

UNIVERSIDADE FEDERAL FLUMINENSE

CARLOS HENRIQUE ZILVES NICODEMUS

Managing Vertical Memory Elasticity in Containers

NITERÓI

2020

UNIVERSIDADE FEDERAL FLUMINENSE

CARLOS HENRIQUE ZILVES NICODEMUS

Managing Vertical Memory Elasticity in Containers

A thesis presented to the Computing Postgraduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Doctor of Science. Research Area: Computer Science

Advisor:

Eugene Francis Vinod Rebello

NITERÓI

2020

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

N633m Nicodemus, Carlos Henrique Zilves
Managing Vertical Memory Elasticity in Containers / Carlos
Henrique Zilves Nicodemus ; Eugene Francis Vinod Rebello,
orientador. Niterói, 2020.
130 f.

Tese (doutorado)-Universidade Federal Fluminense, Niterói,
2020.

DOI: <http://dx.doi.org/10.22409/PGC.2020.d.10046273786>

1. Computação em Nuvem. 2. Gerenciamento de Memória. 3.
Elasticidade Vertical de Recursos. 4. Containerização. 5.
Produção intelectual. I. Rebello, Eugene Francis Vinod,
orientador. II. Universidade Federal Fluminense. Instituto de
Computação. III. Título.

CDD -

Carlos Henrique Zilves Nicodemus

Managing Vertical Memory Elasticity in Containers

A thesis presented to the Computing Postgraduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Doctor of Science. Research Area: **Computer Science**

Approved in December 2020 by:

Examination Board



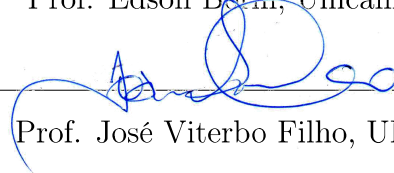
Prof. Eugene Francis Vinod Rebello - Advisor, UFF



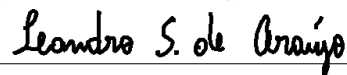
Prof. César Augusto Fonticielha De Rose, PUCRS



Prof. Edson Borin, Unicamp



Prof. José Viterbo Filho, UFF



Prof. Leandro Santiago de Araújo, UFF

Niterói

2020

Acknowledgement

Thanks to all my family and friends who always supported me in this endeavor. My wife Hannah, who was always by my side, even when I thought about giving up everything. To my advisor, Vinod Rebello, who helped to mature the ideas that I had for the VEMoC tool and refine them. To the professors and directors of the Computing Postgraduate Program of the Institute de Computação at UFF, for their patience and support to allow this work could be completed. Finally, I would also like to thank the funding agencies, CAPES and FAPERJ, who each provided me with a postgraduate scholarship for different periods during this work.

Resumo

A adoção da tecnologia de contêiner para implantar uma grande variedade de aplicações e serviços em *clusters*, *data centers* na nuvem e até mesmo em *cloudlets* na borda, tem aumentado continuamente. Essa forte demanda levou ao desenvolvimento de plataformas de orquestração de contêineres capazes de gerenciar de alguns contêineres em computadores simples, com restrição de recursos, para executar milhares de aplicativos em contêineres em centenas de máquinas. Em cada uma dessas escalas, a utilização eficiente de recursos e a maximização do rendimento são dois objetivos importantes ao tentar reduzir os custos operacionais da organização que hospeda esses ambientes. Embora os contêineres alocados sejam executados como processos, logicamente isolados em um único servidor, eles compartilham os recursos do servidor e o *kernel* do sistema operacional. Considerando que as capacidades dos recursos são limitadas, os contêineres co-hospedados podem sofrer alguma degradação de desempenho, ao competir por recursos como CPU, memória e largura de banda de E/S. Mesmo que os contêineres consumam tais recursos elasticamente, os *frameworks* de escalonamento ainda devem alocar contêineres de acordo com a disponibilidade de recursos e limitar a quantidade de recursos que cada contêiner pode usar, para que a interferência seja minimizada. Atualmente, é prática comum ao agendar contêineres, reservar efetivamente a quantidade máxima de memória exigida pelo contêiner, para toda a duração de sua execução. Em vez disso, este trabalho propõe uma ferramenta escalonável de orquestração de contêineres chamada o **VEMoC** (para *Vertical Elasticity Management of Containers*), projetada para ajustar constantemente a quantidade máxima de memória que cada contêiner co-hospedado pode usar naquele momento, com base no comportamento dos aplicativos que cada contêiner está executando. Nossa abordagem visa aumentar o número médio de contêineres que podem ser hospedados em um servidor, sem impactar o desempenho dos contêineres co-allocados. A avaliação experimental do VEMoC mostra que, por meio de ajustes cuidadosos dos limites de memória do contêiner, manipulação de páginas entre memória principal e memória *swap*, e a preempção de contêineres, melhorias na utilização da memória, custos de nuvem e rendimento do trabalho podem ser alcançados sem prejudicar o desempenho dos contêineres em comparação com os mecanismos empregados por sistemas de orquestração de contêineres corporativos, como o Kubernetes. Além disso, o VEMoC é um gerenciador de contêineres agnóstico e modular, que permite que ele seja estendido para usar uma variedade de sistemas de contêineres e empregar novos mecanismos de elasticidade de recursos.

Keywords: Computação em nuvem, gerenciamento de memória, elasticidade vertical, contêinerização, orquestração de infraestrutura.

Abstract

The adoption of container technology to deploy a diverse variety of modern-day applications in clusters, cloud data centers and even cloudlets at the edge has been steadily increasing. This strong demand has led to the development of container orchestration platforms capable of managing a few containers on resource constraint single board computers to running thousands of containerized applications across hundreds of machines. At each of these scales, efficient resource utilization and throughput maximization are two important objectives when trying to reduce the hosting organization's operating costs. Although co-allocated containers execute as logically isolated processes on a single host, they share the host's resources and the operating system kernel. Given resource capacities are limited, co-hosted containers may suffer some performance degradation when competing for resources such as CPU, memory and I/O bandwidth. Even though containers consume such resources elastically, scheduling frameworks must still allocate containers according to resource availability and limit the amount of resources that each container can use so that interference is minimized. Currently, it is common practice when scheduling containers to effectively reserve the maximum amount of memory required by the container for the entire duration of its execution. Instead, this work proposes a scalable container orchestration tool, called Vertical Elasticity Management of Containers (VEMoC), designed to constantly adjust the maximum amount of memory each co-hosted container may use at that moment, based on the behavior of the applications each are running. Our approach aims to increase the average number of containers that can be hosted on a server, without impacting the performance of the co-allocated containers. Experimental evaluation of VEMoC shows that through its careful adjustment of container memory limits, manipulation of pages between memory and swap, and container preemption, improvements in memory utilization, cloud costs, and job throughput can be achieved without prejudicing container performance when compared to mechanisms employed by enterprise container orchestration systems, like Kubernetes. In addition, VEMoC is a modular and container engine agnostic framework, which allows it to be extended to use a variety of container engines and employ new resource elasticity mechanisms.

Keywords: Cloud computing, memory management, vertical elasticity, containerization, infrastructure orchestration.

Contents

List of Figures	ix
List of Tables	xi
List of Acronyms and Abbreviations	xiii
1 Introduction	1
1.1 Motivation and Objectives	3
1.2 Contributions	5
1.3 Thesis Outline	7
2 Container Virtualization	9
2.1 The Core Concepts of Virtualization	10
2.2 Container Namespaces	13
2.2.1 Computational Resource Allocation	14
2.2.2 Disk Storage	15
2.2.3 Communication between Containers	18
2.3 Container Virtualization Technologies	18
2.3.1 Docker	19
2.3.2 Linux Containers - LXC	19
2.3.3 Apache Mesos	20
2.3.4 Google Kubernetes	20
2.4 Cloud Elasticity vs Scalability	20

2.4.1	Horizontal Elasticity of Virtual Clusters	21
2.4.2	Vertical Elasticity in Containers	22
3	Related Work on Memory Elasticity	24
4	The VEMoC Architecture	28
4.1	Cloud Manager	29
4.1.1	Cloud Monitor	29
4.1.2	Cloud Scheduler	30
4.2	Host Manager	31
4.2.1	Host Monitor	31
4.2.2	Request Receiver	32
4.2.3	Container Manager	33
5	Managing Vertical Memory Elasticity	36
5.1	The VEMoC Orchestration Engine	36
5.2	Phase 1: Calculate the Memory Demand of Inactive Containers	40
5.3	Phase 2: Classify Running Containers by their Recent Memory Consumption	43
5.3.1	Container Memory Consumption Rate	43
5.3.2	Container Memory Classification	45
5.3.3	Estimate Container Memory Demand	46
5.4	Phase 3: Passive Memory Limit Reduction	49
5.5	Phase 4: Active Memory Limit Reduction	50
5.6	Phase 5: Increase Container Memory Limits	52
5.7	Phase 6: Pause or Suspend Containers	54
5.8	Phase 7: Start or Resume Inactive Containers	57
5.9	Summary	58

6 Performance Evaluation and Analysis	59
6.1 Experimental Setup	59
6.2 Container State Transition Costs	61
6.2.1 Container Creation and Destruction under Docker and LXC	61
6.2.2 Pausing and Suspending Containers under Docker and LXC	64
6.3 Memory Utilization Effectiveness	69
6.4 Container and Host Manager Overheads	74
6.5 Managing Container Memory Limits	76
6.6 Benefits of Vertical Memory Elasticity	79
6.6.1 Scenario 1: Two 4 GiB J1 jobs are submitted 50 seconds apart to a host with 6 GiB of available memory	80
6.6.2 Scenario 2: One 4 GiB J2 job is followed by one 4 GiB J1 job, 100 seconds later, on a host with 6 GiB of available memory	81
6.6.3 Scenario 3: One 4 GiB J2 job is followed by five 4 GiB J1, at 10 seconds intervals, using all the available memory (23.5GiB) on a host while sharing the Linux operating system.	82
6.7 Scalability Analysis	83
7 Conclusions and Future Work	84
References	88
Appendix A – Container versus Virtual Machines: A Case Study	94
A.1 Availability test	95
A.2 Performance Test	97
A.3 Scalability Test	98
A.3.1 CPU Allocation Test	101
Appendix B – Memory Consumption Trends	105

List of Figures

2.1	Full Virtualization (Host 1) x Containerization (Host 2).	12
2.2	<i>cgroups resource organization for cpu and memory resources allocation.</i> . .	15
2.3	The Rootfs File System and Union File System container storage types. . .	17
2.4	Container Horizontal Elasticity	22
2.5	Container Resource Vertical Elasticity	22
4.1	Architectural Model for VEMoC Container-based Cloud Management . . .	29
4.2	States of the Container Life Cycle	34
5.1	Phases of a VEMoC Management Iteration	37
6.1	Average container initialization and termination times.	62
6.2	Breakdown of LXC container initialization time.	62
6.3	The average times to Pause and Unpause a container in relation to its memory consumption, for Docker and LXC.	65
6.4	The average times to suspend/resume a container in relation to its memory consumption.	66
6.5	Average times to suspend or resume several LXC containers, that consumed 2GB of Memory, sequentially and in parallel.	67
6.6	Suspend and Resume Behaviors of an LXC container that consumes 2 GiB of memory.	68
6.7	Container Memory Utilization vs Effectiveness for a single container. . . .	70
6.8	Container Memory Limit Utilization for 2 containers.	72
6.9	Container Memory Utilization for 4 containers.	73
6.10	The execution times of 4 GiB jobs J1 and J2 with different container mem- ory limits.	75

6.11	VEMoC predicting the memory demand of a <i>J1</i> type job.	77
6.12	VEMoC predicting the memory demand of a <i>J2</i> type job.	78
6.13	VEMoC predicting the memory demand of a <i>J4</i> type job.	79
A.1	GARP scalability for the modeling step	99
A.2	GARP scalability for the test stage	100
A.3	GARP scalability for the projection stage	101
A.4	GARP scalability, for the modeling stage, with fixation of <i>cores</i>	102
A.5	Evaluation of the GARP algorithm regarding the use of <i>hyperthreading</i> . . .	103

List of Tables

6.1	LXC Create, Start and Destroy containers from images of different sizes. . .	63
6.2	Docker Create + Start and Destroy containers from images of different sizes.	63
6.3	Average duration in seconds to Pause and Unpause containers varying memory consumption under Docker and LXC.	64
6.4	Average Suspend and Resume times in seconds for Docker and LXC containers.	66
6.5	The average total times to suspend and resume several LXC containers consuming 2 GiB of memory, with commands executed sequentially and in parallel using threads.	67
6.6	Memory consumption metrics as the <i>CML</i> of a single container is increased. Data presented in Figure 6.7.	71
6.7	Memory metrics for 2 containers as presented in Figure 6.8.	71
6.8	Memory metric values presented in Figure 6.9 for the execution of 4 containers simultaneously.	74
6.9	Average execution times in seconds with 95% confidence intervals for 4 GiB jobs on different platforms.	74
6.10	Comparing allocation strategies under Scenario 1	81
6.11	Comparing allocation strategies under Scenario 2	82
6.12	Comparing allocation strategies under Scenario 3.	82
6.13	Execution of small <i>J1</i> jobs with VEMoC, starting the jobs in intervals of 5 seconds.	83
A.1	GARP: Average time and standard deviation of the three stages	97
A.2	MAXENT: Average time and standard deviation for the three steps	97
A.3	ENVDIST: Average time and standard deviation of the three steps	98

B.1 Verification of all possible combinations for values of the adopted Cgroups
Memory Metrics. 107

List of Acronyms and Abbreviations

AHMU	: Average Host Memory Utilization;
AJTT	: Average Job Turnaround Time;
API	: Application Programming Interface;
CM	: Container Manager;
CML	: Container Memory Limit;
CMU	: Container Memory Usage;
CRIU	: Checkpoint/Restore in Userspace;
HM	: Host Manager;
HMUT	: Host Memory Utilization;
HRM	: Host Reserved Memory;
LRU	: Least Recently Used;
LXC	: Linux Containers;
MCT	: Memory Consumption Trends;
MPF	: Major Page Fault;
MTT	: Memory Trend Timer;
MUE	: Memory Usage Effectiveness;
NAHM	: Non-Allocated Host Memory;
OCI	: Open Container Image;
OS	: Operating System;
PCM	: Page Cache Memory;
QoS	: Quality of Service;
RAM	: Random Access Memory;
RSS	: Resident Set Size;
SU	: Swap Usage;
THM	: Total Host Memory;
TMTF	: Total Memory-Time Product;
VEMoC	: Vertical Elasticity Management of Containers;

Chapter 1

Introduction

Over the last few years, the adoption of cloud computing by users and enterprises has grown rapidly with a substantial offering of services from cloud providers such as Amazon, Microsoft and Google. Through the public cloud's pay-as-you-go model, users can have their applications and services hosted in a malleable online environment, without the need to invest in infrastructure on the user's own premises. Being located at a cloud provider's central data center, however, data communication latencies may often impact user experience. Thus, cloud providers have begun to augment their regional data centers with multiple smaller so called edge data centers, strategically located closer to clients.

However, having restricted amounts of resources, these smaller edge systems typically have to support different types of jobs – from high-availability long-running ones to latency-sensitive, or even deadline constrained batch jobs – each containing scientific and business applications that may employ a variety of technologies from areas such as Big Data and Artificial Intelligence (AI), Internet of Things (IoT) and web services, among others. The majority of cloud infrastructures thus support multi-tenancy, where applications belonging to multiple users may be scheduled on the same shared set of compute resources, allowing for better resource utilization and higher job throughput.

Initially, cloud providers started with virtualization technologies (hypervisors and virtual machines) to offer on-demand execution environments to users [81]. These robust virtual systems, with pre-reserved amounts of computational resources (CPU, memory, disk space, etc), dedicated operational systems, libraries and applications, are capable of simulating physical machines. However, to customize the virtual machines to meet their needs, users tend to require remote access to the host to both configure the system and install libraries to run their applications, all of which requires significant amounts of know-how and time.

More recently, an alternate virtualization technology has been gaining popularity because of its apparent greater portability and ease of use – Containers [9]. A container is a lightweight virtual environment that shares the same operating system with the host, but with its own applications and libraries, and thus has a smaller size compared to an equivalent virtual machine (VM). Deploying an application to a cloud that uses this technology is simple, since containers allow the developer to package up their application with all of the required components such as libraries and other dependencies, and submit them to the provider’s cloud container platform. Currently, Docker [28] is probably the most popular, but a number of other container technologies have also gained a foothold in the community, including Linux Containers (LXC) [14], Containerd [17], CoreOS Rocket (rkt) [64], and OpenVZ [76].

Containers provide advantages for cloud providers too in terms of better levels of consolidation [20] and lower energy consumption [49]. Application performance is often close to that obtained when running on *bare-metal servers* (i.e., without virtualization), since many of the overheads encountered when running applications in VMs are avoided [79]. Since a container has no pre-reserved resources allocated to it, it can use as much of the free resources available on the host (up to predefined limits), as and when its encapsulated application requires. This feature allows applications to avoid the performance degradation that affects VMs when insufficient resources have been pre-allocated.

Cloud providers have obvious financial incentives to invest in more efficient data centers. While the amount of computing carried out in data centers more than quintupled between 2010 and 2018, the amount of energy they consumed worldwide grew only six percent during that period, thanks to improvements in energy efficiency. Nevertheless, data centers are responsible for 1% of the electricity consumed globally. In an attempt to address the environmental impact, law makers intend to force cloud providers to become carbon neutral and energy efficient by 2030. With the forecasts of growth in cloud computing and significant increases energy consumption, technological and infrastructure improvements alone will be insufficient for cloud providers to meet their legal demands. Providers are thus promoting a more efficient form of cloud computing known as *server-less computing*, also related to function-as-a-service, that takes advantage of the flexibility of containers and their ability to consume just the resources they require, i.e., they are *elastic*.

Elastic virtual environments expand and contract resource allocations to meet demand. Two forms of *elasticity* can be employed, based on the characteristics of the

running application: a horizontal and/or a vertical approach. The horizontal approach creates replicas of the running virtual environments in order to distribute the number of requests for a service or split a given workload amongst them. This technique is typically used with micro-services and online applications. Each replica is equal to the original and the quantity of resources allocated is fixed at the time of their creation. The vertical approach changes the resource allocation of a virtual environment dynamically, shrinking or expanding it, based on the application consumption, at run time and without restarting it. This approach can be used for applications and services that do not scale in parallel or for serial workloads. Both approaches aim to avoid performance degradation.

1.1 Motivation and Objectives

Containers, however, do offer less isolation than VMs and this can cause them to interfere with the execution of each other. A resource intensive or run away container could degrade the performance of all containers and the host. Memory, for example, is generally a resource that is difficult to reclaim without the container's application terminating or being killed. In fact, operating systems frequently kill processes when the host is running low on memory and cloud orchestrators, like Google's Borg system, kill any container that consumes more Random Access Memory (RAM) than expected. To address this, a *utilization limit* can be set to the maximum amount of each type of resource that can be allocated to the container, thus controlling the degree of vertical elasticity. While a container will not be able to acquire a resource capacity that exceeds this established limit, these limits can be adjusted during the container's execution without the need to stop or restart the container.

While greater ease and efficiency has thus motivated the increasing adoption of containers, one of the major resource management problems to solve in cloud computing (whether using containers or VMs) is predicting precisely the actual resource requirements [41, 65] of a user's application throughout the duration of its execution. From a cloud provider's point of view, foreseeing the consumption can help maximize the use of its infrastructure by intelligently allocating virtual environments across the available servers. More efficient server utilization will permit a larger number of jobs to be accommodated and reduce costs. For users, this knowledge might help avoid the unnecessary expense of contracting oversized virtualized environments.

In practice, estimating these requirements can be very difficult, especially without

previous knowledge of the service or application's behavior. Users, themselves tend to request more resources than necessary for their applications, i.e., *over-provisioning*, and thus incur additional financial expenses, and possibly longer waiting times if the job needs to be queued. All this adds up to a poorer utilization of the infrastructure and a loss of revenues for cloud provider given that servers are being sub-utilized or perhaps even being left idle. *Under-provisioning*, on the other hand, occurs when insufficient resources have been reserved for the service or application, which can cause a significant deterioration in performance or even a malfunction in the execution of the application. In both of these scenarios, this will increase the costs for the user.

Cloud orchestration platforms are responsible for allocating virtual environments (be them VMs or containers) to appropriate host servers, where they will share the host's resources with the other co-allocated virtual environments. Each host server is considered to have a fixed total capacity for each type of resource (for example, CPU cores, RAM or I/O bandwidth). Currently, it is common practice to determine the maximum amount of each resource that the user's application requires, and then define these amounts as static utilization limits for its container (in the case of VMs, the capacity of each resource allocated to the VM) for the duration of its execution. This choice is motivated by the option of a less complex approach to resource management. The Kubernetes [44] orchestrator, for example, schedules containers according to one of three quality of service (QoS) classifications:

Guaranteed – A container that is said to require a *guaranteed* quality of service will only be allocated to a host that has enough idle resources to meet the container's predefined utilization limits. This ensures that the sum of all allocated container limits is less than the total capacity of the host. Nevertheless, note that this still means some portion of the resources will remain underutilized since an application may not require a capacity equivalent to its utilization limit for its entire execution.

Burstable – Under this QoS level, the orchestrator reserves or guarantees only a portion of each resource utilization limit, known as the container *request*, but will allow a container to use additional resources, up to its limit, if available. In relation to *guaranteed* QoS, this QoS level aims to make better use of the host resources and improve system efficiency.

Best-effort – In this case, a container does not define any limit or request but is permitted to execute using whatever idle resources the host has available.

While it appears that mixing containers with different qualities of service might improve resource utilization, the problem is that both *best-effort* and *burstable* containers may be placed on hosts with insufficient resources to meet their requirements. Due to their elastic behaviour, the execution of these containers could therefore cause resource contention, by obtaining resources that were idle and then holding on to them when they are required by other containers to whom they were *guaranteed*. In this scenario, containers with no resource guarantees can indeed cause interference and affect the performance other containers including *guaranteed* ones. The only safety mechanism that Kubernetes provides is to kill containers that consume more than their resource utilization limit (if defined), but this will lead to wasted computation and resource consumption.

The aim of this thesis is to address the challenge of trying to maximize resource utilization by allowing, for example, containers with batch or latency insensitive jobs take advantage of these underutilized host resources without compromising the performance of other co-allocated containers. The amount of resources to be allocated to each container must be estimated in advance and carefully controlled to avoid performance degradation or interference between containers due to an unfair distribution of resources. Different from the existing solutions, this problem cannot be solved efficiently by exploiting elasticity alone. Any attempt risks the possibility that resource demands exceed availability so resources might have to be reacquired by the orchestrator and redistributed by throttling, suspending or migrating certain containers. Such an orchestrator requires an enhanced resource management strategy. The benefits, given the existence of some applications that can tolerate delays, will allow these container-based cloud platforms to further consolidate workloads to improve utilization, reduce energy consumption and save in costs for both users and cloud providers.

1.2 Contributions

Given the importance that RAM memory availability plays in application performance, this work has designed a container orchestration tool called VEMoC (*Vertical Elasticity Management of Containers*). The tool exploits vertical memory elasticity to allow cloud providers to increase server utilization, without a deterioration in individual container performance. Note the focus of this work is not implementing elasticity in containers but rather on its orchestration. While container engines implement resource elasticity inherently, under a model of infinite resources, when resources are limited, more complex scheduling decisions beyond the scope of these container engines must be taken. For ex-

ample, if host memory becomes scarce, which containers would be chosen to “collaborate” by donating some or all of their allocated resources to others in need? Future work on VEMoC will look to incorporate other resource types such as CPU cores and I/O bandwidth in to the framework.

The principal contributions of this thesis include:

- The design of the VEMoC scheduling algorithm to manage the distribution of the host’s memory amongst the running containers allocated to that host. Described in detail in Chapters 4 and 5, the algorithm executes a sequence of 7 stages in each scheduling interval to manage the life-cycle of the set of queued containers. The algorithm incorporates a number of novel features:
 - It integrates container life-cycle management, including the pausing and suspension of containers (Section 5.7), and memory elasticity of containers into a unified decision making process.
 - The scheduler operates at a higher temporal resolution than existing elasticity management tools by not having to depend on the synchronous collection of monitoring data relating to the host and running containers. This allows VEMoC both to reduce the amount of extra container capacity that needs to be reserved for unforeseen demands (but is generally unused) and to react faster to changes in individual container requirements. In both cases this helps significantly in improving memory utilization efficiency.
 - Employs a novel and more precise model to predict a container’s memory consumption (Section 5.3) that does not require apriori knowledge about the requirements of the applications running in the container.
 - To address the failings of previous work, particularly in terms of resource utilization efficiency, resource allocation adjustments are not fixed in size nor are they only triggered when statically predetermined thresholds are reached. VEMoC also adapts to changes in the behavior of the operating system as the available memory on the host becomes scarce.
 - In the context of memory consumption, it is prudent to reclaim memory no longer being used by a container. Cloud environments may or may not have automatic garbage collection schemes and, if they do, they often have a significant impact on performance. To address this, VEMoC proposes the use of a generic approach referred to as *memory stealing* (described in Sections 5.4 and 5.5).

- To evaluate the VEMoC scheduling algorithm and compare it with existing approaches, a modular, scalable container orchestration system (Chapter 4) has been implemented, with a reusable and expandable framework for the resource management of LXC and Docker containers. This framework can easily be used to develop and test other scheduling algorithms and provides support for other containerization technologies. Additionally, it supports the development of algorithms to manage multiple instances of VEMoC.
- An experimental evaluation on a physical server has shown VEMoC to be more efficient than existing approaches adopted by production services such as [44]. This is the case not only in terms of memory utilization (with VEMoC constantly sustaining utilization efficiencies above 90% while traditional threshold approaches have their peak utilization efficiencies limited by their chosen threshold value, typically between 70% and 80% [2, 48]) and job turn around times (where VEMoC can be as much as twice as fast) but also in terms of lowering costs for clients, being at least 50% more efficient in the majority of the comparisons.

1.3 Thesis Outline

An introduction to containerization is given in Chapter 2, presenting the core concepts and technologies used to create and manage containers as well as citing a few examples of commonly adopted implementations. The chapter also focuses on the existing elasticity techniques available in virtual environments and some of the ideas used as a basis for the design of the VEMoC architecture. Chapter 3 summarizes some of the related work in the literature with a particular focus on research into resource elasticity and memory elasticity in virtual environments (both virtual machines and containers).

Chapter 4 describes the cloud orchestrator framework developed for VEMoC that is composed of two types of controllers: the Host Manager (Section 4.2) that is used to manage the life-cycles of the containers during their respective executions on a server, and; the Cloud Manager (Section 4.1) that receives and manages the system's job requests and collects monitoring data for from all available hosts to aid global scheduling decisions and for further analysis. A detailed description of the VEMoC algorithm is provided in Chapter 5. This includes the steps designed to dynamically manage memory allocations on a given host in order to control the distribution of the available memory between co-allocated containers, and to maximize the memory utilization with minimal impact on the

performance of the applications in execution. Some experimental evaluations and analysis of the results are presented in Chapter 6, including the costs of transitioning through the stages of a container’s life cycle, the overhead of the VEMoC framework as well as comparing VEMoC’s management policy against other three mechanisms commonly used by enterprise orchestrators today, in terms of execution times, memory utilization and cost. VEMoC is shown to be able to manage multiple containers simultaneously, without losing performance or significantly increasing the latency of its execution. Finally, Chapter 7 draws some conclusion and indicates promising avenues for future work.

Also included in this thesis are two appendices with additional background material. Appendix A complements the motivation for this work with an initial case study to compare the performances between the use of containers and virtual machines to run scientific applications. The results of this study helped make the decision to focus on the use of containers as a technology for creating virtual environments in a cloud. Appendix B provides the theoretically foundation for a key, novel aspect of the seven stage VEMoC algorithm – how it uses rates of change in operating system memory metrics to classify containers in its second stage, which determines the respective actions taken for each container at later stages. The simplicity of Algorithm 3 (Section 5.3.2) is derived from an analysis of all possible memory consumption trends that could occur during the execution of a container, based on the memory usage metrics collected by the VEMoC framework.

Chapter 2

Container Virtualization

This chapter provides a brief overview of container virtualization technology, its concepts, how it works, and some of the most common examples in use. To finish, the chapter discusses how this technology can be used to create an elastic environment that is able to adapt to the demands of the application(s) running inside each container.

According to Bernstein [9], most commercial cloud infrastructures, whether for applications or services, widely employ hypervisor-based virtualization technologies, such as VMware [78], Xen [16] and KVM [45], including Amazon Web Services (AWS) and Microsoft Azure. However, some companies are already offering container-based services, such as Amazon EC2 Container Service (ECS), Google Cloud Container Engine and Azure Container Service (ACS), capable of deploying and managing container clusters, as well as providing the necessary scheduling and load balancing across the available physical resources. This new type of service, called *Container as a Service* (CaaS) [61], is being offered by cloud providers as an intermediate service model between that of *Platform as a Service* (PaaS) and *Infrastructure as a Service* (IaaS). However, of these service models, Container-as-a-Service is distinguished by a fundamentally different approach to virtualization: the use of container technology. While computational resources in IaaS are offered to the user in the form of a VM or a container, and in PaaS, virtual environments are offered for specific purposes such as a database service or a web server, in CaaS, containers are available to load images that can contain applications or services ready for use (*microservices*), with the advantage of a lower consumption of computational resources, comparable performance and being more scalable [66].

Container virtualization is not a new concept, as pointed out by Almeida in [23], as it was based on the command *chroot*. *Chroot* [36] was released in 1979, on Unix version 7, as a command to segregate a user's access to the system's root directory ("/" or *root*).

Later, in 2000, this concept was expanded to the *jail* command in the FreeBSD version 4 [72], allowing, in addition to the file system, the network and users to be also separated into subsystems. In 2005, Sun Microsystems launched its Solaris 10 operating system with a functionality known as *Solaris Zones* [62], which is considered one of the origins of container virtualization as it is known today. Each Solaris Zone created an isolated environment for the execution of applications sharing the same image of the operating system, with the computational resource capacity being limited. Such an implementation allowed applications to run without any knowledge of, or suffering interference from, other applications running in different "zones", even from those running with administrator privileges. It was later renamed to Solaris Containers, when the term gained popularity. Currently, one of the main container virtualization technologies on the market is Docker [25], developed by the dotCloud group. Docker was initially developed based on the Linux Container virtualization technology, LXC [14] but currently uses its own virtualization library, the *containerd* [17], maintained by the Open Container Initiative [46]. Much of the work presented in thesis derives from the use of LXC as its base containerization technology.

2.1 The Core Concepts of Virtualization

Currently, a number of competing virtualization technologies exist and differ from each other in the way virtual environments are created, the level of isolation of environments and how computational resources are reserved and used. First, they fall in one of two groups, the majority implement complete virtualization or simply **virtualization**, the second, containerized virtualization or simply **container**.

Complete virtualization aims to create instances called virtual machines (VMs), that are virtualized environments totally isolated from the server or host operating system (OS). VMs have their own operating system that, for example, allows one to have a Windows environment running on a Linux servers or vice versa. VMs have their own set of libraries and services, in addition to their own set of applications, creating a complete environment capable of simulating a real machine. There are two ways of implementing full virtualization, one using the virtual machine manager as a server application and the other where the manager is embedded in the server's OS or is the OS itself, also known as **paravirtualization**.

Under paravirtualization, the virtual machine manager is "embedded" in the kernel of the server OS, through the use of modules, to create a system commonly called *bare metal*,

meaning one that is close to the hardware. This was considered to allow gains of performance through more direct access to the server's computational resources. Examples of this type of model include VMware ESXi [78], Xen Server [16] and KVM [45].

In the case of complete or full virtualization, a virtual machine manager is an application that runs on the server OS, which generates an extra management layer to translate operations. While this can hurt performance in relation to paravirtualization, the performance difference between the two models has fallen drastically due to chip manufacturers incorporating hardware support for virtualization. Given full virtualization does not require modifications to the server OSs, this model has become the most popular. Examples include VirtualBox [54] and VMware Player[77].

To allow access to the physical hardware, VMs use *drivers*, a set of virtual devices that simulate the hardware devices on the server and that perform the function of transmitting access instructions between the VM and the hypervisor. Due to their structure, VMs are considered robust yet heavy environments because they incur small but not insignificant performance overheads compared to the physical machine, and tend to need large storage spaces to hold the complete installation of a new OS.

Today, the primary means of managing VMs is through the **hypervisor**, a set of modules and dedicated instructions for virtualization that are sent to the kernel of the server OS to allow the creation and management of VMs and obtain access to available hardware devices. The hypervisor is responsible for the allocation and use of computational resources by VMs, as well as the control of communication (network) and storage (disk), and thus is the "brain" of the virtualization system. A VM is "interpreted" by the server OS as a single process, regardless of what is running internally, occupying and consuming the resources reserved by the hypervisor.

Container virtualization does not differ from other virtualization technologies in terms of its objectives: to create virtualized environments for the isolation of applications and services, and; to share the computational resources of the same server. But it does differ in the way these environments are created and how they access computational resources available on the server hardware. Figure 2.1 presents a comparison between the structure of a complete virtualization system (Host 1) and the containerized virtualization system (Host 2). The complete virtualization model, also called type 1, is considered the most used virtualization model today, mainly in the cloud, and is considered the main competitor of container virtualization.

In containerized virtualization, the isolation of the virtual environment is reduced to

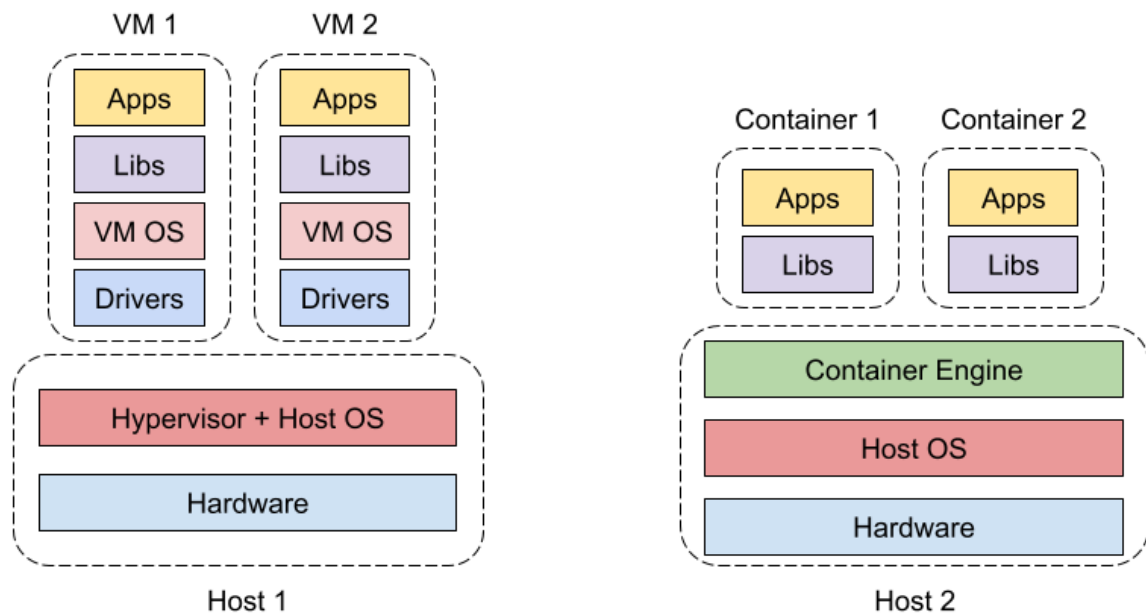


Figure 2.1: Full Virtualization (Host 1) x Containerization (Host 2).

only the libraries, services and applications that the user needs to perform their tasks [32]. In containers, the kernel and OS structures are shared between the server and the containers. Due to this sharing of structures with the host's OS, a container does not need layers of hardware "emulation", which means its performance is close to that of the physical machine. Allowing the container's size to be reduced facilitates its storage and efficiency in the use of computational resources, and is thus often called *lightweight* virtualization [75] since this permits faster start-up times and lower overheads [56]. Note that the virtualization layer runs as an application on top of the operating system (OS) and thus may also be referred to as *operating system virtualization*.

The management of the containers is performed through a service or management application, which allows the creation and maintenance of the containers inside the server OS, using isolation layers called *namespaces*. Due to the weak isolation of the virtual environment, each application or service running internally in a container is viewed as a server OS process, which can have its status modified by the server administrator (root). The containers can only see the status of their own processes.

Another important feature in containers is that they do not require specialized hardware to function and can be run on any machine with an OS capable of running them, including old hardware and different types of architectures. VMs, on the other hand, require that the server have specialized hardware for their correct operation, for example,

the processor should have enabled a set of additional instruction, such as Intel VT [42] and AMD-V [1], which allow access and control of physical machine devices.

2.2 Container Namespaces

A **namespace** is a feature of the Linux kernel that partitions kernel resources so that one group of processes may see one set of resources while another group of processes will see a different set [10] [11]. For example, each process running on a Linux machine is enumerated with a process ID (PID). Each PID is assigned a namespace. PIDs in the same namespace can have access to one another because they are programmed to operate within a given namespace. PIDs in different namespaces are unable to interact with one another by default because they are running in a different context or namespace.

Containers, being symbolic virtualized instances within the server's operating system, use this feature for isolation. A process running in a container under one namespace cannot access the namespace related information outside of its container or information running inside a different container. In fact, a container is composed of several namespaces to limit the visibility that a group of processes has over various system entities such as process trees, network interfaces, user IDs, and filesystem mounts. Thus, namespaces are categorized into one of several groups:

- **cgroup - control group** is one of the principal namespaces of a system, responsible for allowing and/or restricting access to a given computational resource (such as CPU, memory, input and output, network or a device) by processes belonging to a given group.
- **Network** - allows the isolation of system resources connected to the network, whether they be devices, protocol stacks, routing tables, firewalls, ports, sockets, among others.
- **Mount** - The mount namespace is used to isolate mount points such that processes in different namespaces cannot view each others' files, thus providing isolation between container file systems. It creates a subtree of directories for the container, just like the existing file system on the server, but prohibits access to directories above its mount point (where the container was created).
- **PID** - this group isolates the processes running inside a container under the same identifier on the server. This makes it possible to change the states of the processes

of a container as a group, and thus permit functionalities such as the migration of containers between servers.

- **User** - allows the mapping of user identifiers (UID) and group identifiers (GID) between the container and the server, facilitating the identification of the owners of the running processes and increasing security.
- **UTS** - this group provides the isolation of *hostname* and domain (*NIS domain-name*), allowing a container to have its own identifier on the network.
- **IPC** - handle the communication between processes by using shared memory areas, message queues, and semaphores, etc., avoiding conflicts with instances belonging to other groups.

2.2.1 Computational Resource Allocation

The allocation and reservation of computational resources for a container is done through the **cgroups** namespace. *control groups* (cgroups) creates an independent hierarchical system, in the form of a tree, for each type of computational resource (memory, CPU, I/O, network and devices) available in the operating system, associating each of them with a set of processes or tasks, as well as containers [47]. Each leaf node in the tree represents a set of processes, using the same identifier and that will share a certain amount of that resource. Usage limits (quantity, priority, and access) can be defined for each node in each of the existing subsystems.

In Figure 2.2, there are two examples of how cgroups organizes two of the most used resources in a container, the CPU and the memory. The cgroups creates subtrees for each type of task, in this case, containers and databases, and their leaf nodes are represented by a unique identification. The same leaf node identifier will be used in each hierarchy, with only the limits for the use of that particular resource being informed [59].

For processing (CPU), limits can be set with regard to access to the cores of a processor, the execution time on the processor (time slot), or weights for sharing the processing time between groups that will use the same features. In the case of memory, limits on the maximum amount of memory that a given container can access can be imposed. These limits restricts the amount of memory a container can access when there is unused memory available, preventing greedy processes from using all the host resources. Also these limits can be used for several scheduling decisions: suspending a container for lack of

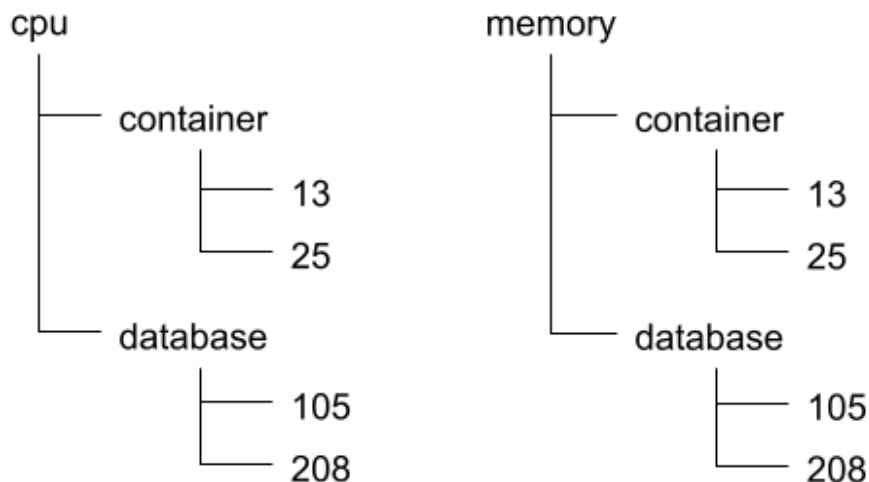


Figure 2.2: *cgroups resource organization for cpu and memory resources allocation.*

memory; increasing the amount of memory and even migrating a particular container due to a lack of resources.

For disk access (*BlkIO*), we can use cgroups to control access to disk read and write functions, whether this limitation is for a particular device, or sets different weights between containers, this enables access privileges for a certain container. The same is true for network traffic, with traffic classes and priorities for outbound traffic being defined for a given container. Incoming traffic is not controlled by cgroups. Setting weights for disk access and network traffic, can prioritise a container with critical communication and storage services, in detriment of general purpose containers executing in the same host, when these resources suffer a lot of competition. If the network or the disk is free, any container can use them without restrictions. In addition to accessing resources, cgroups also controls access to certain devices in the system, allowing tasks to read or write from, for example, GPUs or even manipulating the network interface to create communication tunnels.

2.2.2 Disk Storage

Traditionally, virtual machines (VMs) store their operating system, program libraries, applications and services within virtual drives called *disk images* on a server, the format of which may vary according to the hypervisor used [71]. This disk image acts as an isolated storage system for each VM and prevents the data stored within a VM from

being visible to the server or other VMs. This disk image is a file created together with the VM, specifying the maximum size it can occupy, disk update methods and the persistence of the stored data. Note, however, that a disk image of a VM can occupy large amounts of disk space on the physical machine and so may be divided into several smaller files. When creating the VM, there is an option for the disk to occupy its maximum size, the entire defined space (*fixed-sized file*), or occupy it incrementally, during the life time operation of the VM (*copy-on-write file*) [71].

Containers are created from a template compatible with the kernel of the host server OS, and stored on the host's file system or remotely in the cloud, in which case, the image must be downloaded before the container is created. Unfortunately, storing these container images using existing file systems can be inefficient in terms of disk space and affect the time to start a container.

In Linux, a container is considered to be a process, but the only way to create a new process is forking the existing process. The fork operation creates a separate address space for the child process that has an exact copy of all the memory segments of the parent process. To create a new container, all the files of the image layers need to be copied into the container's namespace. A container is expected to start in a few milliseconds. If a huge payload is needed to be copied at the time of starting a container, this can significantly increase the time to boot the container.

Union File Systems [26] are used to help efficiently share physical memory segments among containers. A Union File System works on top of the other file-systems to give a single, coherent and unified view of the files and directories of separate file-systems. In other words, it mounts multiple directories to a single root, creating an illusion of merging contents of several directories into one without modifying its original (physical) source, as exemplified in Figure 2.3.

Following the *chroot* concept, a folder for the container is created as a subdirectory on the server (which will be the root of the container) that recreates the OS directory structure, based on the container image used for its creation. In this model, the containers will initially occupy the size of the image used but may be increased in accordance with the creation of new files and the installation of new applications and services internally, up to a size limit defined by the virtualization system administrator. An internal user of the container does not have access to the directories and files of the server system, except if there are mappings between the directories and appropriate access permissions, as observed in the */bin* directory of the test container, in Figure 2.3. However, the

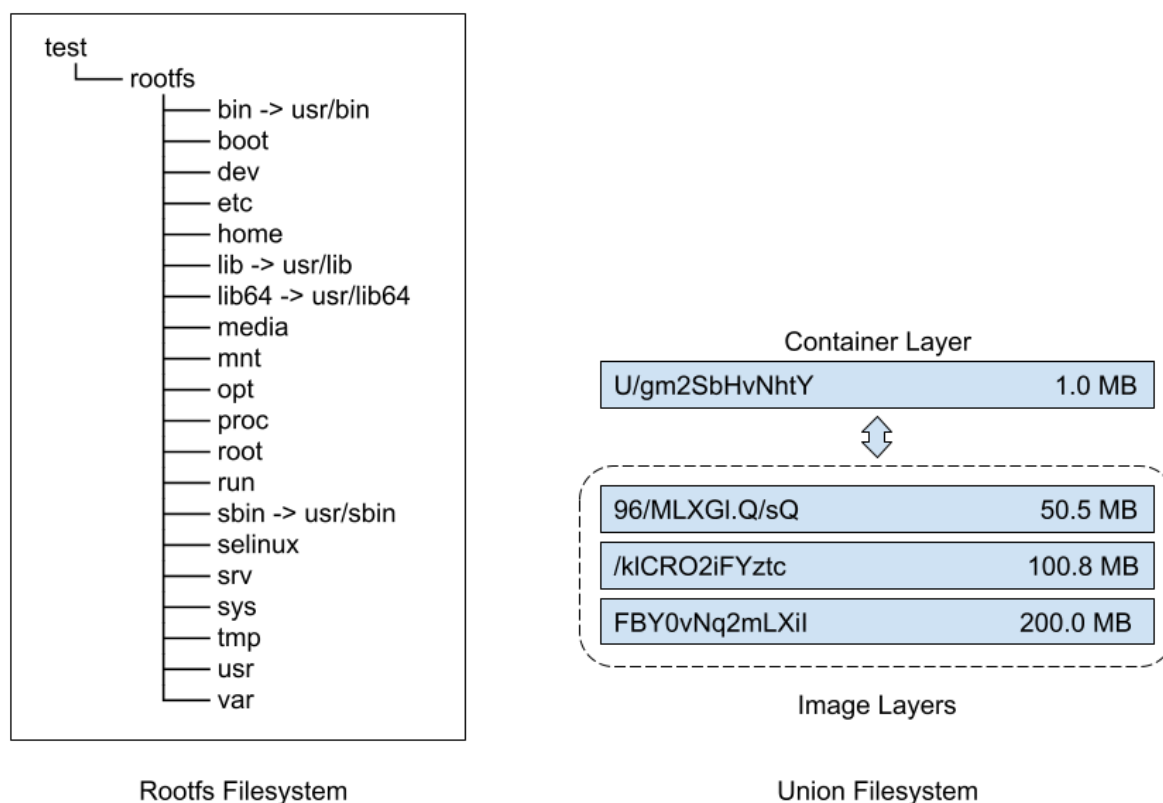


Figure 2.3: The Rootfs File System and Union File System container storage types.

system administrator has access to all directories, being able to change files and internal directories of the container, which reduces file security and reduces the isolation between virtual and server systems.

A container is composed of multiple branches or *layers*. A sandbox of a container is composed of one or more image layers and a container layer mapped using hash algorithms for identification. When created, a container adds a new container layer, with read and write privileges on disk, where any modifications made will be stored, while the other layers below do not change, being configured as read-only content.

Under the Union File System, internal users of the container will continue to view the file system as if they were on a real machine, with a traditional OS directory structure and files, while the server administrator does not have access to the internal content of the container. This increases the security of services and internal data as well as the insulation of the container. It is also possible to map a server directory for access within the container, such as a volume or disk attached to the container, which can be read-only or with write permissions as well. The example of Figure 2.3 shows an image initially containing three layers, where a layer was added to each new version of the

image, containing their respective modifications. The container layer, created as a result of the creation of the container, is integrated with the image layers. In addition, the images can be shared between several containers, allowing a reduction of the total space needed for storage of the containers, a control of its content (based on its versioning) and its distribution through the cloud, since its layers are only accessible to read and each container creates a separate layer to write its own data.

2.2.3 Communication between Containers

A container, initially, is an environment completely isolated from the network, not accessible via the Internet or even the Intranet, thus is a relatively secure environment for critical applications that involve processing sensitive data. However, container virtualization technologies, such as LXC, allow containers to communicate with other containers on the same server, via an internal network or *bridge*, interconnecting them. This communication is useful in cases such as: services stored in different containers, such as a website and its database, for example.

It is also possible to expose the container to the external network by opening specific ports on the server where a container can start listening to communication requests, an approach widely used by online services and websites. However, a problem encountered is the communication between containers running on different servers. Systems like Docker have some technologies that overcome this limitation and allow this type of communication, such as an *overlay network* [27] where the messages exchanged between the participating containers are sent through *gossip* protocols using different types of network tunnels.

2.3 Container Virtualization Technologies

Although the concept of containerization and process isolation has been around for decades, it has recently become a major trend in software development, encapsulating or packaging up software and all its dependencies so that it can run uniformly and consistently on any infrastructure. The technology is quickly maturing, resulting in measurable benefits for developers and operations teams as well as overall software infrastructure.

Containerization allows developers to create and deploy applications faster and more securely since applications need only be “written once and run anywhere.” This portability is important in terms of the development process and vendor compatibility. However,

other notable benefits include fault isolation, ease of management and security. Together, containers and cloud computing are bring application development and delivery to new levels not possible with traditional methodologies and environments.

Several competing container virtualization systems can be found in use, each with different objectives and purposes, serving various market demands such as companies with large scale services, research laboratories, application developers, and even ordinary users. They can be separated into container technology and container managers. The former create and manage containers on just one machine, like Docker and LXC, while the latter manages containers on multiple machines simultaneously, like Apache Mesos and Kubernetes, using one or more of the existing container technologies.

To run a containerized system, we only need a SO that supports one or more of these containerization technologies and container images compatible with the machine's SO. The most common SO used to run containers is any distribution based on Linux Kernel.

2.3.1 Docker

Docker [25] is one of the most widely adopted container virtualization technologies on the market. It was developed by DotCloud, a company offering platform solutions for developers. Initially it was designed so that developers could have development environments that facilitate the sharing of the application among their teams, providing a shared environment and versioning control, in addition to allowing the application to be deployed quickly at a client, without the need to configure the host server, thus providing a ready-to-use environment.

It was originally based on the LXC virtualization system, but later developed its own container virtualization library, *libcontainer*, currently maintained by the Open Container Initiative (OCI)[46] as *Containerd* [17], with the aim of creating a standard for the creation and use of containers. Today, it has several solutions for containerizing services and applications, tailored to meet the needs of different market niches.

2.3.2 Linux Containers - LXC

LXC [14] is an interface that allows you to create containers using the process isolation functions present in the Linux kernel. It was based on *chroot*, extending its original functionality and now offers tools and libraries for the use and development of new container-based solutions. These tools allow the simple and easy creation of containers, without

the need to modify the kernel of the system. Currently maintained by Canonical Ltd., the developers of Ubuntu, it has been modified into a new technology to conform to the OCI standard. Consequently, support for LXC has been reducing as investment is moved to the development of LXD [15], considered to be the successor to LXC. The LXD can manage containers on multiple machines and is capable of running a entire Linux System, from other distributions, inside a container, like a virtual machine can.

2.3.3 Apache Mesos

Apache Mesos [6] is an application isolation solution based on the Linux kernel, but with a different level of abstraction. It is a container manager that offers a client-server environment with features for managing computer resources and scheduling tasks for applications such as Hadoop, Spark, and MPI. Its manager is capable of running instances of these applications on different machines or in the cloud, by requesting the application scheduler for new tasks to be performed on their nodes, through the Mesos Master service. It offers linear scalability, fault tolerance, and high availability mechanisms and is compatible with other container virtualization technologies, such as Docker.

2.3.4 Google Kubernetes

Kubernetes [44] is an open-source system developed by Google, for the creation and management of containers, offering automatic provisioning environments for users' applications. It uses a structure called a **pod**, which can be composed of several containers spread across the network, but which work as a single environment for the application to run. The environments are scalable and incorporate recovery mechanisms and load balancing services. Kubernetes can use several container technologies for its pods. One of the most frequently used is rkt [64], an container engine capable of running pods directly in the Unix process model, without the need for a central daemon running in the SO.

2.4 Cloud Elasticity vs Scalability

One of the main characteristics of cloud computing is the offer of virtualized environments capable of meeting the demands of users' applications and services. *Scalability* addresses the changing needs of an application within the confines of the infrastructure by statically adding or removing resources to meet the application demands if they change between

executions. The purpose of *elasticity* is to match the resources allocated with actual amount of resources needed by the application at any given point in time of its execution. In most cases, both of these are handled by adding resources to existing instances — called scaling up or vertical scaling — and/or adding more copies of existing instances — called scaling out or horizontal scaling.

Elasticity is the ability to grow or shrink infrastructure resources dynamically as needed to adapt to workload changes in an autonomic manner, maximizing the use of resources. During its execution, an application's behavior may vary, impacting the consumption of resources, and where the lack of availability in turn may affect its execution. Furthermore, if the virtual environment where it is being executed has not been configured appropriately, an unexpected execution error may occur. To avoid such problems, one of two elasticity techniques can be used to allow the virtual environment to change its capacity during the execution of the application: horizontal elasticity of a *virtual cluster*, and; vertical elasticity of containers.

2.4.1 Horizontal Elasticity of Virtual Clusters

The horizontal elasticity is directly linked to the running application or service within a virtual cluster. In this model, additional instances or containers are added to, or removed from, the virtual cluster within the cloud, depending on the amount of work the application has to perform.

This is most common form of elasticity implemented in clouds, especially for: running Web services where requests can be distributed across an elastic set of replicated service instances; task management services, where the number of servers in the cluster can change, or; autonomous parallel applications, capable of managing their own workloads, depending on the size of the environment where they are being executed.

A cloud management system that uses this technique can request the creation of new containers, called **replicas** that use the same image of the application that is running in the primary container, and form a larger virtual cluster when the consumption of resources of each container rises close to their reserved limits, as seen in Figure 2.4. With more instances, the application will be able to re-balance the load between them, avoiding overloading the existing containers and possible improve the QoS of the application or the service in execution.

Just as new containers can be added to a virtual cluster, a management system can

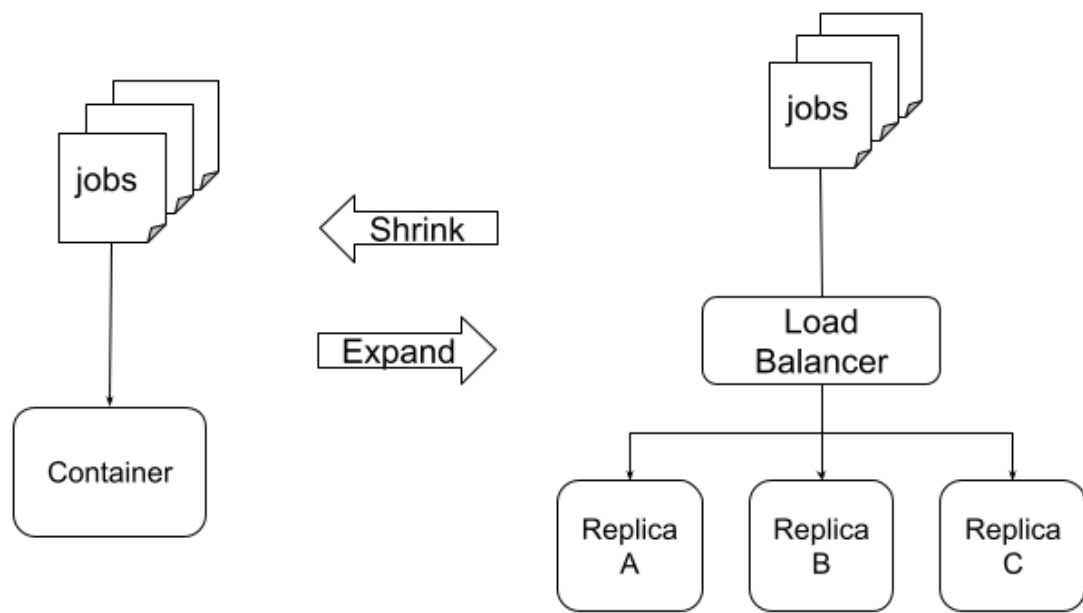


Figure 2.4: Container Horizontal Elasticity

also remove containers if the demand for the service running on the cluster is low. In this case, computational resources will be released to the cloud provider for the creation of other containers for other applications. This offers providers the opportunity to increase the utilization of the cluster, reducing its idleness.

2.4.2 Vertical Elasticity in Containers

Vertical elasticity is the possibility of increasing or decreasing the amount of computational resources in a given virtual environment, based on their demand. Using the resource consumption information from the container, a manager can perform one of the actions proposed and exemplified in Figure 2.5:

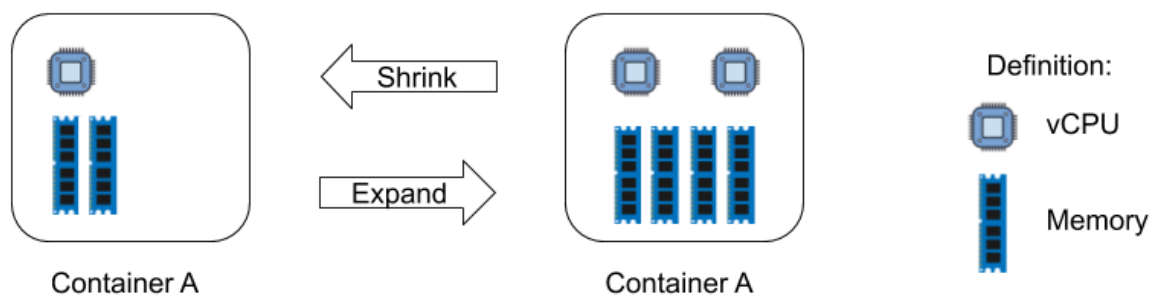


Figure 2.5: Container Resource Vertical Elasticity

1. **Expanding resource capacity** - In this scenario, the application needs more

computational resources than it currently has available for use, which can result in loss of performance and slow execution. The controller, having spare computational resources available, can allocate more resources by an amount compatible with the application's need, thus reducing performance loss. For example, this might be in the form of an increase in the amount of memory that the container can access or the number of vCPUs available for its applications.

2. **Shrinking resource capacity** - In this scenario, a container has more resources than it needs at that given moment, either because it has a reduced workload or because the heaviest part of an application has already been executed. In these cases, the controller can return an amount of the allocated resources back to the host, making it available for allocation to new containers or to others already in execution. For example, when there are sub-utilized or idle vCPUs in a container.

These elasticity techniques can be applied to different types of resources, in particular, for CPU and memory. In the case of CPUs, adding or removing vCPUs will depend on the number of tasks running in the container. One can also define **quotas**, different weights for containers that share vCPUs, defining priorities between them. In this model, larger slices of capacity can be given to applications that need more processing power. Allowing a higher level of sharing can be especially useful for clouds with limited resources, for example, cloudlets or edge computing. In the case of memory, one typically defines the maximum limits on the amount of memory that a container can use.

To support this objective, LXC takes advantage of cgroups [47] to define limits and control access to the resources and devices of a server. With this technology, it is possible to define weights or quotas of CPU usage, allowing a scenario where one can maximize the allocation of virtual environments, using slices of processing time, particularly useful in environments with a high degree of resource sharing and limited amounts of available resources.

This thesis work extends the ideas and concepts of managing memory elasticity in virtual environments presented by [68], in the development of MEC. Unlike MEC, which works with the elasticity of memory in VMs, this thesis is directing its research towards the dynamic sharing of resources, working with the elasticity of memory in containers. The MEC uses memory distribution over VMs, while the VEMoC focuses on managing memory allocations to containers, by adjusting their memory limits dynamically, with LXC and cgroups.

Chapter 3

Related Work on Memory Elasticity

While there exists a number of definitions in the literature for *elasticity* [7, 18, 37, 40], this work defines elasticity as the ability of a system to add and remove resources (such as CPU cores, memory, VM or container instances) “on the fly” to adjust the quantity available to an application in real time. The economics of cloud computing is built on the premise of *multi-tenancy* - where public or private cloud architectures are designed to share computing resources between multiple users. Elasticity is thus one of the fundamental characteristics necessary for cloud computing in order to scale and contract computing resources according to the demands of the application workloads in a timely fashion.

Implementing elasticity efficiently is of financial importance to, and is still a major research challenge for, cloud computing providers. In the literature [3, 18, 37, 51, 52], much attention has been given to the elasticity of virtual machines, while there is comparatively little, as yet, on the elasticity of containers. Furthermore, most of the approaches focus on horizontal elasticity, due to its simplicity as it does not require any extra support from the virtualization engine. Only a few address vertical elasticity, and fewer still focus on memory. Given this relative lack of study and the importance memory has on the performance of applications, especially in the context of high impact fields of research such as Big Data, Artificial Intelligence and Deep Learning combined with the trend towards multi-tenant clouds, this thesis addresses the problem of vertical memory elasticity. Therefore, this chapter briefly discusses some of the most relevant research and tools from the literature that focus primarily on this topic.

Studies such as [21], among others, initially motivated the use of vertical elasticity, in particular, of vCPUs in VMs, by showing that an elastic VM implementation performs better than using multiple VM instances each with a single vCPU. The reason for this

being the fact that the former consumes less resources and avoids increasing overhead costs when scaling-up vCPUs. While many approaches show that they can improve the utilization efficiency of the resource, the majority fail to tackle the other half of the same problem: The question of how to allocate appropriate amounts of a limited resource between competing applications.

One can find a wide range of resource management tools from proposals in the literature to enterprise class tools employed extensively in production, including container-based cloud orchestrators like Docker Swarm [30] and Kubernetes [44], elastic management of VMs, e.g. [13, 34, 48, 68] and, more recently, elastic management of containers [2, 5, 22]. Containers themselves may be deployed on bare metal or, as is common in public clouds, inside VMs [4, 63].

Enterprise class orchestrators [65] have typically focused on horizontal elasticity, changing the number of containers (or pods, as referred to in the context of Kubernetes) based on demand (e.g. requests/second of online applications). Here, the number of containers is scaled up or down according to predefined rules. In an experimental phase, vertical pod autoscaling allows Kubernetes to readjust the pod resource limits, however, each change requires the pod to be restarted, thus limiting its effectiveness [39].

Academic research, like [22] and [5], have applied horizontal elasticity to containers by creating and removing container replicas within a virtual cluster based on the load or the number of application tasks within their architectures. De Alfonso *et al.* [22] used a container to act as a front-end master to receive requests and create worker nodes (container replicas) to then handle each request using Docker Swarm and its scheduling policies to decide where to allocate the replica. This type of approach works well for online and distributed applications, where either tasks are executed based on requests for a unique application (online) or a workflow is divided into different (distributed) tasks. The quantity of resources can then be elastically grown to support the required number of tasks. However this approach cannot benefit serial applications, which are typically offline and executed in batch, but need to execute in a single virtual environment. In this case, the capacity of the environment need to be increased.

Extensive research on container elasticity exists [3] where the main focus, like [22] and [5], has been on horizontal elasticity for applications that can have multiple instances or be distributed. However, vertical elasticity is claimed to be more applicable, efficient, and faster [2, 56] given resource limits are adjusted in line with the run time consumption, without the need for additional hosts, tools, and, in some cases, licenses. Previous work

by [8], [34], and [48], for example, have applied this vertical elasticity to adjust the amount of memory of VMs. Baruchi and Midorikawa compared two metrics for vertical memory elasticity, the Exponential Moving Average of the memory utilization and the number of page faults, and concluded that the latter leads to a better performance when used as the main criteria for allocating memory [8]. The work in [34] proposed a hybrid approach that takes into account the application performance and the resource utilization when realizing vertical elasticity of memory. Control theory is used to synthesize a feedback controller that meets the application performance constraints by auto-scaling the allocated memory, at run time.

Many of these approaches have characteristics in common. Often vertical re-sizing is only triggered when reaching a predefined percentage of capacity or performance. As these amounts or rate increase, the system effectively becomes less sensitive to changes and less efficient. Also, the majority of these proposals only consider the execution of online applications, and when resources on the host become scarce, they always assume that additional capacity is available elsewhere in the environment. Sawamura *et. al.*, however, consider pausing or migrating them should there be insufficient resources to meet the demands of all executing VMs simultaneously [68].

More recently, these elasticity techniques have begun to be applied in a similar fashion to containers [57]. ElasticDocker is an example of a tool to manage CPU and memory limits, making adjustments when consumption reaches a predefined upper or lower threshold [2]. Their approach to manage vertical elasticity in containers, also involves invoking live migration when no more resources are available on the host [2]. However, approaches based on thresholds and fixed elasticity adjustment ratios can fail to provide enough resources in time and, in practice, cause container performance to degrade.

The Vertelas framework supports vertical elasticity when executing containers inside VMs on Amazon Elastic Compute Cloud (EC2) [63]. The scheme maintains multiple VMs, each with different memory configurations, on standby in a suspended state. If a container has insufficient memory, Vertelas suspends it generating a checkpoint with CRIU (Checkpoint/Restore in Userspace) [19] and starts one of the standby VMs with more memory to which it migrates and resumes the container. In effect, this solution does not actively manage the resources of containers, nor is it interested in improving the utilization of already deployed VMs.

Different from ElasticDocker, this thesis uses a set of memory consumption metrics, collected periodically, during a scheduling cycle, to define the amount of memory a con-

tainer may need or give up, over the next cycle, without interfering with the performance of applications running in co-allocated containers. It also allows for pausing or even suspending some containers when resources become scarce so that others can finish their tasks without being prejudiced. These preempted containers resume their execution when enough resources are once again available. More details of our proposed approach can be seen in Chapters 4, which describes the tools components and architecture, and 5, which describes the functionalities present in the orchestration strategy.

Chapter 4

The VEMoC Architecture

This chapter describes the VEMoC framework, its architecture and components, which have implemented to create an elastic container service so that the VEMoC orchestration algorithm (described in Chapter 5) can be evaluated within a cloud-like environment. This architecture is based on how the enterprise container orchestrators manage containers within infrastructures composed of multiple servers, commonly adopted by the principal cloud providers on the market. Typically, cloud providers aim to offer services that allows a user to submit a container image to run their application, within the provider's infrastructure, with little or no knowledge of where their application will run. Our architecture continues to support this model and is designed so that it can be easily incorporated into existing cloud container services or perhaps, more usefully, allow users to create their own portable container service within the VMs they rent from enterprise cloud providers or run on their own infrastructures.

Our VEMoC architectural model has been design to sit between a container orchestration platform and a container engine with the aim of managing the life cycle of containers in a cloud environment or edge infrastructure. It is composed of two management levels: a Cloud Manager and multiple local Host Managers, as presented in Figure 4.1.

As shown in Figure 4.1, the Cloud Manager (CM) is responsible for managing the infrastructure and the user's (jobs) requests. The CM receives monitoring information regarding each of the hosts that comprise the cloud infrastructure and their respective containers. With this information, the CM aims to determine an ideal initial placement of containers to execute the jobs requests it receives.

Each Host Manager (HM) is responsible for creating, executing, and monitoring the containers allocated to it by the CM. In addition, it must also decide how to distribute

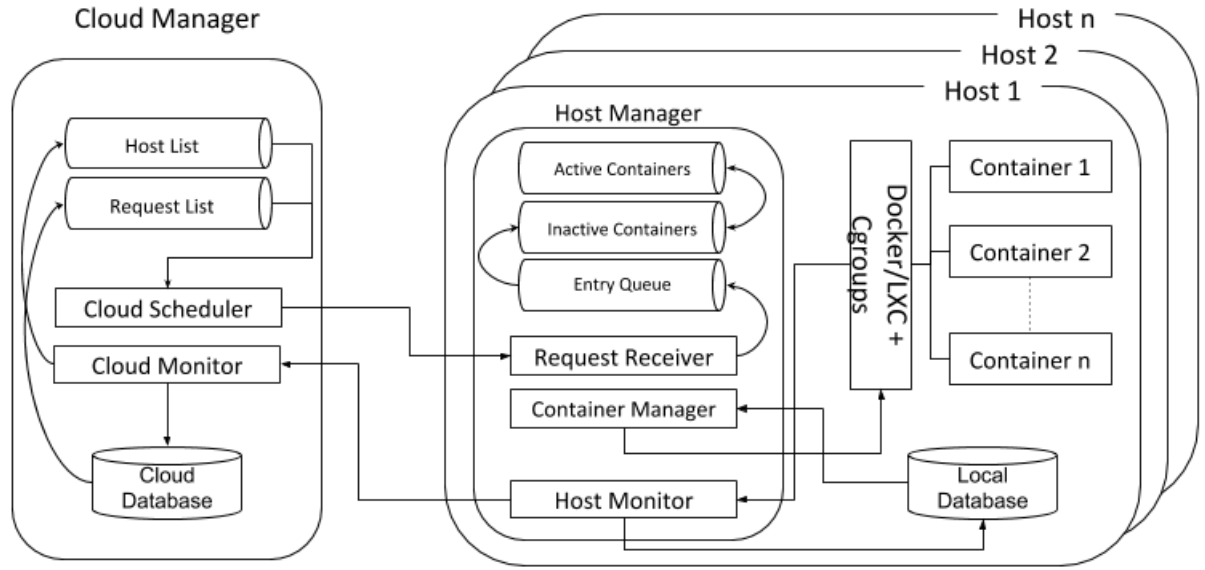


Figure 4.1: Architectural Model for VEMoC Container-based Cloud Management

the host's available resources to each container, adjusting the allocations dynamically, according to application demand at run time. Redistribution of resources may involve other actions including container pausing, suspension and/or migration due to a lack of sufficient available resources on the host, as discussed further in the sections that follow. Cloud management is thus distributed as each Host Manager makes its own local resource allocation decisions independently and autonomously of the Cloud Manager and other Host Managers.

4.1 Cloud Manager

The Cloud Manager (CM) is a service responsible for receiving and managing user's requests (job submissions), collect monitoring information from all hosts and containers in the infrastructure, determining where each container that will compose the request are to be instantiated and create a execution history of each application executed in this architecture. The CM is composed of the Cloud Scheduler, a Cloud Monitor, a database, and two lists: a Host List and a Request List, as shown in Figure 4.1.

4.1.1 Cloud Monitor

The Cloud Monitor periodically collects monitoring data sent from the hosts, updates the host list with the latest resource consumption information, updates the job status

in the request list, and stores monitoring information in the database. The monitoring data contains CPU usage, memory and swap consumption information of each host, and the resource consumption/requirements for each container allocated to it, whether in execution or not. While all data is logged in the database, more recent information is also accessible via the host list. The host list and monitoring information is used by the Cloud Scheduler to make decisions regarding where new jobs (taking into consideration their requirements) should be submitted for execution. Received job submissions are inserted in the request list, which is updated with the progress of such user requests, and follows the container status of each request until its completion.

The container data is also stored in the database for use as historical information to trace their execution and can be used for postmortem analysis of application behavior. In addition, this could also be used to provide finer grained billing since a resource can be charged for the exact usage rather than the current practice of having to pay for a quantity (determined prior to execution) of resources of a predetermined capacity, which are typically oversized for the application's actual needs.

4.1.2 Cloud Scheduler

Each user job request is considered to require a set of containers, where each container may execute an application or a workflow of application tasks. Both classes of applications are considered – *online* ones that must execute continuously and cannot be suspended, and *batch* ones that have a finite execution time. The request can optionally define a minimum and maximum amount of resources, per container, necessary to execute its application or workflow (e.g. quantity of CPU cores and memory). The Cloud Scheduler is the module responsible for orchestrating the execution of these requests, allocating each container instance to an appropriate host in the cloud.

An initial container allocation can be determined by one of many scheduling policies, for example, [43]. However, since a container's resource requirements change over time, the Cloud Scheduler is also responsible for coordinating the migration of containers from an overloaded host to an alternative location. Discussions regarding these orchestration policies have been discussed elsewhere [73, 74] and thus have been omitted here as the Host Manager rather than the Cloud Scheduler is the principal focus of this work.

4.2 Host Manager

The Host Manager (HM) is a service that manages the resources of a single host machine. An instance of this service runs on each host that makes up the cloud infrastructure. Its goal is to schedule container requests received from the Cloud Manager and arbitrate the host's resources elastically between running containers. This is achieved through three modules: the Host Monitor, the Request Receiver and the Container Manager; and two container lists: the Active List and the Inactive List.

4.2.1 Host Monitor

The Host Monitor is the module that determines the capacity of the host (e.g., the number of physical or virtual cores and the amount of physical memory) and constantly collects metrics (every second) to provide statistics regarding the resource utilization of its respective host and the containers under the HM's supervision. Some of the host CPU and memory related metrics include:

- The number of cores available on the host to execute containers – considers the possibility that some of the host's processing capacity may be reserved for other activities;
- Percentage CPU utilization per core;
- In addition to the total memory present in the host, the amount of memory in use, and *free* and *available* memory is collected, together with;
- Total swap memory, swap used, swap free, and swap-in and swap-out traffic.

The Python system and process utility library named `psutil` has various functions to collect resource usage information from operating systems like Linux or Windows and is used to collect host information. For the following container-related CPU and memory statistics, the library APIs (Application Programming Interface) of the respective container technologies are used to consult the following information in the `cgroup` files of each container:

- The set of CPU cores that the container is pinned to;
- Total CPU time utilized by each container in processor cycles or ticks;

- Total memory used by the container as Resident Set Size (RSS) and Page Cache Memory (PCM) – RSS refers to the pages allocated to the container in the main memory, while PCM refers to the pages that originated from the disk and that are now cached in main memory for rapid access;
- Active and Inactive Pages: Active pages are those used recently, while Inactive pages refer to those in the Least Recently Used (LRU) list associated with a container;
- Total amount of memory page in/out operations that were generated from the beginning of the container’s execution and the number of major page faults;
- Total amount of swap memory used by the container, and;
- The current memory and swap limits of each container;

These host and container utilization metrics are then used by the Container Manager to predict future demands of each respective container and determine if these can be met by the host by reallocating resources through some form of appropriate action. Additional necessary information is then derived by VEMoC from this data. *Host Reserved Memory* refers to the total amount of memory that is reserved specifically for operating system processes and other internal host and container services. This changing value is taken into consideration when calculating the amount of memory available for the containers.

It is also assumed that some specific CPU cores of the host can be reserved for exclusive use by the host Operating System (OS) and services *Host Reserved Cores*. While this reduces the chance of containers competing with system processes it does not eliminate this possibility, as there are no guarantees that a system process will not temporarily use an idle core currently assigned to a container.

4.2.2 Request Receiver

The Request Receiver module is responsible for receiving job requests from the Cloud Scheduler and deploying the corresponding containers for execution. Initially, such container requests are queued in a list of containers awaiting execution on the host. Each container request is composed of:

- The container name, based on the request id and container number inside the request;

- The location of the container image that contains the required application and libraries;
- The command and the parameters for the application(s) to be executed within the container;
- (Optional) The maximum expected memory usage by the container, *MaxMemLim* – this is used as a safety mechanism to avoid spurious container behavior that might adversely affect the host and other containers. This parameter can also be viewed as the equivalent of the fixed predefined resource allocation used in conventional (non-managed elastic) container based systems;
- (Optional) The minimum amount of memory that should be allocated during the container’s execution. Again, acts as a safety to guarantee a certain quantity of resources.

For technologies that need to create the containers prior to being able to initialize them, like LXC, the Request Receive Module creates the container before inserting it into the Inactive Container queue, to speed up the container’s startup when its application is scheduled to execute. Just like the monitor modules, the Request Receiver and Cloud Scheduler communicate using sockets over the network.

4.2.3 Container Manager

The Container Manager is the most important component of the VEMoC architecture. It is responsible for managing resource usage (in this work, specifically memory) of each executing container on its respective host. Its purpose is to manage the life cycle of each container allocated to it by the Cloud Manager, from the creation of the container, to determining the maximum amount of each resource the container can use at any given time during execution until it terminates and is destroyed. The state transitions during this life cycle are shown in Figure 4.2 and described below.

- **QUEUED:** the container is created and/or is waiting to be started by the Container Manager, depending on the container technology (e.g. LXC or DOCKER) being employed. LXC containers are created prior to being started while Docker containers are created during the start process;
- **RUNNING:** this state indicates that the container is being executed and consuming resources;

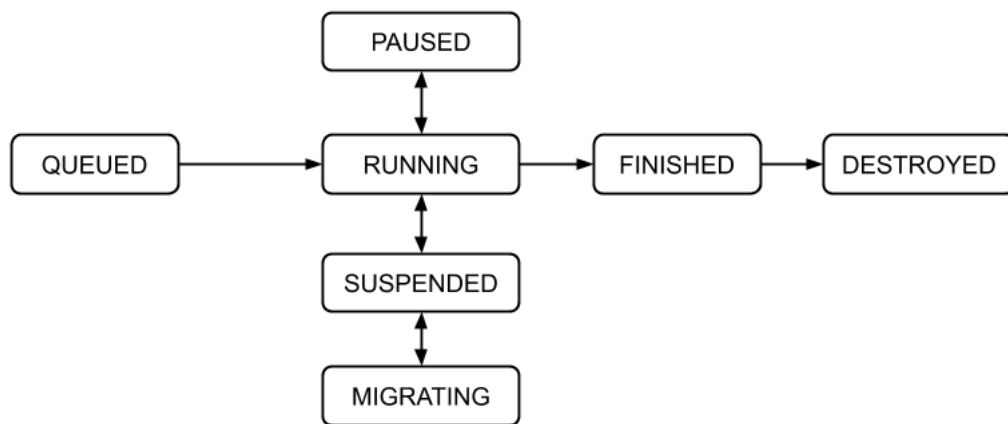


Figure 4.2: States of the Container Life Cycle

- **PAUSED:** indicates that the container's execution has been temporarily stopped but the memory resources it has consumed remains allocated to it;
- **SUSPENDED:** the container has been suspended by the Container Manager, which may occur when there is insufficient memory for an efficient execution of its application. A checkpoint is created and saved on the host's disk to allow the container to be resumed later. When suspended, the resources that were being consumed by the container are released for use by other containers in the same host;
- **MIGRATING:** the container is being migrated to another host either because of a lack of resources in the current host or for server consolidation. The respective checkpoint file will be "migrated" too, if not already on a shared file system, so that it can resume its execution on another host. This behavior is outwith the scope of this work and is not discussed here;
- **FINISHED:** the container ends its execution, with or without errors and the resources used by it have been released back to the host for use by the other containers;
- **DESTROYED:** the container is destroyed by the Container Manager to liberate space on the host's disk. The original base container image is preserved but the instance specific data produced is deleted.

To control the life cycles of the containers allocated to the host, the Container Manager constantly controls the distribution of memory assigned to each container by changing the maximum amount of memory the container can use, and their respective states, at regular scheduling intervals. This scheduling algorithm is described in detail in the following

chapter. To implement the algorithm, the Container Manager uses four auxiliary lists: a list of active containers (ACList), i.e. those in a RUNNING or PAUSED state (as these containers are holding on to or consuming host resources); a list of inactive containers (ICList), i.e. containers in a QUEUED or SUSPENDED state; a list of job requests from the CM (JobReqList); and a Core Allocation List used to identify which CPU cores are in use by the running containers.

With the implementation this architecture and all of its components, the VEMoC algorithm presented next can be evaluated against alternative strategies (see Chapter 6) in a real cloud deployment scenario, whether it be to distribute virtual environments over a cluster or manage all of them in one machine, without the need to resort to simulation.

Chapter 5

Managing Vertical Memory Elasticity

Server consolidation and resource elasticity are two key foundations of resource management in cloud computing. Vertical memory elasticity, if implemented efficiently, can be used as an effective method to assign sufficient amounts of memory to applications (encapsulated in virtual environments such as containers) while allowing cloud providers to instantiate more containers in a physical node without the risk of over-provisioning. This chapter briefly describes the memory management policies adopted by VEMoC to show that, when resources are scarce and insufficient, the consequent performance losses can be mitigated to a certain degree, thus offering an opportunity to improve server utilization. VEMoC combines the use of two preemption schemes (pausing and suspension) with vertical memory elasticity into a coordinated scheduling policy for the effective management of containers.

5.1 The VEMoC Orchestration Engine

Vertical memory elasticity is a technique used to change the amount of memory a virtual environment like a virtual machine or a container can use during its execution, without the need to stop and restart it. In virtualization environments, a VM can choose to use as much of its already allocated memory as it wishes, but should it require more than this, the VM will have to use *swap*, even if the host has additional memory available. Differently, containers are not allocated pre-reserved amounts of memory, but rather have upper bounds that are defined through *cgroups*, which limit the maximum amount each can consume. A container can use any amount of memory up to this limit or that the host

has available (if less than the limit), after which it will be forced to use swap and suffer the consequent performance degradation. Beware however, that if the container application cannot use swap memory, it may terminate abnormally with an out of memory error.

Kubernetes [44] adopts the philosophy that swap space is not needed (and should be disabled on the host) since its containers (pods) should have a memory limit set high enough to meet their respective application's maximum requirements and be allocated to hosts with enough available resources to meet them (Guaranteed Quality of Service). Nevertheless, if host runs low on memory or a container's memory consumption reaches that limit, an *Out of Memory* (OOM) exception may cause the container to be "killed" by the host operating system.

This work proposes a new approach to manage the memory limits of containers, at run time, based on the memory consumption of its running application(s), while trying to prevent the application from losing performance or even being unexpectedly terminated due to a lack of memory. This approach makes a set of iterative management and scheduling decisions at autonomously reconfigurable frequencies in order to predict and react to changes in demand while balancing response times with management overheads.

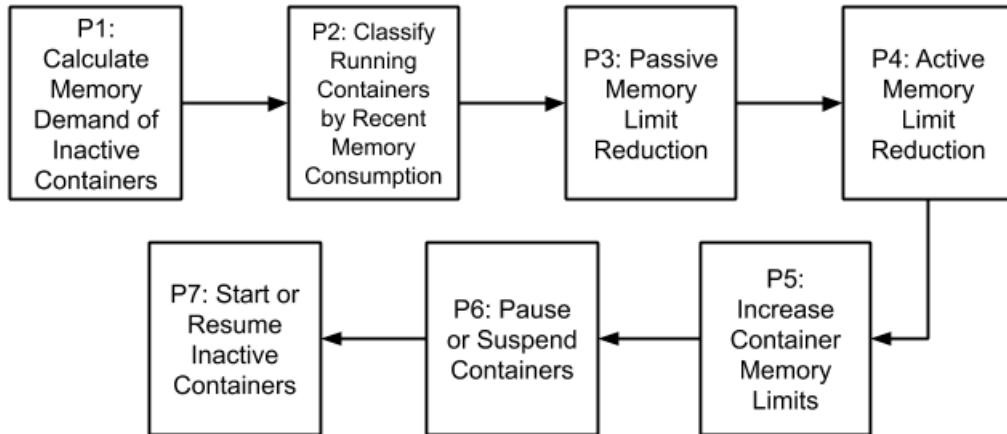


Figure 5.1: Phases of a VEMoC Management Iteration

During each scheduling interval, the VEMoC algorithm follows a sequence of up to 7 phases, P1 to P7 in the Figure 5.1. The objective is to define a *Container Memory Limit* (*CML*) for each running container that is sufficiently large enough to meet the needs of that respective container until the end of the next scheduling cycle, while maximizing the number of running containers (this includes resuming paused containers, restarting suspended ones and/or starting new jobs, if possible) and all without (significantly) degrading their performance. The percentage of the memory allocation limit actually being used at a given time by each container defines its true memory utilization (*MemUtil*).

The aim is therefore to maximize *MemUtil* without slowing the container's application(s) due to swap usage.

Since a lack of a sufficient amount of memory can significantly impact the application performance and that accurately predicting when and by how much the memory limit should be changed is generally difficult for every type of application, overestimating the value of the Container Memory Limit, *CML*, is a common practice to allow for sudden or unforeseen increases in demand for memory. Given this limit is often used to make orchestration decisions, *MemUtil* could in practice be quite inefficient, especially if the limit for each container is defined statically as a worse case maximum value. The VEMoC Host Manager, described in Algorithm 1, dynamically adjusts the *CML* of each running container with three goals: to maintain high but safe *MemUtil* values; to reduce the average turnaround times of batch jobs, and; to improve host memory utilization.

In Algorithm 1, the 7 phases of VEMoC are preceded by setting a few necessary variables that describe some of system capabilities of the host where VEMoC is running (Lines 1 to 4 of algorithm 1). These variables describe: the duration of the scheduling interval that VEMoC uses to collect and analyze monitoring data (*LongInt*, *ShortInt* and determine its base operating frequency *SchedInt*), in seconds; the latency of VEMoC during the last scheduling interval (*SchedLat*), also in seconds; the read and write rates for memory (*MemWrRt*) and swap (*SwapInRt* and *SwapOutRt*) of the host, in memory pages per second; the maximum percentage memory utilization before a container uses swap (*MaxMU*) and the maximum amount of memory the machine can provide (*HMUT*), in memory pages. We use memory pages (in Linux, each memory page equals 4 KB) due to cgroups, which uses this unit of measure for system memory management. Some of this metrics are obtained through experimentation using test applications (the host memory rates and the *MaxMU*), while others can be chosen arbitrarily in a configuration file (for example, *LongInt*). In the specific case of *LongInt*, the smallest value that can be used is 6 seconds in order to have enough monitoring data points to predict the memory demand of a running container.

A brief initial summary of each of the seven phases (P1 to P7) of Figure 5.1 is presented below, while thereafter, the remainder of this chapter will describe in each of these phases in further detail in a separate section of their own.

- In P1 (Lines 7 to 17 of Algorithm 1), VEMoC identifies the amount of memory necessary to start an inactive container. This is either the defined minimum memory requirement to start the execution of new containers or the memory necessary to

restart the execution of a container that has been suspended by VEMoC.

- In P2 (Lines 18 to 25 of Algorithm 1), VEMoC obtains a picture of the current state of the host, and the memory resource allocation and usage by the containers. Using the data from the previous scheduling interval, this phase aims to estimate the demand of each running container for the current interval.
- In P3 (Lines 26 and 27 of Algorithm 1), VEMoC reduces the memory limits of any containers that appear to have an overestimated memory allocation. This reduction aims to make a more conservative estimate of memory usage while still minimizing the chance that container will require the use of memory swap.
- In P4 (Lines 28 and 29 of Algorithm 1), VEMoC “steals” memory from the containers that appear to have inactive memory, forcing them to send some of their stale (least recently accessed) memory pages to swap. This technique consists of temporarily reducing the memory limit below the amount of memory in use. cgroups will then be forced to send the least recently used (LRU) memory pages to swap. The recovered memory can then be made available for use by the other containers in need.
- In P5 (Lines 30 and 31 of Algorithm 1), VEMoC determines the proportion of the available memory that should be added to the reservation of each container that has been predicted to require more than its current limit during the current scheduling interval. All, while at the same time, trying to serve as many containers as possible.
- In P6 (Lines 32 and 33 of Algorithm 1), VEMoC must decide what action to take with respect to those containers that could not be satisfied in Phase 5 and whose predicted consumption will now likely exceed their memory limit before the end of the next scheduling interval. Unlike many of the previous identified elasticity controllers in the literature, VEMoC also considered pausing or suspending containers as part of its elasticity and scheduling policies.
- In P7 (Lines 34 and 35 of Algorithm 1), if Phases 4 and 6 were not necessary, VEMoC will try to assign the remaining available host memory from Phase 5 to containers in the inactive list. The approach will first prioritize the resumption of previously suspended containers and then permit new queued containers to initiate, if there are any and given sufficient memory.

After these phases have been completed, VEMoC calculates the latency to make these decisions in order to precisely factor this into the schedule and the prediction model for

the next interval. This is necessary in order to avoid making future adjustments to the container limits too late.

5.2 Phase 1: Calculate the Memory Demand of Inactive Containers

Each phase is composed of a number of stages or steps. The steps in Phase 1 are summarized as follows. The first step consists of receiving job requests from the Cloud Manager and placing their corresponding containers in the inactive queue (Lines 7 to 10 of Algorithm 1). Next, the algorithm calculates the amount of memory that would be required by those queued containers (Lines 11 to 17). The inactive queue (*ICList*) contains the new containers that have not begun their execution and containers that are currently suspended. To begin its execution, a new container should at least receive its minimum memory allocation (*MinMemLim*). On the other hand, to restart a suspended one, it will need to have the same memory limit (defined as the Container Memory Limit, *CML*) it had at the moment prior to its suspension plus the supplemental amount ΔL that wasn't available when the container was suspended.

The third step of this phase (Lines 18 to 23) consists of calculating how much free memory the host server currently has available to allocate to the non-finished containers on this host, i.e. the currently available *Non-Allocated Host Memory* (*NAHM*). The host's operating system obviously needs to allocate memory for itself and whatever local host services that are considered necessary. However, to improve system performance, operating systems also make use of "free" memory to hold various data structures, for example, for disk caching. Defined here as the *Host Reserved Memory* (*HRM*), Equation 5.1 estimates the amount of memory being used at time t by system related processes by determining the amount of memory being used (*CMU*) by the active containers and measuring the *Host's Available Memory* (*HAM*) that includes both the free and buffered/cache memory. The *Total Host Memory* (*THM*) is the total amount of physical memory on the host.

$$HRM(t) = THM - \left(\sum_{i=1}^n CMU_i(t) \right) - HAM(t) \quad (5.1)$$

Using the data collected by the Host Monitor (in line 18), Equation 5.2 can then calculate *NAHM* at time t by discounting the memory already being used by the host (*HRM*) and already allocated by the Host manager, i.e. the sum of the Container Memory

Algorithm 1 VEMoC Host Manager Algorithm

```

1: LongInt  $\leftarrow$  6; ShortInt  $\leftarrow$   $\max(\text{LongInt}/2, 3)$ 
2: SchedLat  $\leftarrow$  0.1; SchedInt  $\leftarrow$  LongInt
3: HMUT  $\leftarrow$  5976883; MaxMU  $\leftarrow$  0.997
4: MemWrRt  $\leftarrow$  9999; SwapInRt  $\leftarrow$  6600; SwapOutRt  $\leftarrow$  33000
5: while True do
6:   sTime  $\leftarrow$  getActualTime()
7:   for each Container i  $\in$  JobReqList do
8:     InactiveTimei  $\leftarrow$  sTime
9:     ICList.append(i, QUEUED)
10:    JobReqList.remove(i)
11:   ICMem  $\leftarrow$  0
12:   for each Container i  $\in$  ICList do
13:     Statei  $\leftarrow$  getContainerState(i)
14:     if (Statei = QUEUED) then
15:       ICMem  $\leftarrow$  ICMem + MinMemLimi
16:     else if (Statei = SUSPENDED) then
17:       ICMem  $\leftarrow$  ICMem + CMLi +  $\Delta L_i$ 
18:   HostUpdateInfo()
19:   NAHM, HAM, TCML  $\leftarrow$  getHostMemoryInfo()
20:   if HAM <  $2^{18}$  then MUE  $\leftarrow$   $\min(\text{MaxMU}, \text{MaxMU} - (\text{HAM} + \text{TCML} - \text{HMUT}) / ((20 \times 10^9) / 2^{12}))$ 
21:   else
22:     MUE  $\leftarrow$  MaxMU
23:   SpMemCap  $\leftarrow$  MemWrRt  $\times$  (SchedInt + SchedLat)
24:   PauCount  $\leftarrow$  0; PauDemand  $\leftarrow$  0; StealCheck  $\leftarrow$  False
25:   Algorithm 4 (Estimate Memory Demand)
26:   if (NAHM < PauDemand + UrgentNeed + InNeed + ICMem) then
27:     Algorithm 5 (Passive Limit Reduction)
28:   if (NAHM < PauDemand + UrgentNeed + InNeed) then
29:     Algorithm 6 (Active Limit Reduction)
30:   if (PauDemand <> 0) OR (UrgentNeed <> 0) OR (InNeed <> 0) then
31:     Algorithm 7 (Increase Memory Limits)
32:   if (PauDemand <> 0) OR (UrgentNeed <> 0) OR (InNeed <> 0) then
33:     Algorithm 8 (Suspension)
34:   else if (StealCheck = False) AND (length(ICList) <> 0) then
35:     Algorithm 10 (Container Start/Resume);
36:   fTime  $\leftarrow$  getActualTime(); SchedLat  $\leftarrow$  fTime - sTime
37:   if SchedLat < SchedInt then sleep(SchedInt - SchedLat - 0.007)

```

Limit (*CML*) for each of the n running or paused containers (identified by the *ACList*).

$$NAHM(t) = THM - \left(\sum_{i=1}^n CML_i(t) \right) - HRM(t) \quad (5.2)$$

This is implemented in Line 19 of Algorithm 1 by substituting Equation 5.1 in Equ-

tion 5.2:

$$NAHM(t) = HAM(t) - \sum_{i=1}^n (CML_i(t) - CMU_i(t)) \quad (5.3)$$

Our approach manipulates the container’s memory limit (CML) rather than its actual memory consumption since the Host Manager (HM) effectively considers that the CML is the quantity of memory that is guaranteed for that running container, until the next scheduling interval. Since a container will be unable to use more memory than the HM has allocated, it cannot “steal” additional memory and interfere with memory allocation of other containers. On the other hand, the HM must choose the value of each limit carefully. Overestimating the value will leave resources idle while underestimating the demand will force the container to use swap or perhaps even be killed by the operating system because of a lack of available memory.

The concept of HRM goes some way to help prevent the host’s operating system and its processes from stealing the memory of the running containers, given that these processes have a higher execution priority than the co-located containers. Also, since the principal motivation of this work is to improve host utilization, additional hurdles need to be overcome when the sum of the memory limits of the active containers begin to reach the total memory of the host. In practice, containers may be forced to use a swap before actually reaching their respective set memory limits, i.e., prematurely from their point of view. This can be compounded if the host’s processes use more memory than that accounted for by HRM . In this case, this can cause memory theft from a running container, even if the container’s consumption is below its respective CML , forcing it to use swap before reaching the memory limit.

The principal motive behind these scenarios is the fact that modern operating systems proactively swap out pages as memory becomes scarce. This behavior even affects how much of their allocated memory containers can use before data is moved preemptively to swap. To quantify this, we experimentally determined the Memory Usage Effectiveness (MUE) of a container (Line 20). On our test host server, we found that containers can only use around 99.8% of their limit ($MaxMU$ defined in Line 3) before having to use swap if the host’s memory usage does not exceed a 22.8 GiB threshold of the given the 24 GiB physically installed. The Host’s Memory Usage Threshold, $HMUT$, is also defined in Line 3 but in terms of 4096 byte memory pages. However, as the amount of memory being used rises above this threshold, the value of MUE deteriorates, as discussed in the experimental analysis of Section 6.3 to define MUE . The rest of Algorithm 1 steps through the remaining phases, if their respective prerequisite conditions are met.

5.3 Phase 2: Classify Running Containers by their Recent Memory Consumption

This phase consists of classifying each of the active containers held in *ACList* and determining which of the running containers will likely need to have their *CML* increased. The amount of additional memory expected to be required by a container in the near future is estimated based on its recent memory usage behaviour and is predicted using the monitoring data obtained during the previous scheduling interval. This process can be divided into three steps:

- Calculate the memory consumption rate of a container i in the *ACList* (line 7 of Algorithm 4 which calls Algorithm 2).
- Classify container i based on its Memory Consumption Trends (MCTs) (line 8 of Algorithm 4 calls Algorithm 3).
- Estimate the memory demand of container i and determine whether this container will be a candidate to receive more memory or a candidate to give up memory for other containers in need (Algorithm 4).

5.3.1 Container Memory Consumption Rate

In this step, the host and container execution metrics collected (which were described in Section 4.2.1) during two overlapping windows of time (one long, the other shorter) are used to specifically determine (in Algorithm 2) the average rate per second of memory page in (*PgIn*) and page out (*PgOut*) operations for each container i .

Algorithm 2 starts by retrieving the monitored data metrics for two periods of time, one long *LongInt* and the other short *ShortInt* both going back in time from the time stamp of the last recorded set of metrics. The long window gets the metrics collected during the entire previous scheduling cycle, while the short window covers a more recent portion (approximately the last half cycle) in the same period. We use windows of time to factor out noise in the measurements and two different windows (Lines 2 to 8) to avoid misinterpreting the container's memory consumption behavior during the last scheduling cycle by trying to detect changes in the rates of consumption during the cycle. This helps identify transitions in container behaviour early. In addition, the algorithm calculates the average usage of swap memory (*SU*) and the number of major page faults (*MPF*), which represent how many times it was necessary to bring pages in from swap into main memory.

Algorithm 2 Phase 2: *MemConsumptionRate()***Require:** Container i , *LongInt*, *ShortInt*

```

1:  $PgVar \leftarrow 33$ 
2:  $dataLlist, timeLlist \leftarrow getContainerHistory(i, LongInt)$ 
3:  $dataSlist, timeSlist \leftarrow getContainerHistory(i, ShortInt)$ 
4:  $wallLtime \leftarrow timeLlist[0] - timeLlist[1]$ 
5:  $wallStime \leftarrow timeSlist[0] - timeSlist[1]$ 
6:  $MDLat_i \leftarrow sTime - timeSlist[0]$ 
7:  $SU_i \leftarrow (getSU(dataLlist[0]) - getSU(dataLlist[1]))/wallLtime$ 
8:  $MPF_i \leftarrow getMPF(dataLlist[0]) - getMPF(dataLlist[1])$ 
9: if  $MPF_i \leq MTMPF$  then  $MPF_i \leftarrow FALSE$ 
10: else  $MPF_i \leftarrow TRUE$ 
11:  $PgIn_i \leftarrow getPageIn(dataLlist[0]) - getPageIn(dataLlist[1])/wallLtime$ 
12:  $PgOut_i \leftarrow getPageOut(dataLlist[0]) - getPageOut(dataLlist[1])/wallLtime$ 
13:  $PIS \leftarrow getPageIn(dataSlist[0]) - getPageIn(dataSlist[1])/wallStime$ 
14: if  $PgIn_i > 0$  then
15:   if  $PgIn_i \leq PIS$  then  $MC_i \leftarrow PIS + PgVar$ 
16:   else  $MC_i \leftarrow (PIS + PgIn_i)/2 + PgVar$ 
17: else  $MC_i \leftarrow 0$ 
18:  $PgIn_i \leftarrow PgIn_i >> 6; PgOut_i \leftarrow PgOut_i >> 6$ 
19: return  $MC_i, SU_i, MPF_i, PgIn_i, PgOut_i, MDLat_i$ 

```

The variable $PgVar$ is used to compensate for imprecise measurements of memory page metrics by cgroups. On our system, we determined experimentally the variation between successive measurements, in this case, 33 pages. The $MDLat$ is the age of the most recent metrics used to calculate the consumption rates (i.e., how out of date they were), which is necessary to know in order to make accurate predictions. High precision is fundamental to reducing spare reserved memory capacity and achieving high memory utilization but this also increases the risk of using swap due to unforeseen events.

After this, Algorithm 2 then determines the page in ($PgIn$ and PIS) and page out rates ($PgOut$) during each data window (Lines 11 to 13). The $PgIn$ counter is incremented whenever the container needs a new memory page not already in the main memory, independent of its origin, be it a new page or a page from swap. This metric can be combined with the number of major page faults (MPF) to identify, if necessary, the volume of pages brought back to the main memory from the swap, which is located on disk. Similarly, the $PgOut$ counter is incremented when the container removes pages from the main memory, including pages sent to swap during *swap-out* operations.

In Lines 14 to 19, the algorithm then calculates the average memory consumption rate (MC), comparing the numbers of pages created in memory during the long and short windows. By basing decisions on rates of change over neither overly short or lengthy time intervals, instead of using absolute metric values, this allows the algorithm to make

decisions that are less susceptible to noise or sporadic behaviour, and more flexible in terms of the durations of the monitoring data window and the scheduling interval. In addition to being more robust, the VEMoC controller's control logic becomes simpler.

5.3.2 Container Memory Classification

With these rates of change determined, the approach makes a memory usage prediction for each container for a future period of time, i.e., the next scheduling interval, first by classifying the running containers according to one of the three *Memory Consumption Trends*, defined in Algorithm 3. The logic for this implementation has been obtained by the optimization of the 20 viable combinations (out of 108 possible permutations) of the monitored metric rates. For further details regarding this optimization are presented in the Annex B. The following three Memory Consumption Trends (MCTs) help us to identify which containers are candidates for relinquishing memory and which containers will likely need additional memory, based on their behaviour during previous scheduling interval:

- **RISING:** Containers that had a non-zero *PgIn* count during the last scheduling interval, i.e., those that required memory pages to be brought to RAM;
- **FALLING:** Containers that have voluntarily reduced their RAM consumption during the last scheduling interval;
- **STABLE:** These are containers that do not fall into the two previous trends since they had no page major faults occurring during the last scheduling interval, and their memory and swap consumption was around zero, unless they were being forced to swap out memory by VEMoC.

Algorithm 3 identifies the operating conditions that define the MCTs but focuses on the containers that change MCT state and, when this does happen, a timestamp is recorded. For example, any container that is performing swap-out operations (Lines 1 to 6) or that is allocating new memory pages in main memory (Lines 20 to 24) are classified as RISING. Classified as STABLE are those containers that either do not consume any memory (Lines 7 to 10), i.e., have no change their memory usage *CMU*, or those that create more pages in memory than it sends to swap (Lines 17 to 19). The FALLING trend is used to identify those containers that are voluntarily decreasing their memory consumption (Lines 11 to 16).

Algorithm 3 Phase 2: *MemClassification()*

Require: Container i , Swap Usage SU_i , Major Page Faults MPF_i , Page In $PgIn_i$, PageOut $PgOut_i$

```

1: if  $MPF_i = TRUE$  then
2:   if  $MemRepo_i = TRUE$  then
3:      $RepoSI_i \leftarrow TRUE$ 
4:   if  $MCT_i \neq RISING$  then
5:      $MCT_i \leftarrow RISING$ 
6:      $ST_i \leftarrow getActualTime()$ 
7: else if  $(PgIn_i = 0)$  AND  $(PgOut_i = 0)$  then
8:   if  $MCT_i \neq STABLE$  then
9:      $MCT_i \leftarrow STABLE$ 
10:     $ST_i \leftarrow getActualTime()$ 
11: else if  $PgOut_i \geq PgIn_i$  then
12:   if  $SU_i = 0$  then
13:     if  $BlockRepo_i = TRUE$  then  $BlockRepo_i \leftarrow FALSE$ 
14:     if  $MCT_i \neq FALLING$  then
15:        $MCT_i \leftarrow FALLING$ 
16:        $ST_i \leftarrow getActualTime()$ 
17:   else if  $MCT_i \neq STABLE$  then
18:      $MCT_i \leftarrow STABLE$ 
19:      $ST_i \leftarrow getActualTime()$ 
20: else
21:   if  $BlockRepo_i = TRUE$  then  $BlockRepo_i \leftarrow FALSE$ 
22:   if  $MCT_i \neq RISING$  then
23:      $MCT_i \leftarrow RISING$ 
24:      $ST_i \leftarrow getActualTime()$ 
25:  $MTT_i \leftarrow getActualTime() - ST_i$ 
26: return  $MCT_i, MTT_i$ 

```

In each case, the timestamp is kept to identify when the current state began. Each scheduling interval, independently of whether or not container i changed its Memory Consumption Trend MCT_i , its Memory Trend Timer (MTT_i) that maintains the total continuous time the container has been in this state, is updated. This classification helps identify the behavior of the running containers during each scheduling interval and is used to decide if a container will need more memory or perhaps can provide memory for those other containers in need.

5.3.3 Estimate Container Memory Demand

After the memory consumption classification, the running containers whose Container Memory Limits (CML) should be increased are added to one of two lists, based on the motive for their predicted memory consumption, using Algorithm 4. The amount by which

the *CML* should be increased is determined by a number of factors. First, the predicted memory consumption for the next scheduling interval uses one of two average *PgIn* rates per second – one over a longer interval of time and a second over a smaller one – as calculated by Algorithm 2. A function of the higher of the two rates is adopted to account for any acceleration or deceleration in memory consumption during the last scheduling interval. The quantity of memory expected to be consumed is derived by multiplying this rate by the time between when the latest monitoring data were collected and when the *CMLs* will again be updated in the next scheduling cycle (see line 12 of Algorithm 4). Second, the *CML* needs to be adjusted as the amount of the memory limit a container can effectively use (*MUE*) may not be close to 100% (for the host and system installation used in our experiments, the highest value of *MUE* was 99.8% of *MaxMU*) and which decreases when the host begins to run out of memory (see Section 6.3 for further details). Finally, adjustments may be needed if the containers have a predetermined maximum memory consumption limit that should not be exceeded.

If a container’s existing *CML* is not likely to be sufficient to meet the predicted demand for the next cycle, this RISING container will be placed in one of two lists depending on whether or not Major Page Faults (MPFs) occurred during the previous scheduling interval – indicating that the container realized swap-in operations. The *InNeedList* identifies the RISING containers whose *CMLs* would be exceeded if their memory consumption were to be extrapolated using their consumption rate calculated in Algorithm 2 but where no MPFs occurred. The RISING containers in the *UrgentList* while also likely to exceed their *CML*, are presumed to likely incur further MPFs in the near future. If no additional memory is given to the container, pages will have to be swapped out to make room for incoming pages. If the swapped-out pages then have to be brought back to memory at a later time, this will likely negatively impact the container’s execution time. Therefore, priority for extra memory should be given to the container to compensate and avoid page thrashing and future swap usage.

The *ProviderList* holds a list of containers that will not likely require their *CML* to be increased during the next scheduling interval and thus, might be able to release some of their “reserved” memory for the other containers in need. The variables *InNeed* and *UrgentNeed* store the sum total of the each container *i*’s additional memory requirements, ΔL_i , in their corresponding lists. These values are then used in Algorithms 1 and 7 to 9 to determine the total outstanding additional memory demand for the running containers, including the paused ones (Lines 45 to 47 of Algorithm 4) with their respective ΔL value that was calculated during the cycle in which they were paused.

Algorithm 4 Phase 2: Estimate Memory Demand**Require:** Active Containers $ACList$, Schedule Interval $SchedInt$

```

1:  $InNeedList \leftarrow []$ ;  $UrgentList \leftarrow []$ ;  $ProviderList \leftarrow []$ 
2:  $InNeed \leftarrow 0$ ;  $UrgentNeed \leftarrow 0$ 
3:  $MemTest \leftarrow 2000$ ;  $MTMPF \leftarrow 33$ 
4: for each Container  $i \in ACList$  do
5:    $State_i \leftarrow ContainerState(i)$ 
6:   if ( $State_i = RUNNING$ ) then
7:      $MC_i, SU_i, MPF_i, PgIn_i, PgOut_i, MDLat_i \leftarrow MemConsumptionRate(i)$ 
8:      $MCT_i, MTT_i \leftarrow MemClassif(i, SU_i, MPF_i, PgIn_i, PgOut_i)$ 
9:      $CMU_i \leftarrow getMemoryUsed(i)$ ;
10:     $CML_i \leftarrow getMemoryLimit(i)$ 
11:    if  $MCT_i = RISING$  then
12:       $\Delta M_i \leftarrow MC_i \times (SchedInt + MDLat_i + SchedLat)$ 
13:      if  $MTT_i < SchedInt$  then  $\Delta M_i \leftarrow \max(\Delta M_i, SpMemCap)$ 
14:       $\Delta L_i \leftarrow (CMU_i + \Delta M_i) / MUE - CML_i$ 
15:      if  $\Delta L_i > 0$  then
16:        if  $CML_i + \Delta L_i > MaxMemLim_i$  then  $\Delta L_i \leftarrow MaxMemLim_i - CML_i$ 
17:        if  $MPF_i = TRUE$  then
18:           $UrgentList.append(i)$ ;  $UrgentNeed \leftarrow UrgentNeed + \Delta L_i$ 
19:        else
20:           $InNeedList.append(i)$ ;  $InNeed \leftarrow InNeed + \Delta L_i$ 
21:        else  $ProviderList.append(i)$ 
22:    else if  $MCT_i = STABLE$  then
23:      if ( $MTT_i < SchedInt/2$ ) AND ( $RepoSI_i = TRUE$ ) then
24:         $BlockRepo_i = TRUE$ ;  $RepoSI_i = FALSE$ 
25:         $diff \leftarrow (CMU_i - CMURepo_i) >> 6$ 
26:        if  $diff \geq 0$  then  $RepoMLim_i \leftarrow CMURepo_i$ ;  $MemRepo_i \leftarrow FALSE$ 
27:        if ( $BlockRepo_i = FALSE$ ) AND ( $MTT_i < SchedInt/2$ ) then
28:           $NAHM \leftarrow NAHM + CML_i - (CMU_i - MemTest) / MUE$ 
29:           $CML_i \leftarrow (CMU_i - MemTest) / MUE$ 
30:          if  $CML_i < MinMemLim_i$  then
31:             $NAHM \leftarrow NAHM - (MinMemLim_i - CML_i)$ 
32:             $CML_i \leftarrow MinMLim_i$ 
33:           $MemRepo_i \leftarrow TRUE$ ;  $CMURepo_i \leftarrow CMU_i$ 
34:           $LastRepo_i \leftarrow MTT_i$ ;  $UpdateLRU_i \leftarrow TRUE$ 
35:           $setMemoryLimit(i, CML_i)$ 
36:        else if ( $((BlockRepo_i = TRUE) \text{ AND } (MTT_i < SchedInt/2)) \text{ OR } (UpdateLRU_i = TRUE))$  then
37:           $UpdateLRU_i \leftarrow FALSE$ 
38:           $\Delta L_i \leftarrow (CMU_i + SpMemCap) / MUE - CML_i$ 
39:          if ( $\Delta L_i > 0$ ) then
40:            if  $CML_i + \Delta L_i > MaxMemLim_i$  then  $\Delta L_i \leftarrow MaxMemLim_i - CML_i$ 
41:             $InNeedList.append(i)$ ;  $InNeed \leftarrow InNeed + \Delta L_i$ 
42:            else  $ProviderList.append(i)$ 
43:          else  $ProviderList.append(i)$ 
44:        else  $ProviderList.append(i)$ 
45:    else if ( $State_i = PAUSED$ ) then
46:       $PauDemand \leftarrow PauDemand + \Delta L_i$ 
47:       $PauCount \leftarrow PauCount + 1$ 

```

Containers that have just changed their state and become classified as STABLE (i.e. have no memory consumption for at least one scheduling interval) are subject to a temporary, one scheduling cycle reduction to their *CML* in preparation for Phase 4. During this and the following cycle (if the container continues to be STABLE), it is not added to the *ProviderList*. Once all have been categorized, the total demand for additional memory by running containers, including paused ones, can now be easily determined.

Additionally, provider containers can have their memory 'stolen' if the HM detects inactive memory in Algorithm 6 sending this quantity (or a swap-out rate) of inactive memory to swap. However, if the container changes their memory state because of swap-in operations, this container receive a block flag, which prevents it from having memory removed until it regains its stability.

5.4 Phase 3: Passive Memory Limit Reduction

The two previous phases of Algorithm 1 have identified the total predicted memory demand for the next scheduling interval. This third phase is only executed if host does not already have sufficient free memory available (Line 26 of Algorithm 1). to meet the demand. Since, in previous scheduling intervals, the Host Manager may have over estimated a container's memory reservation, Algorithm 5 is used in this phase in an attempt to reduce the *CML* of those containers that have an excess allocation.

The containers, previously classified by Algorithm 3 as either FALLING or STABLE, and RISING containers that did not need to have their *CMLs* increased, will have been added to the *ProviderList*, as candidates to yield part of their memory limit for use by other containers in need of more memory. Note that this does not involve the actual transfer of any memory between containers. This is a bookkeeping operation to redistribute excess amounts of the pre-reserved memory limit amongst containers. Any algorithm, however, must try to avoid reducing the limit too far and cause the container to use swap memory during the next scheduling interval.

Algorithm 5, initially determines how much memory a container in the *ProviderList* is likely to consume until the next scheduling interval, using the current *CMU*, a quantity of spare memory, *SpareM*, and the amount of memory needed by a RISING container *deltaM* (Lines 2 to 4). *SpareM* is the maximum number of pages a container could allocate, based in the *MemWrRt* rate of their host, until the next scheduling interval *SchedInt*, less 20%. Should a previous stable container suddenly start accessing new

Algorithm 5 Phase 3: Passive Memory Limit Reduction

Require: *ProviderList*, Memory Utilization Effectiveness (*MUE*), Non Allocated Host Memory (*NAHM*)

```

1: for each Container i  $\in$  ProviderList do
2:    $\text{deltaM} \leftarrow 0$ ;  $\text{SpareM} \leftarrow \text{SpMemCap} \times 0.8$ 
3:   if ( $\text{MCT}_i = \text{RISING}$ ) then
4:      $\text{deltaM} \leftarrow \Delta M_i$ 
5:   if ( $\text{CMU}_i + \text{deltaM} + \text{SpareM} < (\text{CML}_i \times \text{MUE})$ ) then
6:      $\epsilon_i \leftarrow \text{CML}_i - (\text{CMU}_i + \text{deltaM} + \text{SpareM}) / \text{MUE}$ 
7:      $\text{CML}_i \leftarrow \text{CML}_i - \epsilon_i$ 
8:      $\text{NAHM} \leftarrow \text{NAHM} + \epsilon_i$ 
9:     if  $\text{CML}_i < \text{MinMemLim}_i$  then
10:       $\text{NAHM} \leftarrow \text{NAHM} - (\text{MinMemLim}_i - \text{CML}_i)$ 
11:       $\text{CML}_i \leftarrow \text{MinMemLim}_i$ 
12:       $\text{setMemoryLimit}(i, \text{CML}_i)$ 

```

data, *SpareM* would be the amount of memory consumed before VEMoC could detect, react and update the container's *CML*.

In Algorithm 5, if the sum of *CMU*, *deltaM* and *SpareM* is below the effective amount of memory (this takes into consideration the variable percentage of the total memory limit that can be consumed, Memory Utilization Efficiency *MUE*) the container has access to, the algorithm reduces the *CML* and adds the amount removed to the quantity of currently non allocated host memory *NAHM* for later redistribution (Lines 5 to 8). If the new *CML* is below the container's minimum memory limit (*MinMemLim*), a container may suffer a loss of performance. To avoid this, the *CML* is set to *MinMemLim* and the difference is removed from *NAHM* (Lines 9 to 11).

5.5 Phase 4: Active Memory Limit Reduction

This phase is more aggressive than Phase 3 and attempts to *repossess memory* from STABLE containers in the *ProviderList* by forcing them to *swap-out* some of their inactive memory, memory believed to be no longer necessary, at least in the short term. Note that each container has its own LRU (Least Recently Used) list, identifying its active and inactive memory pages. Through cgroups, the *inactive anonymous* variable returns the size in bytes of the inactive memory pages listed in a container's LRU list. VEMoC tries to decrease the *CML* by an amount equivalent to the remaining memory demand or the quantity of inactive pages available, whichever is smaller.

The LRU list classifies an page as inactive or "cold" when its contents have not accessed by the container application for a determinate period of time, which indicates page

Algorithm 6 Phase 4: Active Memory Limit Reduction

Require: ProviderList, Non Allocated Host Memory (NAHM), Container Memory in Use (CMU_i), Container Memory Limit(CML_i)

```

1:  $RepoInt \leftarrow 50$ ;  $MinRepoThd \leftarrow 256$ 
2:  $TMemReq \leftarrow MemoryNeeded + UrgentlyNeeded + PauDemand$ 
3:  $StealCheck \leftarrow False$ 
4:  $sort(ProviderList, getMTT(C), decrescent)$ 
5:  $Index \leftarrow 0$ 
6: while ( $TMemReq - NAHM > 0$ ) AND ( $Index < length(ProviderList)$ ) do
7:    $i \leftarrow ProviderList[Index]$ 
8:   if ( $MCT_i = STABLE$ ) AND ( $BlockRepo_i = FALSE$ ) AND ( $CMU_i > RepoMLim_i$ )
     then
9:      $\beta_i \leftarrow getMemoryInactive(i)$ 
10:    if  $\beta_i > MinRepoThd$  then
11:       $RepoL \leftarrow min(SwpOutRt \times SchedInt, TMemReq - NAHM)$ 
12:      if  $\beta_i > RepoL$  then
13:         $\beta_i \leftarrow RepoL$ 
14:         $NAHM \leftarrow NAHM + \beta_i$ 
15:         $CML_i \leftarrow CML_i - \beta_i$ 
16:         $StealCheck \leftarrow True$ 
17:         $LastRepo_i \leftarrow MTT_i$ 
18:         $MemRepo_i \leftarrow TRUE$ 
19:        if ( $CML_i \times MUE < RepoMLim_i$ ) then
20:           $NAHM \leftarrow NAHM - (RepoMLim_i - CML_i \times MUE)$ 
21:           $CML_i \leftarrow RepoMLim_i / MUE$ 
22:           $setMemoryLimit(i, CML_i)$ 
23:        else if ( $MTT_i > LastRepo_i + SchedInt \times RepoInt$ ) then
24:           $NAHM \leftarrow NAHM + CML_i - (CMU_i + MemTest) / MUE$ 
25:           $CML_i \leftarrow (CMU_i + MemTest) / MUE$ 
26:           $UpdateLRU_i \leftarrow TRUE$ 
27:          if  $CML_i < MinMemLim_i$  then
28:             $NAHM \leftarrow NAHM - (MinMemLim_i - CML_i)$ 
29:             $CML_i \leftarrow MinMemLim_i$ 
30:             $setMemoryLimit(i, CML_i)$ 
31:        else if ( $MCT_i = STABLE$ ) AND ( $(MTT_i > LastRepo_i + SchedInt \times RepoInt)$ ) then
32:           $BlockRepo_i \leftarrow FALSE$ 
33:     $Index \leftarrow Index + 1$ 

```

candidates to swap-out. The increase in swap can also cause the Linux kernel to classify more pages as inactive, by reducing the period of time a page is considered active or “hot”, adding them to LRU list and thus offering further opportunities to obtain more space in main memory for new pages.

The Linux kernel updates the LRU list whenever a process suffers memory stress. Therefore, as soon as a container is reclassified as STABLE in Phase 2, VEMoC forces it to swap out a small amount of memory, pages being chosen from the LRU list. This serves two objectives: first, for the kernel to update the LRU list by reclassifying the container’s

active and inactive pages, and; second, to test if the container has pages in memory from outside its current working set. If the pages that were swapped out are later required to be brought back to RAM, the container is blocked from suffering larger memory repossession in Phase 4 and the MPFs caused will classify the container as RISING, in urgent need of memory.

Initially, Algorithm 6 calculates the total memory demand ($TMemReq$) originating from Algorithm 4 (Line 2). After this, sorts the *ProviderList*, in descending order of age in the current MCT, prioritizing the containers that have been in a STABLE or FALLING state for the longest, as the candidates to give up a portion of their *CML* (Line 4).

While Algorithm 6 is unable to meet the demand and there are candidates to donate *CML*, it will try to reduce the *CML* of a container from *ProviderList*, by the amount of existing inactive memory sufficient to meet the demand or by the maximum that the host server can swap out in a single scheduling cycle, while maintaining the container's minimum memory requirements (Lines 6 to 22). Once a container has donated some of its allocation, there is an interval of time, *RepoInt* before it can do so again. If a container has not made a *CML* donation for a long time, i.e., longer than the *RepoInt*, Algorithm 6 will try to “steal” some of their *CML* (Lines 23 to 30). This will be repeated until the demand is met or there are no more donors on the *ProviderList*.

5.6 Phase 5: Increase Container Memory Limits

This phase seeks to distribute non-allocated host memory (*NAHM*) to the containers in need of additional memory. In terms of satisfying demand, Algorithm 7 considers three groups of containers: priority is given to paused containers (Lines 1 to 13), followed by those RISING containers already using swap (in the *UrgentList*, Lines 14 to 25) and finally those containers in the *InNeedList* (Lines 26 to 40).

Since there might not be a sufficient amount of available memory for all containers in each group, the containers are ordered by their total run time so far. This heuristic ordering is used in this version of VEMoC as a simple approximation to ordering the containers by the shortest remaining execution time (information we assume is not available a priori to the Container Manager). The motivation being that these containers are most likely to release their resources sooner. Following this ordering, Algorithm 7 adopts an all-or-nothing policy where it increases the limit of container i by the whole requested amount (Δ_i) unless this would exceed the container's maximum $MaxMemLim_i$. If there

Algorithm 7 Phase 5: Increase Container Memory Limits**Require:** *inNeedList*, *InNeed*, *UrgentList*, *UrgentNeed*, *NAHM*

```

1: if (NAHM > 0) AND (PauDemand > 0) then
2:   sort(ACList, getRunningTime(C), decescent)
3:   Index  $\leftarrow$  0
4:   while (NAHM > 0) AND (Index < length(ACList)) do
5:     i  $\leftarrow$  ACList[Index]
6:     if (getContainerState(i) = PAUSED) AND ( $\Delta L_i \leq$  NAHM) then
7:       CMLi  $\leftarrow$  CMLi +  $\Delta L_i$ 
8:       NAHM  $\leftarrow$  NAHM -  $\Delta L_i$ 
9:       setMemoryLimit(i, CMLi)
10:      UnpauseContainer(i); Statei  $\leftarrow$  RUNNING
11:      PauDemand  $\leftarrow$  PauDemand -  $\Delta L_i$ 
12:      PauCount  $\leftarrow$  PauCount - 1
13:      Index  $\leftarrow$  Index + 1
14:   if (NAHM > 0) AND (UrgentNeed > 0) then
15:     sort(UrgentList, getRunningTime(C), decescent)
16:     Index  $\leftarrow$  0
17:     while (NAHM > 0) AND (Index < length(UrgentList)) do
18:       i  $\leftarrow$  UrgentList[Index]
19:       if NAHM  $\geq$   $\Delta L_i$  then
20:         CMLi  $\leftarrow$  CMLi +  $\Delta L_i$ 
21:         NAHM  $\leftarrow$  NAHM -  $\Delta L_i$ 
22:         setMemoryLimit(i, CMLi)
23:         UrgentNeed  $\leftarrow$  UrgentNeed -  $\Delta L_i$ 
24:         UrgentList.remove(i)
25:       Index  $\leftarrow$  Index + 1
26:   if (NAHM > 0) AND (InNeed > 0) then
27:     sort(InNeedList, getRunningTime(C), decescent)
28:     Index  $\leftarrow$  0
29:     while (NAHM > 0) AND (Index < length(InNeedList)) do
30:       i  $\leftarrow$  InNeedList[Index]
31:       if (InNeed > NAHM) AND (MemRepoi = TRUE) AND (BlockRepoi =
32:         FALSE) then
33:         InNeed  $\leftarrow$  InNeed -  $\Delta L_i$ 
34:         inNeedList.remove(i)
35:       else if NAHM  $\geq$   $\Delta L_i$  then
36:         CMLi  $\leftarrow$  CMLi +  $\Delta L_i$ 
37:         NAHM  $\leftarrow$  NAHM -  $\Delta L_i$ 
38:         setMemoryLimit(i, CMLi)
39:         InNeed  $\leftarrow$  InNeed -  $\Delta L_i$ 
40:         inNeedList.remove(i)
41:       Index  $\leftarrow$  Index + 1

```

is insufficient *NAHM* available to cover Δ_i , then the container is left unsatisfied and the algorithm checks the following containers to see if the remaining *NAHM* can completely meet their respective demands.

If VEMoC cannot satisfy the needs of all the containers, Algorithm 1 will attempt

to adopt preemptive measures in Phase 6 to resolve the issue. On the other hand, if, however, all containers have received their requested allocation and there is still some non-allocated memory left, Algorithm 1 will move directly to Phase 7 and consider starting new containers or resuming the suspended ones both being held in the list of inactive containers *ICList*.

5.7 Phase 6: Pause or Suspend Containers

If the previous phase was unable to attend the predicted additional memory requirements of all of the containers for the next cycle, this sixth phase looks to sacrifice one or more of the containers to benefit the others. The approach proposes a combination of preemption strategies - pausing and suspending containers. While suspension is used to release memory to redistribute among the remaining containers in need, pausing is used to protect the performance of a running container that might otherwise have to use swap. While pausing a container appears to bring little gain since its resources are not released back to the system, there are two benefits: it reduces competition to access limit resources, e.g. memory bandwidth, and; if only for a relatively short time while waiting for memory to become available, could help avoid costly swap-in operations later that it might be incurred if left running. While containers with online applications should not be paused or suspended, this approach permits containers with batch-type applications to take advantage of spare unused capacity on host servers.

In terms of scheduling, the costs of *pausing* and *unpausing* a container, and *suspending* and then *resuming* a container, can have a significant impact on what and when certain decisions have to be made. As the experimental analysis concluded in Section 6.2.2, the latency to carry out suspension operations depend principally on the container's memory consumption at the time of the operation.

The proposed preemption scheme is divided in two algorithms: Algorithm 8 handles the special case where only one container requires memory, while Algorithm 9 covers the general case where more than one requires more memory and thus all of the running and paused containers are candidates for suspension. When there is only one active container on the host, care should be taken to avoid deadlock: Lines 9 to 16 of Algorithm 8 address the case where the system is executing a single container and this container is in need of memory, while Lines 17 to 23, handle the scenario where the system no longer has running containers and so any available memory can be allocated to the single paused container.

Even though there is not enough memory to meet the container's demand, the algorithm will resume it after increasing the CML by what ever spare NAHM it has left. The final scenario (Lines 5 to 8) is where only one of a number of running containers should have its CML increased. Here, we opt to pause a container that is already using swap but leave a container in the *InNeed* list running as we are still not certain that it is short of memory.

Algorithm 8 Phase 6: Handle Single Container Demand

Require: ACList, MemoryNeeded, ProviderList, NAHM, PauCount, PauDemand

```

1: RunCount  $\leftarrow$  length(ACList) – PauCount
2: NumCount  $\leftarrow$  length(UrgentList) + length(InNeedList)
3: if (NumCount + PauCount > 1) then Algorithm 9
4: else if (NumCount = 1) then
5:   if (RunCount > 1) AND (UrgentlyNeeded <> 0) then
6:     i  $\leftarrow$  UrgentList[0]
7:     PauseContainer(i)
8:     Statei  $\leftarrow$  PAUSED
9:   else
10:    if UrgentlyNeeded = 0 then
11:      i  $\leftarrow$  InNeedList[0]
12:    else
13:      i  $\leftarrow$  UrgentList[0]
14:      CMLi  $\leftarrow$  CMLi + NAHM
15:      setMemoryLimit(i, CMLi)
16:      NAHM  $\leftarrow$  0
17: else if (RunCount = 0) then
18:   i  $\leftarrow$  ACList[0]
19:   CMLi  $\leftarrow$  CMLi + NAHM
20:   setMemoryLimit(i, CMLi)
21:   UnpauseContainer(i)
22:   Statei  $\leftarrow$  RUNNING
23:   NAHM  $\leftarrow$  0

```

Algorithm 9 employs container suspension. However, different from cost of pausing a container which can be completed almost instantly, this operation can be costly in terms of latency. The suspended container's memory can only be returned to host for redistribution once the suspension operation is complete, which (as seen in the experimental analysis of Section 6.2.2) can be relatively time consuming as it depends on amount of memory being used by the container at the time of suspension.

VEMoC thus opts to suspend one of the containers in need, which also reduces the demand. The *RUNNING* or *PAUSED* container with a *CML* sufficient to attend the predicted demand of the remaining active containers and that has the smallest memory consumption *CMU* is chosen. If none satisfy this condition, VEMoC chooses the one with the largest *CML* from the same group. If there are containers that did not receive

Algorithm 9 Phase 6: Container Suspension**Require:** ACList, MemoryNeeded, ProviderList, NAHM, PauCount, PauDemand

```

1:  $SuspThd \leftarrow 768 \times 1024$ 
2:  $MemDemand \leftarrow UrgentlyNeeded + MemoryNeeded + PauDemand$ 
3:  $Cands \leftarrow NumCount + PauCount - 1$ 
4:  $MemAvail \leftarrow NAHM$ 
5:  $sort(ACList, Largest.CML.First())$ 
6:  $Index \leftarrow 0$ 
7:  $Hold \leftarrow FALSE$ 
8: while ( $Cands > 0$ ) AND ( $Index < length(ACList)$ ) do
9:    $i \leftarrow ACList[Index]$ 
10:   $State_i \leftarrow getContainerState(i)$ 
11:  if ( $i \in UrgentList$ ) OR ( $i \in InNeedList$ ) OR ( $State_i = PAUSED$ ) then
12:    if  $Hold = TRUE$  then
13:      if  $CML_i \geq MemDemand - \Delta L_i$  then
14:        if ( $CMU_i < CMU_j$ ) OR ( $CML_j < MemDemand - \Delta L_j$ ) then  $j \leftarrow i$ 
15:         $Cands \leftarrow Cands - 1$ 
16:      else
17:         $j \leftarrow i; Hold \leftarrow TRUE$ 
18:     $Index \leftarrow Index + 1$ 
19:  if  $Hold = TRUE$  then
20:    if ( $j \in UrgentList$ ) then
21:       $UrgentNeed \leftarrow UrgentNeed - \Delta L_j$ 
22:       $UrgentList.remove(j)$ 
23:       $NumCount \leftarrow NumCount - 1$ 
24:    else if ( $j \in InNeedList$ ) then
25:       $MInNeed \leftarrow MInNeed - \Delta L_j$ 
26:       $inNeedList.remove(j)$ 
27:       $NumCount \leftarrow NumCount - 1$ 
28:    else if ( $State_j = PAUSED$ ) then
29:       $PauDemand \leftarrow PauDemand - \Delta L_j$ 
30:       $PauCount \leftarrow PauCount - 1$ 
31:       $UnpauseContainer(j)$ 
32:       $ContainerSuspend(j)$ 
33:       $NAHM \leftarrow NAHM + CML_j$ 
34:      if  $CMU_j \leq SuspThd$  then
35:        Algorithm 7 (Increase Container Memory Limits)
36:  if ( $NumCount > 0$ ) then
37:     $Index \leftarrow 0$ 
38:    while ( $Index < length(UrgentList)$ ) do
39:       $i \leftarrow UrgentList[Index]$ 
40:      if  $\Delta L_i / SwapInRt > SchedInt + SchedLat$  then
41:         $PauseContainer(i)$ 
42:       $Index \leftarrow Index + 1$ 

```

what they need, VEMoC calculates whether it would be worthwhile to pause them until the next cycle rather than leave them running. To avoid waiting until the next cycle for the memory of the suspended container to become available (detected automatically as

NAHM in future scheduling intervals), VEMoC estimates if the memory will be available by the middle of the next cycle and if the waiting containers will have enough memory capacity to continue running without impediment until then.

A special case is where only one remaining container requires memory. If it is the only active container, VEMoC gives it what ever spare *NAHM* remains and leaves it running. If it is a paused container, it is unpaused and VEMoC does the same. If there are other containers running and the container is in *UrgentList*, it may be paused, if worthwhile, (it is not suspended as there is no other container to benefit from its resources).

The suspension of a container is initiated from its respective container daemon with a system call to the CRIU [19] checkpoint package. This program generates a checkpoint of the running container and stops the container so that its execution can be resumed later on the current host or on another one, if available, after migrating the suspended container. While the operating system is expected to release the memory of suspended containers as soon as possible, its availability will be identified as NAHM in future scheduling intervals.

5.8 Phase 7: Start or Resume Inactive Containers

This last phase is only executed if the host had sufficient non allocated memory to address the predicted need of all of its running containers. Here, VEMoC uses any remaining *NAHM* to first resume the suspended containers and then queued new ones. Initially, the algorithm sorts the Inactive Container List (*ICList*) based on their queued time, thus attempting to start (or resume) the oldest container first, available memory permitting. Algorithm 10 repeats this action over the entire *ICList* as long as there is memory available. The queue time is measured from the moment the container arrived at the host and indicates how long the container has existed in the host.

If a container state is *SUSPENDED*, the VEMoC algorithm can only resume a container if the amount of memory available is sufficient to cover the memory limit *CML* plus its memory demand prior to its suspension (Lines 3 to 13). Similarly, if the container state is *QUEUED*, it will only be started if there is sufficient available memory to attend the application's minimum memory limit, as defined during the user's job submission (Lines 14 to 24).

The Cloud Manager monitors the state of the queues of each of its hosts, in order to balance the workload. Given that containers held in *ICList* are initial candidates for migration, the Cloud Manager is responsible for determining if the container should be

Algorithm 10 Phase 7: Start or Resume Inactive Containers**Require:** $ICList$, $NAHM$

```

1:  $sort(ICList, longest.QueuedTime.first())$ 
2:  $Index \leftarrow 0$ 
3: while ( $NAHM > 0$ ) AND ( $Index < length(ICList)$ ) do
4:    $i \leftarrow ICList[Index]$ 
5:    $State_i \leftarrow getContainerState(i)$ 
6:   if  $State_i = SUSPENDED$  then
7:     if  $CML_i + \Delta_i \leq NAHM$  then
8:        $NAHM \leftarrow NAHM - CML_i - \Delta_i$ 
9:        $CML_i \leftarrow CML_i + \Delta_i$ 
10:       $ResumeContainer(C_i)$ 
11:       $State_i \leftarrow RUNNING$ 
12:    $Index \leftarrow Index + 1$ 
13:  $Index \leftarrow 0$ 
14: while ( $NAHM > 0$ ) AND ( $Index < length(ICList)$ ) do
15:    $i \leftarrow ICList[Index]$ 
16:    $RequestMemory_i \leftarrow getRequestMemory(i)$ 
17:   if  $RequestMemory_i \leq NAHM$  then
18:      $NAHM \leftarrow NAHM - RequestMemory_i$ 
19:      $CML_i \leftarrow RequestMemory_i$ 
20:      $StartContainer(i)$ 
21:      $State_i \leftarrow RUNNING$ 
22:      $RepoSI_i \leftarrow FALSE$ ;  $BlockRepo_i \leftarrow FALSE$ 
23:      $RepoMLim_i \leftarrow MinMLim_i$ 
24:    $Index \leftarrow Index + 1$ 

```

migrated to another host. The design of the Cloud Manager, being responsible for the initial container allocation and any subsequent migrations, will be discussed elsewhere, as it is not the focus of this work.

5.9 Summary

This chapter has discussed in detail the design and mechanics of the VEMoC controller to manage vertical memory elasticity of containers. To evaluate the VEMoC algorithm and find out how containers behave under various scenarios, the following chapter first presents a series of experiments to motivate the configuration of VEMoC. This is then followed by a comparison, in terms of a number of metrics, of the performance of VEMoC with common scheduling algorithms used by commercial container orchestrators.

Chapter 6

Performance Evaluation and Analysis

To configure the VEMoC algorithm and calibrate the values of some of the system specific variables appropriately, it is necessary to understand the behavior of containers and the host under different situations. To this end, after an initial section describing the setup and configuration of the computing environment used in the evaluation, the following three sections each present a series of experiments carried out to analyse aspects often overlooked in other implementations: the costs of changing container states including container suspension and performance difference between two leading container technologies; the necessity of the proposed *Memory Utilization Effectiveness* (MUE) function to facilitate the use *CML* as a form of precisely controlling the memory allocation of a container, and; the overheads associated with container virtualization and managing vertical elasticity, respectively. After this, the next two sections help visualise the key adjustments VEMoC makes to *CML* when executing jobs and then compares the performance of VEMoC with some common scheduling approaches used by other state-of-the-art container orchestrators. The final section, briefly evaluates the scalability of the VEMoC architecture in terms of being able to manage increasing number of containers.

6.1 Experimental Setup

To evaluate the quality of our proposal, the VEMoC Host Manager has been implemented in Python 3 and is capable of managing and monitoring the life cycle of containers while applying vertical memory elasticity to improve server utilization. The python library *psutil* (version 5.6.3) is used to collect the host metrics, while for container technology,

the tool supports both LXC (version 3.2.1) with its python API *python36-lxc* (3.0.4) and Docker Community Edition (version 19.03.6) with its API (3.7.3).

The VEMoC system is composed of two modular services, the Cloud Manager and the Host Manager, with their respective components mentioned in Section 4. Communication between managers occurs through sockets while access to a MariaDB database server (10.4.12) and a MongoDB server (2.6.12) for the Local Database makes use of compatible python MySQL connectors (2.2.9). We use the relational database (MariaDB) to store the control information for the VEMoC architecture, such as the list of users, container execution requests, executed container execution history, among other management information. To store the metrics collected by the Host Manager, we use the non-relational database (MongoDB), due to the better performance when handling large volumes of data.

Since the experiments presented here focus on maximizing host resource utilization, all job requests from the Cloud Manager are sent to the same host. The host in question runs CentOS Linux 7.7 with kernel version 4.20.11 on two Intel Xeon X5650 processors (each with 6 cores at 2.67GHz) with hyperthreading enabled, 24 GiB of DDR3 RAM memory, 8 GiB of Swap memory and 2 TB of disk space.

To simulate container applications, and precisely control memory access and usage, some experiments presented in this chapter employ a number of synthetic memory intensive jobs (*J1* and *J2*), proposed in [67, 68] and a modification version of job *J3*, denoted here as job *J4*. These jobs were designed highlight a classic failing of many existing elasticity controllers to be able to differentiate between an application's memory consumption and its real requirements. By requirements, we mean the actual amount of memory necessary to execute without suffering performance degradation. These proposed jobs access a vector of a predetermined size sequentially, and repeatedly, a fixed number of times to model different degrees of temporal data locality. This, these jobs can be used to exemplify the main memory usage behaviors typically found during the execution of an application, such as the continuous and increasing memory consumption (*J1* and *J2*) with different localities or the allocation of the entire data set and its manipulation over time (*J4*).

While *J1* accesses the elements of its entire vector of size M sequentially, and repeatedly, for a fixed number of iterations, *J2* accesses its vector (with the same size M) in blocks of size w (where $w = M/4$ in these experiments for illustrative purposes), for the same number of iterations over each block. *J3* is a mixture of two jobs, the first iteration

traverses the entire vector while subsequently iterating three times over each block. However, in our modification of *J3*, job *J4* iterates over each block backwards, from the last block of vector to the first, performing vector shrink and releasing the memory allocated in the visited vector. All jobs have the same amount of work and memory demands, but because of their distinct memory access patterns, their requirements differ over time. This allows us to investigate how data locality and types of swap usage impact memory elasticity and container scheduling. The vector size is large enough so that it and a block do not fit into the host's cache hierarchy.

6.2 Container State Transition Costs

Linux Containers, or LXC for short, is an operating system level virtualization technology that allows the creation and running of multiple isolated Linux environments on a single host. LXC uses cgroups to partition resource allocation into so called isolated *namespaces*. Docker containers are considered more portable since they package a single application and all of its dependencies in a virtual environment that can run on any Linux server since the container itself relies on the functionalities provided by the infrastructure's operating system.

6.2.1 Container Creation and Destruction under Docker and LXC

Given the significant popularity of both Docker and LXC containers, the first set of experiments presented here aim to identify the transition times between container states, in relation to the respective container technology being employed. The graph in Figure 6.1 presents the average times in seconds to initialize a container from the inactive queue, and the time to remove the finished container from the host and liberate its allocated disk space. The results show that Docker appears to have a significant advantage over LXC when starting a container (5 times quicker).

While Docker provides a single command to start a container, LXC breaks this operation into two steps: first, container creation decompresses the container image to a specific sub-directory, after which the container can be booted. The average times for each of these steps is presented in Figure 6.2. Having separate commands, however, means they can be executed at different times. In this work, a container is created on submission to the host and is inserted in the inactive queue. Later, when the HM determines appropriate, the container will be started, assigned resources, and moved to the active queue. This

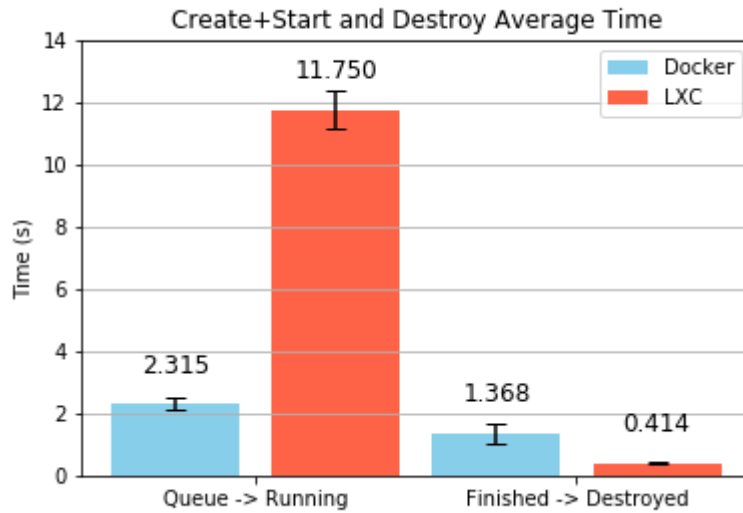


Figure 6.1: Average container initialization and termination times.

can help hide at least some of this relatively high overhead. If the average queuing time were to be larger than the average time to create a LXC container, the effective cost to start a container would then be significantly lower than Docker (less than 4% of the time in comparison).

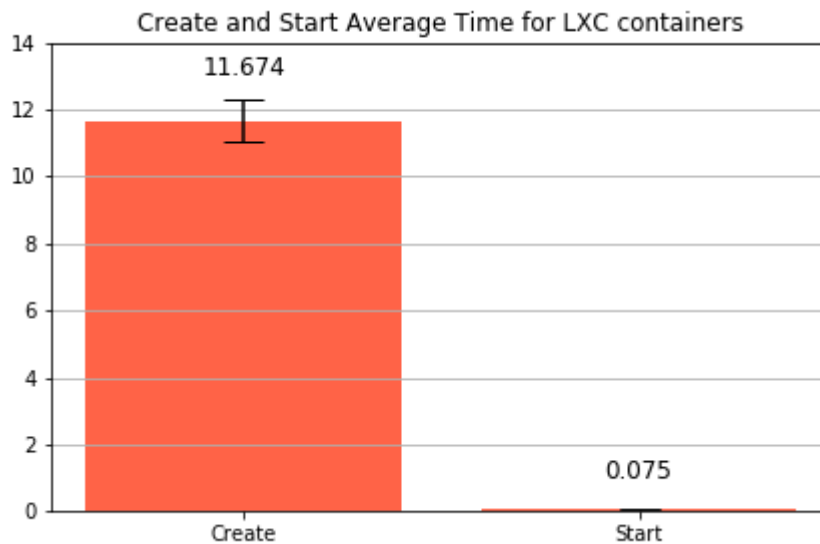


Figure 6.2: Breakdown of LXC container initialization time.

To verify the influence of the image size on the creation time, we repeat the experiment with 2 other images, based on real applications: *OpenJDK*[55], a programming language framework for Java and *Elasticsearch*[33], a distributed data analyzer. For LXC containers, the time to create and destroy a container depends on the size of the container image after being decompressed, as seen in Table 6.1. For Docker containers, the time does not

depend on the size of the image, but the time to create a new layer to save the changes made by the container instance, as seen in Table 6.2. These tables present the average times to create, start and destroy a container under both technologies. Although the same image is used for both, for LXC containers, the docker image is converted to an Open Container Initiative (OCI) [12] image with Skopeo [69]. This conversion can generate an image with different size from the original docker image, normally a little smaller than the original, as observed in the Tables 6.1 and 6.2. Docker images are downloaded to the local image repository apriori to minimize the influence on the creation and start time of the Docker container. If an image were not to be available in the local repository, Docker would need to download the image from an external hub first, adding the download time to the creation and start time. Subsequent container creation operations need only to create the new layer for each instance.

Table 6.1: LXC Create, Start and Destroy containers from images of different sizes.

Image	Create (s)	Start (s)	Destroy (s)	Size
Jobs	11.165	0.099	0.326	211 MiB
OpenJDK	20.617	0.096	0.331	498 MiB
Elasticsearch	31.371	0.100	0.393	735 MiB

Table 6.2: Docker Create + Start and Destroy containers from images of different sizes.

Image	Create and Start (s)	Destroy (s)	Size
Jobs	2.778	1.201	368 MiB
OpenJDK	2.711	1.224	511 MiB
Elasticsearch	2.821	1.539	811 MiB

Although the Docker image is larger, the slower creation time for LXC can be explained as follows. The storage driver used by the Docker for this test is the DeviceMapper [29, 31] that allows containers to share a base image. One image is generally composed of several read only layers. Each layer is a storage structure where successive changes made to the image are saved, and represents the version of that image, much like a software version controller. When a docker container is created, the image layers are mounted as read only and a new layer is created with read/write permissions to save the changes made during the execution of this instance of the container.

On the other hand, LXC must unpack the base image, which is stored in a compressed OCI image format [12], and recreate the file system tree structure, Root Filesystem or rootfs, inside a specific sub-directory in the host's file system. This structure recreates a copy of the file system hierarchy of the operating system to which the libraries, archives and application files required during the execution of the container are added. The du-

ration of these unpacking and file system operations account for almost all of the cost to initiate a submitted LXC container.

When destroying a container, the removal of the disk layer in a Docker container is four times slower than the sub-tree removal for a LXC container. Being the last transition in the life cycle of a container, since image sizes are relatively small in term device storage capacities, the cost to destroy a container and free up disk space has little impact on container management performance but has been included here for completeness.

6.2.2 Pausing and Suspending Containers under Docker and LXC

In terms of dynamic container management, the costs of *Pausing* and *Unpausing* a container, and *Suspending* and then *Resuming* a container, are an important factor in scheduling decisions. The costs in terms of scheduling latency depend principally on the container's memory consumption at the time of the operation. To investigate how the transition times between the states RUNNING and PAUSED and SUSPENDED, respectively, vary, we control the memory consumption of a *J1* job and trigger the transition of container state when the consumption reaches a predetermined size, under both Docker and LXC. The Pause operation, denominated Freeze in the terminology of container technologies, consists of pausing every processes running in the container, and the container itself. But, different from suspension, the container's resources are not returned to the operating system after the freeze operation, nor is its state saved to disk (which would facilitate its migration to another host). Unpausing the container returns it to a running state as shown earlier in Figure 4.2.

Table 6.3: Average duration in seconds to Pause and Unpause containers varying memory consumption under Docker and LXC.

Memory Used	Pause		Unpause	
MU_i	Docker	LXC	Docker	LXC
1 GiB	0.076	0.002	0.080	0.002
2 GiB	0.072	0.002	0.074	0.002
3 GiB	0.079	0.002	0.076	0.002
4 GiB	0.077	0.002	0.077	0.002
5 GiB	0.076	0.002	0.076	0.002
6 GiB	0.076	0.002	0.075	0.002
7 GiB	0.075	0.002	0.080	0.002
8 GiB	0.073	0.002	0.075	0.002

Analyzing the graphs in Figure 6.3 and Table 6.3, the times to pause and unpause a container are extremely small in relation to suspending or resuming a container, being on

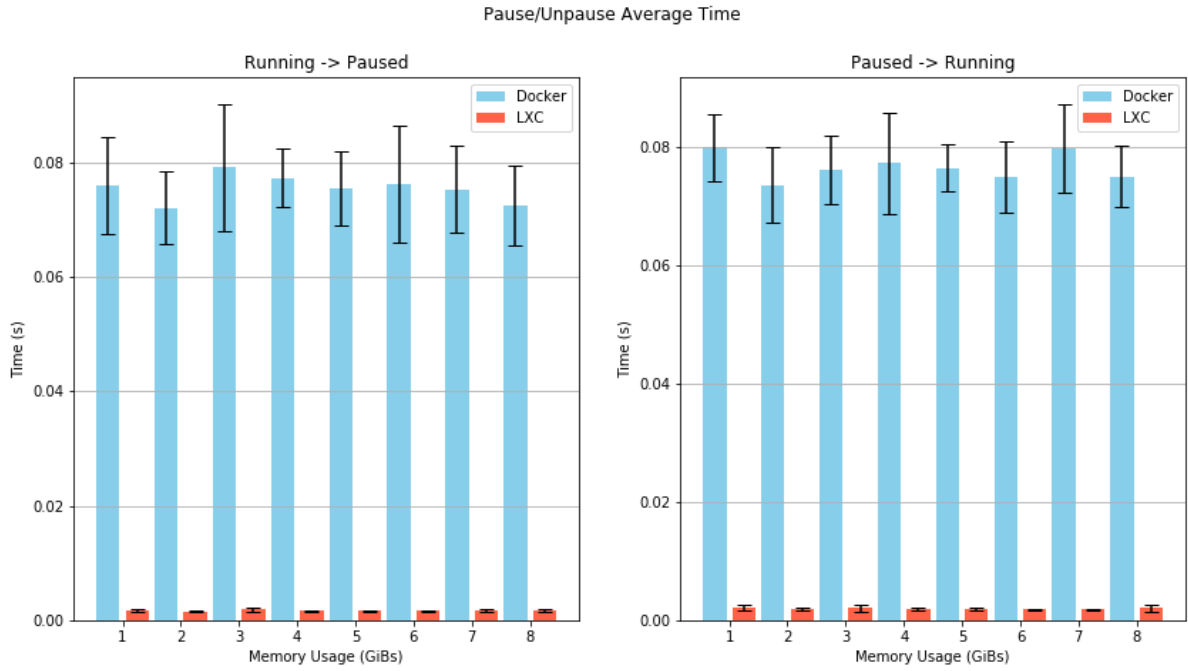


Figure 6.3: The average times to Pause and Unpause a container in relation to its memory consumption, for Docker and LXC.

average 0.076 seconds for Docker but an order of magnitude smaller for LXC (around 0.002 seconds), for containers consuming between 1 and 8 GiB of memory. Effectively, for both technologies, the transition cost to pausing or unpause a container will have little effect on performance, only the duration the container remains in the paused state. VEMoC aims to only pause in need containers (i.e., those that are swapping pages between memory and swap) for the short time it takes to reclaim memory from suspended containers.

With respect to suspending and resuming containers, both technologies use CRIU [19] (version 3.14) to generate a checkpoint that can be used later to restore the container's execution from the point it was suspended. The delay to carry out these operations increases for containers with a higher memory consumption. This means that suspending containers with lower memory consumption (MU) should allow the host to recover and redistribute the corresponding memory earlier.

As shown in Figure 6.4 and Table 6.4, the corresponding suspend and resume operations in Docker are consistently much slower, with times increasing faster as the amount of memory used by a container becomes larger. LXC suspend operations are one order of magnitude faster than those of Docker, with the resume operations being one to two orders better. Although container suspension may be a relatively new functionality in Docker space, and further optimizations may be developed in the future, the stark contrast in performance is surprising. The times obtained for Docker might be acceptable

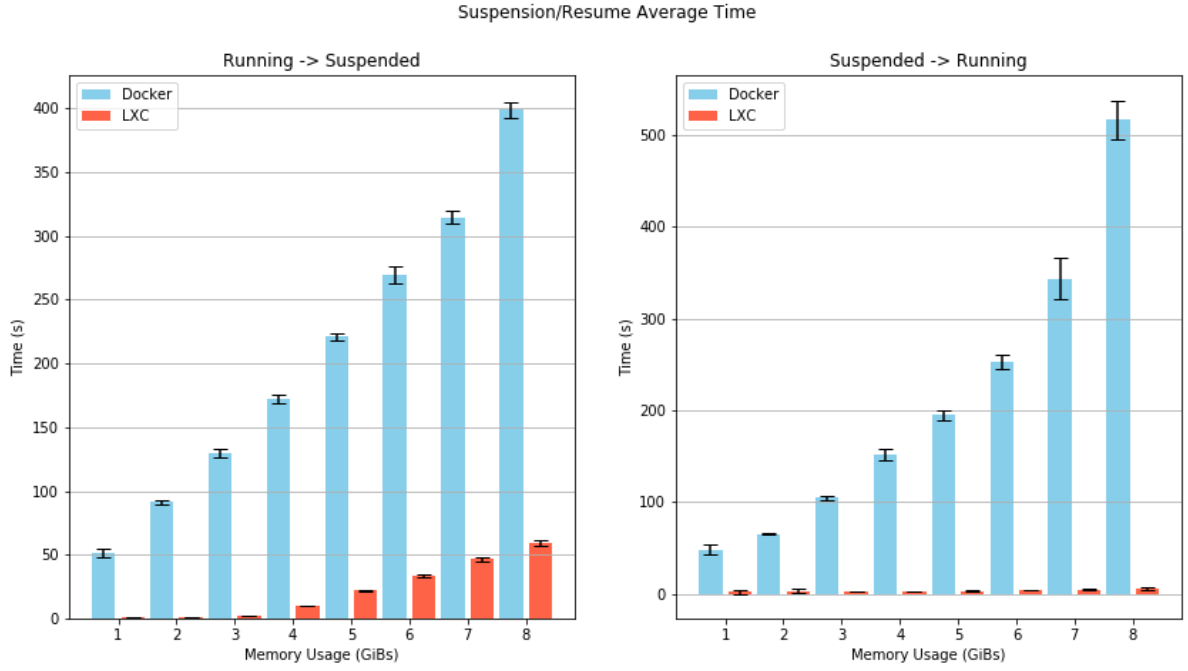


Figure 6.4: The average times to suspend/resume a container in relation to its memory consumption.

Table 6.4: Average Suspend and Resume times in seconds for Docker and LXC containers.

Memory Used MU_i	Suspend		Resume	
	Docker	LXC	Docker	LXC
1 GiB	55.858	0.774	47.719	1.679
2 GiB	91.501	1.277	64.738	2.151
3 GiB	129.656	2.084	104.363	2.327
4 GiB	172.211	10.420	151.435	2.479
5 GiB	220.848	22.050	194.819	3.156
6 GiB	269.416	33.914	252.532	3.999
7 GiB	314.342	46.714	343.421	4.623
8 GiB	398.821	59.130	516.615	5.781

for checkpointing and restarting an extremely long running container that is executing on a device that may fail. However, for applications of shorter duration, or big data applications, the overheads may make the adoption of Docker as a container manager impractical.

This aim of this work is not to enter in to the merits of one technology over the other, but rather to gain an insight into the costs of operations relevant to managing the vertical elasticity of containers. In the following experiments, we chose to use LXC because of its significantly lower cost to suspend and resume a container, which are key operations in the context of vertical memory elasticity.

Returning to the specific question of suspending LXC containers, from Figure 6.4 and Table 6.4, we observe that the time taken for the container’s resources to be released increases almost exponentially with respect the amount of memory being used by the container. The resources of a suspended container will only be available after the checkpoint is created and the container stops. Thus, the redistribution of memory by VEMoC must wait the suspension operation of the container to complete. At first sight, from the point of view of the host manager, it appears that, in most cases, it would be better to suspend two smaller containers than a single one of the equivalent total size.

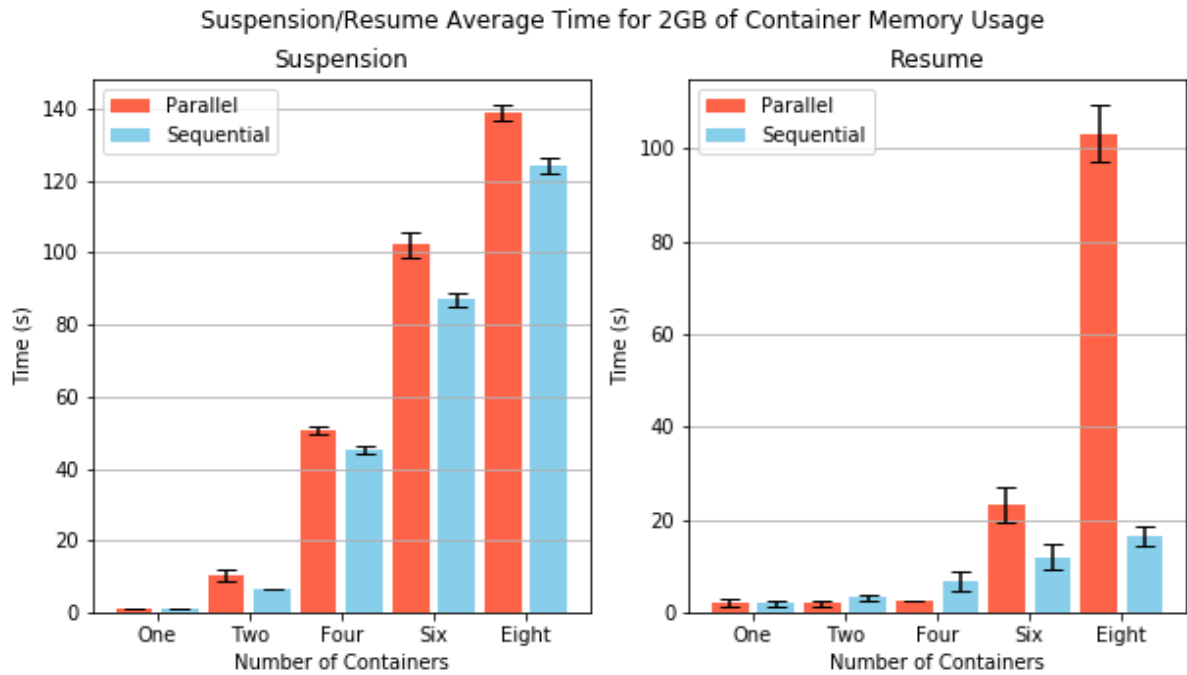


Figure 6.5: Average times to suspend or resume several LXC containers, that consumed 2GB of Memory, sequentially and in parallel.

Table 6.5: The average total times to suspend and resume several LXC containers consuming 2 GiB of memory, with commands executed sequentially and in parallel using threads.

No. of Containers	Suspension Time (s)		Resumption Time (s)	
	Sequential	Parallel	Sequential	Parallel
1	1.233	1.246	2.007	2.021
2	6.521	10.390	3.176	2.079
4	45.239	50.608	6.844	2.608
6	86.907	102.244	12.032	23.297
8	124.377	138.827	16.427	103.212

Figure 6.5 presents the average times to suspend 2, 4 and 8 containers (using 2 GiB of RAM) during the same scheduling interval. Two approaches are considered: the first

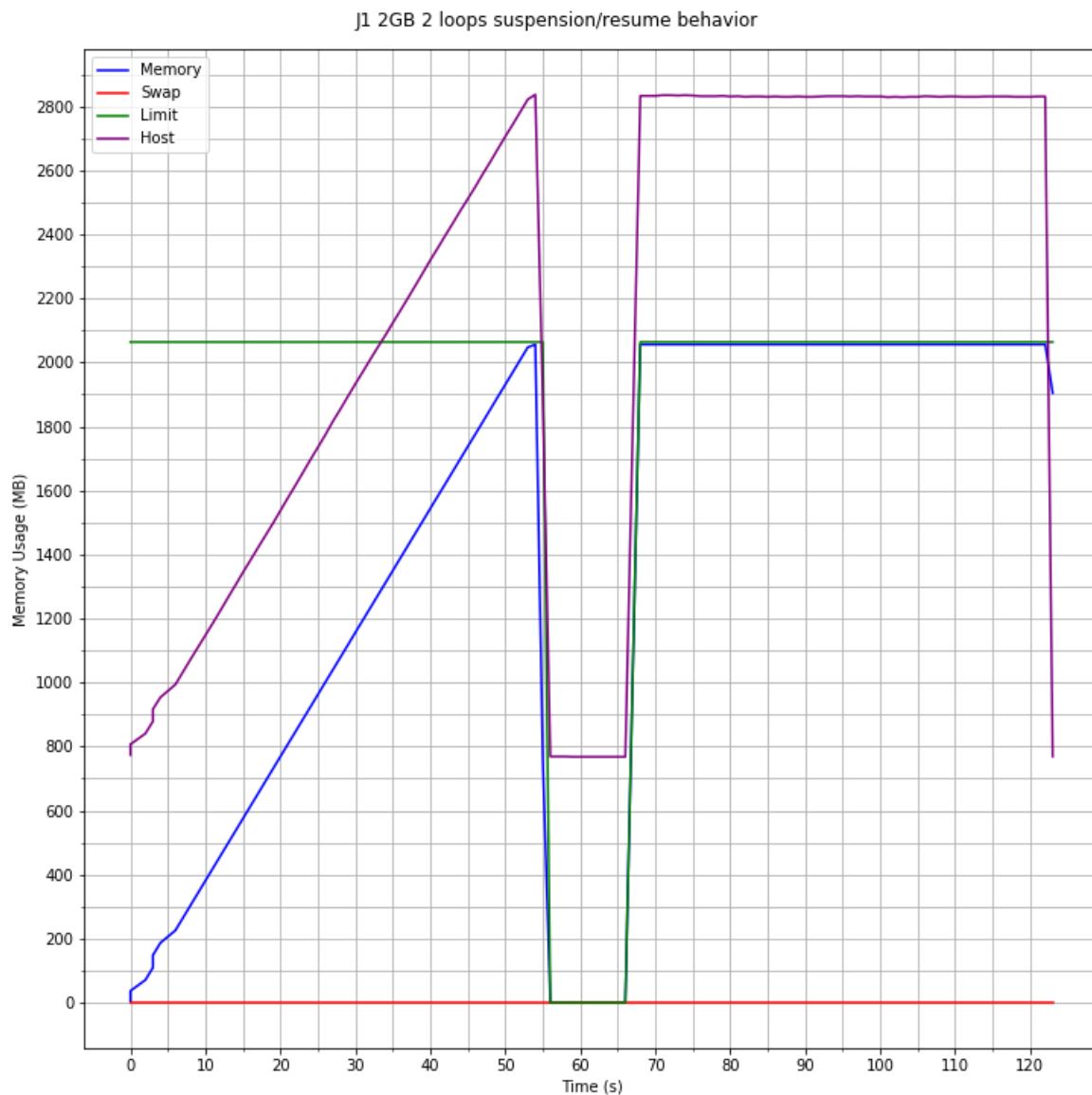


Figure 6.6: Suspend and Resume Behaviors of an LXC container that consumes 2 GiB of memory.

is a sequential approach where containers are suspended one command at a time, and; second, by using threads to issue commands in parallel, the containers are suspended simultaneously. Note that the durations are now longer than expected, in relation to the sum of individual times presented in Figure 6.4. When suspending a container, its checkpoint (image and state) has to be written to disk and the larger the checkpoint size the longer the operation takes. If we overlap multiple suspension operations, the contention to access disk impacts the suspension time of all containers. Furthermore, as disk writes are non-blocking operations, subsequent sequentially issued suspension commands also take longer since operations are being delayed by earlier ones. For example, if the host manager needs 4 GiB of memory, based on Table 6.4, it appears to be better to choose to

suspend two 2 GiB containers rather than one of 4 GiB, but suspend them one at a time, based on Table 6.5, rather than suspending both at the same time. In terms of resuming multiple suspended containers, it appears that the best approach would be to allow up to four containers to be resumed in parallel.

To observe the memory availability on the host during the suspension and resumption of a container, we monitor the execution of a two loop *J1* job that requires 2 GiB + 8 MiB, as seen in Figure 6.6. The 8 MiB of additional memory gives the experiment target threshold to initiate the suspension function once $MU \geq 2$ GiB. The container is resumed from the suspended state 10 seconds after the end of the suspension operation. The *CML* was set to 2 GiB plus 16 MiB to avoid forcing the container to use swap. The monitor was set to collect container information every second and then go back to sleep.

Analysing the Figure 6.6, the host memory consumption (purple line) accompanies the consumption behavior of the container (blue line) in execution. Note the host OS consumes almost 800 MiB of RAM for its own data structures. 54 seconds into the execution, the suspension initiates and after 2 seconds (2 monitoring cycles) completes, stopping the container and releasing its 2 GiB of memory back to the host. Having been recognized as available on the host, this memory could be used immediately for allocation to other containers. The resume operation is scheduled to commence 10 seconds after the suspension finished. The operation allows the container to restart its execution from the point immediately before the suspension, restoring the same memory resources that were used earlier and continuing the execution without subsequently affecting its performance.

6.3 Memory Utilization Effectiveness

Given a container memory limit (*CML*), it's fair to assume that the container will have unrestricted access to this amount of memory. However, there are scenarios where this is not the case, especially given that the objective is to maximize the utilization of the host's memory. One must remember that this is a parameter of cgroups, used to guarantee that the consumption of memory by a container does not exceed this predefined value. Its definition does not necessarily imply that a container will be able to use this amount of memory. For example, a container with a *CML* of 2 GiB does not guarantee that an application that needs 2 GiB will be able to execute without an out-of-memory (OOM) error or having to use swap. Thus, an elasticity controller cannot use *CML* alone to manage memory elasticity.

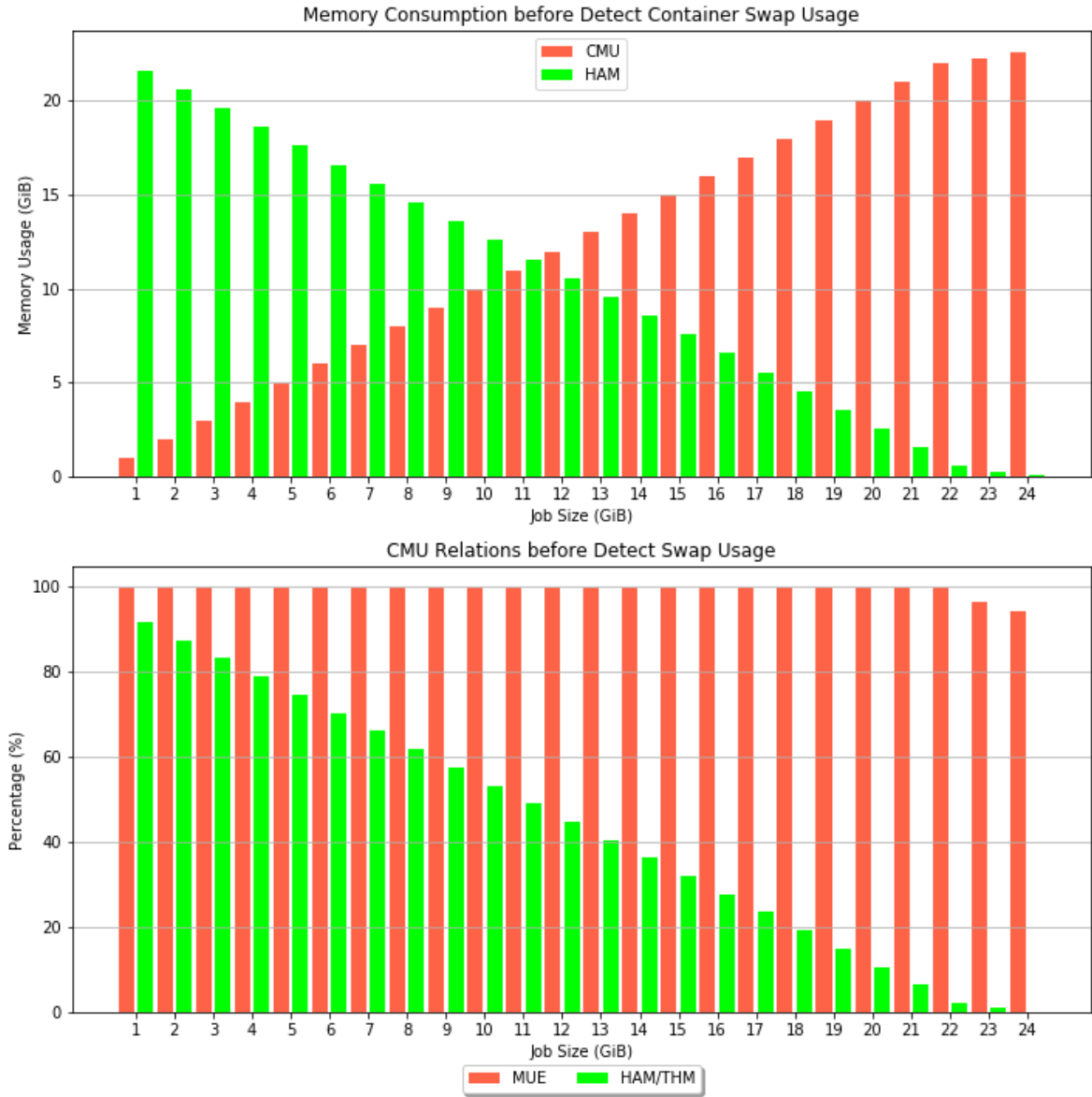


Figure 6.7: Container Memory Utilization vs Effectiveness for a single container.

The following experiment aims to discover how much of a container's *CML* can effectively be used before the operating system begins moving data to swap, i.e., aims to find its maximum memory utilization. The experiment first executes a single container with different memory limits on a server with 24 GiB of physical RAM, starting with *CML* of 1 GiB followed by increments of 1 GiB at a time. The container runs a *J1* job that consumes 128 MiB more memory than its *CML* to guarantee all the memory defined by *CML* is used, while sending the rest to swap. Figure 6.7 presents the Memory Usage Effectiveness (*MUE*) obtained by a single container, based on Equation 6.1 that uses the Container Memory Usage at the first instant that the use of swap is detected.

$$MUE = \frac{CMU}{CML} \quad (6.1)$$

Table 6.6: Memory consumption metrics as the *CML* of a single container is increased. Data presented in Figure 6.7.

	CML	CMU	MUE (%)	HRM	HTMU	HAM
1		0.997	99.749	0.582	1.579	21.585
2		1.996	99.781	0.581	2.577	20.587
3		2.994	99.790	0.589	3.583	19.581
4		3.992	99.791	0.594	4.586	18.578
5		4.990	99.795	0.596	5.585	17.579
6		5.988	99.797	0.599	6.587	16.578
7		6.986	99.798	0.597	7.583	15.581
8		7.984	99.798	0.618	8.602	14.562
9		8.982	99.797	0.616	9.598	13.566
10		9.980	99.799	0.622	10.602	12.562
11		10.978	99.801	0.623	11.601	11.564
12		11.976	99.801	0.629	12.605	10.559
13		12.974	99.801	0.633	13.607	9.557
14		13.972	99.802	0.631	14.603	8.561
15		14.970	99.801	0.631	15.601	7.562
16		15.968	99.801	0.639	16.608	6.556
17		16.966	99.802	0.637	17.603	5.561
18		17.964	99.802	0.636	18.601	4.563
19		18.962	99.801	0.640	19.602	3.561
20		19.960	99.802	0.648	20.608	2.555
21		20.958	99.802	0.649	21.608	1.556
22		21.956	99.802	0.649	22.605	0.559
23		22.212	96.576	0.650	22.863	0.302
24		22.582	94.092	0.564	23.146	0.082

Table 6.7: Memory metrics for 2 containers as presented in Figure 6.8.

	TCML	TCMU	MUE (%)	HRM	HTMU	HAM
2		1.995	99.741	0.542	2.536	20.636
4		3.991	99.780	0.538	4.529	18.643
6		5.987	99.790	0.550	6.538	16.634
8		7.983	99.791	0.579	8.562	14.609
10		9.979	99.795	0.573	10.552	12.620
12		11.975	99.795	0.578	12.554	10.618
14		13.972	99.797	0.582	14.553	8.619
16		15.968	99.799	0.579	16.547	6.624
18		17.964	99.798	0.428	18.392	4.780
20		19.960	99.798	0.464	20.423	2.748
22		21.956	99.800	0.450	22.406	0.766
24		22.111	92.128	0.616	22.727	0.446

Observing Figure 6.7 and its corresponding Table 6.6, the containers were able to use 99.8% of their limit before having to use the swap, but only while the host's memory usage does not exceed 22.8 of the 24 GiB physically installed. After that, the *MUE* drops dramatically, affecting not only the container, but the other processes in the system, as noted in the column *HRM*, which represents the memory usage of the other host processes.

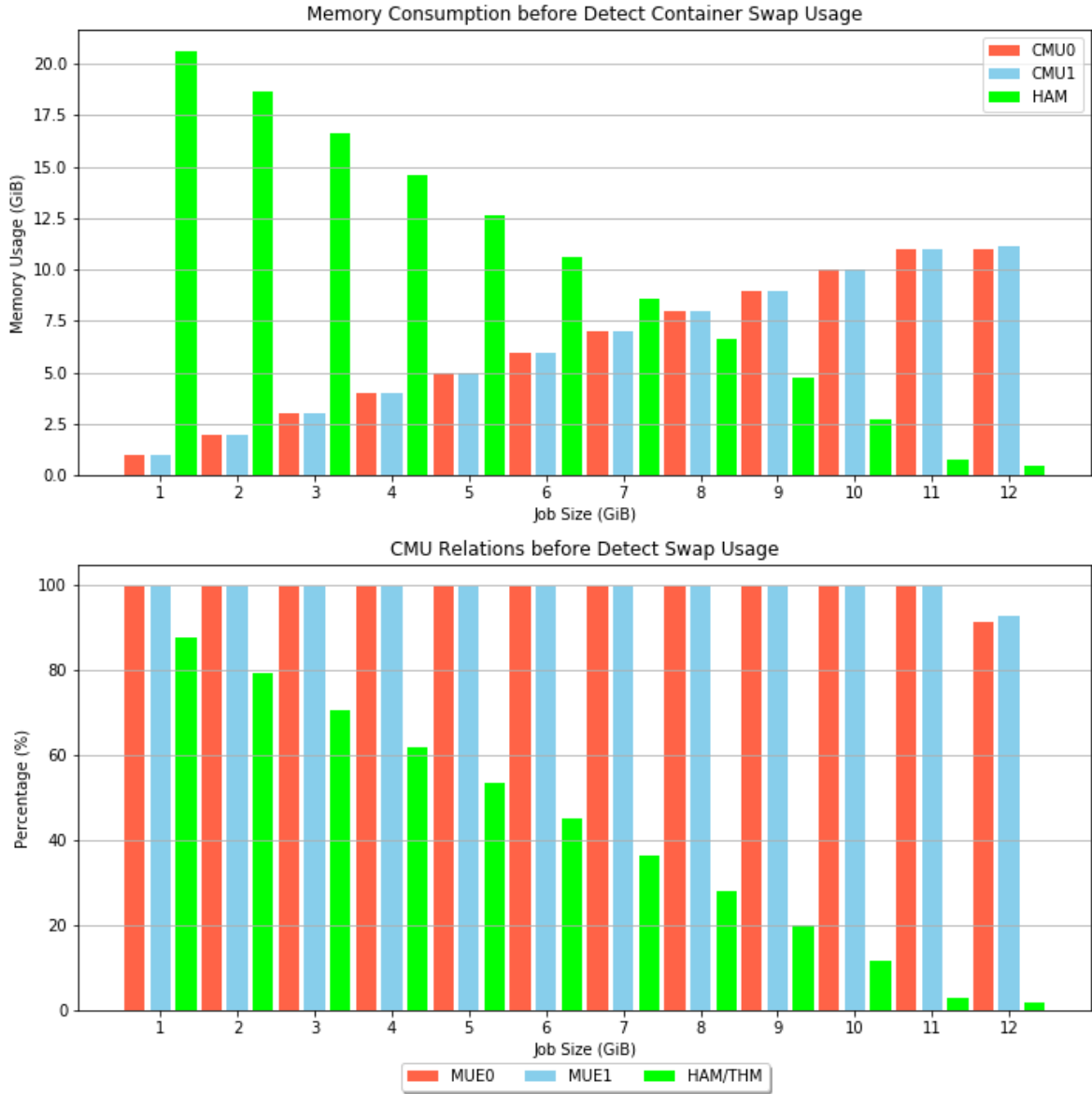


Figure 6.8: Container Memory Limit Utilization for 2 containers.

The following experiments extend this investigation to consider multiple concurrently executing containers. *TCML* and *TCMU* are the sum of their respective *CMLs* and *CMUs*, while *HTMU* is the host's total memory usage. Given *CMU* is the maximum quantity of memory used before swap usage occurred, *MUE* continues to be calculated

as the ratio of $TCMU$ to $TCML$. As shown in Figure 6.8 and Table 6.7, we can observe that the pattern remains with MUE tapering off as $HTMU$ reaches the 24 GiB limit. However, the fall is now more distinct as the two containers start to affect each other, reducing the amount of memory that can be used effectively. A similar trend was observed when running four containers simultaneously, Figure 6.9 and Table 6.8 that shows MUE drops under 90%, and the total memory utilization, $TCMU$, does not even reach 22 GiB.

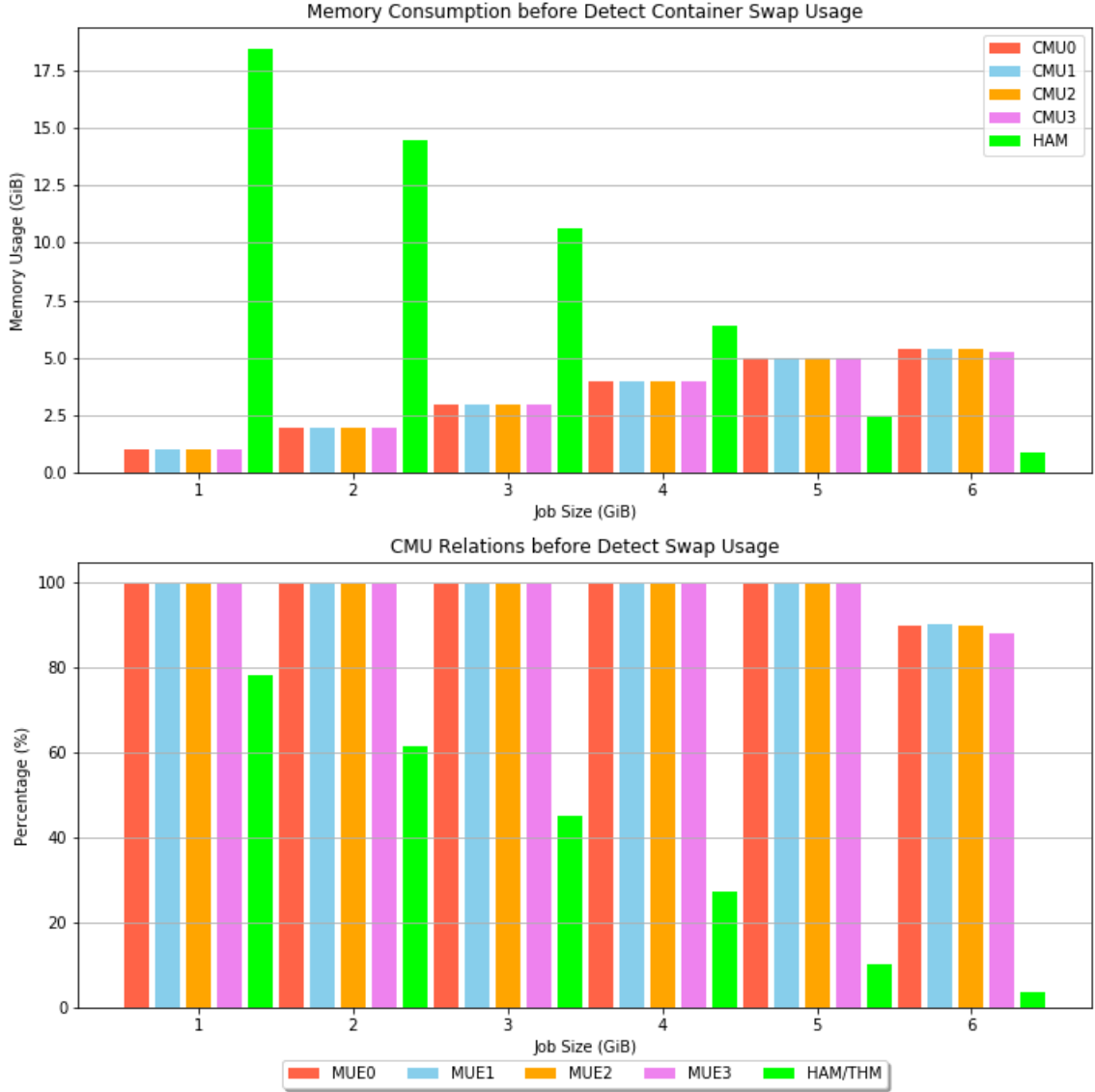


Figure 6.9: Container Memory Utilization for 4 containers.

Based on these experiments, we choose an upper bound for the value of $HTMU$, in terms of memory pages, to determine the value of the constant Host Memory Usage Threshold ($HMUT$), as shown in Line 3 of Algorithm 1 presented earlier in Section 5.2) and used a best-fit linear regression analysis to derive a function to adjust the percentage MUE at each scheduling interval, in relation to the current amount of memory available

Table 6.8: Memory metric values presented in Figure 6.9 for the execution of 4 containers simultaneously.

TCML	TCMU	MUE (%)	HRM	HTMU	HAM
4	3.990	99.757	0.745	4.735	18.440
8	7.983	99.783	0.737	8.719	14.456
12	11.975	99.791	0.603	12.578	10.598
16	15.967	99.792	0.788	16.755	6.421
20	19.959	99.795	0.784	20.744	2.431
24	21.479	89.497	0.817	22.296	0.880

on the host, *HAM*, as described in Lines 20 to 22 of Algorithm 1, during Phase 1 of VEMoC.

6.4 Container and Host Manager Overheads

To determine reference execution times for the jobs, this set of experiments executed a varying number of 4 GiB J1 or J2 jobs concurrently, on 3 different platforms: 1) directly on the host server; 2) inside a LXC container; and 3) inside a LXC container managed by VEMoC. Each execution was repeated 20 times, with Table 6.9 showing the average times with 95% confidence intervals.

Table 6.9: Average execution times in seconds with 95% confidence intervals for 4 GiB jobs on different platforms.

Number of Concurrent Containers	1	2	5
J1 job directly on the host	420.5 \pm 0.7	423.7 \pm 1.1	439.8 \pm 4.1
J1 job inside a LXC container	422.7 \pm 0.4	433.7 \pm 3.7	452.7 \pm 1.5
VEMoC managed J1 LXC container	422.8 \pm 0.9	428.4 \pm 2.5	445.3 \pm 1.5
J2 job directly on the host	422.2 \pm 0.7	422.9 \pm 0.6	436.5 \pm 7.0
J2 job inside a LXC container	421.2 \pm 0.6	428.7 \pm 5.0	450.0 \pm 1.4
VEMoC managed J2 LXC container	420.9 \pm 0.6	427.1 \pm 2.3	445.9 \pm 3.2

The times show in Table 6.9 that there is little overhead between all three scenarios. In particular, VEMoC does not cause interference, while adjusting *CML* to dynamically provide sufficient memory for the application. As the number of running jobs increases, so does their average execution times as the jobs compete for memory bandwidth, as expected. Although not conclusive, the results hint at VEMoC having a beneficial impact on LXC container management.

To demonstrate how an application might be (or not be) affected by wrong decisions when setting *CML*, we compare executions of 4 GiB *J1* and *J2* jobs with different fixed

values for container's *CML*. Remember that the codes of two jobs are almost identical, both access each element of their vector 4 times. Given an unlimited amount of memory, both jobs consume 4 GiB. The difference then between them is only their memory access pattern, with *J1* successively accessing each element of the vector once from the beginning to the end before returning to access the first element again. In the case of *J2*, the vector is divided into four blocks, with the elements of the first block being accessed sequentially in a cycle 4 times before moving to the next block and repeating the same process before progressing. The execution times of each job are shown in Figure 6.10.

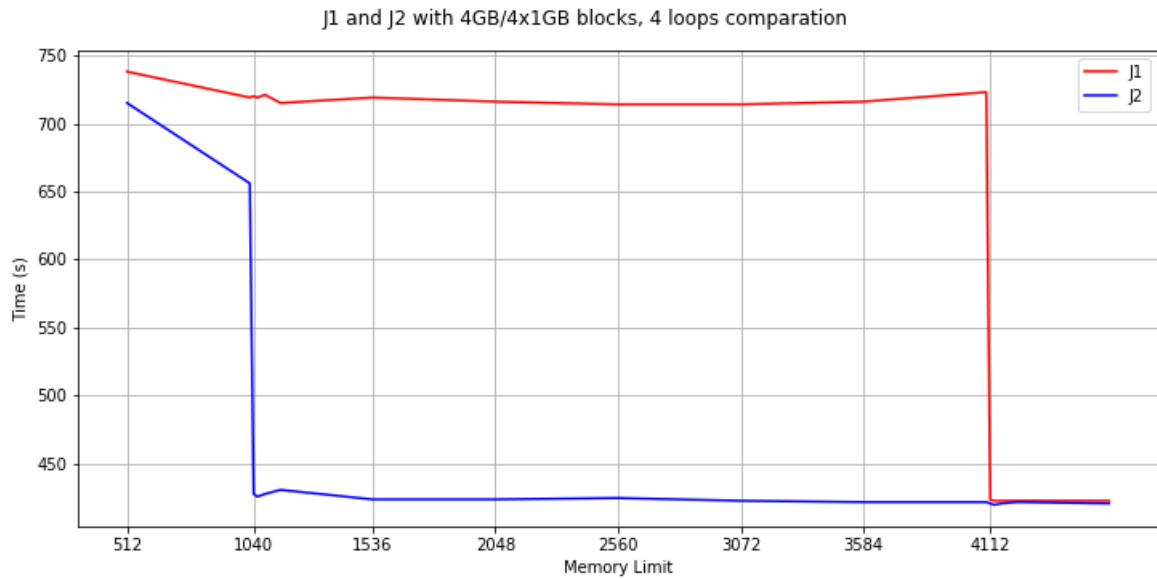


Figure 6.10: The execution times of 4 GiB jobs *J1* and *J2* with different container memory limits.

The performance of job *J1* suffers degradation as *CML* drop below the value equivalent to the size of the vector. On the other hand, *J2* continues to execute with a consistent duration while the *CML* remains larger than the size of the block (1 GiB in this case). Since both jobs require at least 4 GiB memory, when executed with a smaller *CML*, the respective containers will be forced to use swap. The difference in their execution times is due to their different working sets and the Linux kernel's least recently used (LRU) page substitution policy when swapping pages out of memory. In the case of *J1*, the least recently used pages that are swapped to disk to make room for the required pages are themselves the ones that will need to be accessed again by the application in the near future. The increase in the execution time in relation to one with a *CML* of a little more (16 MiB) than 4 GiB is due to the additional delay caused by the swap in operations. A slowdown for *J2* only occurs when the *CML* is below 1 GiB + 16 MiB (16 MiB being the amount of memory required by the container process itself to execute). Notice that

the 1 GiB limit coincides with the size of the vector processed within a block. Because of data locality in $J2$, the working set size to process a block is only 1 GiB. The pages of memory that are swapped out are no longer accessed, thus no swap in operations are required to execute $J2$ although swap is being used. As one can see from Figure 6.10, the differences in both the execution times and memory consumption are glaring. As far as we know, VEMoC is the only container elasticity controller to make this distinction and try to take advantage of it, although some previous work has been proposed in the context of virtual machines [67, 68]

6.5 Managing Container Memory Limits

VEMoC dynamically adjusts the limit CML on the memory a container has access to during its execution, without compromising its performance and without exceeding the minimum and maximum memory limits of the job. To exemplify the dynamic changes made to CML , this section first presents the execution of an individual $J1$, $J2$ and $J4$ job, each with a vector size of 4GiB, divided into 4 blocks in the case of jobs $J2$ and $J4$, and 1 block for $J1$. The Max Memory Limit ($MaxML$) is set to 16 MiB above the vector size to allow the execution of 4 loops with sufficient available memory. All jobs start with the Minimum Memory Limit ($MinML$) of 512 MiB.

The $J1$ job represents those applications that initially allocate their datasets to memory before processing the data during its execution. The staircase behavior present in Figure 6.11 shows VEMoC (by way of Algorithms 2, 3, 4 and 7) proactively increasing the CML of the container during the allocation of the entire vector in memory (and execution of the first cycle). When the job completes this allocation phase its memory consumption remains stable (after 110 seconds, approximately), VEMoC maintains $CML = MaxML$ during the rest of the job execution, with sufficient spare memory to avoid swap usage and performance degradation. Look carefully at what happens between 105 and 110 seconds, you might see a dip in the CML . This behaviour is related to discussion related to non-working set memory at the end of the previous section and will become clearer in the next example.

$J2$ aims to represent jobs who work on parts of their datasets, one or a few at a time. Figure 6.12 shows that a $J2$ job reaches its first stability phase (after 25 seconds, approximately, the equivalent to a quarter of the time to allocate 4 GiB). The pattern of consumption repeats for each additional one of the four blocks. At similar points in

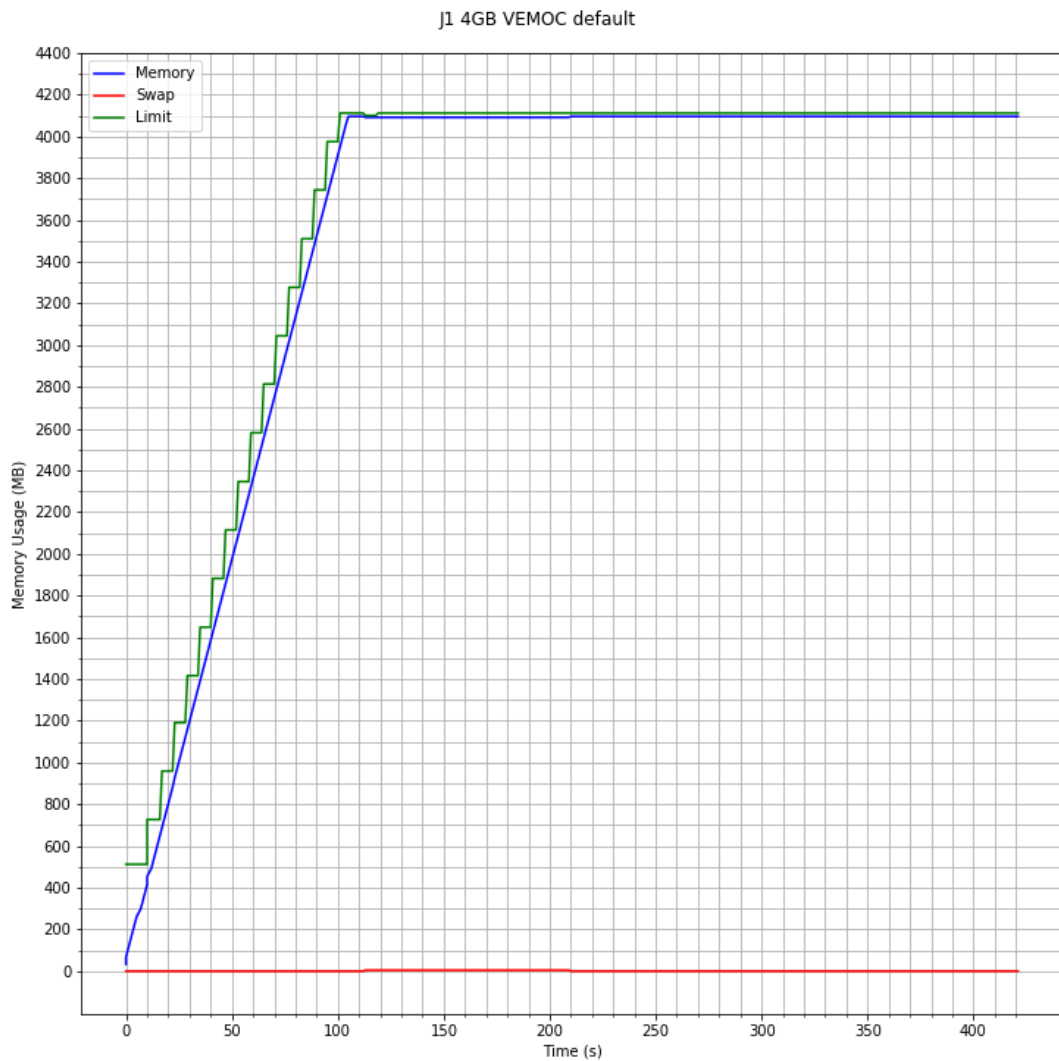


Figure 6.11: VEMoC predicting the memory demand of a *J1* type job.

the first three phases, one can see that the *CML* is significantly higher than the memory consumption of the container. VEMoC has included sufficient "spare capacity" to guard against sudden changes in the container's memory consumption, until the next scheduling cycle. This spare capacity is maintained as long as the host has enough available memory but can be donated (through algorithm 5 to RISING containers in need additional memory, should the host not have enough memory to serve them.

The valley presented in the green line of the *CML*, is where Algorithm 4 forces the Linux kernel reclassify the containers active and inactive pages, given the container has reached a STABLE state. This is so that Algorithm 6 can decide whether is container could be a candidate from which memory can be removed, should the host not have enough in the future. A small test is carried out by forcing a small amount of the container's memory to swap. VEMoC then waits to see it is then swapped back in. If not then the



Figure 6.12: VEMoC predicting the memory demand of a *J2* type job.

container hold data from outside its current working set in memory. In Figure 6.12, we see shortly afterwards that memory consumption returns to the previous level indicating that the data is still part of the container’s working set.

J4 represents the somewhat rarer case where applications free up memory they are no longer using during their execution. As seen in Figure 6.13, the initial behavior of *J4* is identical to that of *J1*, where the vector is being allocated to memory. When the job finishes processing a block, *J4* shrinks the vector, freeing the memory used by the removed block. This is detected by VEMoC in Algorithm 4, which adjusts the *CML* appropriately, while again updating the LRU page list, and then incorporating the spare capacity, in case new data needs to be brought into memory before the next cycle.

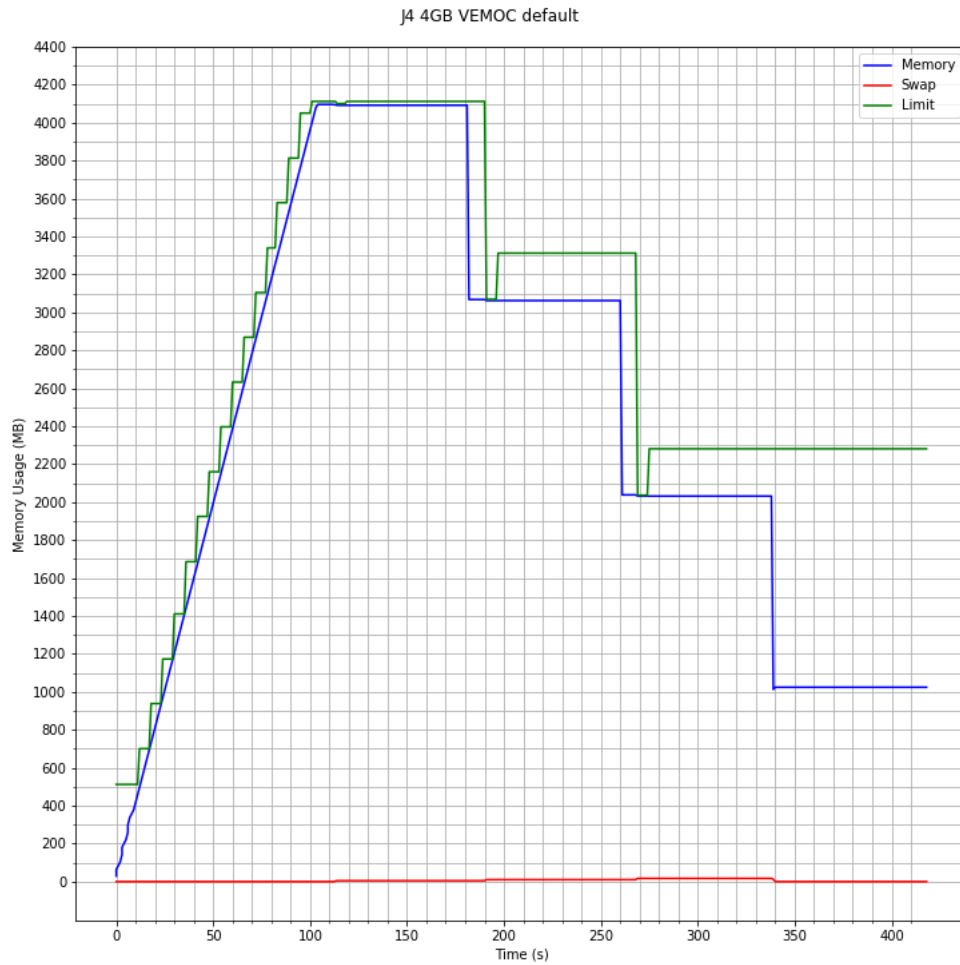


Figure 6.13: VEMoC predicting the memory demand of a *J4* type job.

6.6 Benefits of Vertical Memory Elasticity

The following three experiments demonstrate VEMoC’s ability to predict and adjust the *CML* to meet the memory demands of each job in scenarios where insufficient memory is available (and migration has been disabled). All containers begin with a minimum memory limit of 512 MiB and are pinned to distinct CPU cores for the duration of their executions.

The performance of VEMoC is compared with the following three commonly adopted forms of defining memory limits for containers (loosely based on Kubernetes QoS terminology):

- *Guaranteed* – with the limit set at the maximum amount of memory required, the job will only be submitted when that amount is available;
- *Fair Share* - Prior to execution, the available memory is divided equally among the

jobs to be executed;

- *Best Effort* – Run jobs on arrival if the minimum memory limit is available. Containers can use what ever free memory is available, limited only by their own maximum memory limit.

These comparisons are based on five 5 different metrics:

- The *Total Scenario Execution Time* (TSET) is the wallclock time from the first job submission to end of the last job, in seconds;
- The *Average Job Turnaround Time* (AJTT) in seconds;
- The *Average Memory Utilization* (MemUtil) is the average for the scenario of each of the job’s average percentage *MemUtil* (i.e., $\frac{CMU}{CML}$) over their respective execution;
- The *Total Memory-Time Product* (TMTP) is a cost metric (in millions of page-seconds) for clients, which considers the duration a quantity of memory was reserved for, in that scenario, and;
- The *Average Host Memory Utilization* (AHMU) percentage that represents the effective use of the host’s memory during TSET.

6.6.1 Scenario 1: Two 4 GiB J1 jobs are submitted 50 seconds apart to a host with 6 GiB of available memory

With insufficient memory for both identical jobs, orchestration with a *Guaranteed* policy will run the jobs sequentially, while a *Fair Share* policy reserves 3 GiB for each. In the case of *Best Effort* and VEMoC, ultimately, the amount of memory each job receives, and when, depends on the relative delay between submissions. The difference between the two resulting schedules is that VEMoC suspends the second job to release its memory for use by the first (Phase 6). From the data in Tables 6.4 and 6.5, VEMoC does not need to pause the first job since it will not be affected by the suspension delay. If the interval between jobs had been larger, enough so that the first job had received all the memory it required, the second job would have been paused by VEMoC, avoiding the suspension and resume costs. Table 6.10 presents the metrics for each scheme and the percentage difference in relation to VEMoC (red is how much worse, blue is how much better, compared to VEMoC, the traditional approaches performed).

Table 6.10: Comparing allocation strategies under Scenario 1

	TSET	AJTT	MemUtil	TMTTP	AHMu
Guaranteed	843.2 (4.9%)	606.3 (2.6%)	87.1 (10.1%)	886.98 (10.9%)	58.1 (5.7%)
Fair Share	1245.2 (54.9%)	1193.2 (101.8%)	96.4 (0.5%)	1876.71 (134.6%)	92.2 (49.7%)
Best Effort	1284.0 (59.7%)	1236.9 (109.2%)	95.2 (1.8%)	2019.6 (152.5%)	92.8 (50.6%)
VEMoC	804.0	591.2	96.9	799.8	61.6

VEMoC performs better than the three orchestration policies in all of the metrics, except for *AHMu*. VEMoC has a slightly better TSET and AJTT in relation to Guaranteed, because the second *J1* job took advantage of some spare capacity before being suspended when the host has used up its available memory. Fair Share and Best Effort both force the containers to use swap which delays their executions significantly. However, since both containers are executing concurrently, *AHMu* is high since all the available memory is being used.

Nonetheless, our proposal reserves 152% and 134% fewer pages than Best Effort and Fair Share, respectively, which means that VEMoC is the cheapest alternative for the user, even compared to the Guaranteed, the most commonly employed policy by cloud providers.

6.6.2 Scenario 2: One 4 GiB *J2* job is followed by one 4 GiB *J1* job, 100 seconds later, on a host with 6 GiB of available memory

Memory accesses for the *J2* job have more locality than those of *J1*. Phase 4 allows VEMoC to identify that *J2* may have pages in memory that it no longer needs and forces *J2* to swap them out, reducing its RAM consumption. VEMoC is thus able to obtain the memory required for *J1* without having to suspend or pause either job. Under a Guaranteed policy, the jobs again run sequentially. However, under both Fair Share, where the memory is divided evenly between both jobs, and Best Effort, where the *J1* jobs consumes memory faster to obtain the 4 GiB it needs thus leaving the *J2* job with just 2 GiB, the jobs without their required amount of memory are forced to use swap. Remember though that job *J2*'s performance is not affected by swapping out pages and thus both policies obtain a better TSET and AJTT (see Table 6.11) than Scenario 1. In this scenario, although Best Effort now turns out to be better than Fair Share because *J1*

did not need to use swap, VEMoC is clearly still more efficient.

Table 6.11: Comparing allocation strategies under Scenario 2

	TSET	AJTT	MemUtil	TMTTP	AHMu
Guaranteed	841.1 (58.8%)	582.8 (36.7%)	73.1 (22.3%)	884.8 (50.6%)	48.8 (28.0%)
Fair Share	793.2 (49.8%)	561.8 (31.8%)	83.1 (11.7%)	883.6 (50.4%)	60.5 (10.8%)
Best Effort	637.9 (20.4%)	483.1 (13.2%)	71.1 (24.4%)	1002.7 (70.7%)	65.3 (3.7%)
VEMoC	529.7	426.2	94.1	587.5	67.8

6.6.3 Scenario 3: One 4 GiB J2 job is followed by five 4 GiB J1, at 10 seconds intervals, using all the available memory (23.5GiB) on a host while sharing the Linux operating system.

This scenario is designed to stress the host, given this workload would be require the containers to consume all the physical memory in the system to achieve good performance, without leaving any for the operating system and other hosts processes. Table 6.12 shows VEMoC again performs better and suffers less interference than the other three mechanisms because of its awareness of *MUE* and its perception of *HRM*, reflected by the lower *MemUtil* values. Guaranteed started all but one job on arrival, while Fair Share ran all jobs concurrently (as reflected by a higher AHMu) but with less than ideal amounts of memory. With 3 of the J1 jobs acquiring sufficient memory helped Best Effort’s AJTT.

Table 6.12: Comparing allocation strategies under Scenario 3.

	TSET	AJTT	MemUtil	TMTTP	AHMu
Guaranteed	867.8 (59.6%)	506.9 (3.1%)	82.6 (11.8%)	2788.4 (5.4%)	42.8 (42.2%)
Fair Share	779.6 (43.4%)	663.6 (35.0%)	84.9 (9.3%)	4093.0 (54.7%)	73.7 (0.4%)
Best Effort	715.1 (31.5%)	543.4 (10.5%)	72.6 (22.4%)	4407.8 (66.7%)	63.0 (14.9%)
VEMoC	543.6	491.7	93.6	2644.9	74.0

Moreover, VEMoC achieves an average of 74% host utilization, while allocating at least 50% fewer pages than Best Effort and Fair Share, and a per container latency that is at least 11% better than them. In summary, VEMoC is consistency better independently of the metric adopted.

6.7 Scalability Analysis

To evaluate the ability of VEMoC to manage increasing numbers of containers, a final scenario is considered where several small *J1* jobs (with a working set of 1 GiB) are submitted for execution at 5 second intervals between them, simulating the submission of micro-services to a (function-as-a-service) host. Each job is executed in a distinct container whose execution is managed by VEMoC, executing under its default configuration where its host monitor is activated every second and the scheduling interval was 6 seconds.

Table 6.13: Execution of small *J1* jobs with VEMoC, starting the jobs in intervals of 5 seconds.

N ^o of Jobs	AJTT	Monitor Avg. Time	CM Avg. Time
1	106.1 \pm 0.6	0.045 \pm 0.01	0.106 \pm 0.01
5	115.3 \pm 3.8	0.247 \pm 0.01	0.575 \pm 0.04
10	119.5 \pm 5.9	0.507 \pm 0.03	1.171 \pm 0.14
15	118.5 \pm 5.3	0.940 \pm 0.01	2.262 \pm 0.30

Table 6.13 presents the *AJTT*, the average time to monitor all of the running containers, and VEMoC's Container Manager (CM) processing latency per scheduling cycle. After completing their respective cycles of execution, both the host monitor and VEMoC container manager sleep for the time remaining to complete their respective monitoring or scheduling interval. The execution times were collected from the tool's log file of the same execution.

Analysing the results presented in the Table 6.13, VEMoC is able to collect statistics from up to 15 containers, simultaneously, within the same second. The host monitor collects statistics from each container's cgroup files through the LXC API. On our host system, this information is stored on standard SATA hard disks and without redundancy or mirroring (RAID), thus improvements might be expect through the optimized use of high speed storage devices, such as Solid State Drives (SSD).

As for the Container Manager, it was able to manage the same number of containers in less than half of the total scheduling interval of 6 seconds, demonstrating that VEMoC can scale from a few containers with heavy jobs to many containers with small jobs. Nevertheless, these are still varying non-trivial latencies that have had to be taken into consideration by the prediction model used by VEMoC. Although, we do not expect to running significantly higher numbers of containers, these two intervals could be enlarged to permit the effective management of greater numbers of container per host.

Chapter 7

Conclusions and Future Work

The rapidly increasing adoption of cloud computing has turned it into a multi-billion dollar business. Cloud providers offer computing resources and services, on demand, over the Internet, while aiming to quickly adjust the amount of resources to meet customer needs, without over- or under-provisioning, and thus be able to charge each user for exactly what was used. With high computing capacities capable of processing large volumes of data, the cloud is making computing more accessible, reducing operational costs and accelerating the time to generate results and develop products. With the increasing demand for the use of the cloud, these providers must continue to search for new solutions to make the utilization of server machines in their data centers even more efficient, as well as guaranteeing good levels of quality of service for their users at lower costs and in a more sustainable way.

To make computing more efficient, cloud providers offer virtual environments that share the available computing infrastructure. Much has been analyzed regarding the use of container virtualization technology [32, 58]. Containers are minimalistic virtual environments, with resource allocation being based on the application to be executed, that share the operating system with the host server, and do not require specialized hardware for virtualization, as opposed to virtual machines. However, this technology is still in the process of consolidation and continues to face several challenges related to the dynamic allocation of resources, and the monitoring and management of the quality of service being offered to cloud users [3].

A preliminary evaluation, summarised in Appendix A, concluded that, in general, container virtualization has performance and efficiency advantages over virtual machines. Results such as these appear to be motivating an increased migration towards container based cloud services including *serverless computing*. Increasing the efficiency of these

types of services is of financial and operational importance to cloud providers and motivated the theme of this thesis. Additionally, with the trend in microprocessor development towards an ever-increasing numbers of cores per socket over the last decade, the performances of applications have become increasingly more sensitive to the availability of memory. This thesis has therefore presented VEMoC, a container orchestration tool to manage the life cycle of containers allocated to a host for execution. VEMoC is equipped with an algorithm to enhance the containers with vertical memory elasticity in order to increase server utilization without adversely affecting individual container performances.

This thesis initially introduced the basic concepts of containerization and elasticity in Chapter 2, while briefly describing in Chapter 4 the design and implementation of a scalable modular cloud architecture that collects a variety of resource utilization metrics used by VEMoC to make iterative memory allocation decisions during scheduling cycles. The cloud management architecture itself is a conventional two-level hierarchy that encompasses multiple VEMoC instances. Described in detail in Chapter 5, the VEMoC orchestration algorithm executes a sequence of 7 stages in each scheduling interval to manage the life-cycle of the set of queued containers. The algorithm incorporates a number of novel features, including: low overheads, thus higher performance, due to a simple control logic afforded by evaluating container consumption trends rather than making decisions based on absolute values; a focus on the distribution of the host's memory resources and not just individual container's elastic allocation; the recovery of both unused and underutilized memory; uses a proposed Memory Utilization Effectiveness (*MUE*) metric to take into consideration the change in behaviour of the Linux operating system under extremely high memory utilization, the scenario where an orchestrator like VEMoC is of fundamental importance, and; the possibility of stopping or pausing the execution of some containers to favor the execution of others.

A series of experimental evaluations elaborated in Chapter 6 demonstrated the technique of vertical memory elasticity, with Docker and LXC containers within the context of the VEMoC architecture. In order to make accurate predictions, the first experiments aim to understand the mechanics of how containers behave with respect to changes in memory usage limits, the duration of container state transitions and data collection latencies, and the costs to pause/unpause and suspend/resume containers with varying amounts of memory consumption. One aspect, not previously seen discussed in other research, is that a container ability to utilize all of the memory within its memory limit (*CML*) depends on the proportion of free memory on the host. A new metric *MUE* was proposed to address this issue.

A comparative evaluation on a physical server has shown VEMoC to be more efficient than existing approaches adopted by production services such as [44]. This is the case not only in terms of memory utilization (with VEMoC constantly sustaining utilization efficiencies above 90%) and job turn around times (where VEMoC can be as much as twice as fast) but also in terms of lowering costs for clients, being at least 50% more efficient in the majority of the comparisons. Note that traditional threshold or capacity-based vertical elasticity approaches have their **peak** utilization efficiencies bounded by their chosen threshold value, typically between 70% and 80% [2, 48]. The sustained or average memory utilization over the execution of the container can therefore be significantly lower than this. By considering memory read/write speeds, together With the asynchronous monitoring data collection and low scheduling cycle time, VEMoC is able to track the memory consumption of containers and adjust their limits, without causing their respective applications use swap unnecessarily.

Another aspect that has not been discussed much is VEMoC’s robustness. While operating systems frequently kill processes when the host runs low on memory, cloud orchestrators, like Google’s Borg system, may kill containers that consume more RAM than expected. In fact, Kubernetes [44] adopts the philosophy that swap space is not needed (and should be disabled on the host) since its containers (pods) should have a memory limit set high enough to meet its application’s maximum requirement and only be allocated to a host with resources available to meet this limit (Guaranteed QoS). Nevertheless, if a host runs low on memory or a container reaches that limit, an Out of Memory (OOM) exception may cause the container to be “killed” by host’s OS. This leads to wasted resource consumption and longer completion times.

VEMoC does not require a priori knowledge about the requirements of the applications running in the container instead uses page level predicted memory consumption rates, rather than arbitrary usage thresholds for fine-grained vertical elasticity of memory, combined with the prudent use of memory stealing and container preemption. Across a set of scenarios, comparing VEMoC with 3 scheduling strategies widely used in other container orchestrators, VEMoC demonstrated superior performance, even where memory becomes extremely scarce. Moreover, the VEMoC demonstrated its scalability by managing 15 containers running simultaneously consuming less than half of the scheduling cycle time available. Through its dynamic mechanism VEMoC is able to adjust for varying monitoring and scheduling times, which permits it to scale to cope with larger quantities of containers on each host of a cloud infrastructure.

Some of the work and results derived from this thesis have been presented at the 2020 IEEE International Conference on Utility and Cloud Computing [53]. Although the experimental results presented in this thesis were primarily obtained with LXC containers, VEMoC does work with other container engines such as Docker. In the specific case of Docker, some improvements in the host monitor are needed because a limitation found in the Docker API that makes it impossible to monitor the host and multiple containers simultaneously, every second. This perhaps could be resolved by substituting the default Docker API function to collect the monitoring metrics with an container independent one that can parse the cgroups system files directly. Furthermore, the container suspension functionality available in Docker is still experimental, which is perhaps motive for its relative poor performance in relation to LXC. One can only expect the suspension and resumption costs for Docker to improve with further development in the future. Meanwhile, alternative checkpoint library implementations for Docker and other container engines should be evaluated.

Other avenues of future work also include extending VEMoC to other container engines through the creation of a corresponding compatible container module for each new container engine. This would also allow VEMoC to be integrated with other orchestration systems, like Kubernetes, to offer more flexible autoscaling features. Furthermore, having a modular architecture, VEMoC can also be used for the elaboration and evaluation of new scheduling policies for container-based infrastructures.

The VEMoC architecture and tool has been developed and deployed in real environments, including public cloud infrastructures, within evaluations carried out *in situ*, rather than using data collected through simulations. VEMoC also supports the elastic management of CPU utilization and follow on work might want investigate how to integrate CPU throttling with memory elasticity to reduce the number of expensive preemption events to improve performance and costs.

References

- [1] ADVANCED MICRO DEVICES, INC. AMD Virtualization. <http://www.amd.com/pt-br/solutions/servers/virtualization>, 2016. Accessed on: 10/08/2016.
- [2] AL-DHURAIBI, Y.; PARAISO, F.; DJARALLAH, N.; MERLE, P. Autonomic vertical elasticity of docker containers with elasticdocker. In *2017 IEEE 10th international conference on cloud computing (CLOUD)* (2017), IEEE, pp. 472–479.
- [3] AL-DHURAIBI, Y.; PARAISO, F.; DJARALLAH, N.; MERLE, P. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing* 11, 2 (2018), 430–447.
- [4] AL-DHURAIBI, Y.; ZALILA, F.; DJARALLAH, N. B.; MERLE, P. Coordinating vertical elasticity of both containers and virtual machines. In *CLOSER 2018 - The 8th International Conference on Cloud Computing and Services* (2018).
- [5] ANTON, V.; RAMÓN-CORTÉS, C.; EJARQUE, J.; BADIA, R. M. Transparent execution of task-based parallel applications in docker with comp superscalar. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on* (2017), IEEE, pp. 463–467.
- [6] APACHE SOFTWARE FOUNDATION. Apache mesos. <http://mesos.apache.org/>, 2017. Accessed on: 01/07/2017.
- [7] BADGER, L.; GRANCE, T.; PATT-CORNER, R.; VOAS, J. Cloud computing synopsis and recommendations. Tech. rep., National Institute of Standards and Technology, 05 2012.
- [8] BARUCHI, A.; MIDORIKAWA, E. A survey analysis of memory elasticity techniques. In *Euro-Par 2010 Parallel Processing Workshops*, vol. 6586 of *LNCS*. Springer, 2011, pp. 681–688.
- [9] BERNSTEIN, D. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.
- [10] BIEDERMAN, E.; KERRISK, M. Namespaces - overview of linux namespaces. <http://man7.org/linux/man-pages/man7/namespaces.7.html>, 2017. Accessed on: 15/06/2017.
- [11] BIEDERMAN, E. W.; NETWORX, L. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium* (2006), vol. 1, pp. 101–112.
- [12] BOULLE, J. Open container initiative image format specification. <https://github.com/opencontainers/image-spec/blob/master/spec.md>, 2017. Accessed on: 05/03/2020.

- [13] CALATRAVA, A.; ROMERO, E.; MOLTÓ, G.; CABALLER, M.; ALONSO, J. M. Self-managed cost-efficient virtual elastic clusters on hybrid cloud infrastructures. *Future Generation Computer Sys.* 61 (2016), 13–25.
- [14] CANONICAL LTD. Linux containers - lxc. <https://linuxcontainers.org/lxc/introduction>, 2020. Accessed on: 27/02/2020.
- [15] CANONICAL LTD. What is lxd? <https://linuxcontainers.org/lxd/introduction/>, 2020. Accessed on: 22/06/2020.
- [16] CITRIX SYSTEMS INC. Citrix hypervisor. <http://xenserver.org/>, 2020. Accessed on: 10/08/2020.
- [17] CLOUD NATIVE COMPUTING FOUNDATION. Containerd. <https://containerd.io/>, 2020. Accessed on: 20/07/2020.
- [18] COUTINHO, E. F.; SOUSA, F.; REGO, P.; GOMES, D.; DE SOUZA, J. Elasticity in cloud computing: A survey. *Annals of Telecommunications* 70, 7 (2015), 289–309.
- [19] CRIU CONTRIBUTORS. Checkpoint/restore in userspace - CRIU. <https://criu.org/>, 2017. Accessed on: 15/06/2017.
- [20] CUADRADO-CORDERO, I.; ORGERIE, A.-C.; MENAUD, J.-M. Comparative experimental analysis of the quality-of-service and energy-efficiency of vms and containers' consolidation for cloud applications. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2017)* (sept 2017), pp. 1–6.
- [21] DAWOUD, W.; TAKOUNA, I.; MEINEL, C. Elastic virtual machine for fine-grained cloud resource provisioning. In *Proc. of the 4th International Conference on Global Trends in Computing and Communication Systems* (2012), Springer, pp. 11–25.
- [22] DE ALFONSO, C.; CALATRAVA, A.; MOLTÓ, G. Container-based virtual elastic clusters. *Journal of Systems and Software* 127 (2017), 1–11.
- [23] DE ALMEIDA, J. M. B. Container, o novo passo para a virtualização. *Mini Paper Series of Tecnology Leadership Council Brazil* 10, 234 (2015). Disponível em https://www.ibm.com/developerworks/community/blogs/tlcbr/resource/mp/TLC-BR_Mini_Paper_Ano_10_N_234_Containers.pdf?lang=en.
- [24] DE SOUZA MUNOZ, M.; DE GIOVANNI, R.; DE SIQUEIRA, M.; SUTTON, T.; BREWER, P.; PEREIRA, R.; CANHOS, D.; CANHOS, V. Openmodeller: a generic approach to species' potential distribution modelling. *GeoInformatica* 15, 1 (2011), 111–135.
- [25] DOCKER INC. Docker engine user guide. <https://docs.docker.com/engine/userguide/>, 2016. Accessed on: 01/07/2016.
- [26] DOCKER INC. About images, containers, and storage drivers. <https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/>, 2017. Accessed on: 17/06/2017.

- [27] DOCKER INC. Docker overlay network - get started with multi-host networking. <https://docs.docker.com/engine/userguide/networking/get-started-overlay>, 2017. Accessed on: 17/06/2017.
- [28] DOCKER INC. Docker get started. <https://docs.docker.com/get-started/>, 2020. Accessed on: 27/02/2020.
- [29] DOCKER INC. Docker storage drivers. <https://docs.docker.com/storage/storagedriver/select-storage-driver/>, 2020. Accessed on: 27/02/2020.
- [30] DOCKER INC. Swarm mode overview. <https://docs.docker.com/engine/swarm/>, 2020. Accessed on: 27/02/2020.
- [31] DOCKER INC. Use the device mapper storage driver. <https://docs.docker.com/storage/storagedriver/device-mapper-driver/>, 2020. Accessed on: 22/06/2020.
- [32] DUA, R.; RAJA, A. R.; KAKADIA, D. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on* (2014), IEEE, pp. 610–614.
- [33] ELASTICSEARCH B. V. What is elasticsearch? <https://www.elastic.co/pt/what-is/elasticsearch>, 2020. Accessed on: 22/06/2020.
- [34] FAROKHI, S.; JAMSHIDI, P.; LAKEW, E. B.; BRANDIC, I.; ELMROTH, E. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *Future Generation Computer Sys.* 65 (2016), 57–72.
- [35] FELTER, W.; FERREIRA, A.; RAJAMONY, R.; RUBIO, J. An updated performance comparison of virtual machines and linux containers. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2015), pp. 171–172.
- [36] FRIEDL, S. Go directly to jail: Secure untrusted applications with chroot. *Linux Magazine* (2002).
- [37] GALANTE, G.; DE. BONA, L. C. E. A survey on cloud computing elasticity. In *2020 IEEE/ACM 5th International Conference on Utility and Cloud Computing (UCC)* (2012), p. 263–270.
- [38] GELLER, G. N.; MELTON, F. Looking forward: Applying an ecological model web to assess impacts of climate change. *Biodiversity* 9, 3-4 (2008), 79–83.
- [39] GOOGLE. Vertical pod autoscaling. <https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler>, 2020. Accessed on: 05/07/2020.
- [40] HERBST, N. R.; KOUNEV, S.; REUSSNER, R. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing (ICAC 13)* (San Jose, CA, June 2013), USENIX Association, pp. 23–27.

- [41] HONG, C.-H.; VARGHESE, B. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. *ACM Comput. Surv.* 52, 5 (Sept. 2019).
- [42] INTEL CORPORATION. Intel Virtualization Technology (Intel VT). <http://www.intel.com.br/content/www/br/pt/virtualization/virtualization-technology/intel-virtualization-technology.html>, 2016. Accessed on: 10/08/2016.
- [43] KLOH, H.; REBELLO, V. E. F.; BOERES, C.; SHULZE, B.; FERRO, M. Static job scheduling for environments with vertical elasticity. *Concurrency and Computation: Practice and Experience* 32, 19 (2020).
- [44] KUBERNETES CONTRIBUTORS. Kubernetes. <https://kubernetes.io/>, 2017. Accessed on: 01/07/2017.
- [45] KVM CONTRIBUTORS. Kernel virtual machine. http://www.linux-kvm.org/page/Main_Page, 2016. Accessed on: 10/08/2016.
- [46] LINUX FOUNDATION. Open container initiative. <https://www.opencontainers.org/>, 2017. Accessed on: 10/06/2020.
- [47] MENACE, P.; JACKSON, P.; LAMETER, C. Cgroups. <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>, 2017. Accessed on: 15/06/2017.
- [48] MOLTÓ, G.; CABALLER, M.; DE ALFONSO, C. Automatic memory-based vertical elasticity and oversubscription on cloud platforms. *Future Generation Computer Systems* 56 (2016), 1–10.
- [49] MORABITO, R. Power consumption of virtualization technologies: An empirical investigation. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)* (Dec 2015), pp. 522–527.
- [50] MUHAMMAD, H. htop - an interactive process viewer for unix. <http://hisham.hm/htop/>, 2016. Accessed on: 01/07/2016.
- [51] NAJJAR, A.; SERPAGGI, X.; GRAVIER, C.; BOISSIER, O. Survey of Elasticity Management Solutions in Cloud Computing. In *Continued Rise of the Cloud: Advances and Trends in Cloud Computing*, Z. Mahmood, Ed. Springer New York, July 2014, pp. 235–263.
- [52] NASKOS, A.; GOUNARIS, A.; SIOUTAS, S. Cloud Elasticity: A Survey. In *Algorithmic Aspects of Cloud Computing (ALGO CLOUD 2015)*, I. Karydis, S. Sioutas, P. Triantafyllou, and D. Tsoumakos, Eds., vol. 9511, LNCS. Springer Berlin, Germany, July 2016, p. 151–167.
- [53] NICODEMUS, C. H.; BOERES, C.; REBELLO, V. E. F. Managing vertical memory elasticity in containers. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)* (2020), pp. 132–142.
- [54] ORACLE CORPORATION. Oracle VirtualBox. <https://www.virtualbox.org/>, 2017. Accessed on: 01/07/2017.

- [55] ORACLE CORPORATION. Openjdk. <https://openjdk.java.net/>, 2020. Accessed on: 22/06/2020.
- [56] PAHL, C. Containerization and the paas cloud. *IEEE Cloud Computing* 2, 3 (2015), 24–31.
- [57] PARAISO, F.; CHALLITA, S. AND AL-DHURAIBI, Y.; MERLE, P. Model-driven management of docker containers. In *9th IEEE International Conference on Cloud Computing (CLOUD 2016)* (San Francisco, CA, USA, Jun 2016), pp. 718–725.
- [58] PEINL, R.; HOLZSCHUHER, F.; PFITZER, F. Docker cluster management for the cloud - Survey results and own solution. *Journal of Grid Computing* 14 (2016), 265–282.
- [59] PETAZZONI, J. Anatomy of a container: Namespaces, cgroups & some filesystem magic - linuxcon. <https://pt.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon>, 2016. Accessed on: 15/06/2016.
- [60] PHILLIPS, S. J.; DUDÍK, M.; SCHAPIRE, R. E. A maximum entropy approach to species distribution modeling. In *Proc. of the 21st International Conference on Machine Learning* (2004), pp. 83–90.
- [61] PIRAGHAJ, S. F.; DASTJERDI, A. V.; CALHEIROS, R. N.; BUYYA, R. Efficient virtual machine sizing for hosting containers as a service. In *2015 IEEE World Congress on Services (SERVICES)* (2015), IEEE, pp. 31–38.
- [62] PRICE, D.; TUCKER, A. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA* (2004), vol. 4, pp. 241–254.
- [63] QAZI, K. Vertelas-automated user-controlled vertical elasticity in existing commercial clouds. In *2019 4th International Conference on Computing, Communications and Security (ICCCS)* (2019), IEEE, pp. 1–8.
- [64] REDHAT INC. Coreos rkt. <https://coreos.com/rkt/>, 2020. Accessed on: 05/07/2020.
- [65] RODRIGUEZ, M. A.; BUYYA, R. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience* 49, 5 (2019), 698–719.
- [66] ROUSE, M.; BIGELOW, S. Containers as a Service (CaaS). <http://searchitoperations.techtarget.com/definition/Containers-as-a-Service-CaaS>, 2017. Accessed on: 01/07/2017.
- [67] SAWAMURA, R.; BOERES, C.; REBELLO, V. E. F. Evaluating the Impact of Memory Allocation and Swap for Vertical Memory Elasticity in VMs. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (Oct 2015), pp. 186–193.
- [68] SAWAMURA, R.; BOERES, C.; REBELLO, V. E. F. MEC: The memory elasticity controller. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)* (Dec 2016), pp. 111–120.

- [69] SKOPEO CONTRIBUTORS. Skopeo. <https://github.com/containers/skopeo>, 2020. Accessed on: 20/07/2020.
- [70] STOCKWELL, D. The garp modelling system: problems and solutions to automated spatial prediction. *International Journal of Geographical Information Science* 13, 2 (1999), 143–158.
- [71] SU, Y.; LIAO, X.; JIN, H.; BELL, T. Data hiding in virtual machine disk images. In *IEEE International Conference on Computer and Information Technology (CIT)* (2010), pp. 2278–2283.
- [72] THE FREEBSD COMMUNITY. Jails. <https://docs.freebsd.org/pt-br/books/handbook/jails/>, 2020. Accessed on: 27/02/2020.
- [73] VALENCIA, J. Combining VM preemption schemes to improve vertical memory elasticity scheduling in clouds. Master’s thesis, Pós Graduação em Computação, Instituto de Computação, Universidade Federal Fluminense, 2018.
- [74] VALENCIA, J.; BOERES, C.; REBELLO, V. E. F. Combining VM preemption schemes to improve vertical memory elasticity scheduling in clouds. In *IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)* (2018), IEEE Computer Society, pp. 53–62.
- [75] VAUGHAN-NICHOLS, S. J. New approach to virtualization is a lightweight. *Computer* 39, 11 (2006).
- [76] VIRTUOZZO INTERNATIONAL GMBH. Openvz. <https://openvz.org>, 2020. Accessed on: 05/07/2020.
- [77] VMWARE INC. VMware workstation player. <https://www.vmware.com/br/products/workstation-player.html>.
- [78] VMWARE INC. vsphere e vsphere with operations management. <http://www.vmware.com/br/products/vsphere.html>, 2016. Accessed on: 10/08/2016.
- [79] XAVIER, M. G.; NEVES, M. V.; ROSSI, F. D.; FERRETO, T. C.; LANGE, T.; DE ROSE, C. A. F. Performance evaluation of container-based virtualization for high performance computing environments. In *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2013), pp. 233–240.
- [80] ZARCO-GONZÁLEZ, M. M.; MONROY-VILCHIS, O.; ALANÍZ, J. Spatial model of livestock predation by jaguar and puma in Mexico: Conservation planning. *Biological Conservation* 159 (2013), 80 – 87.
- [81] ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications* 1, 1 (2010), 7–18.

APPENDIX A – Container versus Virtual Machines: A Case Study

In order to compare the performance of containerization with the classic virtualization technology, we used as a case study, an important scientific biodiversity tool, called *Openmodeller* (OM) [24], developed by the Environmental Information Reference Center (CRIA) together with other national and international partners, and widespread in the biological and ecological community [38] [80]. The OM is responsible for modeling ecological niches using a large amount of data to be processed and shared between the different instances of the application, during a scientific experiment.

The modeling of ecological niches - ENM (*Ecological Niche Modeling*) is a procedure for determining the extent of the geographical distribution of species, predicting the distribution of species in different geographical and climatic contexts, as well as studying other aspects of evolution and ecology . This modeling has been used in several situations, such as: searching for rare or endangered species; identification of suitable regions for the (re)introduction of species; forecasting the impact of climate change on biodiversity; definition and assessment of protected areas; preventing the spread of invasive species, among other applications.

Openmodeller can be considered a fundamental application for several countries, especially Brazil, to comply with their agreements with the United Nations regarding the Earth's biodiversity by 2020. We are providing a cloud ENM service to the scientific community.

In our experiments, we performed the three stages of OM sequentially:

- Modeling (om_model): responsible for generating distribution models from an XML request file, containing the information necessary for their execution (modeling algorithm, set of species location points and environmental layers) .

- Test (`om_test`): evaluation of the distribution model generated in the previous step, checking if the entry points have the same spatial references as the points generated in the modeling.
- Projection (`om_proj`): uses the model generated in the modeling stage and the *templates* used to generate a distribution map, storing its result in an image file, to facilitate the visualization of the results obtained .

For the modeling stage, OM has several algorithms with different functionalities. In our experiments, we use the following algorithms:

- GARP-BS [70]: an implementation of a genetic algorithm for the creation of ecological niche models.
- MAXENT [60]: a machine learning algorithm to calculate the probability of an epistemic distribution.
- ENVDIST: a genetic algorithm based on metrics of environmental dissimilarities.

For the tests, 32 GB of data were made available for the environmental layers, information about the environment in which the species will be projected, in the OM. These data were shared through the NFS version 3 protocol, between the server and the VMs, while in the containers, we use a direct mapping mechanism of local directories, also between the server and the containers. The test host is running CentOS Linux 7.7 with kernel 4.20.11, has two Intel Xeon X5650 (6 cores at 2.67GHz) with hyperthreading enabled, 24 GB of DDR3 RAM memory, 8 GB of Swap memory and 2 TB of disk space.

Over the next few sections, we will demonstrate here some experiments that we have carried out so far to support the use of containers as the basis for a computational cloud aimed at high performance scientific applications, evaluating from the availability of the service, to a performance comparison between containers, virtual machines and real machines, analyzing the results and indicating the paths we followed in this work.

A.1 Availability test

In this first experiment, we compare the times of creating a virtual machine and a container, to evaluate the time needed to provision a system for the execution of a user's

application. For the creation of the virtual machine, we use the KVM **virt-clone** command and consider the cloning time as the VM creation time. We opted for cloning the VM instead of creating a new VM, due to the time required for the OS installation to take time, which would make the test uneven, since the container uses pre-defined images of the OS for its creation. For containers, we use the mechanism for the direct execution of a command within it. This method was used because we wanted the necessary time from creation to the execution of a common command, such as the directory listing on Linux (`ls` command), and its finalization (closing the container).

Ten VM clones and ten container executions were performed in Docker and we took an average of the times obtained using the Linux *time* command. The average time obtained in the cloning of the VM was 29.95 seconds, with a standard deviation of 8.67, while the average time for creation, execution of the command, and completion of the container was 2.72 seconds with a standard deviation. 0.31.

Analyzing the times obtained, we can see that a container will be available to the user in an 1100% faster time, approximately. This is due to the large disk size of an VM, as the size of its disk file (equivalent to the HD of the VM), depends on the size that the OS will occupy, in addition to the files of the users and / or applications made available and when cloning such VM, we made a copy of the disk file, generating a new configuration file, as some identifiers are unique among VMs, such as the network interface (NIC), where a new MAC address is defined for the new VM. The larger the disk of a VM, the longer the time for cloning.

In the case of containers, a pre-created image based on an OS is used (in our case, a Linux CentOS) equal to that of the server and a new layer (in the case of Docker) is created, to store the modifications made in the container after creation. When the container is created for the first time, the virtualization system performs a *download* of the image and stores it internally on the server, generating an image in *cache*, which speeds up the creation of containers based on the same Image. We do not consider this first creation because its time is very different from the creation of the container with the image already in *cache*. This first creation in our test, was approximately 34 seconds, for the image of the standard CentOS system from the Docker repository.

Thus, we can say that a container will be available to the user in a much shorter time than a VM, however lean, which makes them more acceptable for the development of a computing cloud environment, where the time needed for the provisioning is relatively important for service quality.

A.2 Performance Test

In this experiment, the objective was to evaluate the performance of virtual environments of VMs (KVM) and containers (Docker) in comparison to direct execution on the server for scientific applications. We used OM and the three algorithms mentioned earlier: GARP, MAXENT, and ENVDIST. An instance of OM was run on each of the virtualization technologies and on the server. We repeated each test ten times and the average time of each step was recorded for the three algorithms and their respective standard deviations.

Table A.1: GARP: Average time and standard deviation of the three stages

GARP (time in seconds)			
	Modeling	Testing	Projection
Physical Machine	655,128 \pm 2.50	54,186 \pm 0.96	291,429 \pm 32.65
Virtual Machine	690,563 \pm 3.07	81,220 \pm 2.43	313,727 \pm 28.60
Container	654,305 \pm 2.43	53,686 \pm 0.65	291,967 \pm 28.95

In the table A.1, we can see that, for the GARP algorithm, the time obtained in the container was slightly better (in milliseconds) than the time obtained directly on the server, for the modeling and testing steps, while in VM this time was 5.4% and almost 50% worse, for the same stages, respectively. The modeling step has a lot of intensive processing (for needing and processing countless entry points and data) and the GARP algorithm is long lasting (more than 10 minutes), so the loss generated by the *hardware* emulation was not so harmful, considering the virtualized environment, but when the process is short, like the test step, this loss is noticeable.

In the projection stage, the time obtained by the container was very close to that obtained on the server, also in the milliseconds, while the virtual machine showed a loss of approximately 7.7%. At this point, the algorithm spends much of the time generating an image file that demonstrates the results obtained in the modeling, requiring the reading of the entry points and other data for this generation. In addition, the image has a size equivalent to the number of entry points, the more points, the larger the file. Therefore, the VM presented worse results than the container for this algorithm.

Table A.2: MAXENT: Average time and standard deviation for the three steps

MAXENT (time in seconds)			
	Modeling	Testing	Projection
Physical Machine	61,557 \pm 0.52	53,116 \pm 0.75	35,505 \pm 0.18
Virtual Machine	90,785 \pm 3.02	81,156 \pm 2.85	40,135 \pm 0.27
Container	61,490 \pm 0.60	52,907 \pm 0.44	35,835 \pm 0.48

Table A.3: ENVDIST: Average time and standard deviation of the three steps

ENVDIST (time in seconds)			
	Modeling	Testing	Projection
Physical Machine	52,561 \pm 0.07	53,080 \pm 0.71	30,968 \pm 0.42
Virtual Machine	81,589 \pm 3.17	78,939 \pm 2.87	34,499 \pm 0.40
Container	52,695 \pm 0.21	52,607 \pm 0.11	31,081 \pm 0.27

In the tables A.2 and A.3, of the MAXENT and ENVDIST algorithms, respectively, the difference between containers and VMs was more noticeable, being around 55.2% worse than the server, in the modeling step of the ENVDIST algorithm and 52.8% for the MAXENT algorithm step, algorithms considered less demanding in terms of processing, and the same processes being considered short, on average of 1 minute, considering the time of the physical machine. This difference was small, for the projection stage, where the VM still presents the worst result, as observed in the GARP algorithm of the table A.1.

Analyzing the results obtained in this test, it was observed that for algorithms considered fast or short, the loss caused by the emulation of *hardware*, in the case of VMs, was very harmful, being this greater than 50%, in several cases, while the container presented results very close or even a little better than on the server, demonstrating that with containers we can have some of the main characteristics of virtualization technologies with a performance close to the real (directly on the machine).

A.3 Scalability Test

Most of the high performance scientific applications used in scientific research have the possibility of processing large volumes of data in parallel, in *clusters* and computational clouds, to reduce their execution time, the analysis of scalability was idealized for virtualization platforms compared to the physical environment (server).

The GARP algorithm of OM was used, as it is the algorithm that consumes more computational resources than the three tested algorithms and because it is the algorithm with the longest execution. The experiment was repeated 10 times for 4, 8, 12, 16, 20, and 24 VMs or containers, simulating a parallel environment with a high workload. Each VM or container executes a complete OM instance (modeling, testing and projection), on the same shared data set (the environmental layers). This test aims to simulate the use of several users simultaneously in a shared environment. OpenMPI was used to initialize

OM in all containers or VM simultaneously.

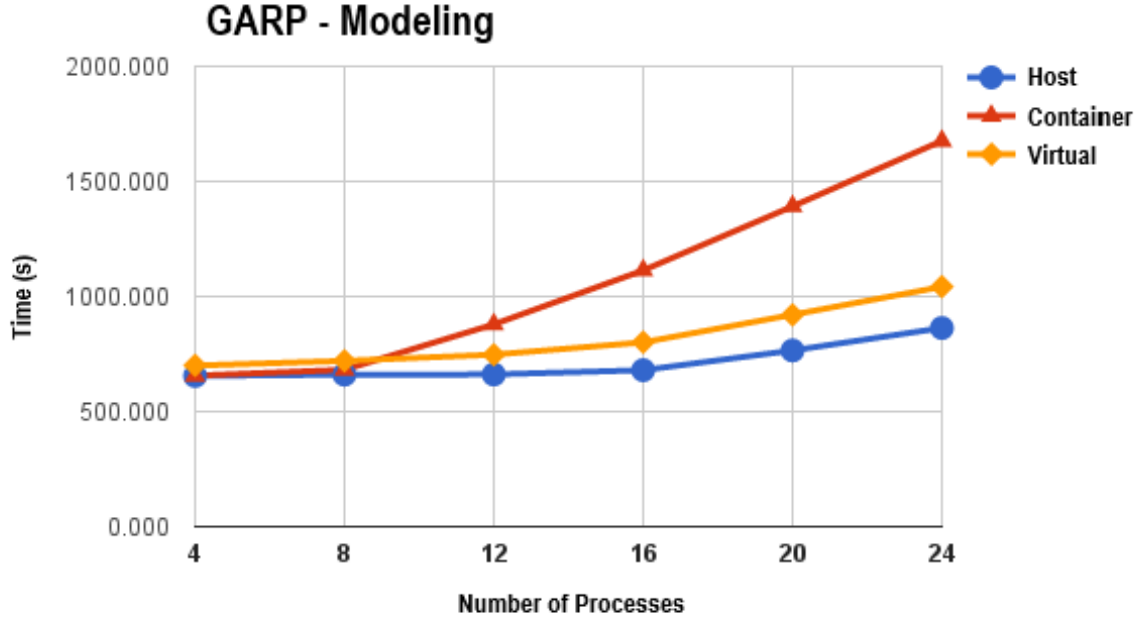


Figure A.1: GARP scalability for the modeling step

In the graph A.1, it is observed that the execution of the containers presented a performance similar to the execution of the physical machine, while the number of processes/containers was less than the number of physical cores available on the server (8 processes) however, when reaching the limit of physical cores (12 cores) and starting to overload them (from 12 cores), the execution in the containers suffered a significant loss, being up to 94% worse than the execution in the physical machine and even 60% worse compared to execution in the VM (for 24 processes). This is because, during modeling, the GARP algorithm performs several iterations of its genetic algorithm to obtain several sets of results, selecting the best ones as output, which requires many disk access operations to read the environmental layers, which are shared by all containers using a Docker directory mapping function. In VMs, these environmental layers are shared by the host through the NFS protocol, and do not generate an overhead on disk access. The difference in performance of the VMs, compared to the physical machine, involves yet another processing overhead, common for this virtualization technology, due to several factors, such as the abstraction of the host hardware for the VM, the interpretation of system instructions between VM and host, among others, as discussed in [79] and in [35].

In Figure A.2, we can see that the performance in the container was better than in

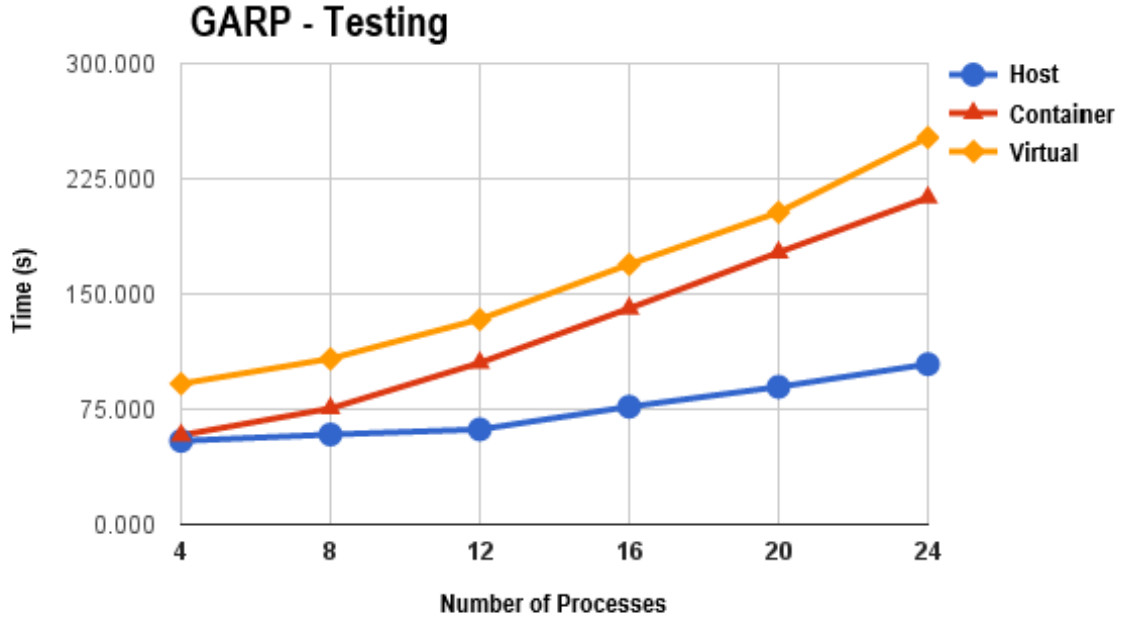


Figure A.2: GARP scalability for the test stage

the VM, however it was worse (reaching twice) compared to the execution on the physical machine. In the Test phase, there is a check if the occurrence points, used for the generation of the model, were properly referenced in the modeling result and the same environmental layers used in the modeling are used. There are fewer read-on-disk operations, so the loss of performance involves more processing, as there was CPU competition between the containers, by not using the virtual cores available through *hyperthreading*. VMs, on the other hand, uses virtual cores, but suffers a greater loss of performance than containers. This phenomenon can be observed using a Linux system command called `htop` [50] that graphically displays the use of processing resources (physical and virtual cores) in addition to the consumption of memory and processes being executed.

Figure A.3 shows the repetition of the behavior observed in Figure A.1, because to make the projection, OpenModeller needs to read a *template*, which is a file that contains information like the size of the cells and the spatial references used to generate the distribution map, corresponding to the environmental layer used in the modeling. Subsequently, an image has to be generated, which requires concurrent reading operations on the same file (the environmental layers and *templates*) and the writing of the process image file on each container to disk.

We can conclude that Docker container management was not able to efficiently handle

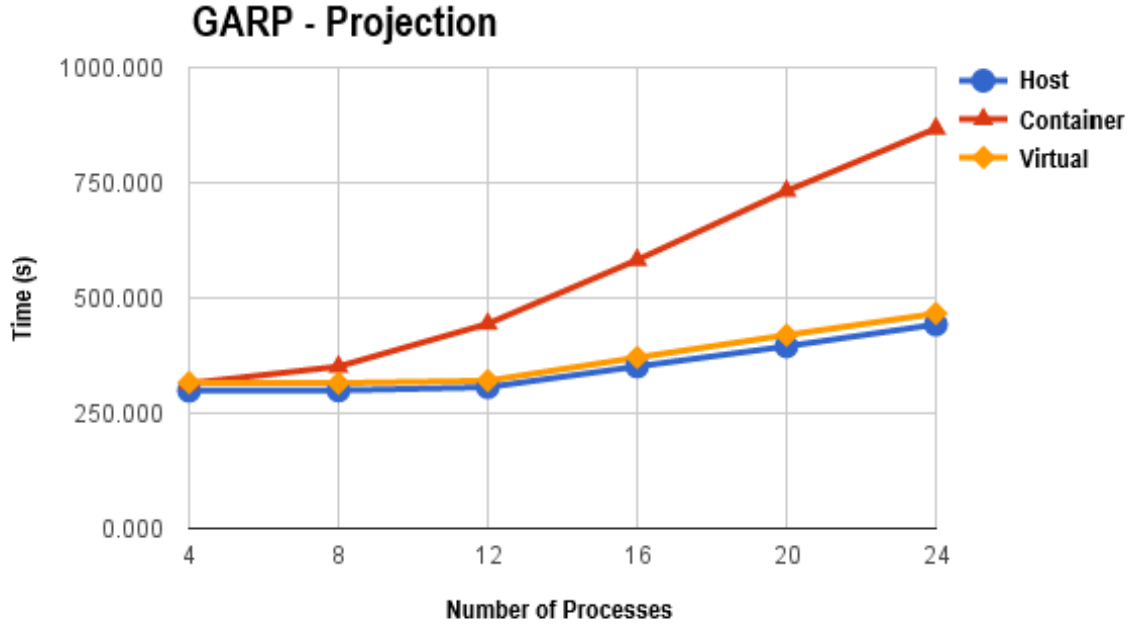


Figure A.3: GARP scalability for the projection stage

the multiple concurrent disk access operations over shared data, as well as not using the virtual cores available by *hyperthreading*, generating competition for processing between processes, while the execution in the VM suffered a loss of performance already pointed out by other works, for applications with intensive processing.

A.3.1 CPU Allocation Test

After carrying out the scalability tests using OM, questions arose about the reasons that led the containers to present bad results, when the number of containers was equal to or greater than the number of *cores* of the server, even knowing that in smaller instances the result was always close to that obtained on the physical machine.

It was initially considered that the problem could have occurred due to the sharing of the database (*layers*), however, when analyzing the consumption of resources during execution, using the tool **htop** present in Linux, it was concluded that the problem was the misuse of the available *cores*, where the virtual *cores* provided by the *hyperthreading* technology of the processors on the server was not used.

To remove this doubt, the scalability test was repeated, setting a *core* for each running container. To perform this operation, Docker was replaced by the virtualization system

for LXC [14] containers. Cgroups [47] was used to fix the *core*. The GARP algorithm of OM was used and we analyzed the modeling step, one of the heaviest OM steps and where the largest consumption of CPU resources is concentrated and for the longest time. The test was repeated ten times and the average time obtained in each run was used.

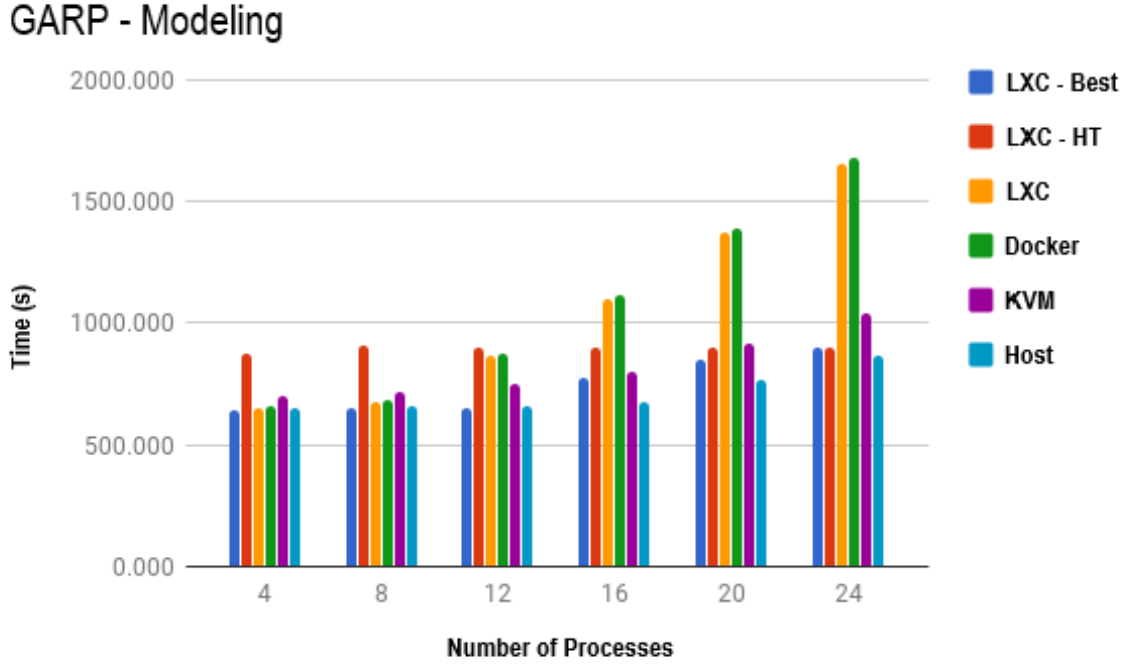


Figure A.4: GARP scalability, for the modeling stage, with fixation of *cores*.

In Figure A.4, we have *LXC-Best*, which corresponds to the setting of a *core* per container, first allocating the physical *cores* (the first 12) and then virtual ones (from 13 to 24). The *LXC-HT* corresponds to the equal allocation between physical and virtual *cores*, that is, in each column, half of the *cores* are physical and the other half their virtual pairs. In addition, we used the standard execution of LXC and compared it with the data obtained in the test of Figure A.1.

Analyzing the graph of Figure A.4, we can conclude that fixing the *cores*, using the physical *cores* first, is the most effective way to execute applications inside the containers, presenting results very close to those obtained in the physical machine. We can also observe that the *LXC-HT* presented a constant behavior, regardless of the number of instances being executed, for using in a balanced way, physical and virtual *cores*.

Considering the result obtained with the fixation of the *cores*, we believe that the problems obtained previously are related to how the virtualization systems are seeing the processors, since it was possible to obtain satisfactory results with the fixation, while the

standard execution of the LXC presented close to Docker, which is slightly better.

To evaluate the influence of *hyperthread* and a multiprocessor architecture *multicore* on the functioning of scientific applications in virtualized environments, we use the results obtained with twelve processes, which is equivalent to the number of physical *cores* available on the server used for the tests. LXC was executed with *hyperthread* disabled, *LXC without HT*, and compared with the results already presented.

GARP - Modeling (12 cores)

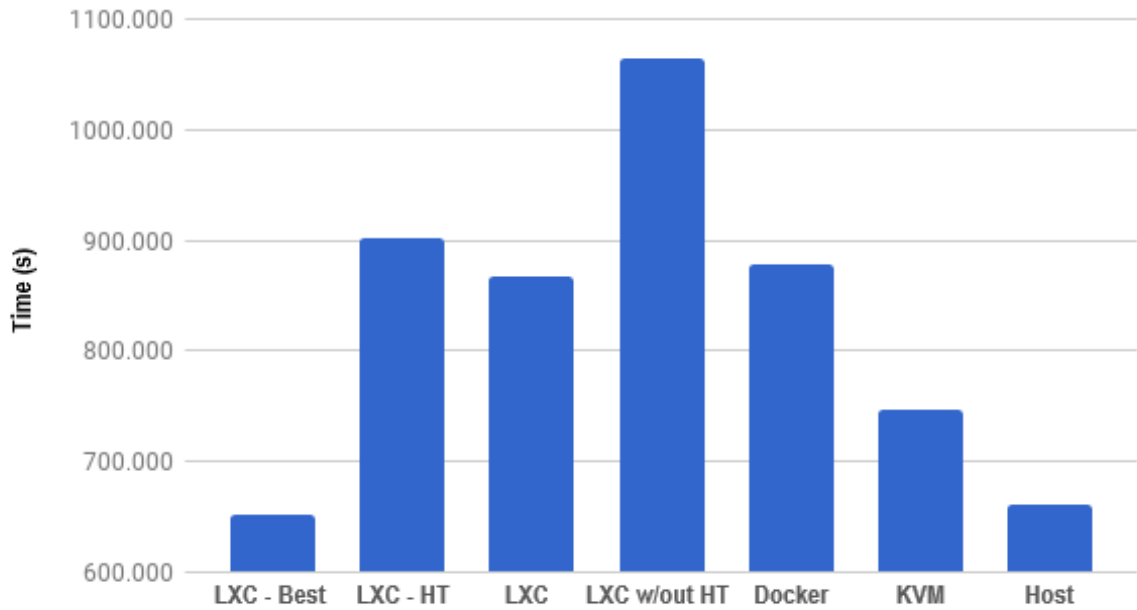


Figure A.5: Evaluation of the GARP algorithm regarding the use of *hyperthreading*.

By looking at Figure A.5, we can conclude that *hyperthread* has some influence on the behavior of containers, since the time obtained with it disabled was almost twice the best result (LXC-Best). In addition to the influence of *hyperthread* it is possible to verify that the virtualization system is not "seeing", on its own, all the available *cores*, being necessary to fix them manually.

Using Htop, to accompany the test, it was observed that the container virtualization systems, both LXC and Docker, did not use the *cores* of the second processor, which can be confirmed by the result obtained in the column *LXC without HT* of the graph A.5.

We can conclude that container virtualization really presents better results than VMs, with a performance close to that of the physical machine, which makes it an excellent candidate to be used in the creation of a cloud service. It is also possible to conclude the need for more intelligent systems, capable of reserving computational resources, for the

management of the cloud environment, since to obtain a performance close to the real, we need to control the resources and maximize their use.

APPENDIX B – Memory Consumption Trends

Using Cgroups, VEMoC is able to access several metrics that help capture the behavior of a container and the applications running inside it. However, since duration of scheduling cycles can vary, in order to obtain accurate predictions VEMoC calculates the rate of change of key metrics to make the decision less time sensitive. A further hypothesis is that approach leads to a simpler management control function than using the actual metric values. This appendix aims to present the rationale for the simple logic, and thus fast execution, of Algorithm 3 used in Phase 2 to classify containers (see Section 5.3.2).

VEMoC analyses the following five Cgroup memory metrics to implement vertical memory elasticity:

- *PgIn*: the number of memory pages created in the main memory, be it a new page or a page brought from the disk (swap-in).
- *PgOut*: the number of pages sent to disk (swap-out) or pages freed from main memory.
- Major Page Faults (*MPF*): the number of times a page was brought back from swap to the main memory.
- Memory Usage (*MU*): indicates whether the amount of memory used increased, decreased or remained stable.
- Swap Usage (*SU*): the same as *MU* but for the swap. We also classify the type of swap operation that is taking place: only swap-in (SI), only swap-out (SO), or both swap-in and out (SISO).

With these metrics, VEMoC classifies the containers behavior into one of three distinct groups:

- **RISING:** Containers that increase their memory consumption, during the period of time;
- **FALLING:** Containers that have voluntarily reduced their RAM consumption during the period of time;
- **STABLE:** These are containers that do not fall into the two previous trends since they had no page major faults occurring during the period of time, and their memory and swap consumption was around zero, unless they were being forced to swap out memory by VEMoC itself.

How VEMoC comes to this conclusion is motivated by Table B.1, which presents all possible combinations (108 in total) of the previous memory metrics. However, after careful analysis, only 20 of these combinations can actually occur in practice (lines in red with an Id).

Table B.1: Verification of all possible combinations for values of the adopted Cgroups Memory Metrics.

Id	PgIn	Vs	PgOut	MPF	MU	SU	Swap Use	Comment	Action
1a	Pi = Po = 0			= 0	= 0	= 0	None	All ops in mem	STABLE
	Pi = Po = 0			= 0	= 0	> 0		Not possible	
	Pi = Po = 0			= 0	= 0	< 0		Not possible	
	Pi = Po = 0			= 0	> 0	= 0		Not possible	
	Pi = Po = 0			= 0	> 0	> 0		Not possible	
	Pi = Po = 0			= 0	> 0	< 0		Not possible	
	Pi = Po = 0			= 0	< 0	= 0		Not possible	
	Pi = Po = 0			= 0	< 0	> 0		Not possible	
	Pi = Po = 0			= 0	< 0	< 0		Not possible	
	Pi = Po = 0			= 0	< 0	= 0		Not possible	
	Pi = Po = 0			= 0	= 0	> 0		Not possible	
	Pi = Po = 0			= 0	= 0	< 0		Not possible	
	Pi = Po = 0			= 0	> 0	= 0		Not possible	
	Pi = Po = 0			= 0	> 0	> 0		Not possible	
	Pi = Po = 0			= 0	> 0	< 0		Not possible	
	Pi = Po = 0			= 0	< 0	= 0		Not possible	
	Pi = Po = 0			= 0	< 0	> 0		Not possible	
	Pi = Po = 0			= 0	< 0	< 0		Not possible	
	Pi = Po = 0			= 0	< 0	= 0		Not possible	
	Pi = Po = 0			= 0	< 0	< 0		Not possible	
	Pi = Po = 0			= 0	< 0	> 0		Not possible	
	Pi = Po = 0			= 0	< 0	< 0		Not possible	
	Pi = Po = 0			= 0	= 0	= 0		Not possible	
	Pi > Po = 0			= 0	= 0	= 0		Not possible	

Table B.1 continued from previous page

Id	PgIn	Vs	PgOut	MPF	MU	SU	Swap Use	Comment	Action
		Pi > Po = 0		= 0	= 0	> 0		Not possible	
		Pi > Po = 0		= 0	= 0	< 0		Not possible	
2a		Pi > Po = 0		= 0	> 0	= 0	None	Increasing MU	RISING
		Pi > Po = 0		= 0	> 0	> 0		Not possible	
		Pi > Po = 0		= 0	> 0	< 0		Not possible	
		Pi > Po = 0		= 0	< 0	= 0		Not possible	
		Pi > Po = 0		= 0	< 0	> 0		Not possible	
		Pi > Po = 0		= 0	< 0	< 0		Not possible	
		Pi > Po = 0		> 0	= 0	= 0		Not possible	
		Pi > Po = 0		> 0	= 0	> 0		Not possible	
		Pi > Po = 0		> 0	= 0	< 0		Not possible	
		Pi > Po = 0		> 0	> 0	= 0		Not possible	
		Pi > Po = 0		> 0	> 0	> 0		Not possible	
2b		Pi > Po = 0		> 0	> 0	< 0	SI		RISING
		Pi > Po = 0		> 0	< 0	= 0		Not possible	
		Pi > Po = 0		> 0	< 0	> 0		Not possible	
		Pi > Po = 0		> 0	< 0	< 0		Not possible	
		0 = Pi < Po		= 0	= 0	= 0		Not possible	
		0 = Pi < Po		= 0	= 0	> 0		Not possible	
		0 = Pi < Po		= 0	= 0	< 0		Not possible	

Table B.1 continued from previous page

Id	PgIn	Vs	PgOut	MPF	MU	SU	Swap Use	Comment	Action
		$0 = \text{Pi} < \text{Po}$		$= 0$	> 0	$= 0$		Not possible	
		$0 = \text{Pi} < \text{Po}$		$= 0$	> 0	> 0		Not possible	
		$0 = \text{Pi} < \text{Po}$		$= 0$	> 0	< 0		Not possible	
3a		$0 = \text{Pi} < \text{Po}$		$= 0$	< 0	$= 0$	None	Application freeing up memory	FALLING
3b		$0 = \text{Pi} < \text{Po}$		$= 0$	< 0	> 0	SO	Forced inactive memory to swap	“STABLE” FALLING?
		$0 = \text{Pi} < \text{Po}$		$= 0$	< 0	< 0		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	$= 0$	$= 0$		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	$= 0$	> 0		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	$= 0$	< 0		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	> 0	$= 0$		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	> 0	> 0		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	> 0	< 0		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	< 0	$= 0$		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	< 0	> 0		Not possible	
		$0 = \text{Pi} < \text{Po}$		> 0	< 0	< 0		Not possible	
4a		$\text{Pi} = \text{Po} < 0$		$= 0$	$= 0$	$= 0$	None	App freeing up as much as it uses. Container is technically STABLE	Given Pi, RISING
4b		$\text{Pi} = \text{Po} < 0$		$= 0$	$= 0$	> 0	SO	Must be due to CML limiting MU	RISING
		$\text{Pi} = \text{Po} < 0$		$= 0$	$= 0$	< 0		Not possible	

Table B.1 continued from previous page

Id	PgIn	Vs	PgOut	MPF	MU	SU	Swap Use	Comment	Action
	Pi = Po <> 0			= 0	> 0	= 0		Not possible	
	Pi = Po <> 0			= 0	> 0	> 0		Not possible	
	Pi = Po <> 0			= 0	> 0	< 0		Not possible	
	Pi = Po <> 0			= 0	< 0	= 0		Not possible	
	Pi = Po <> 0			= 0	< 0	> 0		Not possible	
	Pi = Po <> 0			= 0	< 0	< 0		Not possible	
4c	Pi = Po <> 0			> 0	= 0	= 0	SISO		RISING
4d	Pi = Po <> 0			> 0	= 0	> 0	SO	With some SI but less, looks stable	RISING
4e	Pi = Po <> 0			> 0	= 0	< 0	SI	App frees up some too	RISING
	Pi = Po <> 0			> 0	> 0	= 0		Not possible	
	Pi = Po <> 0			> 0	> 0	> 0		Not possible	
	Pi = Po <> 0			> 0	> 0	< 0		Not possible	
	Pi = Po <> 0			> 0	< 0	= 0		Not possible	
	Pi = Po <> 0			> 0	< 0	> 0		Not possible	
	Pi = Po <> 0			> 0	< 0	< 0		Not possible	
	Pi > Po <> 0			= 0	= 0	= 0		Not possible	
	Pi > Po <> 0			= 0	= 0	> 0		Not possible	
	Pi > Po <> 0			= 0	= 0	< 0		Not possible	
5a	Pi > Po <> 0			= 0	> 0	= 0	None	App frees mem	RISING
5b	Pi > Po <> 0			= 0	> 0	> 0	SO		RISING

Table B.1 continued from previous page

Id	PgIn	Vs	PgOut	MPF	MU	SU	Swap Use	Comment	Action
	Pi > Po	<	>	= 0	> 0	< 0		Not possible	
	Pi > Po	<	>	= 0	< 0	= 0		Not possible	
	Pi > Po	<	>	= 0	< 0	> 0		Not possible	
	Pi > Po	<	>	= 0	< 0	< 0		Not possible	
	Pi > Po	<	>	> 0	= 0	= 0		Not possible	
	Pi > Po	<	>	> 0	= 0	> 0		Not possible	
	Pi > Po	<	>	> 0	= 0	< 0		Not possible	
5c	Pi > Po	<	>	> 0	> 0	= 0	SISO		RISING
5d	Pi > Po	<	>	> 0	> 0	> 0	SO	With some but less SI	RISING
5e	Pi > Po	<	>	> 0	> 0	< 0	SI	With some SO or App frees mem	RISING
	Pi > Po	<	>	> 0	< 0	= 0		Not possible	
	Pi > Po	<	>	> 0	< 0	> 0		Not possible	
	Pi > Po	<	>	> 0	< 0	< 0		Not possible	
	0 < Pi	<	Po	= 0	= 0	= 0		Not possible	
	0 < Pi	<	Po	= 0	= 0	> 0		Not possible	
	0 < Pi	<	Po	= 0	= 0	< 0		Not possible	
	0 < Pi	<	Po	= 0	> 0	= 0		Not possible	
	0 < Pi	<	Po	= 0	> 0	> 0		Not possible	
	0 < Pi	<	Po	= 0	> 0	< 0		Not possible	

Table B.1 continued from previous page

Id	PgIn	Vs	PgOut	MPF	MU	SU	Swap Use	Comment	Action
6a	0	<>	Pi < Po	= 0	< 0	= 0	None	App frees mem, currently classed as RISING but no swap	FALLING
6b	0	<>	Pi < Po	= 0	< 0	> 0	SO	Swap out more than Pi. Or could be forced swap. FALLING or STABLE	STABLE
	0	<>	Pi < Po	= 0	< 0	< 0		Not possible	
	0	<>	Pi < Po	> 0	= 0	= 0		Not possible	
	0	<>	Pi < Po	> 0	= 0	> 0		Not possible	
	0	<>	Pi < Po	> 0	= 0	< 0		Not possible	
	0	<>	Pi < Po	> 0	> 0	= 0		Not possible	
	0	<>	Pi < Po	> 0	> 0	> 0		Not possible	
	0	<>	Pi < Po	> 0	> 0	< 0		Not possible	
6c	0	<>	Pi < Po	> 0	< 0	= 0	SISO	Also App frees mem	RISING
6d	0	<>	Pi < Po	> 0	< 0	> 0	SO	With some but less SI	RISING
6e	0	<>	Pi < Po	> 0	< 0	< 0	SI	App frees mem perhaps some SO	RISING

A few observations with regards to Table B.1:

- From Algorithm 2 (Section 5.3.1), note that the page in and page out values (Pi and Po) VEMoC calculates are currently the average page counts over its long interval, which are then normalised to a per second rate before have their precision is scaled down to cover the 33 page variance observed in the monitored metrics $PgIn$ and $PgOut$;
- Similarly MU and SU are the rate of change values in memory and swap usage, respectively, measured in memory pages (of 4096 bytes) per second, i.e., $MU_i \gg 12$ and $SU_i \gg 12$;
- Given condition is in blue, the combination or condition in light red is not possible;
- Remember tendencies are considered over a number of time intervals to reduce the impact of noise.

Given the viable combinations, it is relatively simple to then optimise to determine the appropriate action. For example, when Major Page Faults occur ($MPF > 0$), the container state ($State_i$) should be *RISING*, thus covering Ids 2b, 4c, 4d, 4e, 5c, 5d, 5e, 6c, 6d, and 6e. This optimization leads to the *MemClassification()* function described in Algorithm 3 (see Section 5.3.2) used in Phase 2 of VEMoC to define the Memory Consumption Trend of a container.