created with a memory allocation $ma(vm_i, 0)$, when the memory size is measured in vm_i via a system call like `top`, one will notice that some of the memory is not available. Let the total amount of memory visible internally by vm_i be $tm(vm_i, t) = cm(vm_i, t) + fm(vm_i, t)$, where $cm(vm_i, t)$ is the amount of memory being consumed by the running applications and guest operating system and $fm(vm_i, t)$, the free memory available, at time t. Note that the total memory $tm(vm_i, t)$ is constant if $ma(vm_i, t)$ does not change. The memory overhead $mo(vm_i)$ can then be measured as the following:

$$mo(vm_i) = ma(vm_i, t) - tm(vm_i, t) \tag{3.1}$$

Given the environment described, VMs were configured with different combinations of $maxml()$ and $ma()$ in order to measure the memory overhead, as shown in Figure 3.1. Values for $maxml(vm_i)$ ranged from 2 to 24GB, and $ma(vm_i, t)$ varied in steps of 2GB up to $maxml(vm_i)$. The results show that $mo(vm_i)$ is not constant, but rather the larger the value of $maxml(vm_i)$, the higher the overhead. While these memory overheads are around the typical value of 3% [15] of $maxml(vm_i)$, note that $mo(vm_i)$ does not appear to depend on $ma(vm_i, t)$.

For example, when $maxml(vm_i)$ is set with a value of 12288MB, $mo(vm_i)$ is 377MB, regardless of the $ma(vm_i, t)$ value, i.e. setting $ma(vm_i, t)$ with 12288MB or setting $ma(vm_i, t)$ with 1024MB, $mo(vm_i)$ will be 377MB. For the guest operating system, according to Equation 3.1, the $tm(vm_i, t)$ will be 11911MB in the first case, and 647MB in the second case, as shown in Table 3.1 for $ma(vm_i)$ from 12288MB down to 512MB.

Table 3.1: Corresponding total memory $tm(vm_i)$ and memory overhead $mo(vm_i)$, for each memory allocation $ma(vm_i, t)$, with a fixed $maxml(vm_i)$ value of 12288, in MB

$\boldsymbol{ma(vm_i, t)}$	$\boldsymbol{tm(vm_i, t)}$	$\boldsymbol{mo(vm_i)}$
12288	11911	377
8192	7815	377
4096	3719	377
2048	1671	377
1024	647	377
512	135	377

Therefore, when defining the memory allocation for vm_i, one should consider that a share of $ma(vm_i, t)$ will not be available to the guest OS. The amount lost is not related to $ma(vm_i, t)$, but rather to the maximum memory value $maxml(vm_i)$.

Concerning the free memory $fm(vm_i, t)$ for a workflow W_k in virtual machine vm_i, and considering the memory consumed by just the guest OS as $cm(vm_i, 0)$, Equation 3.1 can be rearranged as follows:

$$ma(vm_i, t) = fm(vm_i, t) + mo(vm_i) + cm(vm_i, 0) \tag{3.2}$$

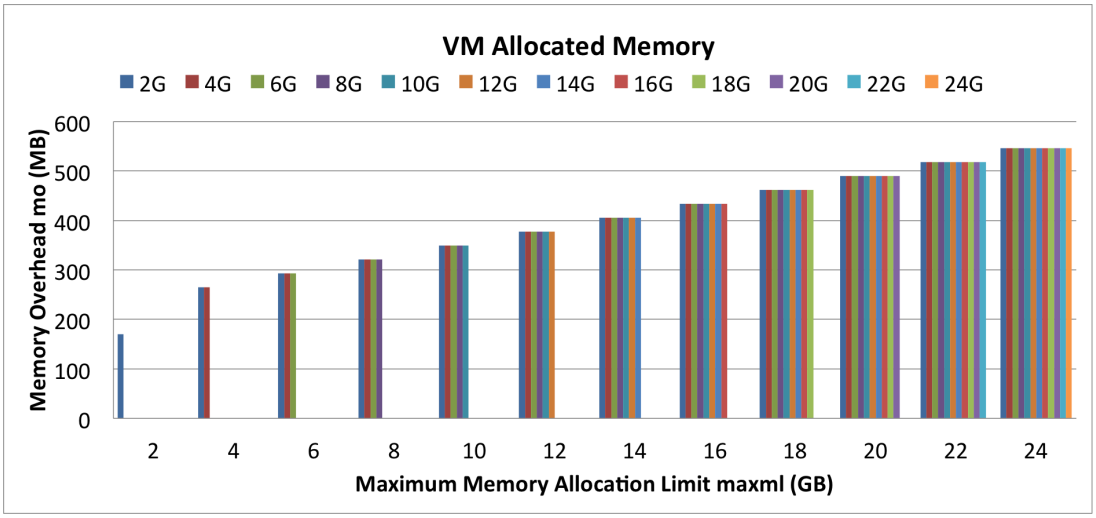

Figure 3.1: Relationship between $mo(vm_i)$ and $maxml(vm_i)$

3.2.2 Swap Activation Threshold

If the OS in vm_i needs more memory and there is no available frame, elected pages will be moved to the swap space to be reloaded later if required. Since swap is typically located on hard drives, which have a slower access times than physical memory, its use should be avoided if at all possible (by providing more memory) to prevent any additional delays to an application's execution. A potential indicator of imminent swap usage might be to monitor the amount of free memory available [13].

Let the swap activation threshold, $sat(vm_i)$, be the minimum amount of free memory $fm(vm_i, t)$ remaining in vm_i before the system starts to use swap space. The following experiment aims to identify the value of $sat(vm_i)$ and its relation to $maxml(vm_i)$ and $ma(vm_i, t)$. Let job J_1 with $ws(J_1, t) = pdr(J_1)$ (i.e. J_1 accesses its entire vector sequentially, and repeatedly, a fixed number of times) be executed on vm_1, configured with different pairs of values for $maxml()$ and $ma()$. Values for $maxml()$ ranged from 2GB to 24GB, and $ma()$ varied from 2GB up to $maxml(vm_1)$ in steps of 2GB. Also, $pdr(J_1)$ was set to $ma(vm_1, t)$ so that J_1 would require to use swap.

Figure 3.2 presents the values of $sat(vm_1)$ and their intervals of confidence, grouped in accordance with each value of $maxml(vm_1)$. In each group, each bar corresponds to a different value of $ma()$. Although there is a reasonable amount of variation around when the SO decides/needs to use swap, it seems fair to say that $sat()$ depends more on the value $maxml()$ than $ma()$.

Hence, in order to avoid swap consumption by a job J_k executing on vm_i, the memory

allocation limit should be the smallest value that satisfies:

$$maxml(vm_i) \geq mo(vm_i) + sat(vm_i) + max_{\forall t}(cm(vm_i, t)) \tag{3.3}$$

and during execution $ma(vm_i, t)$ should be adjusted dynamically according to $cm(vm_i, t)$ which reflects the current memory requirements of J_k at time t. Note that using swap may not always have a detrimental impact on the execution and thus, this memory allocation may still be an overestimate.

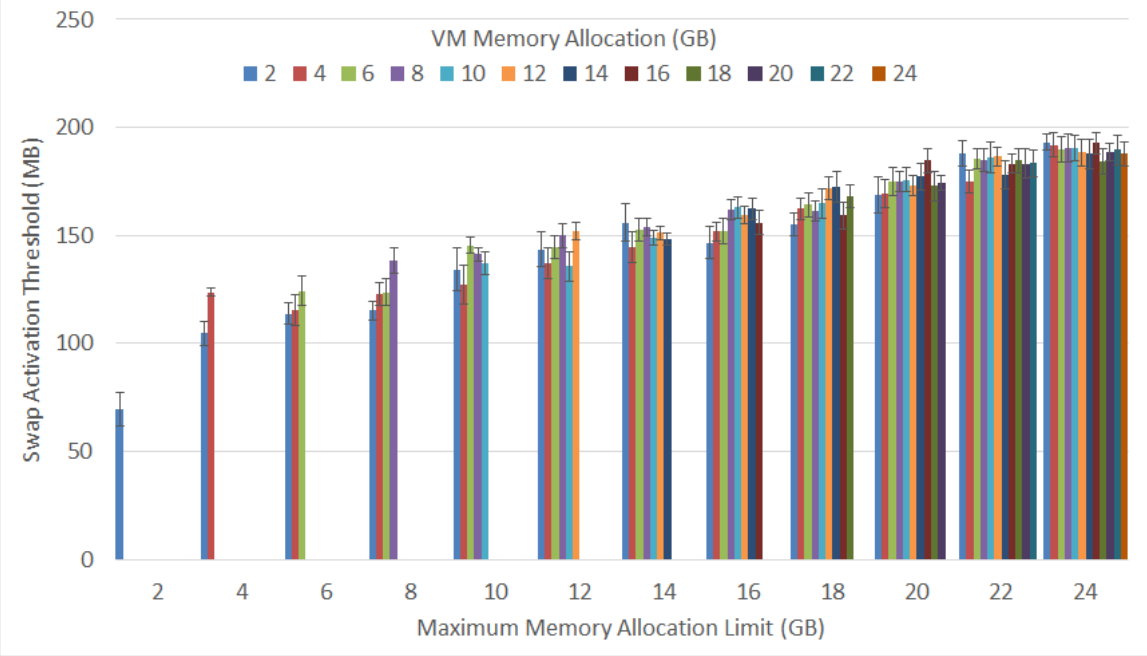

Figure 3.2: Swap Activation Threshold in relation to $ma()$ and $maxml()$) configurations.

Although it is valuable to monitor free memory in a VM, it is as important to monitor swap consumption as it may give a possible indication of performance degradation and how much additional memory might be needed. Let $sc(vm_i, t)$ be the amount of data in swap on vm_i at time t, as reported by the guest OS. Figure 3.3 shows $fm(vm_i, t)$ and $sc(vm_i, t)$ during the execution of J_1 on vm_1 with $ma(vm_1, t) = 1.8\text{GB}\ \forall t$ and $pdr(J_1) = 1.8\text{GB}$. Since the memory consumption required by this job grows steadily, the free memory decreases accordingly until $sat(vm_i)$ is reached. After this point, $fm(vm_i, t)$ remains relatively constant, while swap consumption increases to satisfy $ws(J_1, t)$. If no corrective action is taken, this might delay the job's execution.

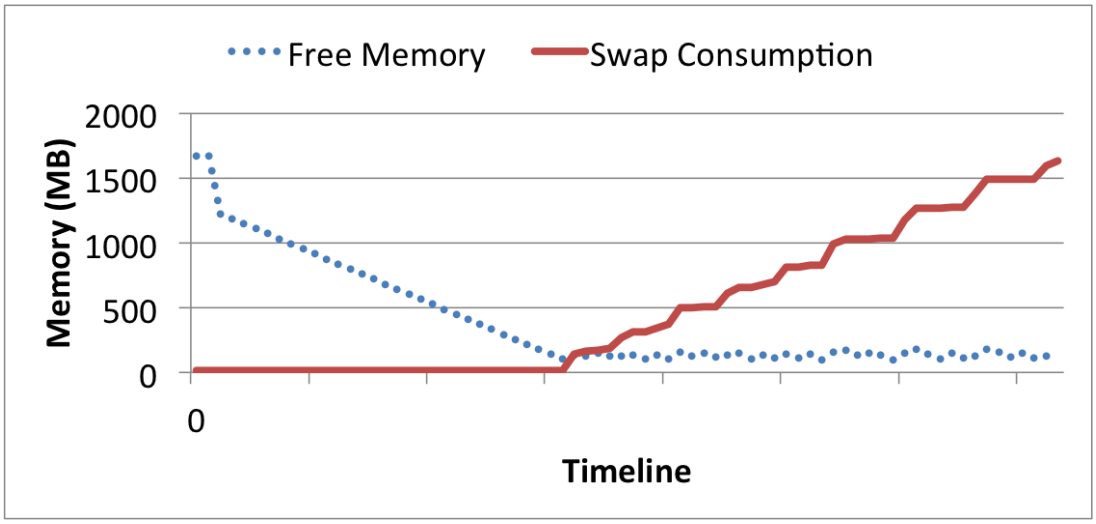

Figure 3.3: Free memory and swap consumption during the execution J_1

3.2.3 Performance impact of distinct working sets

When executing an application on vm_i, it is desirable to have the host provide the VM with sufficient allocated memory to avoid unnecessary use of swap. To introduce the impact of page swapping, let J_1, J_2 and J_3 be three jobs with the same memory requirements, i.e. $pdr(J_1) = pdr(J_2) = pdr(J_3) = 1.6$GB. However, while $ws(J_1, t) = pdr(J_1)$, i.e. J_1 traverses the entire vector four times, $ws(J_2, t) = pdr(J_2)/4$ and means that J_2 first traverses a quarter of the vector four times, before traversing the next quarter, and so on. J_3 is a mixture, first traversing the whole vector once, then each quarter a further 3 times.

This experiment aims to evaluate how the performance of the jobs differ when executing in a VM with different amounts of allocated memory. Let $maxml(vm_i) = 2GB$ so that there will be just sufficient memory when $ma(vm_i, 0) = 2$GB. Each job was executed in isolation on a VM with $ma(vm_i, t) = m\%$ of $maxml(vm_i)$, where $m = 100, 90, \ldots 30$.

Figure 3.4 presents the job execution times normalised with respect to the fastest time obtained with $ma(vm_1, t) = maxml(vm_1)$. In other words, if $et(J_k, m)$ is the execution time of $J_k, k = 1, 2, 3$ on vm_1 with $m\%$ of $maxml(vm_1)$, then the execution time ratio, $etr(J_k, m)$ is $\frac{et(J_k,m)}{et(J_k,100)}$. Note that the three jobs have the same number of operations so $et(J_1, 100) = et(J_2, 100) = et(J_3, 100)$.

The differences in performance between J_1 and J_2 (or J_1 and J_3) when m is between 40% and 90% can be explained by the larger working set of J_1. Since J_1 sweeps the memory locations in sequence, all memory pages are required in quick succession. Since

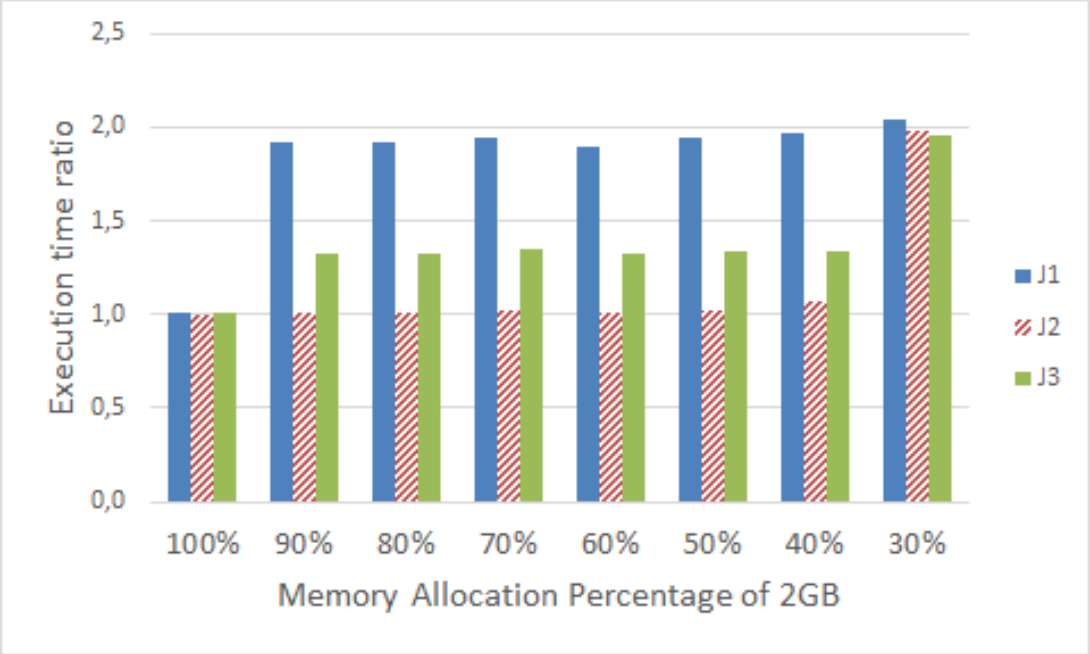

Figure 3.4: Relative job performance on a VM with different amounts of allocated memory

$maxml(vm_i) = 2GB$ and $pdr(J_k) = 1.6GB$, and given the considerations discussed in Subsections 3.2.1 and 3.2.2 that $mo(vm_i) + sat(vm_i) + cm(vm_i, 0) \approx 400$MB, then even a 10% reduction in allocated memory, forces all three jobs to require swap (since $pdr(J_k) > fm(vm_i, t)$). This reduction impacts the performance of the jobs differently, and quite significantly in the case of J_1. On the other hand, J_2 exhibits near optimal performance with as little as $m = 40\%$ of $maxml(vm_i)$. With an average working set size of $ws(J_2, t) = pdr(J_2)/4$ elements occupying ≈ 400 MB, vm_i requires at least 800MB, to run J_2 without needing to recover data from swap. When less memory is available (30% of $maxml(vm_1)$), a significant performance drop occurs, as in the case of J_1. For J_3, the first iteration causes the average 40% delay for values of m between 90% and 40% and, as for J_2, a further drop in performance occurs when m is lower than 40% when heavier swap usage occurs.

Note that $m = 30\%$ of the ideal $ma(vm_i, t)$ is not the same as 30% of memory available for the $pdr(J_j)$. If so, the performance degradation should occur when $m < 25\%$. To determine the exact percentage of $ma(vm_i, t)$ to fit 25% of the $pdr(J_2)$ (or J_3), we should consider the memory overhead $mo(vm_i)$, the $sat(vm_i)$ and guest OS memory $cm(vm_i, 0)$. In the conducted experiments, $mo(vm_1) = 171MB$, $sat(vm_1) = 60MB$ and $cm(vm_1, 0) = 175MB$. Using Equation 3.2 and adding the required memory, $ma(vm_i, t) = mo(vm_i) + cm(vm_i, 0) + sat(vm_i) + 0,25 * pdr(J_i)$, results in 806MB. Any $ma(vm_i, t)$ less than 806MB will not provide enough memory for block size 400MB, which is 25% of the $pdr(J_2) =$ 1.6GB. In experiment above, when $m = 40\%$, $ma(vm_i, t) = 2048MB * 0,4 = 819MB$,

which is higher than 806MB, and therefore, no degradation for J_2 occurs. On the other hand, when $m = 30\%$, $ma(vm_i, t) = 2048MB * 0,3 = 614MB$, and since 614MB is less than 806MB, there is degradation in performance for J_2.

3.2.4 Evaluating Swap Usage

Taking a closer look at swap, Figure 3.5 presents the memory utilisation and swap consumption, $sc(vm_i, t)$, of J_1, J_2 and J_3 during the execution time of the longest job J_1, as reported by the guest OS. The jobs were executed in a VM with a quantity of memory equivalent to 70% of $maxml(vm_1)$, which based on Figure 3.4 causes J_1 and J_3 to be delayed.

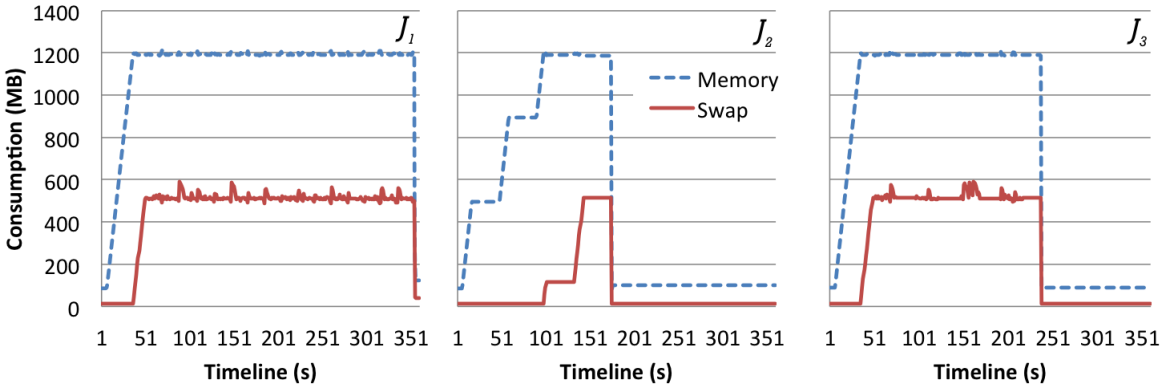

Figure 3.5: Memory utilisation and swap consumption for J_1, J_2 and J_3

From Figure 3.5, note that the jobs have the same peak memory and swap consumption, around 1200 MB and 510 MB, respectively. Interestingly, J_2 does indeed use swap without affecting the execution time, as shown in Figure 3.4. To differentiate between the executions of J_1 and J_3, one must look carefully at the subtle noise or fluctuations ($< 20\%$) in $sc(vm_i, t)$. This indicates that data is being exchanged between memory and disk. While there is a stepped increase in swap consumption for J_2, there is no noise during the time interval, thus indicating good data locality. Initially, J_3 accesses the entire data vector in the same way as J_1 causing the use of swap around the 50 seconds mark, but finishes earlier because it swaps in and out fewer pages. Without capturing the amount of data or duration that data was being written/read to/from swap, $sc(vm_i, t)$ alone does not help differentiate between the executions of J_1 and J_3.

To delve a little further, let swap-in $si(vm_i, t)$ and swap-out $so(vm_i, t)$ represent the amount of data read from and written to swap since the last VM boot, respectively, as provided by the guess OS. Figure 3.6 reports the delta values of swap-in and swap-

out between two successive measurements for the three jobs, i.e. the values reported are $si(vm_i, t) - si(vm_i, t-1)$ and $so(vm_i, t) - so(vm_i, t-1)$, respectively. While J_1 is constantly swapping in and out pages, J_3 uses swap when changing working sets. From J_1 and J_3 we see that simultaneous occurrences of swap in and swap out has high impact on performance.

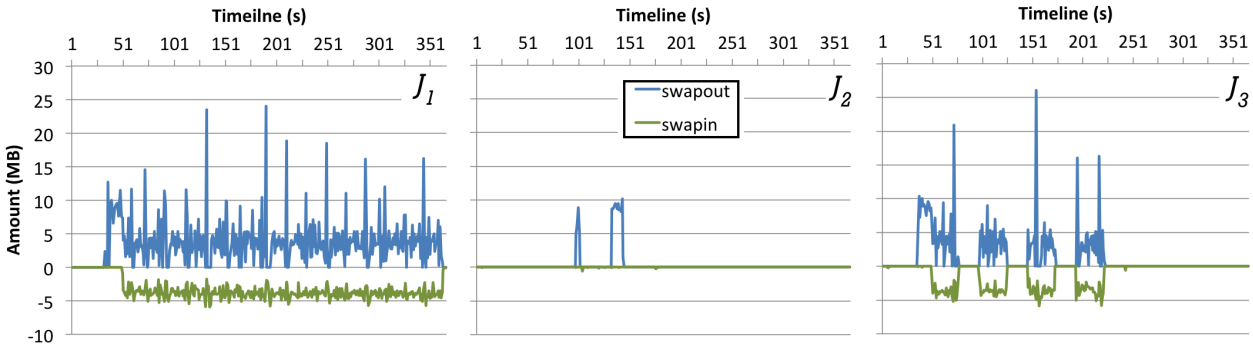

Figure 3.6: Swap-out and swap-in for J_1, J_2 and J_3

To capture this behavior, let $siso(vm_i, t)$ be the swap-in and swap-out indicator for vm_i, calculated by counting the number of concurrent swap-in and swap-out events within the previous scheduling interval of I_{host} seconds up to time t, as follows:

$$siso(vm_i, t) = \sum_{k=t-I+1}^{t} f(k) \tag{3.4}$$

where function $f(k)$ is defined as:

$$f(k) = \begin{cases} 1 & \text{if } si(vm_i, k) > si(vm_i, k-1) \wedge so(vm_i, k) > so(vm_i, k-1) \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

Figure 3.7 shows the calculated $siso(vm_i, t)$ indicator for the same experiment with $m = 70\%$. The indicator varies according to the fluctuation of swap-in and swap-out, and can signal, in the case of J_1, when page swapping is taking place.

3.2.5 Collective Impact of Swap Usage in Concurrent VMs

To investigate if memory constrained VMs affect other VMs executing on the same host, Figure 3.8 presents the execution times of two jobs of type J_1, each running in their own VM, vm_1 and vm_2, with $ma(vm_1, t) = ma(vm_2, t) = 2$GB. The following configurations were considered: [1G & 1G] where both VMs receive jobs with $pdr(J_i) = 1$GB; [1G & 2G]

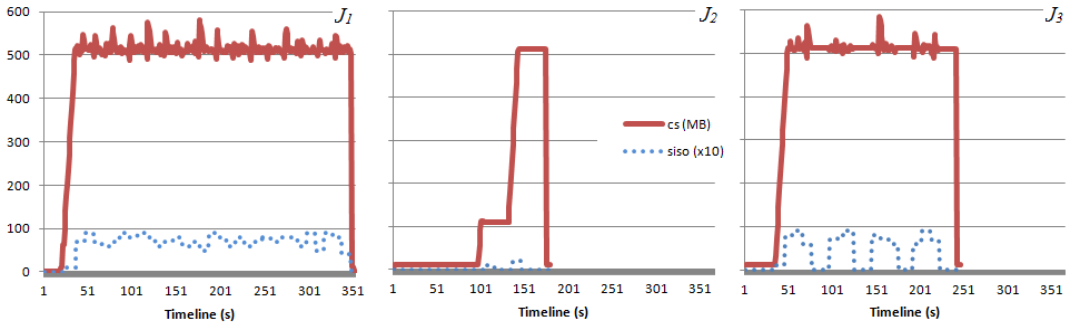

Figure 3.7: The $siso(vm_i, t)$ indicator and swap consumption for $m = 70\%$

vm_1 and vm_2 receive jobs with $pdr(J_i)$ equal to 1GB and 2GB, respectively; and [2G & 2G] each VM receives jobs with $pdr(J_i)$=2GB. Thus, in this experiment, both VMs have enough free memory to run jobs with $pdr(J_i) = 1$GB but not with $pdr(J_i) = 2$GB. Note that, with enough memory, the execution times of both configurations of jobs would be the same since the job with twice the vector size, $pdr(J_i) = 2$GB, has only half of the number of iterations.

As expected, the jobs in configuration [1G & 1G] achieve their minimum execution times, while in configuration [1G & 2G], the second job had its time doubled due to the use of swap. However, when both jobs require swap in their respective VMs, the times are proportionally worse and even slower than a sequential execution. This is caused by competition for disk access on the host, given that the host has sufficient available CPUs and memory.

The results of the concurrent execution of n VMs on the same machine, where $n = 2, \ldots 20$ and each vm_i was allocated 2GB of memory to execute the same job J_1 with $pdr(J_1) = 2$GB, are shown in Figure 3.9. Executing these VMs concurrently (the line named "swap" since their jobs use swap) is worse then executing them sequentially. Note that the physical machine has sufficient available memory to support 10 VMs, after which swap on the host is activated and this accounts for the increased gradient. Even if there was free memory available in the host (as is the case for less than 10 VMs), execution times are still slow since there is no manager to allocate this spare memory appropriately. While running VMs sequentially (which will commit less host memory) is better, any management tool should try to reduce the number of VMs in swap and avoid swap usage on the host. The line "ideal" identifies the obtainable performance if the host were to have enough physical memory (n×2.4GB) to run the n virtual machines concurrently without them having to use swap.

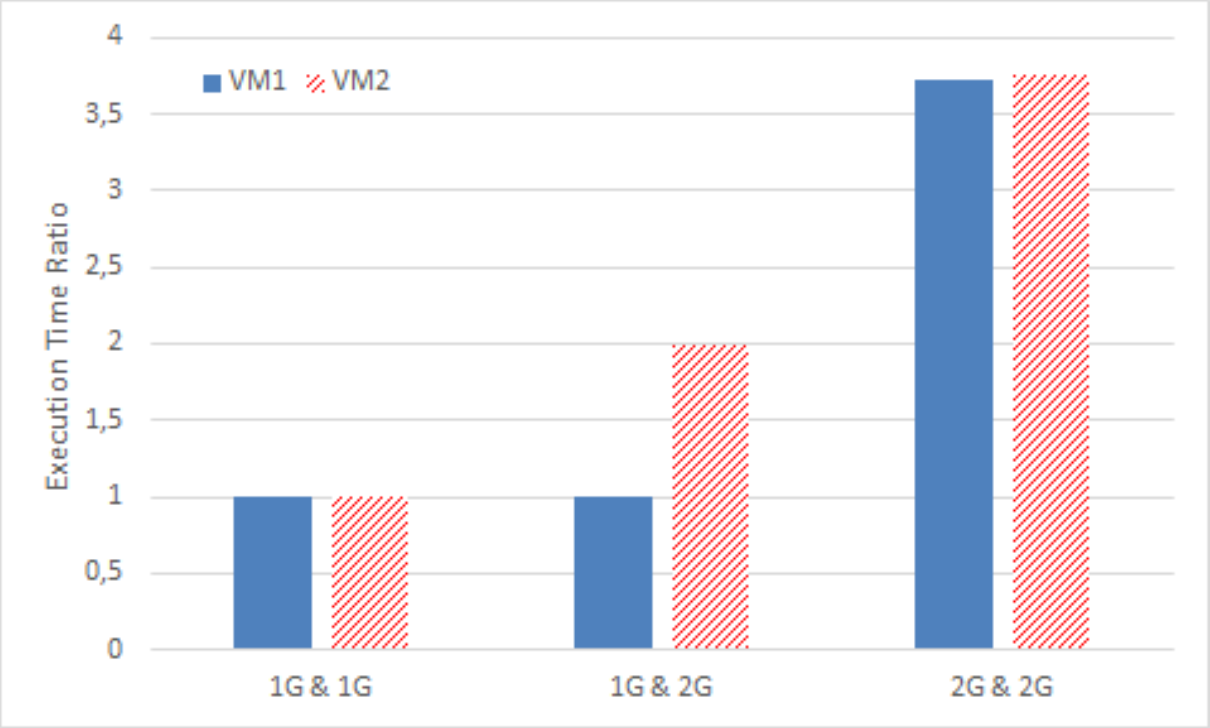

Figure 3.8: Concurrently VMs with and without memory constraints

3.2.6 Summary

This chapter identified metrics to be used by a management tool to control dynamically the amount of memory allocated to each VM during their concurrent execution with other VMs. In the next chapter, a new tool based on the insights obtain here will be presented.

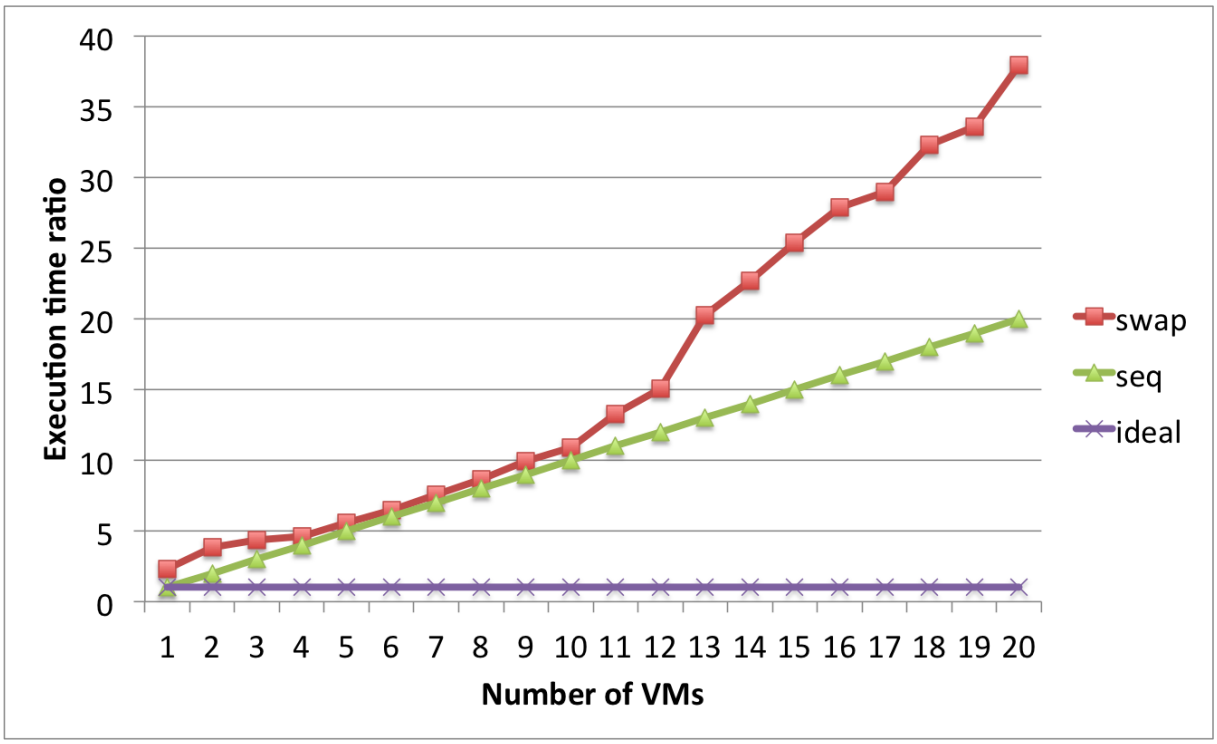

Figure 3.9: Sequential and concurrent execution of VMs, when using swap and not (ideal execution)

Chapter 4

MEC - A Memory Elasticity Controller

As technology pushes up processor core counts in servers and resource providers aim to maximize utilization through improved consolidation rates, memory is becoming relative scarce. Chapter 3 highlighted the performance impacts of insufficient memory on applications and the need to manage memory allocation in multi-tenanted virtualized environments. To address this, the vertical Memory Elasticity Controller (MEC) dynamically manages the memory allocated to VMs, and their corresponding states, running in a cloud environment via a three level hierarchy: the virtual machine layer (VML); the physical machine layer (PML); and the cloud layer (CL), as seen in Figure 4.1. This chapter presents an overview of the MEC framework, starting with an explanation of these layers.

4.1 MEC layers

In the VML, a $Monitor_i$ ((a) in Figure 4.1) is deployed in each VM vm_i and is responsible for gathering and sending information reported by the guest operating system to its local PML. At the PML, on each physical host machine pm_j, a $HostManager_j$ (element (c)) is responsible for receiving requests to instantiate new VMs and allocating physical resources to them. Periodically, each $HostManager_j$ consumes the buffered information obtained by the monitors of the VMs running on pm_j (element (b)) to decide how the host's available memory will be distributed among them during the next scheduling window. While new requests for incoming VMs are placed in the $VMQueue_j$ (element (d)) and moved to the $ActiveVMs_j$ when chosen for execution, the $HostManager_j$ can also move running VMs from the $ActiveVMs_j$ to the $VMQueue_j$ should their execution be suspended (paused).

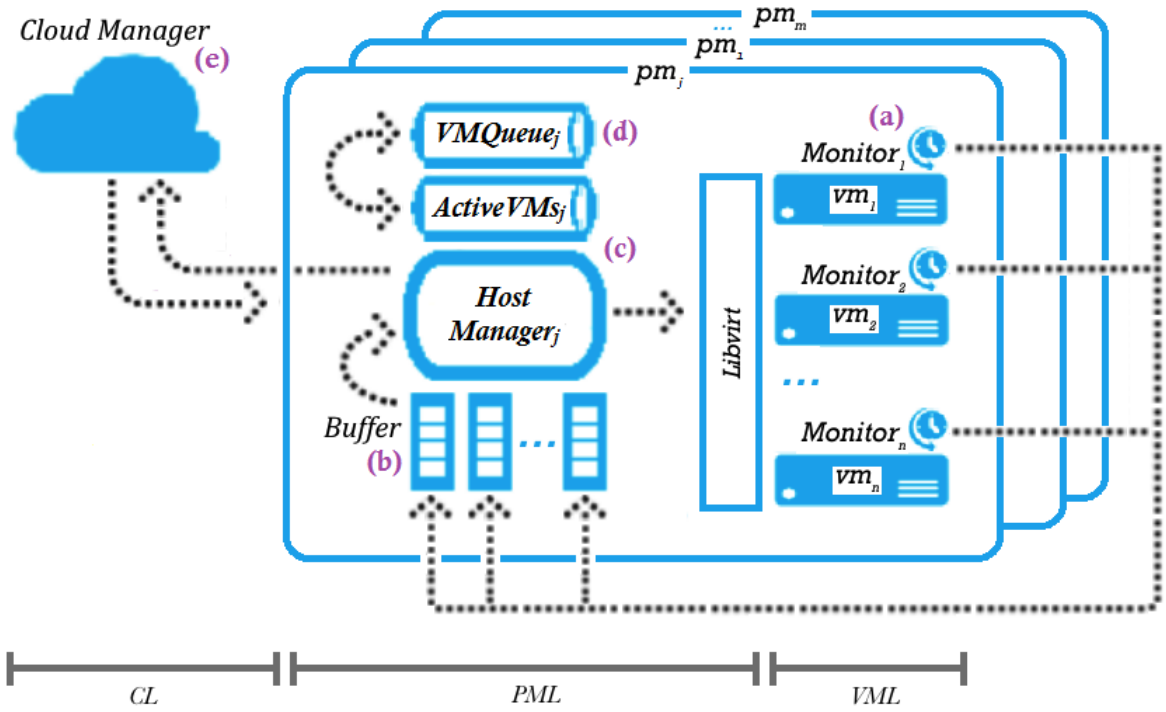

Figure 4.1: The Memory Elasticity Controller (MEC) Architecture

Information regarding the state of the host machine pm_j is sent to the CL where the $CloudManager$ (element (e)) decides to which $HostManager_j$ requests to execute new VMs should be sent, taking into consideration resource availability. In addition, the CL may also define the migration of previously scheduled VMs between physical machines. Following the typical cloud system hierarchy, this multilevel architecture separates issues that needs to be addressed (load balancing at the CL; resource (memory) sharing and local VM throughput at the PML, and; application performance at the VML) to improve the overall global cloud system throughput.

Figure 4.2 shows the communication between the layers of MEC. In the CL layer, the $receiveVmReq$ receives the VM execution request from the user, while $receivePmInfo$ receives information regarding the state of each host machine, sent by the $sendPmInfo_j$ at the PML layer. Also at the PML layer, the $receiveVmExec_j$ receives VM execution requests for pm_j sent by the Cloud Manager, and the $receiveVmInfo$ receives information collected and sent by $Monitor_i$ deployed in each virtual machine running on pm_j.

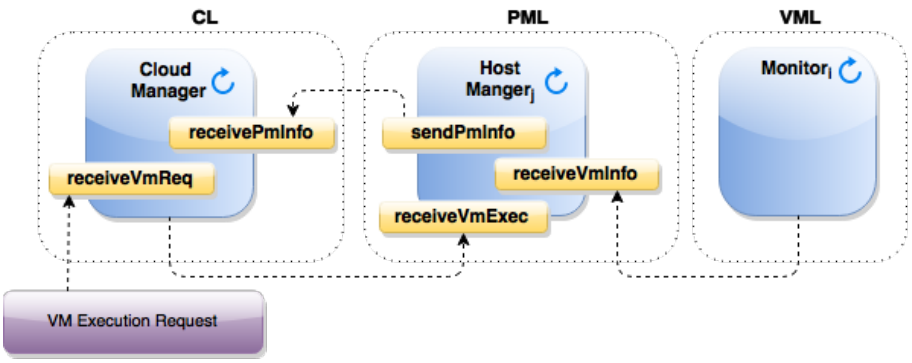

Figure 4.2: Communication between the component and their threads of the layers of MEC

4.2 The Host Manager

The main objective of $HostManager_j$ is to find an appropriate memory allocation $ma(vm_i, t)$ for each running VM vm_i at time t, and at intervals of I_{host} seconds, based on the recent memory consumption of vm_i and the combined requirements of the other running VMs on pm_j. To do so, $HostManager_j$ analyses the information collected by the monitors during the last scheduling time interval $[t - I_{host}, t]$ and, if necessary and possible, adjusts $ma(vm_i, t)$ of each vm_i with the purpose of avoiding a performance degradation of the VM's respective applications during the next scheduling period $[t, t + I_{host}]$.

At time intervals of $I_{monitor}$ seconds, each $Monitor_i$ in vm_i, collects and sends the following information obtained from the vm_i's guest OS:

- $tm(vm_i, t)$: total memory of vm_i at time t;
- $fm(vm_i, t)$: free memory of vm_i at time t;
- $si(vm_i, t)$: amount of swap-in since the last vm_i boot at time t;
- $so(vm_i, t)$: amount of swap-out since the last vm_i boot at time t.

Note that $I_{host} > I_{monitor}$ so that a time series of data related to memory usage during the time window $[t - I_{host}, t]$ can be analyzed. Initially, a simple linear regression is used to predict vm_i's memory requirement at $t + I_{host}$, but other more sophisticated schemes will be investigated in future work.

4.2.1 Virtual Machine State Transitions

At every interval I_{host}, the $HostManager_j$ considers the monitoring information placed in the buffers and the lists of inactive and active VMs, $VMQueue_j$ and $ActiveVMs_j$, respec-

tively. Actions can be taken for each vm_i in pm_j, depending on its current state, defined by $vstate(vm_i, t)$. Figure 4.3 shows the VMs states considered and possible transitions among them.

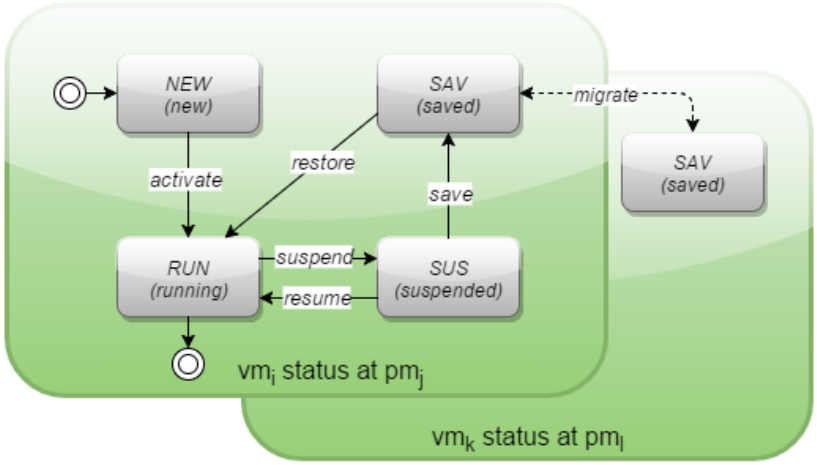

Figure 4.3: VM status and actions

When a VM request is first submitted to $HostManager_j$ by $CloudManager$, vm_i is inserted in $VMQueue_j$ and its state is set to NEW. When a decision to instantiate a VM in a NEW state for execution is taken, its state is switched to running (the RUN state) and the VM is moved from $VMQueue_j$ to $ActiveVMs_j$. Depending on the system load and available free memory in the host machine pm_j, a running VM may be suspended, having its state set to SUS. This state indicates that a VM is not running (no use of CPU), but still maintains the memory allocated to it. This state serves to prevent an application from suffering significant performance degradation by being forced to use swap. The intention is to return the VM to execution (i.e. the RUN state) shortly when enough memory can be found to meet its requirements. From a SUS state, a VM can become SAV (saved), where the VM's memory and context is saved to disk, and its allocated memory is then released to be distributed to other VM's in need. VMs in a SAV state may also be migrated to other physical machines, depending on CL decisions to improve overall performance. From the SAV state, the VM can also return to a RUN state and continue its execution. Note that the estimated amount of memory required by the VM was calculated prior to its suspension. Table 4.1 shows the VMs states and their descriptions, where the column "Description" summarizes the events that led the VM to the respective state, column "Memory" reports whether there is memory allocated to it, and the last column reports the queue in which the VM resides.

Table 4.1: Virtual machines states $vstate(vm_i, t)$ and descriptions

State	Description	Memory	Location
NEW	New VMs that were submitted from the CL, received by the Host Manager, and are waiting to be initiated	Not allocated	$VMQueue_j$
RUN	VMs that were started by the Host Manager and are currently running	Allocated	$ActiveVMs_j$
SUS	Running VMs that were suspended due to performance reasons. If resumed, it continues execution where it left off	Allocated	$VMQueue_j$
SAV	Suspended VMs that were saved to disk to release memory for other VMs. If restored, it continues execution where it left off	Not allocated	$VMQueue_j$

4.2.2 The Host Manager Algorithm

Algorithm 1 summarizes the sequence of operations carried out by $HostManager_j$ at each scheduling interval. The memory allocated to VMs that terminated during the last interval will be freed and returned to the host. The amount of available free memory at time t on the actual physical machine pm_j is read from $hfm(pm_j, t)$ and stored in the *current host free memory* variable $chfm$ (line 2). With this in hand, the host manager first attempts to meet the predicted requirements of each running VM (line 3). This function, described in detail later in Subsection 4.2.4.1, decides which VMs will have their requests for additional memory upheld, based on $chfm$ and memory to be released by other VMs. VMs whose requests cannot immediately be fulfilled are suspended and moved to the $VMQueue_j$. In order to keep delays to attend requests to a minimum, these changes to already running VMs are committed through the libvirt API layer (line 4) before the host manager progresses on to manage the updated list of inactive VMs, in line 5. At this point, the host is possibly short on free memory to allocate to the VMs waiting in this list. Tough decisions may be required and the strategy adopted is described in Subsection 4.2.4.2. Finally, any proposed changes are implemented in line 6 and the $HostManager_j$ sends the updated $chfm$ and $VMQueue_j$ to the CL (line 7) before being put to sleep until next scheduling interval time $t + I_{host}$ (lines 8 to 9).

4.2.3 The Host Manager Parameters

The main objective of $HostManager_j$ is to update the amount of memory allocated $ma(vm_i, t)$ to vm_i at each time step t, based on its prior memory consumption. To acheive this, the $HostManager_j$ analyses the information stored in $Buffer_j$ during the time interval $[t - I_{host}, t]$ and specifies an amount of memory, $ma(vm_i, t)$, which should

Algorithm 1 $HostManager_j$

```
1: while true do
2:    chfm ← hfm(pm_j, t);
3:    manage_active_VMs(ActiveVM_j, chfm);
4:    commit_changes(ActiveVM_j);
5:    manage_inactive_VMs(VMQueue_j, chfm);
6:    commit_changes(VMQueue_j);
7:    send_pm_info(chfm, VMQueue_j);
8:    actual = get_time();
9:    sleep(I_host − (actual − t));
10: end while
```

be allocated to each vm_i and that should be enough to avoid any significant performance degradation during the next time period $[t, t + I_{host}]$.

At this PM layer, two parameters are calculated: memory shaping $ms(vm_i, t)$, which aids the calibration of allocated memory to vm_i and also helps to avoid swap usage, and; the swap-in/swap-out indicator, $siso(vm_i, t)$, of vm_i, proposed in this work to detect performance degradation caused by using swap.

4.2.3.1 Memory Shaping

This work proposes the *memory shaping* factor $ms(vm_i, t)$ as the amount of memory to be added (in the case of a shortage of memory in the VM) or subtracted (in case of a surplus of memory) to the current memory allocation $ma(vm_i, t)$ in order to attend vm_i's memory requirement during the next scheduling interval. This estimated factor is derived with the aim to avoid swap usage and, consequently, to execute the applications with minimal overhead and while improving throughput.

Given the information collected by all monitors $Monitor_i$ from VMs running in a physical machine pm_j from time $t - I_{host}$ to current time t, the $HostManager_j$ calculates the difference between the consumed memory (cm) at moments $t - I_{host}$ and t as:

$$\Delta_{cm}(vm_i, t) = cm(vm_i, t) - cm(vm_i, t - I_{host}) \tag{4.1}$$

Using a simple linear extrapolation, the expected amount of available free memory in vm_i at time $t + I_{host}$ (in the future) is based on $\Delta_{cm}(vm_i, t)$, i.e. the memory consumption during the next interval is assumed to be the same as the current one:

$$efm(vm_i, t + I_{host}) = fm(vm_i, t) - \Delta_{cm}(vm_i, t) \tag{4.2}$$

Note that, although a more precise estimation method could have been used to calculate the memory to be consumed, this work opted to adopt a simplistic but generic estimation method and focus on the elasticity framework rather than having to consider the different behaviors of specific applications.

Recall from Section 3.2.2 that the swap activation threshold $sat(vm_i)$ is the minimal amount of free memory remaining in vm_i before the system starts to use swap space. In order to derive the memory shaping value $ms(vm_i, t)$ while avoiding swap usage, the free memory should be at least equal to the swap activation threshold $sat(vm_i)$, i.e., $efm(vm_i, t + I_{host}) + ms(vm_i, t) \geq sat(vm_i)$. Since we do not want to add memory unnecessarily, then

$$ms(vm_i, t) = sat(vm_i) - efm(vm_i, t + I_{host}) \tag{4.3}$$

Substituting Equation 4.2 in Equation 4.3, it follows that

$$ms(vm_i, t) = sat(vm_i) - (fm(vm_i, t) - \Delta_{cm}(vm_i, t)) \tag{4.4}$$

Lets consider the following three memory consumption scenarios:

Scenario 1: $\boldsymbol{\Delta_{cm}(vm_i, t) > 0}$ and $\boldsymbol{ms(vm_i, t) > 0}$

Figure 4.4 shows the free memory collected at times $t - I_{host}$ and t, and the $sat(vm_i)$ limit. In this scenario, it is assumed that memory was consumed i.e. $\Delta_{cm}(vm_i, t) > 0$, causing free memory to drop from $fm(vm_i, t - I_{host})$ to $fm(vm_i, t)$. At time $t + I_{host}$, since we expect that the same amount of memory will be consumed, the estimated free memory $efm(vm_i, t + I_{host})$ is predicted to be $fm(vm_i, t) - \Delta_{cm}(vm_i, t)$. However, in this case, $efm(vm_i, t + I_{host})$ would be below $sat(vm_i)$ and therefore, in order to avoid swap usage, memory should be added so that the free memory would at least be $sat(vm_i)$, i.e. memory shaping $ms(vm_i, t)$ is set to $sat(vm_i) - efm(vm_i, t + I_{host})$, resulting in Equation 4.4.

Scenario 2: $\boldsymbol{\Delta_{cm}(vm_i, t) > 0}$ and $\boldsymbol{ms(vm_i, t) \leq 0}$

Figure 4.5 shows when $\Delta_{cm}(vm_i, t) > 0$ and $efm(vm_i, t + I_{host})$ is above $sat(vm_i)$ by applying Equation 4.4. In this case, memory can be released, keeping the current free memory $fm(vm_i, t)$ above the $sat(vm_i)$ value. Based on the estimation, the amount left

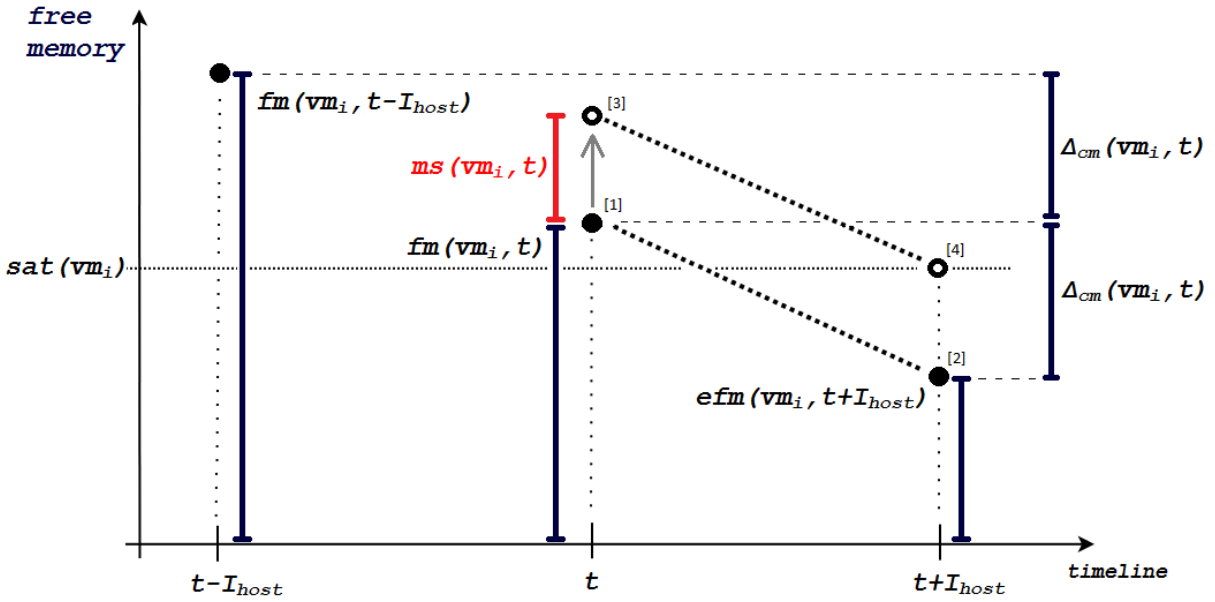

Figure 4.4: Calculating positive memory shaping $ms(vm_i, t)$ for $\Delta_{cm}(vm_i, t) > 0$. The current free memory $fm(vm_i, t)$ is shown at (1). The estimated free memory $efm(vm_i, t + I_{host})$ is calculated at (2). The memory shaping $ms(vm_i, t)$ is then calculated at (3), to avoid letting the free memory fall below the $sat(vm_i)$ value before the next scheduling event at $t + I_{host}$ (4).

above $sat(vm_i)$ will be just enough to meet the expected memory consumption $\Delta_{cm}(vm_i, t)$ until the next scheduling event at $t + I_{host}$ without activating swap.

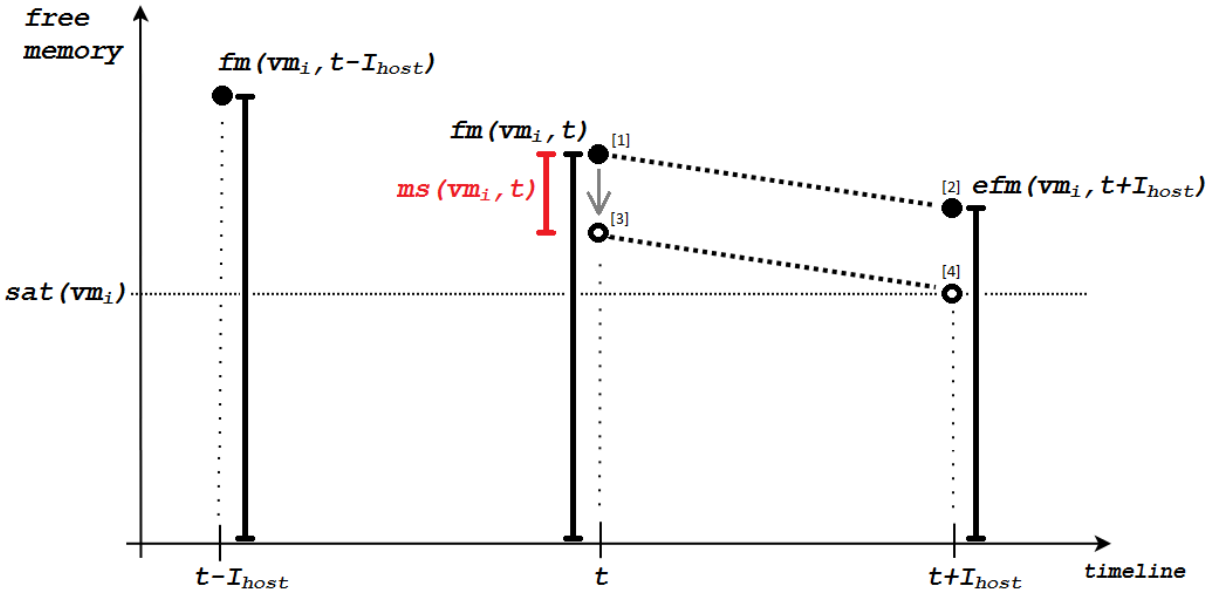

Figure 4.5: Calculating memory shaping $ms(vm_i, t)$ when $\Delta_{cm}(vm_i, t) > 0$. The current free memory $fm(vm_i, t)$ is shown at (1) and the estimated free memory $efm(vm_i, t + I_{host})$ at (2). The memory shaping $ms(vm_i, t)$ is calculated at (3) to avoid leaving "idle" unused memory but still keeping free memory above $sat(vm_i)$ limit until next scheduling time $t + I_{host}$ at (4).

Scenario 3: $\Delta_{cm}(vm_i, t) \leq 0$

The case where $\Delta_{cm}(vm_i, t)$ is negative means that memory consumption has decreased between $t - I_{host}$ and t. In this case, if $ma(vm_i, t)$ is decreased so that the estimated free memory $efm(vm_i, t + I_{host})$ equals the $sat(vm_i)$ value, the current free memory $fm(vm_i, t)$ would in turn fall below this limit and could result in the need to use swap or worse cause the VM to crash. Figure 4.6 shows a diagram where the memory consumption over two scheduling periods is considered to drop.

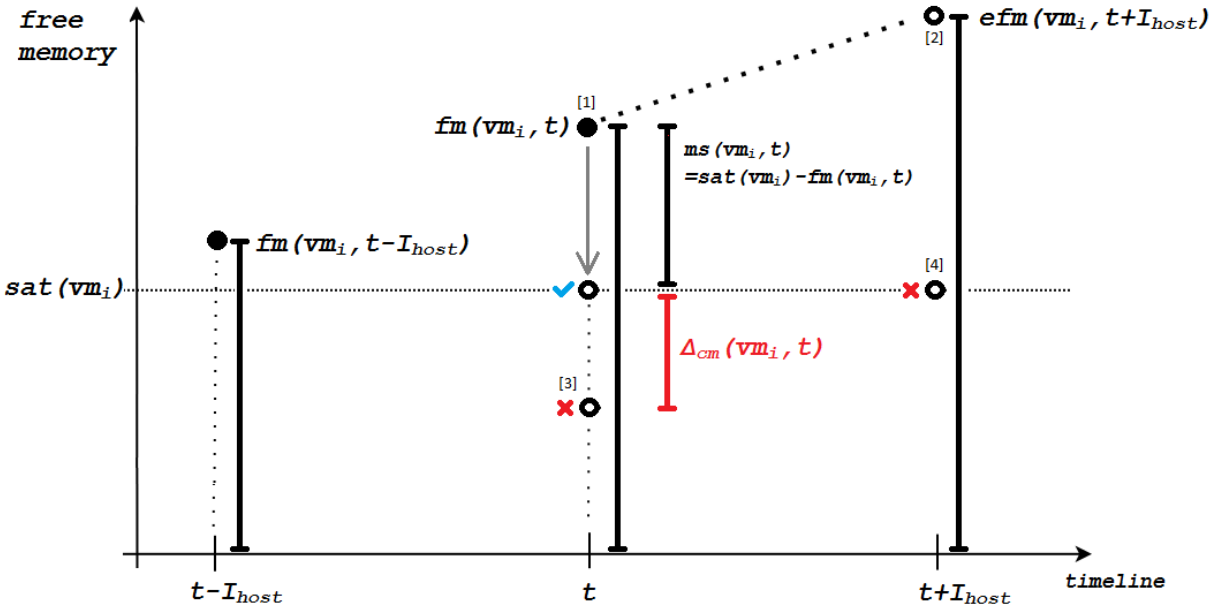

Figure 4.6: Calculating memory shaping $ms(vm_i, t)$ when $\Delta_{cm}(vm_i, t) < 0$. The current free memory $fm(vm_i, t)$ is shown at (1) and the estimated free memory $efm(vm_i, t + I_{host})$ at (2). If the $ms(vm_i, t)$ calculation where to consider the negative $\Delta_{cm}(vm_i, t)$ and keep the free memory equals to the $sat(vm_i)$ value at the time $t + I_{host}$ (in the future (4)), then the current memory allocation $ma(vm_i, t)$ would be decreased by $ms(vm_i, t)$ in such a way that would leave $fm(vm_i, t)$ below $sat(vm_i)$ (at (3)) at the current time t.

In this case, the $\Delta_{cm}(vm_i, t)$ is not considered and the memory shaping $ms(vm_i, t)$ is set with the amount that will adjust the current free memory $fm(vm_i, t)$ to the $sat(vm_i)$ value. From Equation 4.4. thus considering $\Delta_{cm}(vm_i, t) = 0$, it follows that:

$$ms(vm_i, t) = sat(vm_i) - fm(vm_i, t) \tag{4.5}$$

In summary, the value for memory shaping is the amount of memory that should be added to the current memory configuration $ma(vm_i, t)$, and represents the minimal amount that should be given to or the maximum amount that should be taken from vm_i in order to maintain the amount of free memory equal or above the swap threshold $sat(vm_i)$. A positive $ms(vm_i, t)$ indicates there is memory shortage and $ma(vm_i, t)$ should

be increased, while a negative value indicates memory excess and $ma(vm_i, t)$ could be decreased.

On the other hand, when memory consumption increases and free memory $fm(vm_i, t)$ reaches the swap activation threshold $sat(vm_t)$, the amount of free memory becomes constant while swap consumption increases, as seen in Section 3.2.2 (Figure 3.3). In this case, the $ms(vm_i, t)$ value should also consider the amount of swap memory, otherwise it will results in a value lower than what it should be. Specifically, if swap-out is positive, the memory allocated to the VM is not enough and therefore, the amount of memory swapped-out should be added to $ms(vm_i, t)$. Also, a negative value of consumed memory should be discarded, since occasionally the measurement of memory can capture the deallocation of space being moved to swap as memory being freed for release. The amount of memory swapped-out $totalso(vm_i, t)$ is defined as follows: since $so(vm_i, t)$ returns the amount of swap-out space since last boot, the difference in values between time $t - I_{host}$ and t gives the amount of data written to the swap in that period. Therefore, between time period $[t - I_{host}, t]$, the total swap out on vm_i is given by:

$$totalso(vm_i, t) = so(vm_i, t) - so(vm_i, t - I_{host}) \tag{4.6}$$

One final adjustment to $ms(vm_i, t)$ may be necessary, in relation to $maxml(vm_i)$. Recall that the maximum memory allocation limit $maxml(vm_i)$ is the upper limit for $ma(vm_i, t)$. When $ms(vm_i, t)$ is positive, the final $ma(vm_i, t)$ can exceed the $maxml(vm_i)$ value. In this case, $ms(vm_i, t)$ should be limited as follows:

$$ms(vm_i, t) = maxml(vm_i) - ma(vm_i, t) \tag{4.7}$$

so that the new $ma(vm_i, t + I_{host})$ is equals to $maxml(vm_i)$.

Finally, Algorithm 2 presents the calculation of the memory shaping function $ms(vm_i, t)$, based on the swap consumption in the last scheduling period, and considering the values of consumed memory at time $t - I_{host}$ and t, the current free memory $fm(vm_i, t)$ and the $sat(vm_i)$ value. The algorithm first calculates the difference between the consumed memory between $t - I_{host}$ and t and resets it if it is negative (lines 1 to 3). The total memory swapped-out is also calculated in line 5. The memory shaping value is then calculated in line 6, based on Equations 4.4, 4.5 and 4.6, where $\Delta_{cm}(vm_i, t)$ is ignored if lower than zero, and $totalso$ is added if higher than zero. Finally, the amount of memory added or subtracted to the VM memory allocation $ma(vm_i, t)$ respects the limit $maxml(vm_i)$ (lines 8 to 9).

Algorithm 2 $memory\_shaping(vm_i, t)$

1: $\Delta_{cm}(vm_i, t) = cm(vm_i, t) - cm(vm_i, t - I_{host})$;
2: **if** $\Delta_{cm}(vm_i, t) < 0$ **then**
3: $\Delta_{cm}(vm_i, t) \leftarrow 0$
4: **end if**
5: $totalso(vm_i, t) = so(vm_i, t) - so(vm_i, t - I_{host})$;
6: $ms(vm_i, t) \leftarrow sat(vm_i) - (fm(vm_i, t) - \Delta_{cm}(vm_i, t)) + totalso(vm_i, t)$;
7: /* cannot exceed $maxml(vm_i)$ */
8: **if** $ms(vm_i, t) + ma(vm_i, t) > maxml(vm_i)$ **then**
9: $ms(vm_i, t) \leftarrow maxml(vm_i) - ma(vm_i, t)$;
10: **end if**
11: return $ms(vm_i, t)$

4.2.3.2 The swap-in and swap-out indicator

The swap-in/swap-out indicator, $siso(vm_i, t)$, has been proposed as an attempt to identify the specific fluctuations in swap usage, during the time period $[t - I_{host}, t]$, that can cause significant performance degradation. As seen in the experiment in Subsection 3.2.4, concurrently occurring swap-in and swap-out should be avoided. For the purpose of detecting this particular behavior while executing a job in vm_i, readings of swap-in and swap-out are collected at intervals of $I_{monitor}$ by each monitor $Monitor_i$ during the last scheduling interval $t - I_{host} + 1$ to t.

As previously defined, let $si(vm_i, t)$ and $so(vm_i, t)$ be the volume of data swapped-in and swapped-out, respectively, since vm_i was last rebooted, as indicated by VM's OS at time t. But the $siso(vm_i, t)$ indicator is calculated by only counting occurrences of simultaneous swap-in and swap-out activity, as shown in Algorithm 3, based on Equations 3.4 and 3.5 from Section 3.2.4.

Algorithm 3 $swapin\_swapout(vm_i, t)$

1: $siso(vm_i, t) \leftarrow 0$;
2: **for** $k = t - I_{host} + 1, \ldots, t$ **do**
3: **if** $(si(vm_i, k) > si(vm_i, k - 1) \wedge (so(vm_i, k) > so(vm_i, k - 1))$ **then**
4: $siso(vm_i, t) \leftarrow siso(vm_i, t) + 1$;
5: **end if**
6: **end for**
7: return $siso(vm_i, t)$;

4.2.4 Memory State Transitions

By requiring a dynamic approach to identify changes over time of a variable number of unknown executing applications, MEC monitors changes to free memory and swap usage through the VM's operating system over the last scheduling interval and, by extrapolating memory consumption over the next scheduling interval, decides how to adjust the memory allocation of each VM. While this approach may be effective to predict when to increase the memory allocated to a VM, it is less so when memory could be recovered from VM, i.e. when an application or applications could actually execute with less memory without losing performance. The current mechanism relies on the VM's applications to free memory, and is unable to detect that the memory page working set of an application has reduced in size. To address this issue, our model is associated with concept of VM memory states.

The memory state, denoted as $mstate(vm_i, t)$, associated with vm_i when the VM state $vstate(vm_i, t) = RUN$, can be one of the following: in flux, FLU, when the memory allocation of the VM is fluctuating, i.e. memory was added to or removed from the VM at the last scheduling interval; stable, STB, when there was no variation in memory consumption and no fluctuations in swap-in and swap-out (known as *stable memory condition*); held, HLD, is when the VM is considered to be a candidate for a non voluntary reduction in allocated memory, or; frozen, FRZ, when an attempt to remove memory from the VM caused an increase in swap-in and swap-out, and consequently $ma(vm_i, t)$ should be returned to its previous value. A VM's initial state is FLU, while $scount(vm_i)$ counts the consecutive intervals in the current state.

The *stable memory condition* is true when the memory shaping $ms(vm_i, t)$ is zero, meaning that vm_i is not releasing nor requiring memory, and when the swap-in/swap-out indicator $siso(vm_i, t)$ is also zero, i.e.:

$$stable(vm_i, t) \leftarrow (ms(vm_i, t) = 0 \text{ AND } siso(vm_i, t) = 0) \tag{4.8}$$

This work proposes to carefully extract memory from VMs that have been stable for a predetermined number of consecutive intervals, $slimit(STB)$, and only when memory is required by the Host Manager. A VM will transition to STB when the stable memory condition is true except if the VM was recently frozen ($mstate(vm_i, t - I_{host}) = FRZ$). If VM remains stable for $scount(vm_i) = slimit(STB)$ scheduling iterations, the VM transitions to HLD otherwise, at earlier interval, it would have returned to FLU.

Once in HLD, the VM will be tested to see if memory can be removed without adversely affecting the performance of the VM as follows: $PERC$ is a predefined percentage of the actual memory allocation $ma(vm_i, t)$ to be extracted from vm_i. This amount of memory will be removed and held in $reserved(vm_i)$ (it is not added to the available free memory of pm_j) until the next scheduling interval so that the $HostManager_j$ has a chance to detect if this change caused requests for memory or swap usage within the VM. If the swap-in/swap-out indicator $siso(vm_i, t)$ remains zeroed, the VM transitions from HLD to STB otherwise to FRZ.

Arriving in the STB state from HLD means that the VM continues executing without signs of performance degradation, so the reserved memory $reserved(vm_i)$ is released and will be made available to the $HostManager_j$ to allocate to other VMs on the same machine. Transitioning to FRZ state from HLD means the extracted memory should be returned to vm_i and, to avoid hurting performance again, the VM is frozen for $slimit(FRZ)$ scheduling intervals, preventing renewed attempts in the short term to forcefully remove memory from this VM. However, through normal execution, should a frozen VM begin to exhibit concurrent swap-in and swap-out or memory release, it will transition to the FLU state to be a candidate to have its memory allocation adjusted. Figure 4.7 presents the memory state diagram and Table 4.2 shows memory state transitions.

The purpose of using memory states is to save memory by finding the lowest possible $ma(vm_i, t)$ to fit the application working set $ws(W_k, t)$ without generating a high drop in performance. Since memory extraction can cause delay, it must be carefully executed. For this, time limits are defined for certain memory states. For state STB, the memory will only be extracted if the *stable memory condition* is true during a number of scheduling periods, defined as $slimit(STB)$. Defining $slimit(STB)$ with a low value may cause memory to be extracted from VMs that are not stable, which is undesirable due to unpredictable memory usage. On the other hand, setting a high value will take longer to reach the desired $ma(vm_i, t)$.

For state FRZ, it is assumed that the lowest possible $ma(vm_i, t)$ was found, since the memory allocation value is the one prior to the extraction that generated $siso(vm_i, t)$ to raise. The idea is to keep the VM in this state for $slimit(FRZ)$ scheduling periods, so that further memory extractions, which may cause undesired delay, is avoided. However, this state can change under certain conditions: when $siso(vm_i, t)$ is higher than zero, meaning that the $ma(vm_i, t)$ is not enough to fit the $ws(W_k, t)$ of the running application; or when $ms(vm_i, t)$ is lower than zero, which indicates that memory was released, and the working

set is now different. In both cases, the VM is set to FLU and a new $ma(vm_i, t)$ will be recalculated since the new working set is not known. Now, another situation that may occur is that, if a VM is in FRZ state and $ms(vm_i, t)$ higher than zero but $siso(vm_i, t)$ is zero, at this moment no memory is added to $ma(vm_i, t)$ on an attempt to provide memory strictly when in necessary. In this case, the frozen state should not be changed since it does not indicates an actual necessity to increase memory. If the positive $ms(vm_i, t)$ is a real request for more memory (probably due to enlarging the $ws(W_k, t)$), it will most likely cause $siso(vm_i, t)$ to raise, making the memory state change to FLU. If the conditions for interrupting the FRZ state are not true, the VM leaves the state only when counter $scount(vm_i)$ reaches $slimit(FRZ)$, changing to the STB state. This is a time period to check if the VM is actually stable, before another memory extraction round begins.

The value for $slimit(FRZ)$ should also be chosen with care. Setting it with a low value may cause a higher number of attempts to extract memory bellow its working set, generating undesirable delay. On the other hand, setting it with a high value might be ineffective if the application's working set drops to a lower value than the current one, loosing opportunity for reaching the lowest possible $ma(vm_i, t)$ value and saving memory. The FLU and HLD memory states do not require any time limit.

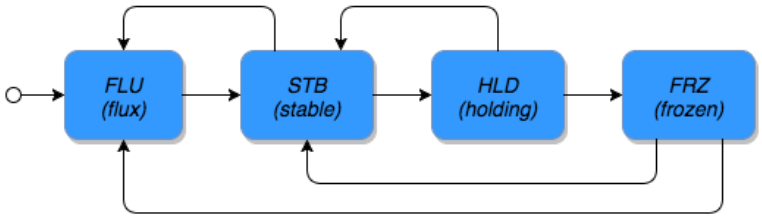

Figure 4.7: Memory State transitions diagram

Table 4.2: Memory State transitions

$mstate(vm_i, t - I_{host})$	Transition Condition	$mstate(vm_i, t)$	Action
FLU	$stable(vm_i, t)$ = TRUE	STB	–
STB	$stable(vm_i, t)$ = FALSE	FLU	–
	$stable(vm_i, t)$ = TRUE AND $scount(vm_i) = slimit(STB)$	HLD	Extract mem
HLD	$siso(vm_i, t) = 0$	STB	Release mem
	$siso(vm_i, t) > 0$	FRZ	Return mem
FRZ	$siso(vm_i, t) > 0$ OR $ms(vm_i, t) < 0$	FLU	–
	$siso(vm_i, t) = 0$ AND $scount(vm_i) = slimit(FRZ)$	STB	–

4.2.4.1 Managing Active VMs

Algorithms 4 to 8 outline the main steps to manage the memory state transitions and memory requirements of the active VMs. It first calculates the memory requirement

$ms(vm_i, t)$ and the swap-in/swap-out indicator $siso(vm_i, t)$ (lines 3 and 4) for all of the VMs in $ActiveVM_j$ and inserts each transitioning vm_i in one of the following lists according to their new memory state: $FRZlist$, which lists all vm_i that were in HLD but when they had $reserved(vm_i)$ memory extracted were no longer stable, and will transistion to the FRZ memory state; $STBlist$ stores VMs that have become stable, which can be VMs that had there memory reduced without causing swap-in/swap-out usage, or VMs coming from FLU, HLD or FRZ states (note that already stable VMs are not inserted since no action on them is necessary); $HLDlist$ lists those that have remain stable for $slimit(STB)$ intervals and are candidates to have memory extracted; and finally, the $FLUlist$ lists all of the VMs that are predicted to require or release memory, including those transitioning from STB or FRZ as well as those already in state FLU.

The VMs that were in the memory states FLU or STB are treated first (lines 7 to 16), checking if vm_i is stable and inserting it in the appropriate list. The HLD state are treated between lines 17 and 22, inserting the vm_i into the $STBlist$ list if extraction succeeded or into the $FRZlist$ list if not. The last state FRZ is verified between lines 23 and 28, checking if vm_i should return to state FLU (line 24) in case $siso(vm_i, t)$ is higher than zero or when $ms(vm_i, t)$ is lower than zero; or to state STB (line 26) when $scount(vm_i)$ reaches the limit $slimit(FRZ)$.

Although $FRZlist$, $STBlist$ and $HLDlist$ do not require any specific ordering, the $FLUlist$ is divided into negative and positive $ms(vm_i, t)$ values. Negative $ms(vm_i, t)$ are treated first (since memory will be freed), updating the $chfm$ with memory that is being released. VMs with a positive $ms(vm_i, t)$ are ordered in accordance with decreasing values of $siso(vm_i, t)$, breaking ties with the smallest $ms(vm_i, t)$. This ordering gives MEC the opportunity to analyze first the VMs with the highest swap-in and swap-out usage accumulated during the period $[t - I_{host}, t]$. The reasoning behind this is that by providing more memory to these VMs will hopefully increase overall performance. Breaking ties with the smaller $ms(vm_i, t)$ requirement will encourage a wider distribution of available memory among VMs and will lead to a smaller collective use of swap.

The management of VMs in $FRZlist$ is shown in Algorithm 5, with $reserved(vm_i)$ being the amount of memory previously extracted and is now being returned to vm_i, i.e., added back to the current $ma(vm_i, t)$. The $mstate(vm_i, t)$ is set to FRZ and its counter $scount(vm_i)$ reset.

Algorithm 6 shows the management of VMs in $STBlist$, where the amount of memory $reserved(vm_i)$ is released if the previous memory state was HLD. Any VM transition

Algorithm 4 *manage_active_vms*$(ActiveVMs_j, chfm)$

for all $(vm_i \in ActiveVMs_j)$ **do**
  // Calculating ms and siso:
  $ms(vm_i, t) \leftarrow memory\_shaping(vm_i, t)$;
  $siso(vm_i, t) \leftarrow swapin\_swapout(vm_i, t)$;
  $scount(vm_i) \leftarrow scount(vm_i) + 1$;
  $prev \leftarrow mstate(vm_i, t - I_{host})$;
  if $prev == FLU$ OR $prev == STB$ **then**
    if $stable(vm_i, t)$ **then**
      if $prev == FLU$ **then**
        $STBlist = STBlist \cup \{vm_i\}$;
      else if $scount(vm_i) == slimit(STB)$ **then**
        $HLDlist = HLDlist \cup \{vm_i\}$;
      end if
    else
      $FLUlist = FLUlist \cup \{vm_i\}$;
    end if
  else if $prev == HLD$ **then**
    if $siso(vm_i, t) > 0$ **then**
      $FRZlist = FRZlist \cup \{vm_i\}$;
    else
      $STBlist = STBlist \cup \{vm_i\}$;
    end if
  else if $prev == FRZ$ **then**
    if $siso(vm_i, t) > 0$ OR $ms(vm_i, t) < 0$ **then**
      $FLUlist = FLUlist \cup \{vm_i\}$;
    else if $scount(vm_i) == slimit(FRZ)$ **then**
      $STBlist = STBlist \cup \{vm_i\}$;
    end if
  end if
end for
manage $FRZlist$, $STBlist$, $FLUlist$ and $HLDlist$;

Algorithm 5 Management of $FRZlist$ list

for all $vm_i \in FRZlist$ **do**
  $ma(vm_i, t) \leftarrow ma(vm_i, t) + reserved(vm_i)$;
  $reserved(vm_i) \leftarrow 0$;
  $mstate(vm_i, t) \leftarrow FRZ$;
  $scount(vm_i) \leftarrow 0$;
end for

from a different previous state (FRZ, FLU or HLD) will have $mstate(vm_i, t)$ set to STB and its counter $scount(vm_i)$ is set to zero.

Algorithm 6 Management of $STBlist$ list

```
1: for all vm_i ∈ STBlist do
2:   if prev == HLD then
3:     chfm ← chfm + reserved(vm_i);
4:     reserved(vm_i) ← 0;
5:   end if
6:   mstate(vm_i, t) ← STB;
7:   scount(vm_i) ← 0;
8: end for
```

The management of VMs in $FLUlist$ can be seen in Algorithm 7. The $FLUlist$ is ordered so that VMs with negative $ms(vm_i, t)$ are placed in the beginning of the list. Then VMs with positive $ms(vm_i, t)$ are sorted by decreasing $siso(vm_i, t)$, breaking ties with the smaller $ms(vm_i, t)$ first. For each active VM, the algorithm checks if $chfm$ is large enough to attend vm_i, taking into consideration $sat(pm_j)$[1]. If there is enough free memory, the $ma(vm_i, t)$ and $chfm$ values are updated (lines 3 to 5). Otherwise, $siso(vm_i, t)$ is verified and if it is higher than zero, vm_i is suspended, i.e., $vstate(vm_i, t) = SUS$, (line 8). Suspending vm_i will not free any memory, but will avoid further degradation in performance due to its swap usage if vm_i were to continue to be executed. If $siso(vm_i, t)$ is zero, no action is performed since, even though there is not enough spare memory available to meet its projected requirement, vm_i is apparently not being adversely affected yet which indicates a reduced priority for immediate intervention. In any case, $mstate(vm_i, t)$ is set to FLU (line 11).

As seen in Section 3.2.3, depending on the application, the amount of memory may be overestimated and its reduction may not cause high degradation. For example, for job $J1$, a 10% reduction of its allocated memory caused a high degradation in performance, while a reduction of up to 60% for $J2$ did not cause any noticeable delay. Therefore, as in the case of J_2, the memory allocation might be an overestimate and this is can be an opportunity for memory extraction.

However, since it is difficult to predict precisely the behavior of the application being executed, the extraction should be carried out carefully and only when necessary. In order to avoid swap usage, the memory extracted from the vm_i is only effectively returned to the host machine in the next time step if $siso(vm_i, t)$ does not become positive.

[1]Note that this is the swap activation threshold of the physical host machine, the minimum amount of free memory remaining before system starts to use swap space

Algorithm 7 Management of $FLUlist$ list

1: $FLUlist \leftarrow$ order by $ms(vm_i, t) < 0$, then by high $siso(vm_i, t)$ breaking ties with low $ms(vm_i, t)$;
2: **for all** $vm_i \in FLUlist$ **do**
3: **if** $chfm - sat(pm_j) \geq ms(vm_i, t)$ **then**
4: $ma(vm_i, t) \leftarrow ma(vm_i, t) + ms(vm_i, t)$;
5: $chfm \leftarrow chfm - ms(vm_i, t)$;
6: **else**
7: **if** $siso(vm_i, t) > 0$ **then**
8: $action(vm_i) \leftarrow suspend$;
9: **end if**
10: **end if**
11: $mstate(vm_i, t) \leftarrow FLU$;
12: **end for**

The VMs in $HLDlist$ list are managed in Algorithm 8, where it shows the steps of the extraction mechanism for each vm_i in the list. Note that this extraction is only performed if the host's free memory $chfm$ is close to *the swap activation threshold* value of pm_j, define here as $satc(pm_j)$, where $satc(pm_j) > sat(pm_j)$ and is set in the MEC framework to indicate that the host free memory is reaching the $sat(pm_j)$ value. For example, $satc(pm_j)$ can be set as $satc(pm_j) \leftarrow sat(pm_j) \times 2$. In this case, when $chfm \leq satc(pm_j)$, the reserved memory is calculated based on a percentage $PERC$ of $ma(vm_i, t)$, and a new $ma(vm_i, t)$ is set (line 3 to 4). If $chfm$ is not "close" to $sat(pm_j)$, non voluntary memory extraction is not considered necessary. The $mstate(vm_i, t)$ is set to STB and its counter $scount(vm_i)$ is set to $slimit(STB) - 1$ (line 7 to 8), making the VM eligible for memory extraction again in the next period $t + I_{host}$ should it continue with stable memory consumption.

Algorithm 8 Management of $HLDlist$ list

1: **for all** $vm_i \in HLDlist$ **do**
2: **if** $chfm \leq satc(pm_j)$ **then**
3: $reserved(vm_i) \leftarrow ma(vm_i, t) * PERC$;
4: $ma(vm_i, t) \leftarrow ma(vm_i, t) - reserved(vm_i)$;
5: $mstate(vm_i, t) \leftarrow HLD$;
6: **else**
7: $mstate(vm_i, t) \leftarrow STB$;
8: $scount(vm_i) \leftarrow slimit(STB) - 1$;
9: **end if**
10: **end for**

4.2.4.2 Managing Inactive VMs

After the management of the active VMs, the $HostManager_j$ handles the inactive ones, i.e. those VMs that are either in a NEW, SUS or SAV state. During the lifetime of a VM vm_i, the time spent in these three states is measured by the waiting time $wtime(vm_i)$. The objective of MEC at this point is, after managing and calibrating memory of the active VMs, to decide which of the currently inactive ones, if any, can be transitioned to the RUN state given the remaining free host memory.

Firstly, the suspended VMs with the highest waiting time $wtime(vm_i)$ are considered: if the host has enough memory, that is, $chfm - sat(pm_j) \geq ms(vm_i, t)$, then memory is given to vm_i and it will be resumed. Otherwise, the suspended VM with the lowest waiting time will be saved, $vstate(vm_i, t) = SAV$. In this case, the allocated memory $ma(vm_i, t)$ of the saved VM is added to the host available memory $chfm$. Choosing to save the VM with the lowest $wtime(vm_i)$ is likely to have the least impact on a VM's execution performance. However, as future work, alternative strategies should be investigated in conjunction with the cloud layer VM scheduling algorithm.

After evaluating all of the suspended VMs in $VMQueue_j$, first the SAV VMs and then NEW VMs remaining in the list are evaluated in order of wait time, being restored or activated, respectively, if sufficient host memory is available. The priority being given to suspended VMs (note that these suspensions occurred during management of active VMs earlier in the evaluation at the current time t) is an opportunity to quickly restart VMs that are still in memory. However, if there is still not enough available memory in the host machine they will be saved and their allocated memory will be freed up. The act of suspending and resuming a VM has a lower overhead when compared to saving and restoring a VM. Finally, saved VMs are given priority over new ones in order to avoid starvation.

Algorithm 9 describes the management steps for inactive VMs. First, the VMs are ordered by $vstate(vm_i, t)$ (SUS, SAV and then NEW) breaking ties with decreasing $wtime(vm_i)$ (line 1). Then variables p and q are set with their initial values (lines 2 to 3). The p variable holds the suspended vm_p identifier with the highest $wtime(vm_i)$ to be resumed, while q variable holds the suspended vm_q identifier with the lowest $wtime(vm_i)$ to be saved. For each vm_i with $vstate(vm_i, t) = SUS$, if the host's free memory is enough to attend the VM (line 6), the memory allocation of vm_p and $chfm$ are updated, the VM is set to be resumed and identifier p receives the VM with the next highest $wtime(vm_i)$ (lines 7 to 10). Otherwise, the vm_q is saved to release memory for vm_p, and identifier q

receives the VM with next lowest $wtime(vm_i)$ (lines 12 to 15).

From lines 20 to 30, the VMs in the saved state SAV, set in earlier intervals, are managed before the NEW VMs, according to ordering in line 1. For each vm_p, if the $chfm$ is enough to attend vm_p memory requirement, it can be restored (line 23) or activated (line 25), depending on its current state SAV or NEW, and $chfm$ is updated (line 27), otherwise nothing is done.

Algorithm 9 $manage\_inactive\_vms(VMQueue_j, chfm)$

```
 1: VMQueue_j ordered by vstate(vm,t) and wtime(vm);
 2: p ← 1; // First suspended VM
 3: q ← |VMQueue_j, vstate(vm,t) = SUS|; // Last suspended VM
 4: // Manage suspended VMs:
 5: while p ≤ q do
 6:    if chfm − sat(pm_j) ≥ ms(vm_p,t) then
 7:       ma(vm_p,t) ← ma(vm_p,t) + ms(vm_p,t);
 8:       chfm ← chfm − ms(vm_p,t);
 9:       action(vm_p) ← resume;
10:       p ← p + 1;
11:    else
12:       chfm ← chfm + ma(vm_q,t);
13:       ma(vm_q,t) ← ma(vm_q,t) + ms(vm_q,t);
14:       action(vm_q) ← save;
15:       q ← q − 1;
16:    end if
17: end while
18: // Manage saved and new VMs:
19: p ← p + 1;
20: while p ≤ |VMQueue_j| do
21:    if chfm − sat(pm_j) ≥ ma(vm_p,t) then
22:       if vstate(vm_p,t) == SAV then
23:          action(vm_p) ← restore;
24:       else
25:          action(vm_p) ← activate;
26:       end if
27:       chfm ← chfm − ma(vm_p,t);
28:    end if
29:    p ← p + 1;
30: end while
```

4.2.4.3 Committing changes

After making the decisions regarding the VMs, it is necessary to commit the changes, during which the $HostManager_j$ calls Libvirt functions to effectively change the memory allocation $ma(vm_i, t)$ and to implement the VM state $vstate(vm_i, t)$, based on assigned actions

$action(vm_i)$ in the Algorithms $manage\_active\_vms()$ and $manage\_inactive\_vms()$. According to Algorithm 1, the procedure $commit\_changes()$ is called at two different moments. The first call is immediately after managing the active VMs, since a quick response from $HostManager_j$ to adjust the memory allocation for these VMs and to suspend VMs that are active with high swap-in/swap-out usage can help reduce any loss in performance especially if there is insufficient memory available on the host machine. After committing the first set of adjustments, the $HostManager_j$ will then consider which inactive VMs should release memory to meet demand, when necessary. This second call commits the adjustments related to these inactive VMs.

Both attributes $ma(vm_i, t)$ and $vstate(vm_i, t)$ of vm_i are be changed during the commitment phase. The change in memory allocation $ma(vm_i, t)$ triggers the change in the total memory $tm(vm_i, t)$ of the operating system at vm_i, providing memory to the running applications to execute without delay. VM states, $vstate(vm_i, t)$, are changed through $action(vm_i)$, which is set with one of the following values: activate, suspend, resume, save, restore and migrate. Table 4.3 shows the possible actions, their descriptions, the associated state transition and equivalent action in the **Libvirt** API.

Table 4.3: Actions and state transitions of virtual machines

Action	**Description**	**Current state**	**Next state**	**Libvirt** API
activate	Start the execution of a VM	NEW	RUN	virDomainCreate
suspend	Suspend the execution of a VM	RUN	SUS	virDomainSuspend
resume	Resume the execution of a VM	SUS	RUN	virDomainResume
save	Save the state of a VM to disk	SUS	SAV	virDomainSave
restore	Restore the state of a VM from disk	SAV	RUN	virDomainRestore
$ma(vm_i, t)$	Change $ma(vm_i, t)$ of a VM	{RUN, SAV, SUS}	{RUN, SAV}	virDomainSetMemory
migrate	Migrate a VM	SAV	SAV	virDomainMigrate

The procedure then traverses the set of VMs and calls the corresponding **Libvirt** function. Before the $HostManager_j$ is put to sleep, it must send information regarding the physical machine state to CL layer, using the procedure $send\_pm\_info()$. Each pm_j that composes the cloud system sends its $chfm$ and information regarding $VMQueue_j$ to the $receivePmInfo$ component, as shown in Figure 4.2. The CL will then make the appropriate scheduling decisions for VM allocation.

Finally, the $HostManager_j$ is suspended for the remainder of the interval, to be activated again at moment $t + I_{host}$. This is achieved calculating how long the current management cycle took, $actual - t$, where t was the moment current management round started and *actual* the current time, and subtracting this from the next interval of I_{host}, as seen in Algorithm 1, line 8.

4.3 The Cloud Manager

The cloud manager is not the main focus of this work, thus only basic outline of what might be required is discussed here. In the CL, the $CloudManager$ is responsible for receiving virtual machines requests for execution and deciding in which pm_j they should be placed, or which already scheduled VMs should be migrated between physical machines. Figure 4.8 shows the diagram of $CloudManager$. From element $receivePmInfo$, it receives information from $HostManager_j$ on each pm_j that composes the cloud, and places it in the $ActivePMs$ list. The $InactiveVMs$ set is then updated with information regarding $VMQueue_j$ of each host. The $CloudManager$ also receives VM execution requests in element $receiveVmReq$, placing them in the $VmReqQueue$ queue. At each time interval defined by I_{cloud}, the $CloudManager$ reads information from $ActivePMs$, $InactiveVMs$ and $VmReqQueue$ and decides if VM migration should be initiated, which VMs should be involved and between which physical machines, or to which physical machines new VMs should be scheduled.

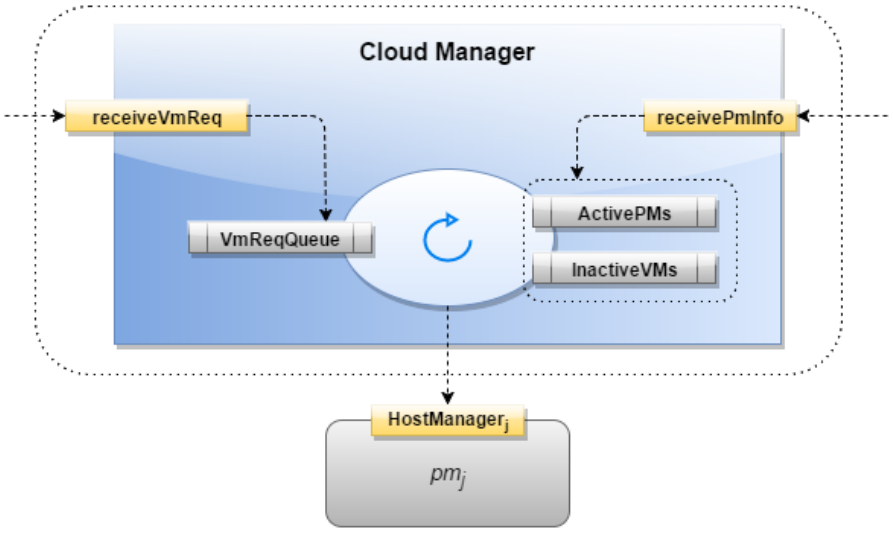

Figure 4.8: The $CloudManager$ at CL layer

The objective of the $CloudManager$ is to distribute the VMs across physical machines. Since each PM sends information of its VMs that are waiting to be executed, as well as its currently free memory, the problem is to determine which PM will receive a VM, to be first started or migrated. In this layer, it can be assumed that the total expected execution time of vm_i is known, and since the time vm_i spent executing is easily extracted from the PML (together with $wtime(vm_i)$), it is possible to calculate the remaining time needed to finish vm_i, define here as $rtime(vm_i)$.

The $ActivePMs$ list can be used to keep the $hfm(pm_j)$ (from variable $chfm$) of each

pm_j that composes the cloud. The following information are set:

- j: identifier of pm_j;
- $hfm(pm_j)$: the host free memory of pm_j.

Then, for each pm_j, the $InactiveVMs$ can be populated with the following information retrieved from the PML, regarding each VM waiting to be executed (from the $VMQueue_j$ queue):

- $wtime(vm_i)$: time spent by vm_i waiting to be executed;
- $rtime(vm_i)$: vm_i expected remaining time to finish;
- $ma(vm_i)$: the memory allocation of vm_i;
- $maxml(vm_i)$: the maximum memory limit of vm_i;
- $vstate(vm_i, t)$: state of vm_i, SAV or NEW;
- pm_j: the physical machine where vm_i is allocated;

With these information, the $CloudManager$ is able to load-balance the cloud system, and define where VMs will be executed, given the existing VMs that are waiting to be executed, together with its waiting time $wtime(vm_i)$ and remaining execution time $rtime(vm_i)$. Then, policies should be defined in the $CloudManager$ to decide how the VMs will be distributed, matching the VM's requirements to PM's memory availability.

This new problem is independent from the one treated in this work at the PML, although related by the parameters that are sent from the PML to the CL. Different policies will be investigated, defined and implemented in future work.

Chapter 5

Experimental Results

5.1 Overview

In this chapter, the results of several experiments using MEC as the execution manager are carried out, while considering different job requirements and examining different aspects of the proposed framework. The experiments are divided in main two sections. The first section aims to test specific criteria and features implemented by MEC. The second section focus on testing MEC against other approaches and implementations, using synthetic jobs and the Parsec Benchmark [4]. Within each test, it is possible to verify how MEC is able to reduce the execution delay and increase the throughput of VMs by calibrating the amount of memory allocated to each VM and controlling their concurrent execution.

These experiments were executed on the same infrastructure mentioned in Section 3.2: a server with two Intel Xeon X5650 2.67Ghz CPUs, with a total of 12 physical cores (hyper-threading was disabled), and 24GB of RAM. The host and guest operating systems were CentOS 6.5, kernel version 2.6.32. The hypervisor was Kernel-based Virtual Machine (KVM) [9] and libvirt was used as an API to manage the virtual infrastructure without over-committing CPUs. The results presented are based on an average of 10 executions.

5.2 Evaluating Specific Characteristics

In this section, some aspects of MEC framework are analyzed, including how the host memory is affected by the virtual machine's memory consumption, the frequency in which MEC is executed, the memory extraction mechanism as a way to predict the memory access pattern, the impact of the chosen priorities and the VM pause functionality.

5.2.1 Host memory consumption pattern under MEC supervision

The virtual memory ballooning technique [22] allows the hypervisor to dynamically take memory from the host machine and give it to the virtual machine, and vice-versa, by deflating or inflating the memory balloon. When the balloon is deflated by the hypervisor, more memory is available for the guest OS, whereas when inflated, memory pages are unmapped from the guest and handed back to the host. This experiment shows the behavior of host memory when using the elastic capability of MEC, and how it can create opportunities for better memory usage.

When the host has sufficient memory, virtual machines can run without the need of memory extraction, pausing or migration. However, when limited, the unsupervised use of this memory by a group of VMs can lead to poor utilization. To show this, three jobs were run in sequence on a single virtual machine and the consumed memory from the host machine was monitored. The jobs specifications can be seen in Table 5.1, where $pdr(J_k)$ is the peak dynamic range of job J_k and $ws(J_k)$ is the working set size of job J_k.

Table 5.1: Jobs specifications

	$pdr(J_k)$ (MB)	$ws(J_k)$ (MB)
J_1	2048	2048
J_2	2048	512
J_3	2048	512

Recall from Section 3.1 that J_1, J_2 and J_3 have the same number of operations to be executed, considering the same peak dynamic range $pdr(J_k)$ size. The difference between them is the manner in which the allocated vector of size $pdr(J_k)$ is accessed. Jobs J_1 and J_2 access their entire working set sequentially and repeatedly a fixed number of times before moving to the next working set. Since $pdr(J_1) = ws(J_1)$, job J_1 has only one working set, while job J_2 has four different working sets. Job J_3 is a mix of these two, accessing its entire vector like J_1 and then accessing each working set like J_2.

Figure 5.1 shows the consumed memory from the host point of view, with MEC enabled (solid line) and with MEC disabled (dotted line), running jobs J_1, J_2 and J_3 in sequence. When the memory management is not active, the host consumed memory remains constant with its maximum value. Even after J_3 is finished, if the VM continues to run without any jobs, the consumed memory at the host would still be around 14250MB. On the other hand, under MEC supervision, the host consumed memory reflects the consumption inside the VM.

The difference between the solid line and dotted line, at each moment, indicates the memory that could be used elsewhere, with, for example, others virtual machines. In this experiment, the difference between the average memory consumption is about 600MB, and shows that, with MEC, the available host memory is better used, and memory overcommitment can be avoided, thus increasing the throughput.

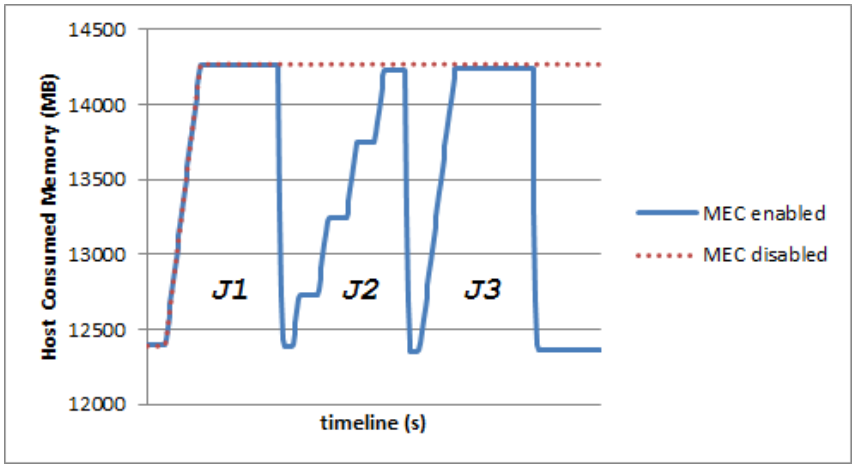

Figure 5.1: Consumed memory from the host point of view

5.2.2 Management frequency

One of the mechanisms exploited when managing vertical elasticity is the release of memory that is not being used from VMs to be made available to others, so that throughput is improved. However, this memory calibration has to be held carefully since low free memory can cause swap activation, if high memory consumption suddenly arises.

The frequency in which the Host Manager is executed, defined by I_{host}, is related with the amount of spare memory[1] that is going to be added to $ma(vm_i, t)$: the smaller the I_{host} value, the faster the response to sudden changes on memory consumption, and thus, lower amount of extra memory is needed. Nonetheless, this frequency is limited to the hypervisor response time and to the frequency of received values from the monitors in the VMs (i.e. from the guest OS).

For example, let four jobs of type J_1 be executed, but with different functions operated on each element of the vector, and with $pdr(J_1) = 1$GB. Table 5.2 shows the different memory consumption speed for each particular function, and for different intervals I_{host} of 20, 10, 5, 2 and 1 second, the required extra memory to absorb the sudden increase

[1] Amount of memory beyond the one calculated with $ms(vm_i, t)$

of memory consumption. One can note that the lower the frequency, the less additional amount of memory is required.

Table 5.2: Memory Consumption rates (GB/s) and Required Extra Memory (GB) for different functions

			I_{host}				
	Function	Rate(GB/s)	20	10	5	2	1
$J_{1:1}$	x^y	0.05	0.91	0.45	0.23	0.09	0.05
$J_{1:2}$	$x * y$	0.29	5.74	2.87	1.43	0.57	0.29
$J_{1:3}$	$= 1$	0.82	16.36	8.18	4.09	1.63	0.82
$J_{1:4}$	$memset()$	2.16	43.12	21.56	10.78	4.31	2.16

Having concluded that, we now redirect our attention to $sat(vm_i)$. In Section 3.2.2, the evaluation held identified the swap activation threshold $sat(vm_i)$ as the minimum amount of free memory remaining before the system starts to use swap space. Figure 5.2 shows the memory allocation $ma(vm_i, t)$ and the swap consumption during execution of J_1 for the function "$= 1$" and $I_{host} = 5$s. Although swap is activated at first, MEC adjusts the $ma(vm_i, t)$, and then, after that, swap is not activated anymore. Although MEC was able to efficiently manage the amount of memory allocated, during the interval in which the manager was not activated, there was an abrupt increase on swap consumption.

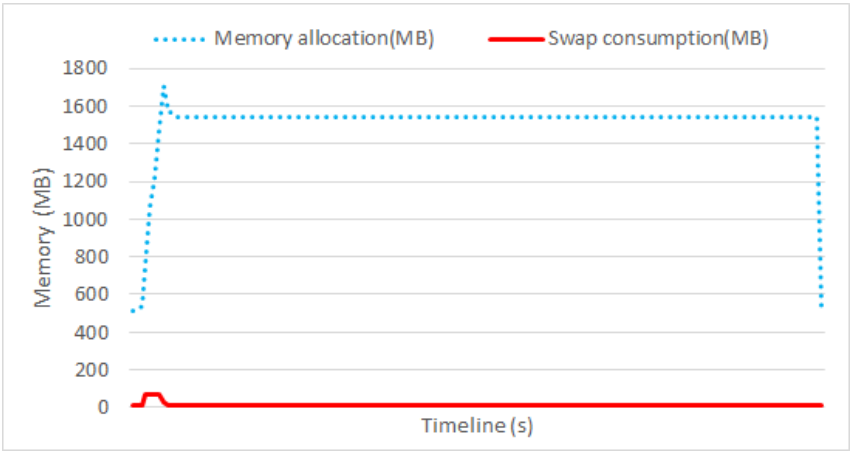

Figure 5.2: Memory and swap consumption during the execution of J_1 using the function "$= 1$" and $I_{host} = 5$s under MEC management

One strategy in order to prevent the slightest chance to access swap when executing jobs in a VM is to add an extra quantity to $sat()$, denoted as $sat_{extra}(vm_i)$. In one side, this strategy may provide good execution performance if a sudden memory consumption occurs, on the other hand, setting this extra memory to $sat(vm_i)$ to each vm_i in pm_j may be undesirable if one wish minimize memory allocation. It is important to report

that $sat(vm_i) + sat_{extra}(vm_i)$ can be used on the $ms(vm_i, t)$ calculations, instead of just $sat(vm_i)$.

In this experiment, there was a delay of 3% on the times when comparing the executions of J_1 with I_{host} = 5s under MEC management with $sat_{extra}(vm_i)$ = 0GB (Figure 5.2) and with $sat_{extra}(vm_i)$ = 4.09GB, which is necessary value of extra memory according to Table 5.2 to avoid swap activation for function "=1". It is interesting to remark that the delay drops to values less than 1% when I_{host} is set to 1s which is negligible considering long batch applications.

Although setting MEC with low I_{host} values prevents delays on execution time, it is important to evaluate MEC's overhead. For this, vm_1 received one job J_1 executing the function x^y and its $ma(vm_1, t)$ (for t =0) was statically set with enough memory to avoid swap. The virtual CPU of vm_1 was fixed to run alone in a processor of the host machine without the interference of MEC. Then the same instance was executed, but now, under MEC management using the identified $sat(vm_1)$ and I_{host} = 1s. In this case, MEC was executed in the same CPU as vm_1. Table 5.3 shows the results of both executions. The delay of 0.18% on the job's execution time shows the low overhead generated by MEC. However, if 0% of overhead is seek, MEC can be set to run in the same CPU of the hosts operating system.

Table 5.3: MEC overhead

Type	**Execution time (s)**	I_{host}(s)	**Delay (%)**
Static	1099.64	–	0
MEC	1101.66	1	0.18

5.2.3 Memory extraction mechanism

As seen in Subsection 4.2.4, when available memory is low, *Host Manager*$_j$ is able to reduce the memory allocation of vm_i even if $ms(vm_i, t)$ does not portray that memory can be reduced (i.e. when vm_i is stable). This experiment has the objective to show the impact of extracting memory from a virtual machine when it is under the stable condition.

As seen in Subsection 3.2.3, jobs behave differently under lack of memory. In that experiment, job J_1 suffered a high delay with a 10% reduction of its ideal amount of memory, whereas J_3 suffered this same delay only when reduction reached 70%. In this experiment, jobs J_1 and J_3 were executed, with MEC managing their memory and with memory extraction enabled. The specification of jobs J_1 and J_3 can be seen in Table 5.4.

The job J_3 was chosen since it represents a type of application that indicates memory consumption of $pdr(J_3)$ units, but with working set $ws(J_3) < pdr(J_3)$ only after the first iteration. In this case, monitoring its memory consumption can be misleading, because its initial memory consumption indicates that J_3 will need all the $pdr(J_3)$ amount. As for J_1, it is an example where there is practically no space for optimization, and making wrong decisions on extraction can lead to high degradation on the performance. In both J_1 and J_3, the memory extraction percentage ($PERC$ from Algorithm 8 in Section 4.2.4.1) to be reserved was set to 5% (a high value can lead to long delays, jobs sudden termination or even VM crash).

Table 5.4: Jobs specifications for memory extraction

	$pdr(J_i)$ (MB)	$ws(J_i)$ (MB)
J_1	1024	1024
J_3	1024	256

Figure 5.3 shows the memory allocation $ma(vm_i, t)$ for job J_3 during its execution time. Initially, the $mstate(vm_i)$ is set to FLU and vm_i receives its additional required memory $ms(vm_i, t)$. Between moments 47s and 121s, there is no more memory consumption and then $mstate(vm_i)$ is set to STB. When memory state continues stable until $scount(vm_i) == slimit(STB)$, memory extraction mechanism begins, gradually, until moment 534s. At this point, $siso(vm_i, t)$ rises, and reserved memory is added back to vm_i, changing $mstate(vm_i)$ to frozen (FRZ). The vm_i continues in this state until $scount(vm_i) == slimit(FRZ)$, in moment 1001s, when $mstate(vm_i) = STB$, when another extraction occurs, causing $siso(vm_i, t)$ to increase, and $mstate(vm_i)$ to set back to FRZ. Another extraction attempt is made at moments 1527s and 2011s, without success. Finally, memory is released at moment 2147s and $mstate(vm_i)$ is set to its default value flux (FLU).

Figure 5.4 shows the same experiment for J_1, where extraction stops at moment 170s, much earlier than in J_3, since its working set $ws(J_1)$ is greater than $ws(J_3)$ in the last 3 iterations. The three attempts to extract memory when running J_1 occurred at moments 670s, 1173s, 1672s due to the timeout of state FRZ, when $mstate(vm_i)$ was changed to STD and another attempt to extract memory was made, without success.

With the purpose of comparing the execution time and memory reduction of J_1 and J_3, MEC was disabled and memory allocation of vm_i was set statically with just enough memory to run J_1 or J_3 without any delay, as indicated in *Static* in column "Type" of Table 5.5. The table also shows the initial memory allocation (Initial $ma()$) when $ma(vm_i, t)$

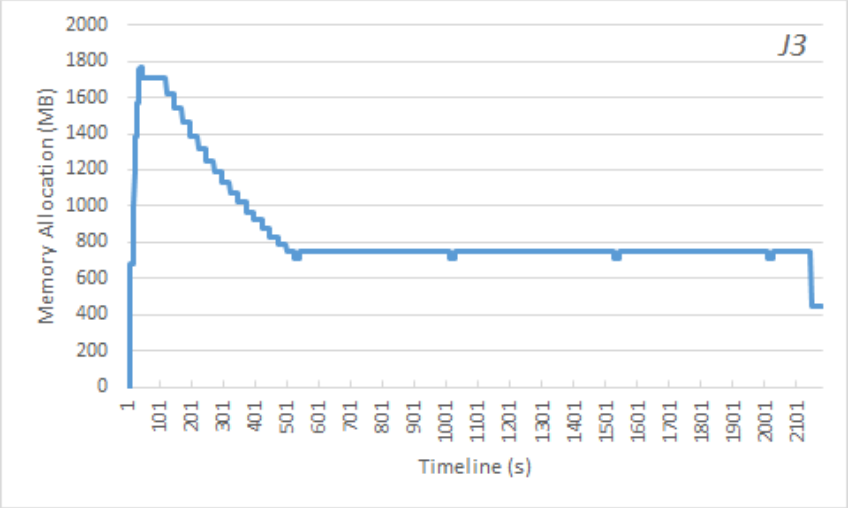

Figure 5.3: The effects on the memory allocation during J_3 execution on vm_i due to memory extraction

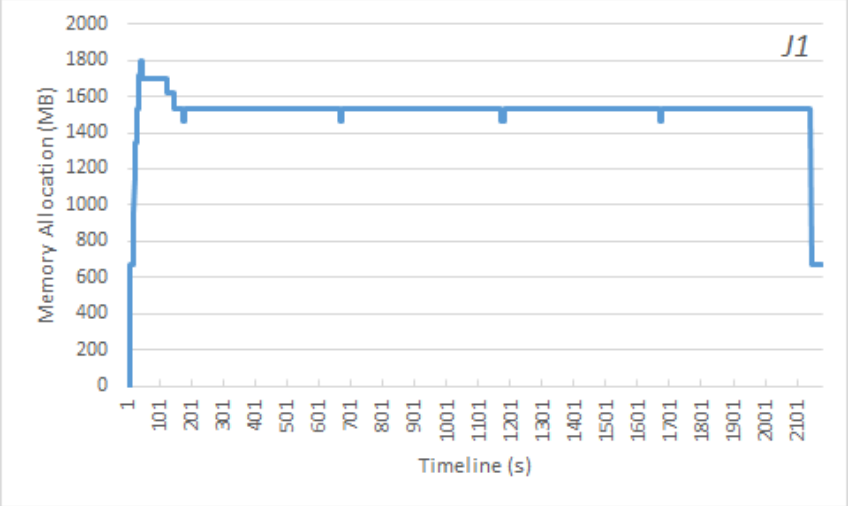

Figure 5.4: The effects on the memory allocation during J_1 execution on vm_i due to memory extraction

starts to be reduced[2], the final memory allocation (Final $ma()$) when $ma(vm_i, t)$ stops being reduced, the memory allocation percentage reduction (Reduc $ma()$), the total execution time (Time) in seconds, and the execution time percentage delay (Delay) when compared with *Stat* execution time.

Table 5.5: Jobs specifications for memory extraction

Type	**Initial *ma* (MB)**	**Final *ma* (MB)**	**Reduc *ma* (%)**	**Exec time (s)**	**Delay (%)**
Static	1510	1510	0	2119	0
J_1	1510	1510	0	2128	0.4
J_3	1510	790	48	2230	5.2

[2]Between 1700MB and 1510MB extraction is not considered since the difference is due to the extra memory set through parameter $sat_{extra}(vm_i)$

In both cases (J_1 and J_3), the memory extraction percentage $PERC$ value was set to 5%. This value should be chosen with care, since a high value can cause longer delays or jobs abrupt termination or even VM crash. Also, using a percentage value instead of a fixed value causes the amount of memory extracted to decrease as the $ma(vm_i, t)$ becomes lower. This is intended, since the lower $ma(vm_i, t)$ gets, the closer to the limit it will be, and taking too much memory at once will cause undesirable effects.

This experiment showed that MEC is able to detect the limit of the allocated memory. When extraction is possible, MEC is able to reduce the memory allocation without a high delay, and when extraction is not possible, MEC is able to react and maintain the job performance. It is import to remember that this memory extraction is only active when the host is under pressure, i.e. free memory on host machine is not enough to attend its VMs. Otherwise, there is no attempt to reduce memory and thus, no delay on the VMs execution occurs.

5.2.4 Prioritizing VMs

When the free memory of the host is not enough to attend all VMs, $HostManager_j$ decides which VMs receive more memory first. In Algorithm 7 in Section 4.2.4.1, it was proposed to give priority to choose those VMs with highest $siso(vm_i, t)$, breaking ties with the lowest $ms(vm_i, t)$. Aiming to show the importance of this priority, the following experiment was carried out: two virtual machines vm_1 and vm_2 were set to run concurrently, where vm_1 executed J_1 and vm_2, J_2. At a certain moment t, high $siso(vm_1, t)$ was detected, while for vm_2 the same did not happen, and the memory available to be distributed at moment t was such that only one VM could be attended.

Two scenarios were defined to test different prioritizing criteria: SC1, where $ms(vm_1, t) < ms(vm_2, t)$; and SC2, with $ms(vm_1, t) > ms(vm_2, t)$. Four priorities were evaluated: the VM with highest $ms(vm_i, t)$ (**High ms**); the VM with lowest $ms(vm_i, t)$ (**Low ms**); distribute the memory evenly among the VMS (**Distribute**); and the VM with highest $siso(vm_i, t)$ (**Siso**).

Figure 5.5 shows the execution time ratio[3] for each scenario and priority. Clearly, for both scenarios, picking up the next VM with highest $siso()$ is the best choice since it captures which VM is under stress mostly. The worst priority is clearly the one that distributes memory evenly, since all VMs compete for swap use.

[3]Execution time normalized with respect to the optimal time

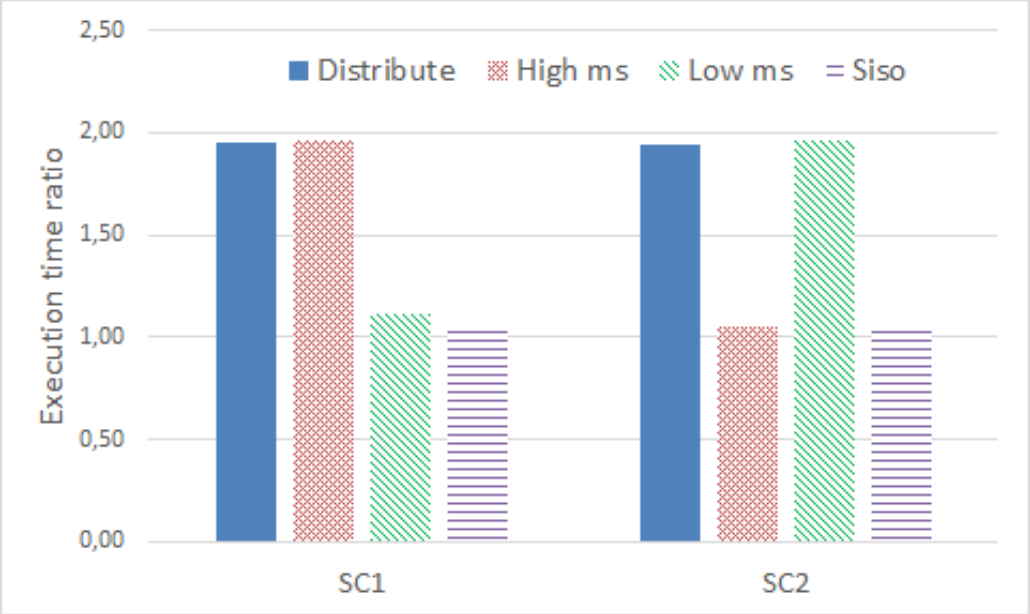

Figure 5.5: Scenarios SC1 and SC2 with different priorities for choosing active VMs

5.2.4.1 Breaking ties with lowest $ms(vm_i, t)$

A third scenario SC3 was defined where four VMs, from vm_1 up to vm_4, are experiencing the same $siso(vm_i, t)$, but requiring different $ms(vm_i, t)$ values. Each one of them executed a job of type J_1. At a certain moment t, the amount of memory required by vm_1 is equals to the summation of the amounts required by vm_2, vm_3 and vm_4 altogether, and also $hfm(pm_j)$ is just enough to attend either vm_1 or all the three remaining VMs.

Figure 5.6 shows the execution time ratio per VM of scenario SC3 for "Distribute", "High ms" and "Low ms" criteria. As can be observed, when memory is distributed equally, the execution time ratio reaches its highest value, due to the concurrent swap usage. For the criteria "High ms", vm_1 is benefited, but overall performance declines since vm_2, vm_3 and vm_4 are under high swap usage. With the criteria "Low ms", the number of VMs with their requirements met are maximized, and thus only vm_1 has its execution delayed (but its execution time ration is only 2.0), and therefore, throughput is maximized.

5.2.5 Pausing VMs

Targeting to offer an opportunity for cloud providers to increase the number of requests being attended, one can consider to execute high number of VMs into one physical machine in order to maximize the throughput. However, the concurrent memory use by many VMs can lead to long execution delays due to concurrent swap usage, depending on the jobs

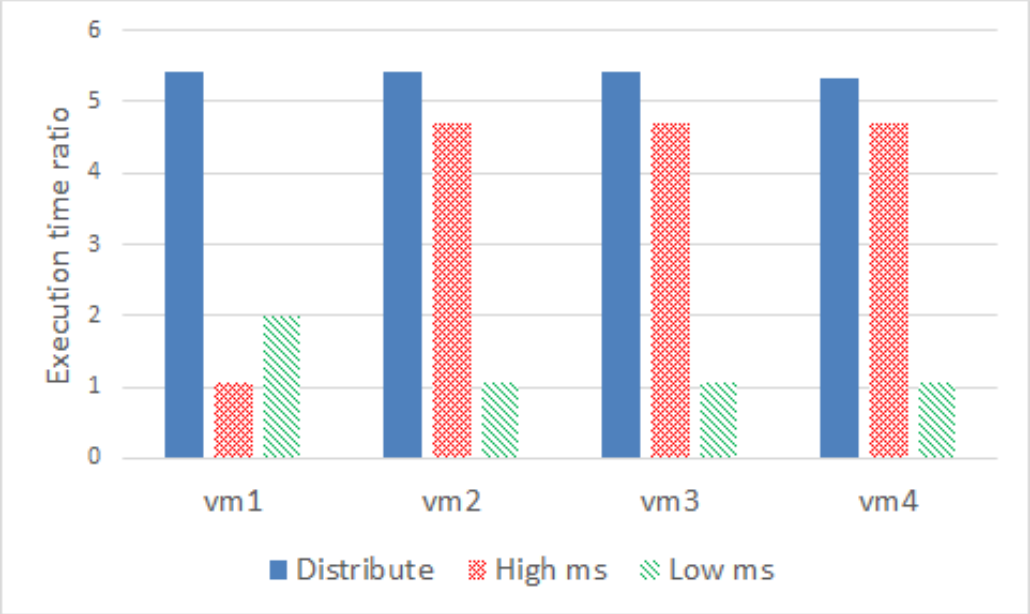

Figure 5.6: Scenario SC3 where VMs face the same *siso*(), breaking ties with the "Distribute", "High ms" and "Low ms" criteria

being executed. This delay can be much longer than if the VMs were executed in sequence.

The extraction of memory can reduce significantly the $ma(vm_i, t)$ of certain VMs, but it is not always possible to be performed since each VM needs a minimum amount of memory. Moreover, the process of extraction is not immediate and requires a number of time steps (specified by $slimit(STB)$). The $siso(vm_i, t)$ indicator can be used to reduce the delay by prioritizing VMs with high $siso(vm_i, t)$, but if there is no more memory left, and the VMs are still experiencing high swap-in/swap-out, MEC should be able to reduce the swap usage impact before relying in the CL for VM migration. The following experiment aims to show the benefits of pausing mechanism of MEC, where VMs are suspended and later saved, if necessary.

In this experiment, vm_1 and vm_2 were executed concurrently, each one running one job J_1 with $pdr(J_1) = 2$GB, both initiating at the same time. In this scenario, and considering the job type, J_1 runs with its optimum time when $ma(vm_i, t) = 3$GB.

Six different situations for running the two VMs were set, as specified in Table 5.6. Static ideal and MEC ideal provides plenty of physical memory to run both VMs and no swap is activated. In Static ma=2G, each VM is set with 2GB, totaling 4GB. The same total amount of memory is made available in the remaining situations. In MEC No Pause, no pausing mechanism is activated (VMs are neither suspended, nor saved), only suspension is enabled in MEC SUS, and both suspension and saving are activated in MEC SAV.

Table 5.6: Comparison on the Pausing Mechanism (Memory Unit in GB)

Type	$\boldsymbol{hfm}$ **(GB)**	$\boldsymbol{ma(vm_i, t)}$ **(GB)**	**Pause**
Static ideal	6	3	No
MEC ideal	6	Dynamic	No
Static $\boldsymbol{ma = 2G}$	4	2	No
MEC No Pause	4	Dynamic	No
MEC Suspend	4	Dynamic	Suspend
MEC Save	4	Dynamic	Save

The results are shown in Figure 5.7. Obviously, both ideal situations (Static and MEC) achieved the optimum execution time. On the other hand, when memory is not enough, Static led to the worse results, since both VMs are accessing swap concurrently. MEC No Pause also showed high delay, though a slight lower value is produced due to the dynamics of memory allocation $ma(vm_i, t)$, since occasionally one VM would end up with more memory than another. MEC SUS shows the importance of avoiding concurrent swap, during which, although one VM continued to run without enough memory, there was no concurrent swap usage due to the suspension of the second VM, and therefore, the delay was lower 33% than the worst case. Finally, MEC SAV resulted in a execution time ratio of two. Even though this seems a high ratio, but since 4GB of free memory is not enough to run both jobs, MEC was able to avoid a huge delay by automatically executing the VMs in sequence, each one close to its optimal time.

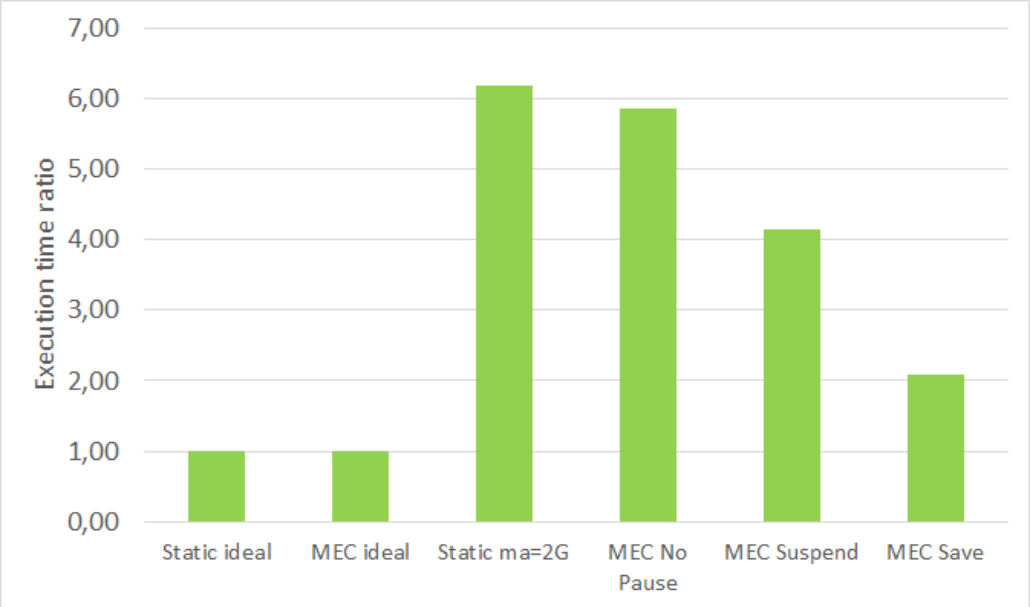

Figure 5.7: Pausing mechanism evaluation when executing two VMs concurrently in different situations

5.3 MEC overall evaluation

The following experiments aim to show that by allowing MEC to elastically manage memory allocation it might be able to reduce execution delays and therefore, to improve performance when compared to the traditional approach where memory is statically allocated, usually evenly, amongst VMs. Furthermore, MEC might also be able to increase the number of VMs that share a physical server without hurting performance as well as reduce the number of required physical machines to host a given number of required VMs.

5.3.1 Synthetic Jobs

In this first experiment, n VMs are executed concurrently, $n = 2, 4, 6, 8, 10$, where the VMs with an even identifier received each a job of type J_1, while the remaining received each a job of type J_2. Both jobs have $pdr(J_1) = pdr(J_2) = 1GB$. Assume, when running n VMs concurrently, that the total amount of memory available for allocation among them is limited to nGB. In the static memory allocation approach, all VMs will receive $ma(vm_i, t) = 1GB$ since both jobs have the same data requirement, but since $pdr(j_i) = ma(vm_i, t)$, there will not be enough memory to run the jobs without using swap. With a manager like MEC, the nGB can be distributed according to the needs of each VM, on demand.

Figure 5.8 shows the execution time ratio for n concurrent VMs, with MEC and static memory allocation. Since J_1 and J_2 use different amounts of swap, there is scope for the manager to orchestrate these demands, by prioritizing VMs with higher $siso(vm_i, t)$. Given that the host has sufficient CPUs and memory, the increasing performance degradation (i.e. the difference in the execution time ratio) as the number of concurrent VMs increases is caused by the fact that their jobs required swap usage. However, the elastic memory allocation provided by MEC obtained a significant reduction, as much as 84% in this example, using the information collected during execution.

The next experiment used a similar configuration to the previous one, except that VMs may now obtain enough memory to avoid swap usage. For $pdr(job_{1,2}) = 1$GB, a memory allocation of $ma(vm_i, t) = 1.4$GB is necessary for this experiment during its whole execution. With the total amount of memory available for allocation still being 10 GB, the question is how many VMs can be executed concurrently?

For the static approach, in order to avoid swap usage, the number of concurrent VMs is limited to $\lfloor 10GB/1.4GB \rfloor = 7$. If more than 7 VMs were to be instantiated, there

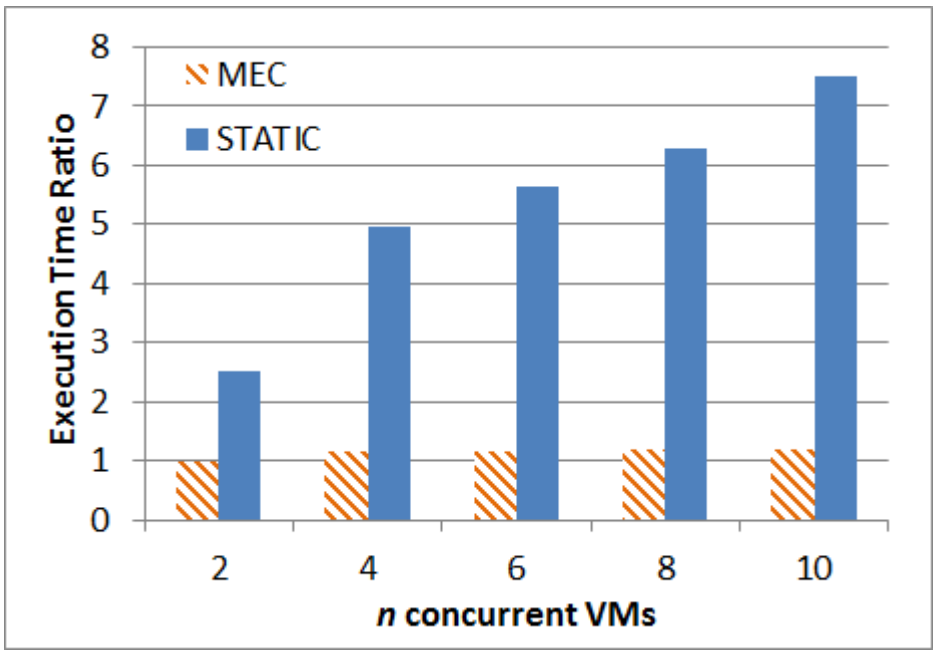

Figure 5.8: Execution time ratio of n VMs under MEC management versus static memory allocation

would be no guarantee that a required throughput criterion could be met. Note that MEC does not know which type of job is executing in each VM. While the results in Figure 5.9 show that, for a mixture of jobs J_1 and J_2, the static approach executes 7 jobs with the expected execution time ratio of one, the execution under MEC management achieves a throughput of 11 – an increase of 57% in the number of VMs on the same physical host. For 12 VMs, the total host memory of 10GB was not enough to run the set of jobs and instead of generating a high delay, VMs were run in sequence due to the pause mechanism, generating a small overhead of 2%. If one considers executing only jobs of type J_2, the results are even better, with an increase of 71% in the number of VMs. While the static approach can still only fit 7 VMs, MEC can up to 12 VMs with an additional delay of no more than 11%.

5.3.2 Parsec Benchmark

The Parsec Benchmark Suite [4] executed in this experiment is composed by several applications, described in Appendix A. Although each application has relative constant memory consumption during its execution, the chosen set of application has high dynamic variations in memory requirements among them. The benchmark was executed on the virtualized environment with the aim to compare the benefits provided by the proposed MEC, against the MOP strategy [13] considering 10% and 30% as the pre-defined

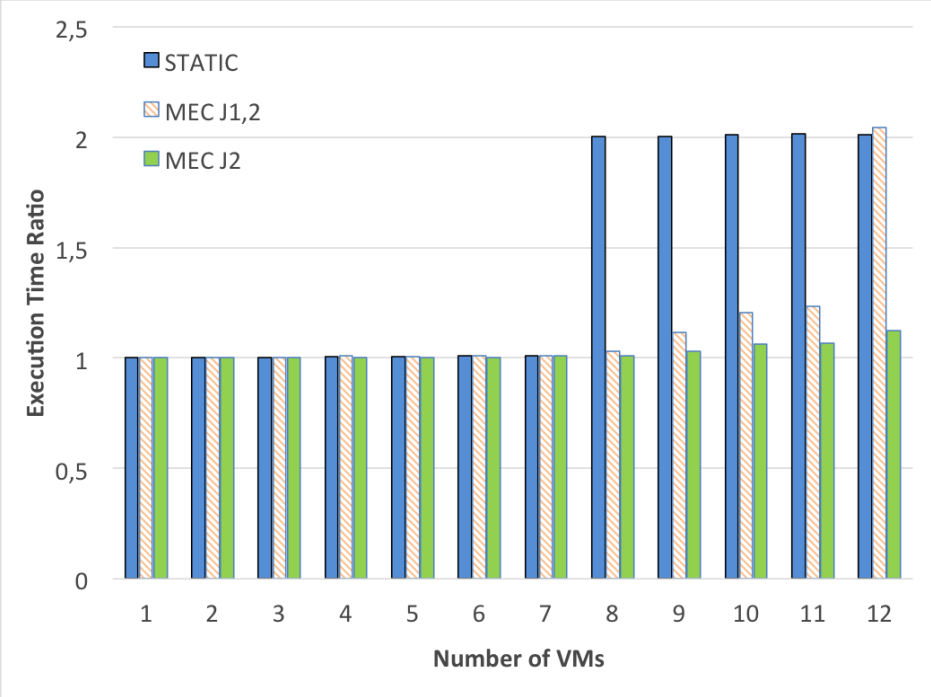

Figure 5.9: Execution time ratio: Static versus MEC

percentages of free memory and also the DMSS strategy [16]. Also, a comparison was carried out with the STATIC strategy, in which VMs are executed with their required memory, and its variant, STATIC-V, where the available memory is evenly divided between VMs. For each experiment, a number of concurrent VMs were executed, each VM running a given combination of 10 Parsec applications sequentially in different orders. Thus, the memory requirements of each VM will vary. Preliminary executions showed that $ma(vm_i, 0)$ =1GB was sufficient to run the set of applications in a single VM without memory related delays. This information was used solely to limit the total physical memory to 6GB, i.e. the sum of all $ma(vm_i, t)$ did not exceed 6GB.

Figure 5.10 compares the execution time ratio in relation to the optimal execution time (6 VMs each one with 1GB of memory) between MEC and STATIC, where in the latter, VMs were set with a fixed $ma(vm_i, t)$ of 1GB. Since the host memory available was set with 6GB, only six VMs can be executed concurrently without loss in performance. When running 7 to 12 VMs, only six VMs were executed concurrently, after which the remaining VMs were run. On the other hand, MEC automatically adjusts the amount of allocated memory and manages to reduce by an average of 31.5% the execution time of the same set of Parsec applications.

A much worse scenario is presented in Figure 5.11 where MEC is compared against STATIC-V approach, in which a static $ma(vm_i, t)$ was set to $\frac{6GB}{n}$, with n being the number

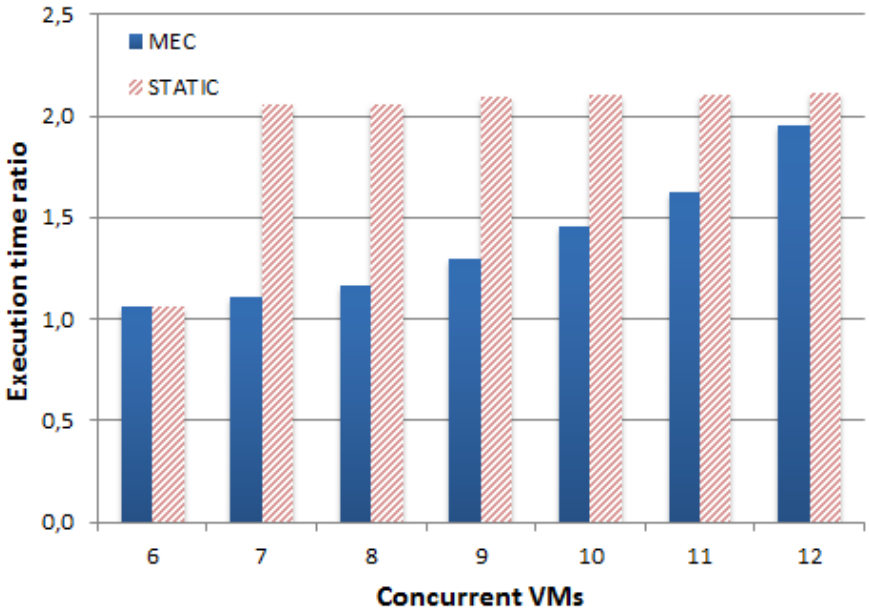

Figure 5.10: Parsec with MEC vs STATIC

of concurrent VMs. In the case of six VMs, $ma(vm_i, t)$ was 1GB, while with seven VMs, $ma(vm_i, t)$ was 878MB. The variant STATIC-V provided much worse results due to the intensive use of swap, since the same amount of available memory as divided between the concurrently executing VMs.

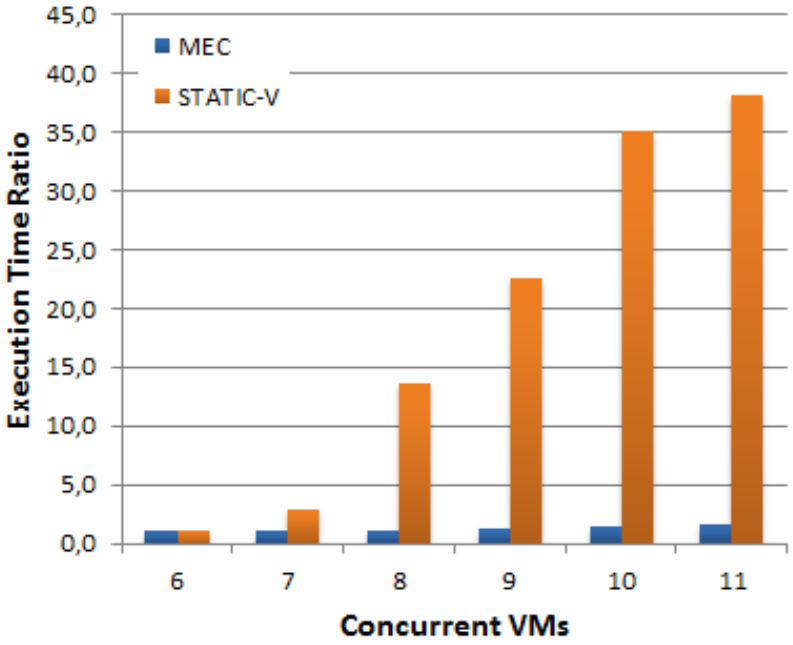

Figure 5.11: MEC vc STATIC-V

The previous results clearly show the dynamic approach of MEC, even with the overheads of detecting and reconfiguring the allocation, to be better than the standard static

strategies. Figure 5.12 compares the MEC scheme with the dynamic approaches from the literature, MOP [13](with two free memory threshold values of 10% and 30%) and DMSS, the closest ones to MEC. The differences between these three approaches are in the metrics and heuristics adopted.

In MOP, the elastic rule was implemented with a few adaptations. In their work, at every time step, the memory allocated to each vm_i is defined as a factor $(1+MOP)$ of the actual memory consumed by vm_i. However, the calculation based solely on the consumed memory might result in the application's abrupt interruption or VM crash, since it does not take into consideration VMs attributes such as memory overhead $mo(vm_i)$ and swap activation threshold $sat(vm_i)$. Therefore, in our version, the amount of memory to be allocated in MOP is the $(1 + MOP)$ times the actual consumed memory plus $mo(vm_i)$ + $sat(vm_i)$. This calculation is also held at every $I_{host} = 5$ seconds, defined in the same manner as in MEC.

For DMSS, the calculation of the VM's memory allocation is similar to our work, therefore no changes were necessary. They also use the Free threshold (FT) to avoid swap usage, similar to the $sat(vm_i)$. The FT in DMSS is dynamic, allowing to adjust an incorrect value set by the administrator during the VM execution, but for our experiments, this feature was not necessary since the correct value of FT was known (which is based on $sat(vm_i)$). The main difference between DMSS and MEC is the memory distribution, when host memory is not enough to attend the VM's requests is the criteria for VMs prioritization. The implementation of the DMSS criteria is shown in Appendix B.

We can clearly see in Figure 5.12 the benefits of MEC when a higher number of VMs need to be executed, since MEC automatically calibrates the amount of memory required by the applications. In the case of a higher pre-defined percentage, the MOP strategy provides a higher amount of memory for an initial number of VMs, but then, remaining VMs will not receive the necessary amount. While DMSS considers the number of page faults and may allocate memory when in fact it is not necessary, MEC looks at the amount of data being swapped in and out in order to define more precisely memory requirements. The two approaches use opposing VMs priorities for memory allocation, since MEC avoids a higher degradation by considering the influence of swap on the performance-memory relationship. As seen in the figure, when 10 to 12 VMs with different requirements are defined, due to the DMSS distribution, a higher amount of VMs in swap occur.

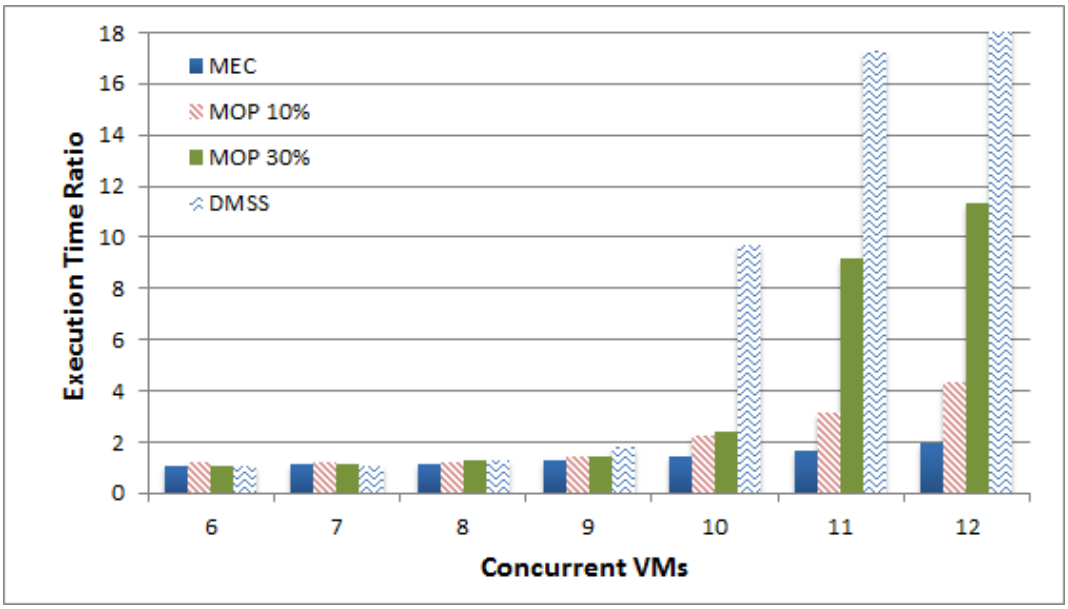

Figure 5.12: MEC vc MOP

Chapter 6

Conclusion and future work

6.1 Conclusion

For cloud providers and data centers, small improvements in memory utilization can lead to higher consolidation rates. The evaluation presented in this work outlined a hypervisor and guest OS independent tool, the *Memory Elasticity Controller* (MEC) that manages the VM's running state and memory allocation on a server, aiming to increase the throughput of a cloud system.

MEC was developed based on a careful examination of the characteristics of the behavior of a virtualized environment, and considered:

- Properties that should be monitored within the guest OS, such as total memory $tm(vm_i, t)$, free memory $fm(vm_i, t)$, swap-in $si(vm_i, t)$ and swap-out $so(vm_i, t)$;
- Characteristics of the VMs, such as the swap activation threshold $sat(vm_i)$, the memory overhead $mo(vm_i)$;
- Attributes of the VMs, such as the maximum memory allocation $maxml(vm_i)$ and memory allocation $ma(vm_i, t)$;
- Dynamic parameters to identify VMs with insufficient memory such as the memory shaping $ms(vm_i, t)$ or VMs suffering significant slowdowns in their execution through the swap-in/swap-out indicator $siso(vm_i, t)$;
- Criteria to prioritize VMs, such as high $siso(vm_i, t)$ or low $ms(vm_i, t)$;
- Configuration parameters of MEC, such as I_{host} and $I_{monitor}$ intervals.

To evaluate the execution of applications in virtualized environments and identify and exemplify crucial configuration criteria and parameters for MEC, the synthetic applications $J1$, $J2$ and $J3$ were created. These jobs represent generic applications with different memory access patterns, and through experiments, showed how different memory and swap usages impact performance and memory under different VM's memory allocations.

The MEC definition also included mechanisms to calculate memory requirement using predictions based on recent memory consumption; to extract memory without the guest OS indicating that memory can be released; and to pause/resume virtual machines when necessary. The collected information identified at the PML is also sent to the CL, giving opportunities for the $CloudManager$ to solve a different problem concerning the load balance of the cloud system.

Also, the memory state transitions defined in this work offer a chance to estimate with a certain accuracy the memory requirements of the VMs in future moments. It provides an opportunity for memory deallocation as a mean to liberate memory for use by others VMs in need on the same machine.

Different to the work of [13, 16], this study focuses on the analysis of memory and swap usage. In particular, how the management of both can be used to increase VM throughput, while attempting to minimize application performance degradation taking into consideration different memory requirements and access patterns. The mechanism proposed does not incur any significant performance overhead (like [23]) nor require apriori application modeling [14]. Memory over-commitment techniques such as Kernel Samepage Merging (KSM), if available, can be applied transparently in conjunction with this proposal.

Finally, MEC was validated through experiments using synthetic applications as well as the Parsec benchmark. The results showed that MEC is capable of increasing the throughput of a server when possible, and maintain performance when not, by managing the VM's memory allocation and running state. Given the high number of physical machines in data centers nowadays, the obtained increases in throughput can have a significant impact in help reduce energy consumption, capital expenses and administration costs.

This work also led to an article titled "Evaluating the Impact of Memory Allocation and Swap for Vertical Memory Elasticity in VMs" [20], presented at the *27th International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD

2015), held in Florianópolis, Brazil.

6.2 Future work

As future work, the free memory estimation can be refined, using different methods to calculate the memory consumption, with adaptive models and linear regression based on all values received from period $[t - I_{host}, t]$, from $Monitor_i$ deployed at each virtual machine.

Another issue, in order to validate the accuracy of memory consumption prediction, alternatives can be evaluated through distinct benchmarks, not restricted to batch applications. Web applications with variable workloads can be included and Java Virtual Machines [1] (JVM) should also be studied since it manages its own memory. In this case, a different approach might be necessary to cope with applications running under a JVM.

Analyses of the relationship between parameters such as I_{host}, $slimit(STB)$ and $slimit(FRZ)$ should also be compared and evaluated. In this work, all mentioned values were set statically, based on experiments, but a more sophisticated approach can be used, with automatic adaptation of I_{host} based on application's demands. Different requirements might need different I_{host} values, and with higher interval, more sophisticated prediction method will be necessary. Values set in $slimit(STB)$ and $slimit(FRZ)$ should also be adaptive with the application's necessity. The adaptive model for memory consumption prediction could be used to define dynamically the values for $slimit()$.

Given the management of the number of vCPUs dynamically in response to requirements of malleable applications is a key feature in the roadmap towards exascale systems, vertical elasticity of other resources such as vCPU can also be integrated into the framework. In this case, a whole new set of parameters can be considered, such as CPU usage and cache-miss rate, as well as considering hardware architectures with non-uniform memory access (NUMA). I/O and network access should also be considered since these also impacts VM migration as well as decisions regarding how to schedule a set of VMs that needs high connectivity across separate PMs.

With the set of information from the Physical Layer, the problem that the Cloud Layer needs to solve will also be addressed. Here, horizontal elasticity policies can be applied, and migration of VMs are regarded and evaluated, comparing live migration with the current migration of saved VMs. Information regarding the time spent in each memory state $mstate(vm_i, t)$ can be used to categorize VMs, for mixing or matching

different types in a physical machine. Also, the waiting time $wtime(vm_i)$, together with the expected remaining time to finish execution $rtime(vm_i)$ can be used to decide which VMs are selected to free memory for other VMs. And in order to increase scalability, the $CloudManager$ could be organized as a P2P network, or divided in sub-clouds, using elasticity as demands increases or decreases in the cloud system.

References

[1] Java virtual machine. http://www.java.com/.

[2] Kernel virtual machine (kvm). http://www.linux-kvm.org/.

[3] Libvirt: The virtualization API. http://libvirt.org/index.html.

[4] The princeton application repository for shared-memory computers (parsec). http://parsec.cs.princeton.edu/.

[5] Vmware. http://www.vmware.com/.

[6] Xen hypervisor. http://www.xenproject.org/developers/teams/hypervisor.html.

[7] Amazon ec2, 2014. Available at http://aws.amazon.com/ec2.

[8] Amazon ec2 auto scaling, 2014. Available at http://aws.amazon.com/autoscaling.

[9] Kvm kernel based virtual machine, 2014. Available at http://www.linux-kvm.org.

[10] Baruchi, A.; Midorikawa, E. A survey analysis of memory elasticity techniques. In *Euro-Par 2010 Par. Proc. Workshops*, vol. 6586 of *LNCS*. Springer, 2011, pp. 681–688.

[11] Coutinho, E.; de Carvalho Sousa, F.; Rego, P.; Gomes, D.; de Souza, J. Elasticity in cloud computing: a survey. *Annals of Telecommunications 70*, 7-8 (2015), 289–309.

[12] Elmroth, E.; Tordsson, J.; Hernández, F.; Ali-Eldin, A.; Svärd, P.; Sedaghat, M.; Li, W. Self-management challenges for multi-cloud architectures. In *Proc. of the 4th European Conf. on Towards a Service-based Internet* (2011), Springer, pp. 38–49.

[13] G. Molto, M. Caballer, E. R.; Alfonso, C. Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements. *Procedia Computer Science 18* (2013), 159 – 168.

[14] Gordon, A.; Hines, M.; Da Silva, D.; Ben-Yehuda, M.; Silva, M.; Lizarraga, G. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *ASPLOS RESoLVE'11: Runtime Environ./Sys., Layering, and Virtualized Environ. Workshop* (2011).

[15] Kleen, A. Where is the kernel memory going? Memory usage in the 2.6 kernel. In *13th International Linux Sys. Technology Conf. (2006 Linux Kongress)* (Sept 2006).

[16] Liu, L.; Chu, R.; Zhu, Y.; Zhang, P.; Wang, L. DMSS: A dynamic memory scheduling system in server consolidation environments. In *15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)* (April 2012), pp. 70–75.

[17] Lu, L.; Zhu, X.; Griffith, R.; Padala, P.; Parikh, A.; Shah, P.; Smirni, E. Application-driven dynamic vertical scaling of virtual machines in resource pools. In *IEEE Network Operations and Management Symp. (NOMS)* (2014).

[18] P. Mell, T. G. The nist definition of cloud computing, 2011. Available at `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[19] Pavlovic, M.; Etsion, Y.; Ramirez, A. On the memory system requirements of future scientific applications: Four case-studies. In *IEEE International Symposium on Workload Characterization (IISWC)* (2011), pp. 159–170.

[20] Sawamura, R.; Boeres, C.; Rebello, V. Evaluating the impact of memory allocation and swap for vertical memory elasticity in VMs. In *IEEE 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (Oct 2015).

[21] Spinner, S.; Kounev, S.; Zhu, X.; Lu, L.; Uysal, M.; Holler, A.; Griffith, R. Runtime vertical scaling of virtualized applications via online model estimation. In *8th IEEE Int. Con. on Self-Adaptive and Self-Organizing Systems* (2014), pp. 157–166.

[22] Waldspurger, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[23] Zhao, W.; Wang, Z. Dynamic memory balancing for virtual machines. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (USA, 2009), pp. 21–30.

APPENDIX A – Parsec Benchmark Applications

This appendix list and describes the applications used from the Parsec Benchmark Suite.

parsec.swaptions: The application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. Swaptions employs Monte Carlo (MC) simulation to compute the prices.

splash2x.raytrace: The Intel RMS application uses a version of the raytracing method that would typically be employed for real-time animations such as computer games. It is optimized for speed rather than realism. The computational complexity of the algorithm depends on the resolution of the output image and the scene.

splash2x.volrend: Computes the cholesky factorization of a sparse matrix

parsec.ferret: This application is based on the Ferret toolkit which is used for content-based similarity search. It was developed by Princeton University. The reason for the inclusion in the benchmark suite is that it represents emerging next-generation search engines for non-text document data types. In the benchmark, we have configured the Ferret toolkit for image similarity search. Ferret is parallelized using the pipeline model.

splash2x.lu_ncb: LU factorization of a dense matrix (non-contiguous block allocation)

splash2x.lu_cb: LU factorization of a dense matrix (contiguous block allocation)

parsec.fluidanimate: This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. It was included in the PARSEC benchmark suite because of the increasing significance of physics simulations for animations.

splash2x.water_spatial: Computes the cholesky factorization of a sparse matrix

parsec.freqmine: This application employs an array-based version of the FP-growth (Frequent Pattern-growth) method for Frequent Itemset Mining (FIMI). It is an Intel RMS benchmark which was originally developed by Concordia University. Freqmine was

included in the PARSEC benchmark suite because of the increasing use of data mining techniques.

parsec.bodytrack: This computer vision application is an Intel RMS workload which tracks a human body with multiple cameras through an image sequence. This benchmark was included due to the increasing significance of computer vision algorithms in areas such as video surveillance, character animation and computer interfaces.

APPENDIX B – DMSS implementation algorithm

This appendix describes the algorithm used to implement the DMSS criteria for memory distribution when the host available memory is not enough to attend the VMs requests.

Few definitions are necessary:

- $ms(vm_i, t)$: the memory shaping, or required memory for vm_i;
- $dms(vm_i, t)$: the difference between $ms(vm_i, t)$ and $ms(vm_{i+1}, t)$, considering that the VMs are already ordered by $ms(vm_i, t)$ decreasingly;
- $ma(vm_i, t)$: the memory allocation of vm_i at current moment t;
- $chfm$: the current host free memory to be distributed.

Algorithm 10 only manages VMs with memory shortage. The memory abundant ones should be treated firstly and is not shown here. The algorithm starts by ordering the VMs with their $ms(vm_i, t)$ value decreasingly (line 1). Then, for each vm_i, it calculates the difference between $ms(vm_i, t)$ value and the value from the next VM, considering the ordering by $ms(vm_i, t)$ (lines 3 to 5). This difference represents the value that should be added to the VM's memory allocation so that its required memory $ms(vm_i, t)$ is not the highest anymore. For the last VM, the difference $dms(vm_i, t)$ is set with its own $ms(vm_i, t)$ (line 7). From lines 9 to 25 , the memory is distributed as follows: if host free memory is enough to attend the first i VMs in order to equalize their $ms(vm_i, t)$ values, then this amount will be added to the VM's memory allocation (line 11). If not, each VM from vm_1 to vm_i will receive an equal share of the last available host free memory (line 13). The actual update of memory allocation is done from line 16 to 18, following by the $chfm$ update with the amount that was used to attend the VMs. If there is no free memory left in the host, then the memory distribution is over.

Algorithm 10 Distribute available memory to VMs with memory shortage

1: Order VMs by $ms(vm_i, t)$ decreasingly;
2: // For each VM with memory shortage, calculate diff ms:
3: **for** $i = 1$ **to** $n - 1$ **do**
4: $dms(vm_i) \leftarrow ms(vm_i, t) - ms(vm_{i+1}, t)$;
5: **end for**
6: // Set diff ms for the last element:
7: $dms(vm_n) \leftarrow ms(vm_n)$;
8: // Calculate the new ma for each VM:
9: **for** $i = 1$ **to** n **do**
10: **if** $chfm \geq dms(vm_i, t) \times i$ **then**
11: $diff \leftarrow dms(vmi)$;
12: **else**
13: $diff \leftarrow chfm/i$;
14: **end if**
15: // Updates the ma of each VM from 1 to n
16: **for** $j = 1$ **to** i **do**
17: $ma(vm_j, t) \leftarrow ma(vm_j, t) + diff$;
18: **end for**
19: // Updates the host free memory:
20: $chfm \leftarrow chfm - (diff \times i)$;
21: // If there is no more memory available, exit:
22: **if** $chfm = 0$ **then**
23: break;
24: **end if**
25: **end for**