UNIVERSIDADE FEDERAL FLUMINENSE

JOSÉ LUIS VALENCIA GUTIÉRREZ

Combining VM Preemption Schemes to Improve Vertical Memory Elasticity Scheduling in Clouds

NITERÓI

2018

UNIVERSIDADE FEDERAL FLUMINENSE

JOSÉ LUIS VALENCIA GUTIÉRREZ

Combining VM Preemption Schemes to Improve Vertical Memory Elasticity Scheduling in Clouds

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

Advisor:
Vinod Rebello

Co-advisor:
Cristina Boeres

NITERÓI

2018

Ficha catalográfica automática - SDC/BEE

V152c Valencia Gutierrez, José Luis
Combining VM Preemption Schemes to Improve Vertical Memory Elasticity Scheduling in Clouds / José Luis Valencia Gutierrez ; Eugene Francis Vinod Rebello, orientador ; Maria Cristina Silva Boeres, coorientadora. Niterói, 2018.
67 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2018.

DOI: http://dx.doi.org/10.22409/PGC.2018.m.06336017729

1. Computação em nuvem. 2. Gerenciamento de memória (Computação) . 3. Elasticidade Vertical. 4. Preempção. 5. Produção intelectual. I. Título II. Vinod Rebello,Eugene Francis, orientador. III. Boeres, Maria Cristina Silva, coorientadora. IV. Universidade Federal Fluminense. Escola de Engenharia.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

JOSÉ LUIS VALENCIA GUTIÉRREZ

Combining VM Preemption Schemes to Improve Vertical Memory Elasticity Scheduling in Clouds

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

Approved in June 2018 by:

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

Prof. Maria Cristina Silva Boeres Ph.D. - Co-advisor, UFF

Prof. Lúcia Maria de Assumpção Drummond D.Sc., UFF

Prof. Alexandre Sztajnberg D.Sc., UERJ

Niterói

2018

To my beloved mother.

Acknowledgements

First and foremost I have to thank my family for their love and support, to my mother and brother, who always believe in me and give me strength to continue.

I am very grateful with my advisers Vinod Rebello and Cristina Boeres for their guidance and support throughout this study, inspiring me to be a better scientist. It has been an invaluable experience to work with and learn from them.

I would like to thank my second family, the people from Nuestro Hogar, for their encourage and help.

Finally, to my friends, for their support and encourage. I cannot list all the names here, but you are always on my mind.

Resumo

Consolidação de servidores e elasticidade de recursos estão entre as dois características mais importantes do gerenciamento de recursos na computação em nuvem. O primeiro tenta melhorar a utilização de recursos reduzindo ou minimizando o número de servidores para uma determinada carga de trabalho. O último tem como objetivo obter ganhos de utilização, tentando explorar as demandas variáveis no tempo dos aplicativos em nuvem durante a execução. Uma das duas formas de elasticidade é frequentemente adotada. Enquanto a elasticidade horizontal se preocupa com a aquisição e liberação de nós computacionais de acordo com a demanda, a elasticidade vertical se concentra na distribuição dos recursos de um nó entre suas máquinas virtuais (MVs) hospedadas ajustando a capacidade dos tipos de recursos alocados a cada MV individual de acordo com necessidades de sua respectiva aplicação. No caso da elasticidade vertical, quando recursos insuficientes estão disponíveis para alocar para uma MV, o desempenho da aplicação sofrerá degradação. Para aplicações on-line, a única alternativa é tentar migrar a MV para outro servidor com capacidade suficiente para hospedar a MV. Por outro lado, ao executar tarefas em lote, a MV restrita de recursos também pode ser suspensa ou salva em disco e revivida em qualquer outro servidor (por meio da migração de MV para outro host) ou no mesmo host, quando os recursos se tornem disponíveis. Como a disponibilidade de memória pode ter uma influência significativa no desempenho do aplicativo e no rendimento do sistema, este trabalho investiga a integração e o impacto da migração da MV como parte de uma estratégia de escalonamento no contexto da elasticidade vertical de memória. Avaliamos uma versão aprimorada e estendida do MEC (Memory Elasticity Controller) e o uso de várias instâncias do MEC sob a orientação do controlador *Memory Elasticity Management in Clouds* (MEMiC) para suportar a execução de aplicações on-line e em lote. As avaliações mostram que a combinação de várias técnicas de preempção pode fornecer melhorias de desempenho e utilização em comparação com as abordagens baseadas em somente migração.

Palavras-chave: Computação em nuvem, Elasticidade vertical, Memoria, Escalonamento de maquinas virtuais, Virtualizção.

Abstract

Server consolidation and resource elasticity are among two of the most important resource management features in cloud computing. The former attempts to improve resource utilization by reducing or minimizing the number of servers for a given workload. The latter aims to obtain utilization gains by trying to exploit the time-varying demands of cloud applications during execution. One of two forms of elasticity is often adopted. While horizontal elasticity is concerned with the acquisition and release of computational nodes in accordance with demand, vertical elasticity focuses on the distribution of a node's resources among its hosted virtual machines (VMs) by adjusting the capacity of the resource types allocated to each individual VM in accordance with its respective application's needs. In the case of vertical elasticity, when insufficient resources are available to allocate to a VM, its application's performance will suffer degradation. For on-line applications, the only alternative is to attempt to live-migrate the VM to another server with enough capacity to host the VM. On the other hand, when running batch jobs, the resource constrained VM could also be suspended or saved to disk and revived wherever (through VM migration to another host) or on the same host, when resources become available. Given that memory availability can have a significant influence on application performance and system throughput, this work investigates the integration and impact of VM migration as part of a scheduling strategy in the context of vertical memory elasticity. We evaluate an improved and extended version of the Memory Elasticity Controller (MEC) and the use of multiple MEC instances under the guidance of the *Memory Elasticity Management in Clouds* (MEMiC) Controller to support the execution of both on-line and batch applications. Evaluations show that combining multiple preemption techniques can provide performance and utilization improvements in comparison to live migration-only approaches.

Keywords: Cloud Computing, Vertical Elasticity, Memory, Virtual Machine Scheduling, Virtualization.

List of Figures

List of Tables

List of Acronyms and Abbreviations

VM	:	Virtual Machine;
PM	:	Physical Machine;
OS	:	Operating system;
MEC	:	Memory Elasticity Controller;
MRVM	:	Migrate running VM;
MPVM	:	Migrate paused VM;
MSVM	:	Migrate suspended VM;
SISO	:	Swap-in/Swap-out;

Contents

Chapter 1

Introduction

Cloud computing offers computational resources that are delivered over the Internet and provided as utilities to be used on demand. Clouds aim to rapidly adjust the quantity of resource appropriately to the customer's needs without over- or under-provisioning, and charge each customer for exactly what used. Thanks to these features, cloud computing has enabled small start-ups and companies to use large computational resources at a low cost, allowing these companies to innovate using new technologies such as big data, deep learning and others, without having to deal with the high acquisition, maintenance and operational costs. Cloud computing is generally considered a commodity business, meaning that the competition largely revolves around price, driving it lower continually. According to the International Data Corporation (IDC), worldwide cloud computing revenues are estimated to reach $554 billion in 2021, more than double those of 2016. As a big business, recent years have seen many new cloud providers emerge into the market to compete with the leaders Amazon Web Services (AWS) and Microsoft Azure, such as the likes of IBM Bluemix, Google Cloud, Oracle and Alibaba among others like LocaWeb and Mandic in Brazil.

Advances in virtualization technologies have made it possible to share large physical servers with multiple cores, memory and other resources between different users in a fully isolated manner through the use of Virtual Machines (VMs). Nevertheless, the user or resource management systems still face the problem of having to configure or determine the appropriate amount of resources to allocate so that the application can execute within a previously defined time frame. Furthermore, application requirements frequently vary during their execution and therefore to tackle this problem, cloud providers must provide tools that exploit elasticity transparently.

Elasticity is a key feature that allows the cloud to allocate or release resources on

demand. While the most common form is horizontal elasticity, which consists of being able to add or remove computational nodes being used by a given application, vertical elasticity is the ability to increase or decrease the amount of resources in each individual node that is allocated to an application. Generally, horizontal elasticity is classified as a coarse grained approach, where virtual resource allocations are pre-configured and remain static, while vertical elasticity is more fine-grained, since the sizes of the virtual resource allocations can change dynamically.

In order to avoid harming the execution of their applications, users tend to overestimate the amount of resources required. In fact, it is common practice to request the amount of resources necessary to satisfy the maximum expected demand. For example, many applications have dynamic memory requirements and, in most cases, all of the allocated memory will not be used for at least some part of execution time. In order to try to take advantage of this idle memory, cloud providers often oversubscribe a physical machine by allowing the instantiation of virtual machines with a total peak memory requirement that is larger than the amount of physical memory available. This practice, however, has been shown to impact the performance of the application negatively [22,28], especially when applications are forced to use swap because of a lack of available memory even for short periods during execution.

Vertical memory elasticity, if implemented efficiently, can be used as an effective method to assign memory to VMs while allowing cloud providers to instantiate more VMs in a physical node without the risks of over-provisioning. In previous work [29], the Memory Elasticity Controller (MEC) is proposed as a framework to manage the transitions of a VM's state based on the requirements of other VMs on a single host and to dynamically assign memory to VMs on demand based on their recent memory consumption. MEC's main objective is avoid the use of swap within the VM and on the host at all costs, while allowing applications to use memory elastically. In order to achieve this, that initial MEC version implemented two countermeasures to be used when a host begins to run out of memory and proposed that VM migration be considered in future work as a third action to further improve VM scheduling in this situation.

The work in this dissertation focuses on modifications to MEC to combine the use of three pre-emption options (VM pausing, suspension and migration) in a cloud or edge computing environment with multiple physical machines [34]. In order to identify and take advantage of idle resources in the system, we introduce a new component in charge of: monitoring multiple MEC instances each running on a different host; scheduling new

job requests to them, and; coordinating VM migrations between appropriate hosts when required. Both scheduling and migration decisions are based on resource availability and the system's current and short term predicted demands. This dissertation analyzes the costs and performance impacts of the pre-emption options and different variants of VM migration, to improve the scheduling policies of MEC. By adding this new method to mitigate the impacts of memory shortages, the framework is able to support both on-line and batch jobs, simultaneously.

1.1 Motivation

Cloud providers must seek to make the most efficient use of their available resources by maximizing their utilization and minimizing power consumption while not violating predefined service level agreements. Given that user applications rarely use all of their resource allocation, all of the time, cloud providers often allow the same physical resource to be shared among multiple users at the same time, while maintaining some form of isolation (through the use of virtual machines or containers) between applications. Under this scenario, however, care must be taken to avoid, when possible, any consequent degradation to the execution times of the applications or jobs.

Cluster or resource management systems are administrative software that automatically orchestrate the allocation of jobs to resources, examples include Mesos [10] and Kubernetes [9] often used in container-based cloud platforms [24], OpenNebula for VM-based clouds and Slurm [18] and TORQUE [32] in High Performance Computing (HPC) environments. In the more established area of HPC, in the past, resource managers had to focus on performance and thus allocated each job to a dedicated set of resources in order to avoid *interference*, i.e. performance degradation, caused by applications competing for the same resource. Since this type of allocation policy can lead to significant sub-utilization of resources, resource managers currently tend to allow the sharing of resources. Unfortunately, they often do so without fully taking in to account the possible negative performance impacts of this decision.

Vertical memory elasticity offers more efficient resource utilization by managing the allocation of memory according to the demand while at the same time trying to avoid performance degradation. MEC manages the state of the VMs to prevent significant performance losses, but when the host machine has run out of resources, negative impact on the performance may be unavoidable, and one or more applications may have to be

paused or suspended to prevent swap usage. This creates the need to find another host within the environment with enough free memory so that the load can be balanced and reduce the performance degradation caused by the memory shortage.

Live migration has been widely adopted as a method to achieve different objectives like load balancing and server consolidation in virtualized systems thus being able to improve performance and throughput when idle resources are available. However this might not always be a viable option and, if existing management systems do not consider other forms of preemption, significant performance degradation could occur.

1.2 Objectives

This work proposes a vertical elasticity based tool to provide efficient memory management and improve the throughput and resource utilization with benefits to both cloud providers and users. While on the cloud provider side, better server consolidation drives lower energy consumption by requiring fewer servers to attend the same workload and increases profit for the existing physical infrastructure, this also reduces the user wait times since more resources are be available for allocation. On the user's side, pricing is based on the amount of requested resources, and even if not consumed completely all of the time, users are charged for idle resources. Vertical elasticity can make a more fine grained pricing model viable where the costumers are charged for what is actually consuming over the execution time, meaning potentially lower costs for users.

VM migration is needed as a form of preemption to take advantage of idle resources in other hosts and relieve load from stressed servers. Live is the most common form of migration used in other approaches as the only preemption mechanism, however, other forms of migration combined with the existing preemption mechanisms (pause and suspend) in MEC yield alternative preemption forms enabling the system to deal with situations where live migration is not an option due to unavailability of an additional host with enough memory, making this tool more generally applicable.

This work aims to extend and adapt a previous version of the single server Memory Elasticity Controller (MEC), with appropriate VM migration policies, to be used within a distributed cloud management framework in order to improve the throughput of multiple job requests of batch and/or on-line applications. The objective is to increase resource utilization by taking advantage of idle resources in virtualized systems with multiple physical servers each managed by a local memory vertical elasticity controller.

1.3 Contributions

The main contributions of this work are listed below:

- A revised memory vertical elasticity-based tool that employs the coordinated use of multiple preemption mechanisms to improve the throughput and resource utilization of servers in a cloud environment while avoiding significant performance degradation due to memory shortages.
- Design and implementation of a framework that orchestrates multiple instances of MEC to operate in a cooperative fashion in a distributed infrastructure that employs virtualization and supports VM migration.
- Integration of multiple migration techniques with pause and suspension to avoid performance degradation even in scenarios when there is not other suitable host with enough memory available to carry out a live migration.
- Analysis of the costs, performance impacts and benefits of the preemption options, and different variants of VM migration, showing scenarios where multiple preemption schemes are needed to avoid performance looses.

1.4 Structure of the dissertation

The remainder of this dissertation is structured as follows: Chapter 2 summarizes some of the related work in this area, describing the techniques adopted to provide vertical memory elasticity. Chapter 3 presents the architecture of the *Memory Elasticity Management in Clouds* (MEMiC) framework in detail and describes the proposed modification to MEC to support VM migration and its integration with the other preemption mechanisms. In addition, the costs of different migration variants are analyzed within the context of the framework. Chapter 4 then evaluates the benefits of using different combinations of preemption schemes as part of the MEC scheduling policy, on system performance, while Chapter 5 draws some conclusions and indicates some future work directions.

Chapter 2

Related Work

Studies like that presented by Dawoud *et al.* [13], among others, have 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 seems to be the fact that the latter is less resource consuming and avoids increasing overhead costs when scaling-up vCPUs. The work of Dawoud *et al.* [13], however, did not consider vertical memory elasticity which also has to tackle another problem: The question of how to allocate appropriate amounts of a limited resource between competing applications.

It has been shown that having a sufficient amount of memory is fundamental for application performance and meeting SLA requirements, whether it be in the context of on-line systems [15], such as web-servers, or batch jobs [28]. However, wasting such a precious resource by statically allocating more than is actually required can lead to reduced throughput or the provisioning of what in essence would be unnecessary additional servers which in any case only increases costs for service providers. Therefore, there is significant interest in efficient schemes to dynamically allocate memory to, and remove memory from, VMs elastically.

Baruchi and Midorikawa compared two metrics for vertical memory elasticity, the Exponential Moving Average 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 [7].

The Vertical Elasticity Manager (VEM) proposed by Moltó *et al.* implements an elasticity rule to maintain a user-defined percentage of free memory in the VM, called the Memory Over-provisioning Percentage (MOP) [22]. The goal is to avoid page thrashing (i.e., swapping pages from memory to disk) in the virtual memory subsystem by scal-

ing up or down the VM's memory. If the percentage of free memory is lower than 80% or greater than 120% of the MOP value, the VM's memory allocation is increased or decreased accordingly. The new VM memory size is calculated based on the used (or consumed) memory and the MOP percentage. Not only can MOP be troublesome for the user to define, but high values lead to wasted memory while a low MOP could hurt performance [28,29]. Furthermore, although the aim is to avoid swap usage, the proposal can fail to achieve this in some scenarios and cause significant performance degradation [28].

Extending the VEM functionality, the authors in [21] proposed a vertical memory over-subscription framework to automatically monitor VMs and dynamically adjust their allocated memory in accordance with the memory requirements of their running on-line applications. The authors exemplified their work, which aims to manage memory within a cloud infrastructure, using OpenNebula [23] as a Cloud Management Platform together with their proposed Cloud Vertical Elasticity Manager (CVEM). A central CVEM receives information from all existing VMs in the cloud and decides whether to enlarge or shrink their memory allocations based on their earlier concept of MOP. Perhaps the most notable contribution has been to address one of the drawbacks of their earlier work. When a host becomes overloaded and the memory available on host becomes scarce, the system may decide to live migrate VMs to a lighter loaded server. However, the work fails to discuss what action should be taken if no such host is available. It is important to point out that CVEM uses over-subscription that needs to be tuned appropriately in order to avoid negative impacts on performance. Furthermore, the strategy depends on the hypervisor's [1] memory ballooning [2] mechanism, which is independent of CVEM, to recover free memory and is thus, subject to policies of that mechanism's implementation. Since these mechanisms are generally only activated when the host server is running low on physical memory, this leads to questions related to how quickly the CVEM scheme can react to changing demands and scheduling requirements of individual VMs.

In order to exploit both types of elasticity, Sedaghat *et al.* proposes a repacking based approach, where vertical elasticity is simulated through horizontal elasticity that configures a new set of VMs with changed configurations based in the current load of the system [30].

A performance-oriented vertical memory controller has been proposed by Farokhi *et*

[1] Software layer that is the interface between the VM and the host OS or hardware, to allow virtualization.

[2] Memory reclamation technique implemented in the hypervisor in order to borrow unused memory from a VM, commonly through a driver installed inside the VM that slowly allocates pages(inflating the balloon) according to the hypervisor requests.

al., based on control theory, to scale up or down the amount of VM memory by considering application performance characteristics such as response time (RT) [16]. Extending this work, Farokhi and others went on to propose a hybrid vertical memory approach where not only was RT considered, but also the average memory utilization, i.e. both application performance and resource capacity characteristics are taken into consideration [15]. In [2], Ahmad *et al.* presents an control theory approach based on previous work of Farokhi *et al.* [15] that is applied to both on-line web applications and batch jobs with a hybrid controller based on capacity and performance that prioritizes the QoS of on-line requests. Vertical elasticity is used to tackle small workload fluctuations, while horizontal elasticity is used to handle larger workload changes. By comparing the response time to an expected time, their strategy identifies if the currently allocated memory is enough for the application to process the current workload.

Also focusing on web services, Sotiriadis *et al.* presents an inter-cloud load balancer that is decoupled from the cloud providers, and aims to avoid downtimes due to failures [31]. Their solution exploits vertical elasticity of CPU, memory and disk resources via instance reconfiguration. The major disadvantage of this form of resource adjustment is that it is not carried out on-line and requires the VM to be stopped in order to commit the changes. The vertical scaling actions are triggered when the resource usage is higher than a pre-defined threshold.

More recently, these elasticity techniques have begun to be applied to containers [25] and to coordinate vertical elasticity of both VMs and containers within the same cloud environment [6]. Al-Dhuraibi *et al.* describe an approach to manage vertical elasticity in containers, invoking live migration when no more resources are available on the host [4].

Many of these approaches have characteristics in common. Often the 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 proposals only consider the execution of online applications, and when resources are scarce on a host, additional ones are always assumed to be available elsewhere, so adopting live migration is the straightforward naive solution that almost all vertical elasticity solutions adopt. As infrastructure utilizations increase, VM migrations need to be considered as part of the scheduling process since performance can be significantly affected through decisions including, for example, avoiding unnecessary VM migrations or choosing the appropriate VM to migrate. In this context, migration itself may be sufficient and should be combined with other preemption

techniques. One can argue that the elasticity controller should be integrated into the environment's application scheduling system.

Our present work aims to extend and adapt a previous version of the single server Memory Elasticity Controller (MEC) with appropriate VM migration policies to be used within a distributed scheduling framework in order to improve the throughput of multiple job requests of batch and on-line applications in cluster/cloud/edge computing environments. Previously, MEC focused on dynamically scheduling VMs with batch jobs to which memory will be allocated elastically. Even before memory on the host becomes scarce, MEC must decide how to distribute memory among the running VMs. MEC is aware that application may ask for more memory than they actually need, so will also attempt to recover memory dynamically. In addition, when insufficient resources are available, scheduling policies determine if and when, VMs should be paused to minimize performance degradation, and suspended to free memory for the remaining executing VMs [29]. The following chapter describes MEC in more details, gives an overview of the proposed distributed scheduling framework and discusses the integration of multiple migration techniques in to MEC.

Chapter 3

MEMiC and VM migration

Vertical memory elasticity improves resource utilization by taking advantage of the time varying memory consumption of applications. However, if the server becomes overloaded, hosted applications could suffer a significant reduction in performance. Existing solutions typically employ some type of preemption scheme (e.g. live migration [21] or VM suspension [29]) in an attempt to reduce or avoid such losses. This work hypotheses that combining multiple preemption schemes will improve the performance of previous resource management solutions. After analyzing and modelling different preemption costs, this chapter describes extensions to the work of Sawamura [27–29] on his vertical Memory Elasticity Controller (MEC), which was focused on manage a single server, in order to support various types of VM migration. The ability to migrate VMs also implies the need for coordinated scheduling between the local MEC instances on each host server. Thus this chapter also introduces the MEMiC Cloud Manager framework that coordinates the allocation of jobs among the physical servers of the environment.

3.1 MEMiC Architecture

This section briefly presents the architecture of the Memory Elasticity Management in Cloud (MEMiC) controller that coordinates resource allocation in a distributed and shared environment. The MEMiC manager is responsible for submitting jobs that where requested and to manage the virtual execution environment, which consists of multiple VMs running across multiple hosts. Each host managed by an instance of the Memory Elasticity Controller that is in charge of adjusting the memory of each VM in the host according to its demands, and manage their states in order to mitigate performance impacts when resource becomes scarce. We evaluate existing migration schemes and incorporate

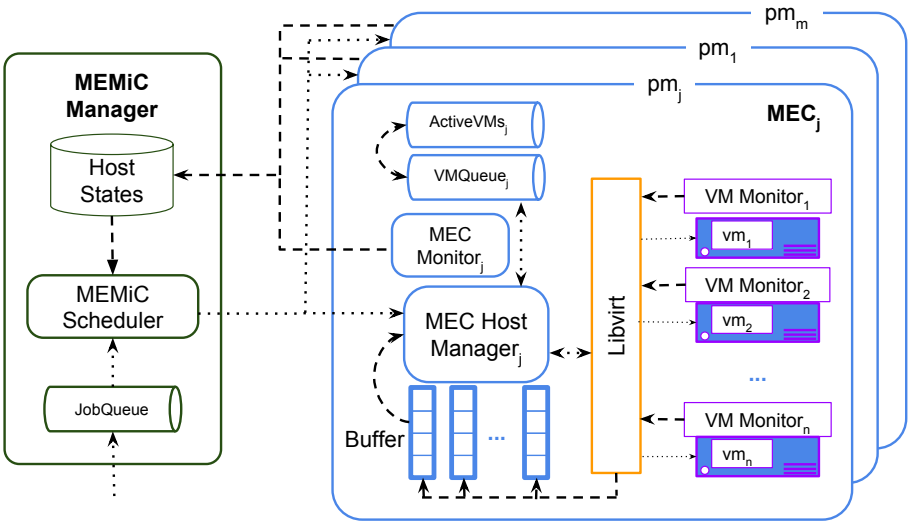

Figure 3.1: Overview of MEMiC architecture

mechanisms into MEC's management of VM states and transitions.

At the top level, as seen in Figure 3.1, the *MEMiC Manager* is composed of a system that collects monitoring information from each MEC instance on host servers in the environment, gathering them in the *Host States database*, and the *MEMiC Scheduler* that analyses this monitoring information in order to decide where to schedule new VM requests held in the job queue or to identify a host (physical host pm_j) to which an already submitted VM could be migrated.

Preemption schemes are necessary to avoid VM performance degradations when a given host does not have enough resources to meet demand. The MEMiC scheduler uses job memory usage profiles to decide statically where to initially allocate the VM for the next job. The MEMiC Scheduler can also dynamically adjust the existing allocation by selecting a specific VM for migration to a different host. In this work, however, the scheduler only identifies the source and destination hosts, decisions regarding which VM is to be migrated are taken by the respective MEC instance on the source host chosen by the MEMiC Scheduler. The reason for this is due to the asynchrony of resource availability and the latency to realize such operations, thus the need to coordinate such actions with alternative preemption schemes.

3.2 Local Memory Elasticity Controllers

Since reserving resources for peak VM consumption will perhaps leave some of these precious resources idle for much of a VM's execution time, an opportunity exists for a more

effective resource sharing approach to be employed. At the lower level of the MEMiC architecture, on each host pm_j, an instance of MEC operates to scale up or down the amount of memory allocated to each running VM on the host. The aim is to improve the throughput of the local VM scheduler by finding an appropriate distribution of the host's memory amongst the local VMs so that their demands continue to be met during execution. When resources are available, allocating memory to a VM on demand is straight forward. On the other hand, when there is not enough memory, under-provisioning will inevitably hurt performance. In this situation, effective management actions should be taken by the local MEC instance at pm_j defined as $HostManager_j$, such as: recovering unneeded memory from VMs; pausing or suspending resource starved VMs and resuming them when memory is again available, or; when identified by the MEMiC Manager, migrate a VM to an alternate host to free up memory for the remaining VMs. Migration, as well as pausing or suspending VMs, are effectively forms of pre-emption.

In order to support migration, the original MEC controller that only paused and suspended VMs has been modified. A brief summary of the underlying MEC architecture is presented next, together with the proposed adaptations that are evaluated later.

3.2.1 MEC and its VM states

At monitoring intervals of $I_{Monitor}$ seconds, each MEC $Monitor_i$ in vm_i, collects the following information from the vm_i's guest OS: total available memory; free memory, and; the amount of swap-in and swap-out since vm_i was last initiated (booted). The motivation for this selection of parameters is based on a detailed discussion presented in [28], which concluded that swap usage has a severe impact on application performance in a virtualized environment. The memory allocated to vm_i in a given time t is defined as $ma(vm_i, t)$, it is adjusted over time and is always greater than the amount of memory in use by the vm_i.

At every interval I_{host}, the MEC $HostManager_j$ considers the monitoring information placed in the buffers and the lists of active and inactive VMs, $ActiveVMs_j$ and $VMQueue_j$, respectively. Based on this information, actions are then taken by the MEC $HostManager_j$ for each vm_i allocated to pm_j, depending on that VM's current execution state and memory usage state. Figure 3.3 presents the VM execution states considered and possible transitions among them.

The job requests considers two parameters defined by the user to be considered for the VM instantiation that will execute it: the initial amount of memory, which should be

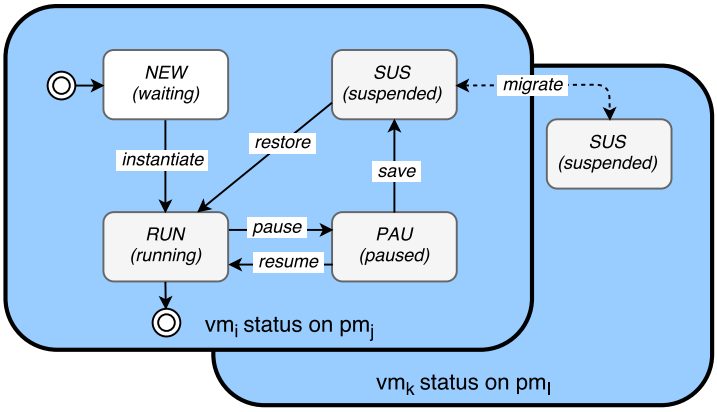

Figure 3.2: Original MEC VM states and their transitions

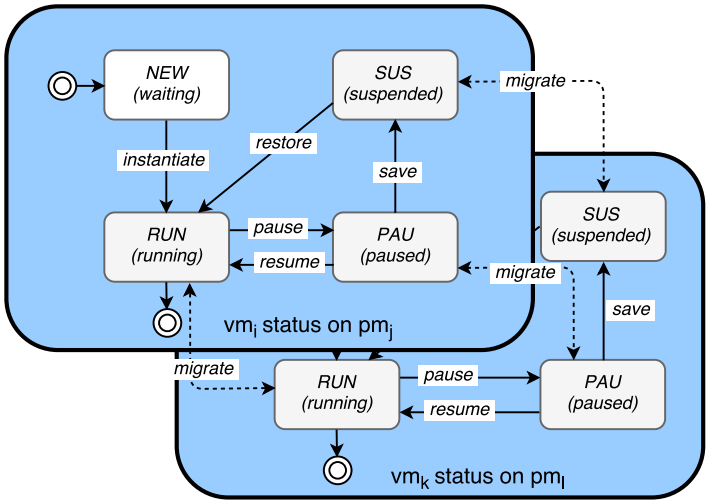

Figure 3.3: Proposed VM states and their transitions

only enough to initialize the operational system, and the expected maximum amount of memory. Setting a lower amount of initial memory allows MEC to distribute the memory among the VMs on demand.

When a VM request is first submitted to $HostManager_j$ by the MEMiC Manager, vm_i is inserted in $VMQueue_j$ and its state is set to NEW. When resources are available (i.e. the initial amount of required memory specified for the VM) and a decision is taken to instantiate the chosen VM in a NEW state for execution, its will be switched to running (RUN) and the VM moved to the $ActiveVMs_j$ queue. MEC detects memory shortages by monitoring both the swap-in and swap-out counters associated with vm_i. Depending on the system load and available free memory in the host machine pm_j, a running VM may need to be paused (PAU) at some point during its execution, meaning that vm_i would no longer be using the CPU(s), but would still retain the memory allocated to it. MEC primarily uses this form of preemption to prevent VMs and their applications from being forced to use swap and suffering significant performance degradation when there is

insufficient memory. From this state, a paused VM can shortly be returned to a running state if enough extra memory can be found by $HostManager_j$ to allocate to it, or may be suspended, where the VM's memory and context are saved to disk, so that its memory allocation can be released for distribution to other VMs in need. Another way to release memory would be to migrate the VM to another host with enough resources, depending on the state of the VM the migration could be executed in different ways.

In the original MEC proposal [29], future work intended to consider migrating VMs in a suspended state, as showed in Figure 3.2. However, some of these operations (e.g. suspension and migration) can take non trivial amounts of time (especially if jobs execution times are a few minutes or less) and might not be particularly useful mechanisms for use as a reactive approach to scaling. Policies are required to take in to consideration the time required to suspend or migrate a VM. As such times may depend on characteristics of the VMs, such policies should determine which VM should be subjected the chosen action. Other external issues may also have to be factored in, for example, the location of the storage where the VM state is saved. Therefore, VM migrations or suspensions should be evaluated with care, taking into consideration not only the amount of memory to be liberated, but also the delay before the memory to be gained from such an action becomes available for reallocation to other VMs. Thus, the reminder of this chapter is focused on how to aggregate these operations in to a coherent scheduling strategy to exploit vertical memory elasticity.

Algorithm 1 $HostManager_j$

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

The actions executed by each MEC instance in a host are shown in Algorithm 1, firstly MEC computes the amount of free memory in the host ($chfm$) in line 2, and then, based on that information, manages each queue and commits the changes (lines 3 to 6). As part of the update to support VM migration in line 7 MEC checks if there exists a migration request received from MEMiC, if so and it is needed, MEC chooses a VM to

Algorithm 2 *manage_inactive_vms*$(VMQueue_j, chfm)$

```
1: VMQueue_j ordered by vstate(vm,t) and wtime(vm);
2: p ← 1; // First paused VM
3: q ← |VMQueue_j, vstate(vm,t) = PAU|; // Last paused VM
4: tbrm ← to_be_released_mem();
5: // Manage suspended VMs:
6: while p ≤ q do
7:    if chfm − sat(pm_j) ≥ ms(vm_p,t) then
8:       ma(vm_p,t) ← ma(vm_p,t) + ms(vm_p,t);
9:       chfm ← chfm − ms(vm_p,t);
10:      action(vm_p) ← resume;
11:      p ← p + 1;
12:   else
13:      if chfm + tbrm <= 0 then
14:         tbrm ← tbrm + ma(vm_q,t);
15:         ma(vm_q,t) ← ma(vm_q,t) + ms(vm_q,t);
16:         action(vm_q) ← suspend;
17:         q ← q − 1;
18:      else
19:         tbrm ← tbrm − ms(vm_q,t);
20:      end if
21:   end if
22: end while
23: // Manage suspended and new VMs:
24: p ← p + 1;
25: while p ≤ |VMQueue_j| do
26:   if chfm − sat(pm_j) ≥ ma(vm_p,t) then
27:      if vstate(vm_p,t) == SUS then
28:         action(vm_p) ← restore;
29:      else
30:         action(vm_p) ← activate;
31:      end if
32:      chfm ← chfm − ma(vm_p,t);
33:   end if
34:   p ← p + 1;
35: end while
```

migrate and begins the migration process as will be explained in figure 3.4. In line 8, the relevant information about the host state is send to MEMiC manager.

The algorithm to manage the inactive VMs invoked in line 5 ($manage\_inactive\_vms()$) originally does not considers that there exists a time delay since the moment a VM suspension decision was taken until the memory is actually released. In order to improve the managing decisions, this delay is considered in the current MEC version as showed in Algorithm 2. This procedure is in charge of managing the inactive VMs in $VMQueue_j$, i.e. the VMs in states NEW, PAU or SUS. Depending on the amount of free memory and the memory demand of these VMs, the procedure decides: if new VMs will be instantiated; which paused VMs will be resumed either to continue executing or to be suspended (in this case, the VM's state is saved to a file in order to release memory); and if suspended VMs will be restored.

For a better understanding, the notation and their meaning used in Algorithm 2 are summarized in Table 3.1.

Table 3.1: MEC notation summary

Symbol	Description
$ms(vm_i, t)$	Memory Shaping: amount of memory calculated to be added to or removed from vm_i at time t
$vstate(vm_i, t)$	vm_i state at time t: RUN, PAU, SUS or NEW
$wtime(vm_i)$	Waiting time of VM that is not running (i.e. vm_i is either in PAU or SUS state).
$ma(vm_i, t)$	Memory Allocated: amount of memory allocated to vm_i at time t
$chfm$	Current host free memory: amount of memory that is available for allocation in the host.
$sat(pm_j)$	Swap activation threshold: minimum amount of free memory before the OS starts using swap

In line 1 of Algorithm 2, $VMQueue_j$ is sorted by the state ($vstate(vm_i, t)$) and decreasing waiting time ($wtime(vm_i)$), being the order of priority of the VM states PAU, SUS and NEW. In lines 2 to 3, the variable p indicates the paused VM that is in $VMQueue_j$ with the longest waiting time, while q indicates the most recently paused VM. In line 4, variable $tbrm$ is set with the amount of memory to be released, computed as the sum of the memory of all the VMs that are being saved or migrated, since $chfm$ will not count this memory as free at this point. From line 7 to 12, if there is enough memory available to satisfy the demand ($ms(vm_p, t)$) of the paused VM vm_p, it will be resumed and the required amount of memory is taken from $chfm$. Otherwise, as seen from line 13

to 19, since there is no memory left, then vm_q (the paused VM with the smallest waiting time) will be suspended in order to release its allocated memory. Note that, the amount of memory will be effectively added to $chfm$ only when vm_q is actually suspended and, to keep track of it, is added to $tbrm$ (differently from [27], where this amount would be added to $chfm$ at this point of the mechanism, representing an immediate release of memory. As a consequence, some paused VM would be wrongly resumed since memory is still not enough, leading to a performance loss). In line 14, the amount of memory that will be released by saving vm_q is added to the temporary variable $tbrm$. Line 13 checks the host free memory also considering $tbrm$. If there is no available memory, vm_q is suspended (as seen in lines 14 to 17). Otherwise, the amount of memory required by vm_q is reserved from $tbrm$ in future iterations of the algorithm. This step is a way to avoid unnecessary suspensions when is expected memory yet to be freed from unfinished past operations.

Finally, by checking memory availability (line 26), either suspended or new VMs can be restored or activated and therefore, their allocated memory must be taken from $chfm$ (lines 25 to 34).

Note that, MEC uses Libvirt [20] C/C++ API functions to commit the changes to the VMs, like the changes on the amount of memory allocated and the preemption mechanisms such as pausing, suspending and migrating.

3.2.2 Supporting VM Migration

Interacting with $HostManager_j$ and the VM queues, the MEC $Monitor_j$ is in charge of reporting the status of the host and the VMs allocated to pm_j to the MEMiC Manager, as seen in Figure 3.1. At the end of each scheduling interval, after the MEC $HostManager_j$ has updated the states of the VMs hosted on pm_j, the MEC $Monitor_j$ collects the following information: the VMs allocated to host pm_j (including those VMs recently designated by the MEMiC Scheduler) and their states; the amount of free memory available on pm_j; and VM scheduling data (such as the duration a VM has either been in a paused or suspended state).

Another observation worth mentioning is that in MEC's original implementation, communications between the VM Monitor and MEC were carried out through UDP sockets, and relies on the local system to update the network configuration after VM migration. In the upgraded version of MEC presented in this work, the VM Monitor communicates through Libvirt by using *channels* so that the VM can re-establish its network connection on a new host after migration automatically and in a way that is independent of the

network configuration, which may be different in each host.

Asynchronously, during each of its own scheduling intervals, I_{sched}, the MEMiC scheduler analyses the *Host States* information to first determine if any of the hosts are overloaded (i.e. the demand for memory exceeds the host's capacity). If so, the scheduler currently attempts to identify another host pm_i with enough free memory to receive a VM from pm_j. Future work will look to use the same technique the MEMiC scheduler uses to statically allocate new jobs [19] in order to predict if candidate hosts will be suitable.

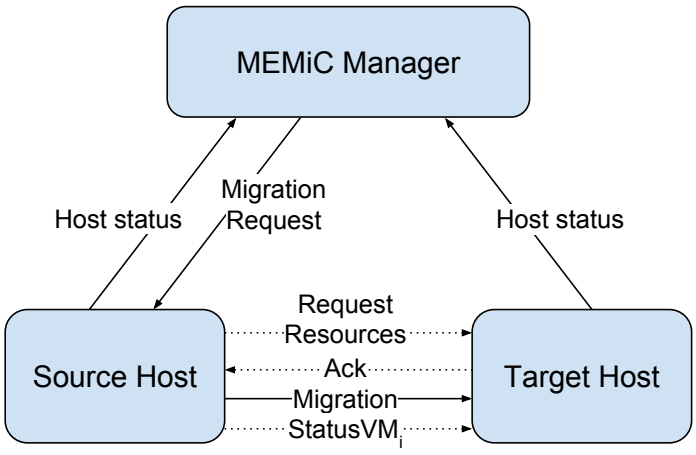

Figure 3.4: Migration mechanism in MEMiC

The MEMiC Manager will determine the source and destination hosts, lets say pm_j and pm_i, respectively, and notify the source pm_j of the existence of a underutilized host through a migration request, as shown in Figure 3.4. It will then be up to MEC $HostManager_j$ on pm_j to accept or not and, if accepted, to determine which VM will be migrated. With the growing maturity of virtualization technology, a number of forms of migration are now available [3]. Consequently, MEC now considers any VM in a post instantiated state (i.e. RUN, PAU or SUS) using a different forms of migraton for each state, as will be discussed in Section 3.3. Note that, in our current implementation, the migration of VMs is handled by LibVirt. Should a migration event fail, LibVirt will inform MEC so that the VM can be restored to its previous VM state prior to migration and rescheduled according to the local policy.

When $HostManager_j$ receives a *migration request* and choses a VM vm_m to be migrated to pm_i, a request is send to pm_i from pm_j to notify the destination and reserve resources, if pm_i does not have enough resources the migration will be rejected, otherwise the migration process can be initiated. First, the state of vm_m is set to a new *migrating* state (MIG) to inhibit any further modifications to the amount of memory allocated to vm_m. Once the migration is completed, vm_m will be returned the same state it had prior

to migration. Once in the MIG state, the migration of vm_m to pm_i can effectively start by using the appropriate hypervisor function calls in accordance with the form of migration to be carried out as explained in next section. Upon completion, $HostManager_j$ sends a notification message with the scheduling information for vm_m (previous state, memory allocated, swap-in and swap-out counters, amount of requested memory, job start, cycles since pause, count of pauses, total time paused) to $HostManager_i$ directly, so that it can start managing vm_m accordingly. Because the VM Monitor in each VM communicates through Libvirt via *channels*, this allows the VM to effectively reconnect and provide status updates sooner after migration to the new host. The MEMiC Manager will recognize the migration operation has completed after receiving the Host States information for pm_i with the state of vm_m.

3.2.2.1 The choice of source and destination hosts at MeMiC Manager level

Algorithm 3 $managing\_migrations\_at\_MEMiCManager_l()$

1: **for all** $pm \in ActivePMs$ **do**
2: **if** $(hfm(pm) < MIN\_FREE\_MEM$ or $qty\_paused(pm) > 0$ or $qty\_suspended(pm) > 0)$ and $qty\_migrating(pm) = 0$ **then**
3: $Sources = Sources \cup \{pm\}$;
4: **end if**
5: **end for**
6: $Targets \leftarrow ActivePMs - Sources$;
7: $source \leftarrow min\_hfm(Sources)$;
8: $target \leftarrow max\_hfm(Targets)$;
9: Send migration request to *source* PM informing who the *target* is

When a new host becomes available and begins to report its status to MEMiC, it is added to the $ActivePMs$ queue. Algorithm 3 summarizes the actions taken by the MEMiC mangager to detect hosts with a shortage of memory and will attempt to find another host to receive a VM from the overloaded host. Firstly, the MEMiC manager will look for hosts in the $ActivePMs$ queue that have a shortage of memory. Four host states are evaluated: the amount of free memory on the host, given MIN_FREE_MEM is the lower threshold on the amount of memory a host must have before being considered short on memory, or if there is any VMs in paused or suspended state, which would have been caused by a shortage of memory on the host, and if from that host a VM is not already being migrated to avoid interference. Hosts with a memory shortage are added to the list of candidates $Sources$ (lines 1 to 5). The hosts that could possibly receive a VM are identified in line 6 by subtracting the hosts in $Sources$ from $ActivePMs$, and placed in $Targets$ list. Finally, the *source* is the host with the minimum amount of memory

available in the candidates list *Sources* (line 7), breaking ties with the number of paused or suspened VMs, and *target* is the host with the maximum amount of free memory in the list of candidates, *Targets*, (line 8), these then constitute the hosts for the next migration request which is sent to the *source* host who chooses the VM to be transferred to the *target* host.

3.3 Integrating VM Migration into MEC

Preemptive process migration in distributed systems has long been exploited to improve load balance [14], availability and fault recovery or to reduce energy consumption, for example. While process migration can incur a significant overhead, this may pale in to comparison with the cost of leaving VMs on a failing node or to compete for resources on an overloaded one. More recently, migration of VMs is playing an important role in the efficient resource management of cloud systems, particularly in the context of horizontal elasticity [5].

Hypervisors implement migration by copying the memory pages of the subject VM to the target host and are commonly helped by a shared storage like NFS or iSCSI, so that a full copy of the VM disk image is avoided. With the growing reliance on virtualization and improved hardware support, more and more efficient variants of the VM migration have been developed.

In the case of *Offline Migration*, the state of the VM is copied only after it has been explicitly paused. Only once the copy has completed can the VM's execution be resumed on the target host. The time consumed by this type of migration depends only on the size of the memory allocated to the VM. On the other hand, *Live Migration* consists of moving a guest VM from the source host to its destination without any noticeable disruption to the execution of the migrated VM [11, 12, 33], using one of the following variants:

- **Pre-Copy Live Migration** is a simple approach, implemented in most hypervisors, where pages of the VM memory are copied successively, starting with the least used pages, until only dirty pages remain. At this point, the VM's execution is paused and these remaining pages together with the CPU and hardware device state of the running VM are copied to allow the VM's execution to be resumed on the target host [8]. The time to perform this type of migration depends not only on the size of the memory (the VM state information is usually very small in comparison), but also on the memory access patterns of the application. If pages become dirty after

they have been copied, they will have to be re-copied.

- With **Post-Copy Live Migration**, the VM's CPU and device state along with a minimum number of pages are copied first and the execution of the VM is started immediately on the target host. Then, the remaining pages are then copied to destination host in background or on demand, when page faults occurs in the running VM. While this scheme reduces the VM downtime and avoids pages being re-dirtied, it can lead to additional execution slowdowns of the VM application due to the resolution of page faults (page copies) over the network to the source host [17]. Post-copy techniques are currently only supported by recent versions of some hypervisors like QEMU [26].
- **Hybrid Post-Copy Live Migration** begins with a bounded pre-copy phase, before carrying out the post-copy phase, to reduce the potential number of future page-fault copies over the network and thus reduce some of the performance degradation that would impact the application [1].

This work adopts Hybrid Post-Copy Migration as one of the pre-emption mechanisms employed to alleviate the consequences of resource shortages on hosts under elastic memory management. Migration serves to move VMs from heavily loaded hosts to less loaded ones. Since each MEC instance only has local information, MEC does not know when, or to where, which one of its VMs could/should be migrated to and thus relies on MEMiC. While memory may be scarce on one host, on another it may relatively plentiful or at least sufficient to meet the demands of a remote VM. But, although the MEMiC Manager maintains the global state of the system to able to identify such instances, migration may not always be a viable option at a specific moment of time. Furthermore, given the fact that VM migrations are both time and memory consuming, MEC must therefore be prepared to take alternative actions to avoid unnecessary performance degradations when memory becomes scarce.

During **Live Migration**, VM memory is being consumed on both the source and destination hosts while the migration is processed. In the case of **Cold** or *Offline Migration*, the VM must first be pre-empted, i.e. its execution is halted, before its memory and state information can be transferred. Under MEC, a halted VM may be in paused (PAU) or suspended (SUS) state. When in the paused state, the VM will occupy memory on both the source and target hosts during migration and thus add to overall memory consumption of the system. Whereas, if the VM has already been suspended, then the VM's memory will have been save to disk and no longer be occupying memory on the source host. Thus,

migration of a VM in this state only involves a file copy. Memory will be allocated to the suspended VM on the destination host only when the VM returns to the *RUN* state.

Based on the performance analysis in [28], MEC pauses VMs that are about to be forced to use swap (due to the lack of available memory) in an attempt to avoid what might be a greater performance degradation should they continue to execute. Other VMs will continue to execute and when they terminate, their memory can quickly be reallocated to the paused VMs. However, if there is a continued demand for additional memory, this needs to be met by removing VMs from memory either by migration to another host or through suspension. All three pre-emption techniques impact the application performance of the VMs and the throughput of the system to varying degrees. Choosing what action should be applied to which VM depends not only on aspects related to the VM's additional memory requirements or remaining execution time, but also, the action's latency, which is related to the total memory footprint and the current state of the VM.

Given the VM states adopted by MEC, it is straightforward to identify the corresponding implementations to permit a VM in each state to be migrated.

- **Migrate a Running VM (MRVM)**, i.e. a VM that is in the *RUN* state, MEC uses live migration (the hybrid post-copy approach). This is the only migration option available for VMs running online applications;
- **Migrate a Paused VM (MPVM)**, i.e. a VM that is in the *PAU* state, MEC invokes offline migration, with the VM eventually being resumed by the MEC instance on the target host;
- **Migrate a Suspended VM (MSVM)**, the image of the VM in *SUS* state will have already been saved, either to a local disk or remote shared storage. For migration, locally stored data will either have to be accessible remotely or copied to the destination.

3.4 Preemption mechanisms evaluation

In order to understand the impact of each preemption option, it is important to analyze the costs associated with these operations on the different aspects of the VM scheduling problem. To exemplify the relative costs of these operations, a first set of experiments was executed with two servers, each with two Intel(R) Xeon(R) X5650 6 core CPUs@2.67GHz

and 24 GB of RAM, running CentOS 7 with Linux kernel version 4.13.4, and interconnected by a dedicated Gigabit Ethernet network. The hypervisor adopted was the Kernel-based Virtual Machine (KVM) - QEMU version 2.9.0 and Libvirt version 3.2.0 was used as an API to manage the virtual infrastructure.

The experiments throughout this work use a couple of the synthetic memory intensive jobs ($J1$ and $J2$), proposed in [28, 29] because they exemplify the need for elasticity controllers to be able differentiate between an application's memory demands and its actual requirements. By requirements, this means the minimum amount memory required to execute the job without suffering performance degradation. These jobs have been designed to access a vector of a predetermined size sequentially, and repeatedly, a fixed number of times. While $J1$ accesses 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 $w = M/4$), for the same number of iterations. Both jobs have the same memory demands, but because of their distinct memory access patterns, their requirements differ over time. This allows us to also investigate how data locality and types of swap usage impact memory elasticity and VM scheduling.

In order to measure the cost of each type of preemption operation, this first experiment was performed using a single VM with enough memory to execute instances of job $J1$, with a vector of size 0.5, 1 2, 4, 6 and 8 GiB, respectively. Note that MEMiC Manager was not required in this experiment.

Table 3.2: Average Pre-emption operation costs in seconds in relation to the amount of memory allocated to a VM

Mem. Size	pVM	rsVM	sVM Local	rtVM Local	sVM NFS	rtVM NFS	mVM
0.5	0.01	0.01	7.4	1.5	15.5	7.1	7.7
1	0.01	0.01	12	1.8	22.6	12.8	12.8
2	0.01	0.01	21.5	2.7	39.6	26.3	21.7
4	0.01	0.01	37.7	3.5	61.1	46.7	39.3
6	0.01	0.01	54	5.9	86.3	69.4	58.3
8	0.01	0.01	70.2	8.5	119.8	90.7	76.3

The times shown in Table 3.2 are the average of 10 executions for different sized instances of job $J1$ and preemption operations, where: pVM refers to the time to pause the VM and, rsVM, the time to resume the VM on the same host; sVM is the time to suspend a paused VM either to local disk (sVM Local) or to remote storage via NFS (sVM NFS); rtVM Local and rtVM NFS are the times to restore a VM from local disk or remote storage via NFS, respectively, and; mVM, the VM migration time, is the overall

elapsed time from the initiation of the migration operation until the moment that the corresponding resources on the source host were released.

While the migration time is proportional to the amount of memory allocated to the VM, the times to perform a live migration, offline migration or copy an image file of the memory from one host to another were found to be very similar. Given we are interested in the relative times of the different operations, only one column of times (mVM) is shown for the different migration types in Table 3.2.

In practical terms, the time to pause and resume a VM is very small and does not depend on the amount of memory allocated to the VM. Also, the time to migrate a running VM (MRVM) is therefore basically the same as the time to migrate a paused one (MPVM) and both approaches are significantly quicker than the time to MSVM. The times for MRVM and MPVM are relevant as they represent the response times to free memory on the host. MEC itself, however, can free up memory by suspending a VM. Suspension causes a VM's memory to be saved to disk, doing so locally has virtually the same cost as migrating a running or paused VM. From Table 3.2 and for the experimental setup used, saving the memory image to local disk is quicker than doing so over the network and the same is true for restoring VMs. So if the intent is to restore the VM on the same host it will be better to save the VM's memory locally. If the plan would be to eventually migrate the VM, it seems better to still save locally, then copy the memory to the destination when known and restore locally than to save the memory to the remote shared storage and later restore this remotely on the destination host (sVM Local + mVM + rtVM Local < sVM NFS + rtVM NFS), for example in this experimental setup.

What Table 3.2 does not show is impact of live migration on the job's own execution time. Figure 3.5 presents the average additional time a migrated job (MRVM) requires over one that was not migrated (No MIG). While the migration delay mVM is proportional to the VM memory size, the results in the figure show that MRVM only delays the job's execution by an almost constant but relatively small amount and, basically, this cost could be considered to be independent of VM memory size. In the post-copy live migration adopted, the delay is due to the need to copy pages remotely over the network, on demand. While these results appear to be line with expectations, note that this experiment was carried in an underloaded dedicated environment.

The next question that arises is which preemption operations are actually required in the context of memory elasticity management and how should they be incorporated into the MEC scheduling policy. Having taken into account these preliminary observations,

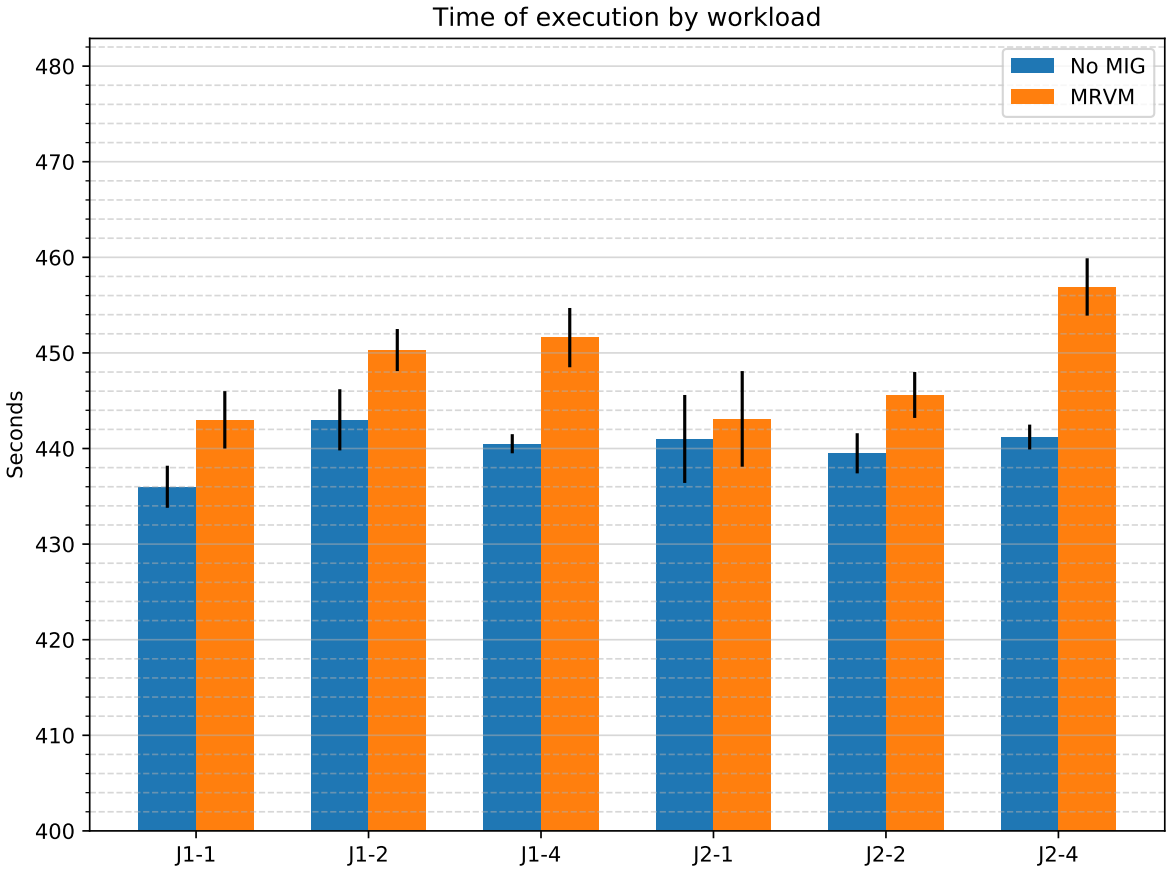

Figure 3.5: Impact of live migration on the average execution times of $J1$ and $J2$ with 1, 2 and 4 GiB, respectively.

the following section aims to analyze experimentally the impact of each migration variant on the execution of the batch jobs under MEMiC management with more than one server, when memory become scarce.

Chapter 4

Experimental Evaluation

Extensive work exists on the development of VM migration schemes and many of these have been incorporated widely into systems that exploit vertical elasticity. Resource elasticity management focuses on improving utilization and thus increasing throughput for a given workload. This chapter presents a set of experiments that aim to highlight the importance of using more than one form of preemption as part of a scheduling strategy to harness shared computing environments. While the effect of preemption may appear to (perhaps) sacrifice the performance of a particular job, the goal is to benefit the workload as a whole. Many existing resource management solutions rely solely on the live migration of VMs as a technique to avoid the penalties of over-provisioning [21]. While migration can improve performance by taking advantage of idle resources, such a resource may not always be available when needed. This chapter evaluates the impact on the makespans of workloads and the benefits of three distinct preemption mechanisms, which have been combined in MEC, under different scheduling scenarios.

4.1 The Test Environment and Synthetic Test Jobs

The experiments presented in this chapter were executed across three servers with the same configuration as that of the experiment described in Section 3.3, i.e., each server with two Intel(R) Xeon(R) X5650 6 core CPUs@2.67GHz and 24 GiB of RAM, running CentOS 7 with Linux kernel version 4.13.4, and interconnected by a dedicated Gigabit Ethernet network. The used hypervisor was the Kernel-based Virtual Machine (KVM) - QEMU version 2.9.0 and Libvirt version 3.2.0 was used as an API to manage the virtual infrastructure. Two of the servers are dedicated to running VMs exclusively under MEC management in a virtual environment and are collectively managed by MEMiC running

on the third server. Each host has 22GiB of RAM memory available for use by the VMs that execute one of the two job types also used in the experiments of Section 3.3. The results presented here are based on the average of 10 executions.

The jobs used in this set of experiments are in fact same synthetic application, but which adopt two distinct memory access patterns. These job types $J1$, $J2$ have been adopted from previous work [28], where this synthetic application was designed to highlight a subtle aspect of how the memory is used, impacts performance. While $J1$ accesses 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 $M/4$), for the same number of iterations and, when sufficient memory is available, the two jobs will have identical execution times. The motivation to use $J1$ and $J2$ is toa analyze the impact of different memory localities on job execution times under memory restrictions and, consequently, identify their effective memory demands and swap usage over time.

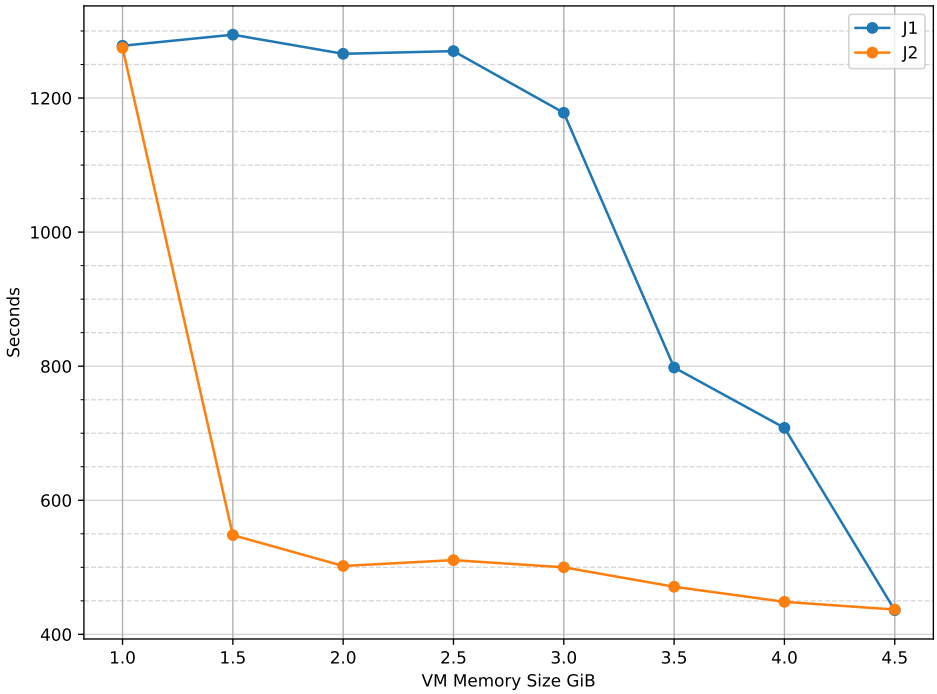

Figure 4.1: Memory-Performance Profile of job $J1$ and $J2$ describes how a job's execution times varies with the amount of available VM memory.

The job execution times for $J1$ and $J2$ for a vector M of size 4 GiB in VMs with different amounts of allocated memory is presented in Figure 4.1. The ideal amount of memory required by the VM is 4.5 GiB (4GiB for the vector plus around 500 MiB for the guest OS) with which each job will execute in a little under 450 seconds. As the amount of available memory in the VM is decreased, the job execution times increase, but at

different rates for each type of job. In the case of $J1$, a 22% (i.e. 1 GiB) reduction in memory availability caused a 77,78% increase in the execution time. With one third of the required memory, the execution time is almost three times longer. This is not the same in the case of $J2$, where the performance degradation is only 22% when 1.5 GiB of memory is available. Because of the better memory access locality of $J2$, the memory page working set is 1/4 of the vector size and the effective total memory demand of the $J2$ VM is in fact only around 1.5 GiB. While with less than this amount, the performance of $J2$ degrades drastically and is equivalent to the swap bound $J1$. On the other hand, with slightly more memory, i.e. 2 GiB, the degradation is half of the previous value (11%). In all cases, these overheads are caused by having to use swap to exchange memory pages to varying degrees. While both jobs request the same amount of memory, in previous work [27], MEC was designed to recognize this difference by analyzing simultaneous swap-in and swap-out (SISO) operations instead of using a page fault counter, adopted in most other approaches, so that it can distributed the host's available memory more effectively [29].

4.2 VM Migration Policy Evaluation

Before analyzing the impact of the different migration operations on the total workload execution times, this subsection presents a brief evaluation of which specific VM should be chosen for migration. Traditionally, as is the case in the CVEM system [21], the VM that is not soliciting additional memory and has the smallest amount of allocated memory from the set of running VMs is the one chosen for live migration. From the investigation in Section 3.3, this decision seems to be motivated by the fact that this choice incurs the shortest migration times. The following experiment aims to confirm if this continues to be a good choice in the context of scheduling VMs with MEC and its preemption schemes to minimize the total execution time of a workload. Under MEC, a VM in any one of the three VM states (i.e. *RUN*, *PAU*, *SUS*) can be chosen for migration (using the respective migration technique, as described in Chapter 3) according to the migration policy being employed.

Given an overloaded physical machine pm_j, MEMiC indicates to $HostManager_j$ that there is another physical machine pm_k with fm_k free memory available. MEC $HostManager_j$ can then select one VM to migrate in accordance with one of the following two migration policies:

Min: selects the VM in pm_j with the least amount of memory allocated, while;

Max Fit: selects the VM with the largest amount of memory that fits in pm_k, i.e. the VM vm_i on pm_j with the largest amount of currently allocated memory that is less than fm_k.

Table 4.1: Workload configurations.

Workload	**# of VMs**			**Total Number**	**Memory** (GiB)	**Total Memory**
	4 GiB	2 GiB	1 GiB			
A	4	3	1	8	23	27
B	4	4	2	10	26	31
C	4	4	4	12	28	34

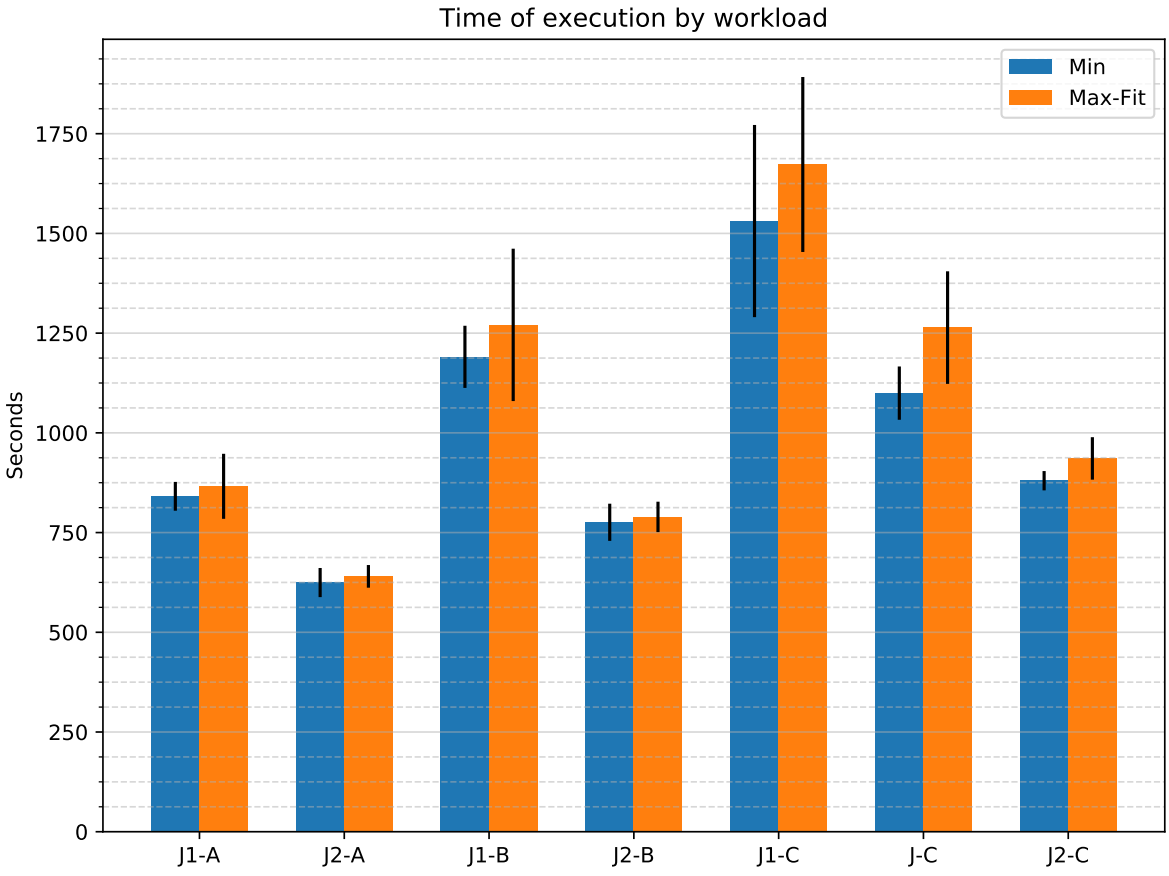

Figure 4.2: Comparison of migration policies with heterogeneous workloads.

In order to compare the Min and Max Fit migration policies, the following experiment was carried out with the workloads described in Table 4.1. For each workload A , B and C, the # **of VMs** specifies the number of VMs each running a job of type $J1$ or $J2$ that requires a vector of size 4, 2 and 1 GiB, respectively; **Total Number** is the total number of VMs/jobs in the workload; **Memory** is the total amount of memory required by the jobs of the workload; and finally, **Total Memory** is ideal amount of memory that

will have to be allocated for all of the VMs taking into account the additional memory overheads related to the guest OS (0.5GiB per VM).

Figure 4.2 presents the execution times for workloads composed exclusively of jobs of type $J1$ or $J2$ or an equal combination of both, as indicated by J-C for just workload C. All seven workloads will overload the memory of the single server to which they were scheduled initially (although sufficient CPU cores are available). However, the second, initially idle, server will provide sufficient memory to help the system meet the memory demands of the three workloads. Note that workloads of jobs $J2$ execute faster than those workloads composed of $J1$ jobs because, as seen in previously in Figure 4.1, $J2$ jobs can execute with less memory meaning that those workloads suffer less memory starvation and will require fewer migrations. On the other hand, migration will improve the distribution of the work and can reduce the total execution time should the host actually be overloaded.

With insufficient memory on the server, as the demand for memory increases further (A < B < C), the evaluation shows that their respective execution times for each workload also increases since it is affected by the migration overhead. On the whole, it appears that the traditional Min policy does provide slightly better performance for both types of jobs considered in this work, although this is not too clear cut or conclusive. While Min frees up a smaller amount of memory on the source host, this is countered by its speedier availability for reallocation to other VMs by MEC on the overload host. Although, the Max-Fit policy does carries out fewer migrations, they do effectively take longer and thus the system under this policy appears to be slightly less reactive. Nevertheless, in various cases, the differences are rather small and thus indicates that further investigation is merited. We leave this investigation for future work and in the remaining experiments, MEC uses the Min policy for a common means of comparison.

4.3 A Comparison of the Preemption Strategies

This section aims to compare the performance gains of using and combining different preeemption schemes including VM pausing, suspension and the VM migration operations **MRVM**, **MPVM** and **MSVM**, described previously in Chapter 3, when a given number of jobs are initially submitted to the same MEC host. In order to over-commit the host's memory to varying degrees, the following set of experiments were executed, separately: k jobs of type $J1$ requiring 4 GiB of memory each, were submitted at intervals of 20 seconds. For each job i, MEC $HostManager_1$ starts a VM vm_i with 1 GiB of initial memory. In

total, each vm_i consumes 4 GiB + $GOSmem$, where $GOSmem$ is the memory used by the guest operating system, which is around 500 MiB. A similar experiment is also carried out with k jobs of type $J2$ in VMs with the same characteristics.

The experiments were carried out for values of k between 4 and 7, in an attempt to analyze the performance impact of increasing VM memory demands on a MEC managed server. The execution times for each workload of k jobs of type $J1$ and $J2$ are shown in Figure 4.3 as Jn-k, $n = 1, 2$ and $k = 4, ..., 7$. For each workload, the figure shows the average execution times or makespans under six different scenarios:

No Preemp: represents a conventional execution scenario where jobs are statically allocated and then executed concurrently without the possibility of being preempted including migration as no alternative resource with sufficient capacity is considered to be available;

NO MIG: is an execution managed by an original version of MEC that only permits the pausing and suspension of VMs. The scheme also describes the situation when VM migration is not possible when no alternative resource with sufficient capacity is available;

MRVM: only allows MEC to migrate a VM that is in a RUN state;

MPVM: MEC can only migrate a VM that is in a PAU state;

MSVM: MEC can only migrate a VM that is in a SUS state;

MECv2: is the combination these individual schemes where MEC must choose when and how to apply the preemption methods including the migration of a VM in any of the three states mentioned above.

The average makespans include the times to launch, manage, and terminate all the k VMs while they each execute their jobs, measured from the start of the first submitted job until the end of the last job to finish (which may not be the last submitted one). Note that MRVM cannot pause or suspend VMs and MPVM cannot suspend VMs. This means that the MRVM scenario is in fact representative of traditional vertical elasticity controllers that only employ live migration, such as the CVEM system [21].

To help understand the results shown in Figure 4.3, Tables 4.2 and 4.3 register the number of times that a VM was paused, suspended or migrated, for each job type $J1$ and $J2$, and k, respectively. Note that these event counts correspond to only one specific

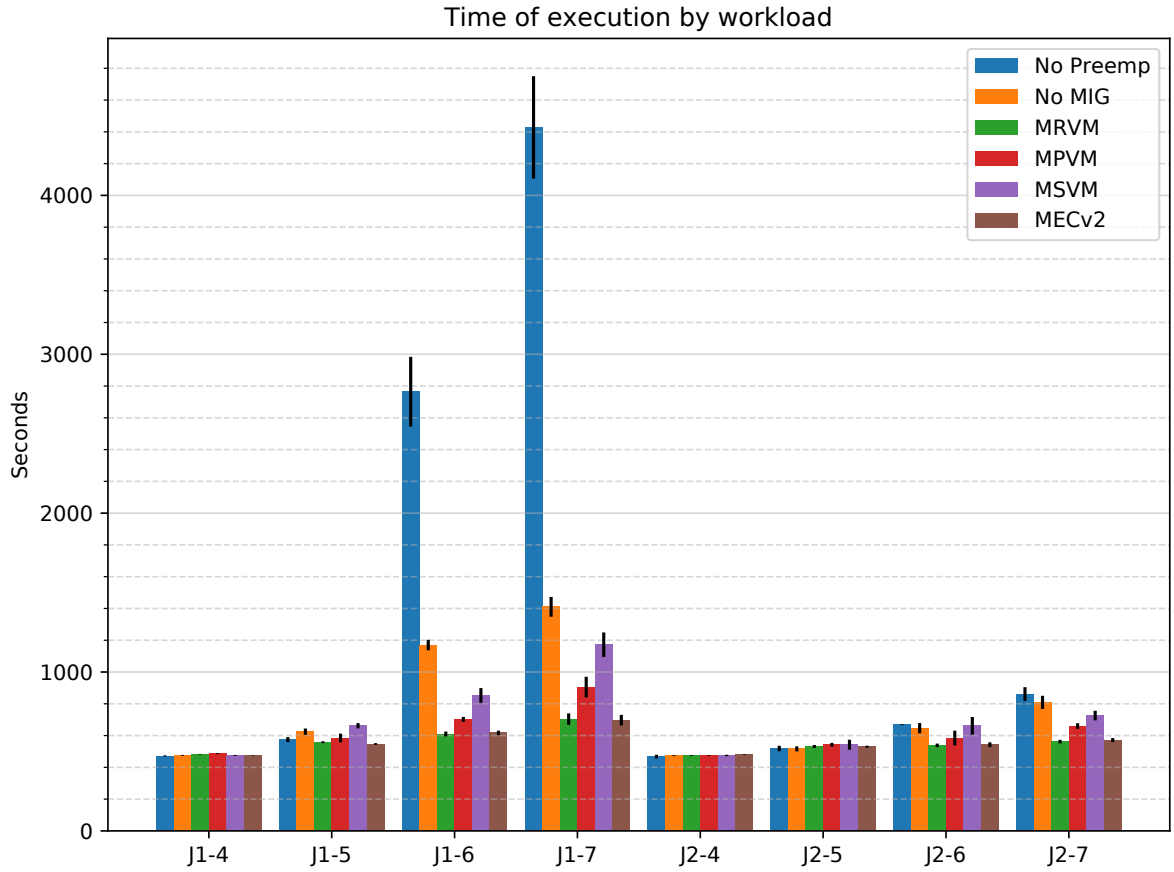

Figure 4.3: Average workload execution times for 4 to 7 jobs of type J1 and J2, without (No Preemp) and with MEMiC. Each remaining color represents different preemption schemes: no migration (No MIG), MRVM, MPVM, MSVM, and MECv2.

execution: the one with the execution time closest to the average of the 10 executions in each experiment. In the case of $k = 4$, where the server has sufficient capacity, there is no need to pause, suspend or migrate any VM and so the counts were omitted from the table.

Although both $J1$ and $J2$ require the same amount of memory (both read and write 4 GiB) during their respective executions, $J2$ has a smaller working set and can execute with less memory than $J1$ without the same significant loss in performance, due to its locality oriented memory access pattern [28]. It is important for any scheduling proposal to discern the difference between such characteristics of jobs so that scarce resources such as memory can be allocated to those VMs that would have their performances degraded the most and perhaps thus avoid unnecessary VM migrations and the use of additional servers.

For $k = 4$, performances for all of the approaches were similar for both job types. An observation in relation to the migration strategies is that the average execution times

Table 4.2: The number of VMs running $J1$ job that were paused (PA), suspended (SU) and migrated (MI) and the respective execution time (T) of k VMs in one sample (the one with median execution time)

Scenario	$k = 5$				$k = 6$				$k = 7$			
	PA	**SU**	**MI**	**T**	**PA**	**SU**	**MI**	**T**	**PA**	**SU**	**MI**	**T**
No Preemp	0	0	0	580.0	0	0	0	2776.7	0	0	0	4830.8
No MIG	1	0	0	626.7	3	2	0	1181.5	10	4	0	1430.3
MRVM	0	0	1	559.4	0	0	2	615.1	0	0	3	720.1
MPVM	2	0	1	588.8	4	0	2	704.0	10	0	3	908.8
MSVM	1	1	1	661.9	4	2	2	870.9	10	4	3	1188.3
MECv2	1	0	1	548.9	1	0	2	621.6	1	0	3	697.6

have the following relationship: MRVM < MPVM < MSVM. Recall that MSVM must first pause and then suspend a VM before the image can be copied to the target server. From Table 3.2, one can see that with migration and suspension costs being effectively the same, MSVM is unable to release its memory sooner than MRVM.

In the case of $J1$ jobs, when $k = 5$, the host server is slightly short of enough memory for some of the VMs and this causes their performances to degrade a little. Migrating one VM is only slightly better than leaving the VMs to execute concurrently (No Preemp). Since No MIG cannot migrate a VM, it pauses a VM, adding a delay to the execution time. For higher values of k, the shortage of memory becomes increasingly acute for the concurrently executing $J1$ jobs which induces significant performance degradation due to swap usage, as seen in the case of No Preemp. Note that the average execution times obtained by MRVM would be the same as those of No Preemp should VM migration not be possible. Through pausing and suspending VMs, MEC without VM migration (No MIG) is able to reduce this degradation for both types of jobs. As expected, migration can improve performance further by taking advantage of the additional memory on the second server. However, note that MECv2 is able to pause VMs in conjunction with migration to provide slightly better performance than MRVM.

These observations can be corroborated by looking at Tables 4.2 and 4.3 that count the number of pause, suspension and migration events during an execution of $J1$ and $J2$ jobs, respectively. One can notice an increasing number of VM pause and suspension events due to performance damaging swap usage in the No MIG version as k increases. Some of the paused VMs are suspended in order to release memory. In the case of MRVM, a smaller number of VMs are paused and no VM is suspended, since memory is released due to the migration of running VMs. In MPVM, VMs are obviously paused, but no VMs are suspended, since the necessary memory is released via offline migration.

Table 4.3: The number of VMs running $J2$ job that were paused (PA), suspended (SU) and migrated (MI) and the respective execution time (T) of k VMs in one sample (the one with median execution time)

Scenario	$k=5$				$k=6$				$k=7$			
	PA	**SU**	**MI**	**T**	**PA**	**SU**	**MI**	**T**	**PA**	**SU**	**MI**	**T**
No Preemp	0	0	0	525.6	0	0	0	668.9	0	0	0	866.0
No MIG	0	0	0	519.7	0	0	0	646.8	2	1	0	821.8
MRVM	0	0	1	532.0	0	0	2	539.1	0	0	2	565.5
MPVM	0	0	0	547.3	2	0	1	585.7	4	0	1	661.5
MSVM	0	0	0	542.7	6	1	1	664.0	2	2	1	727.1
MECv2	0	0	1	531.6	0	0	2	550.2	1	0	3	574.2

Where VMs are running $J2$ jobs, the results show a similar trend, although execution times were significantly quicker. While the number of VMs were the same for both $J1$ and $J2$ experiments ($k = 5, 6, 7$), in the case of $J2$, the memory requirements at each scheduling cycle were smaller than in the case of $J1$. Therefore, the server suffers less memory pressure and, as seen in Tables 4.2 and 4.3, the number of pause, suspension and migration events is much smaller than the corresponding $J1$ executions (e.g. see No MIG).

The scenario in this experiment assumes that there is a spare idle server available immediately should overloading occur and thus clearly favors migration capable schemes by harnessing additional resources to reduce the makespan. Traditional schemes like MRVM will obviously fair very well, especially if one assumes a classic Cloud infrastructure with near "infinite" resources. But, if one were to consider an Edge Computing infrastructure, where resources are limited and resource sharing is more prominent, should this additional resource not be available, MRVM would have the same behavior as No Preemp as it does not have the option of pausing or suspending VMs. By integrating preemption schemes into its scheduling policies, MECv2 is more flexible in its decisions while still being competitive in situations amenable to migration.

In more realistic scenarios where additional resources may not be available immediately, it is possible to show that the performance benefits of MECv2 will lie between those obtained with MECv2 in relation to No Preemp and MECv2 in relation with MRVM. Our objective is not to show that one scheme is better than another but rather to motivate the need to consider alternatives to solely live migration based solutions, e.g. the adoption of other preemption schemes in conjunction with various forms of migration, which can be combined to be used cooperatively in an integrated fashion as part of a resource manager's scheduling strategy.

4.3.1 Not all swap usage is that bad!

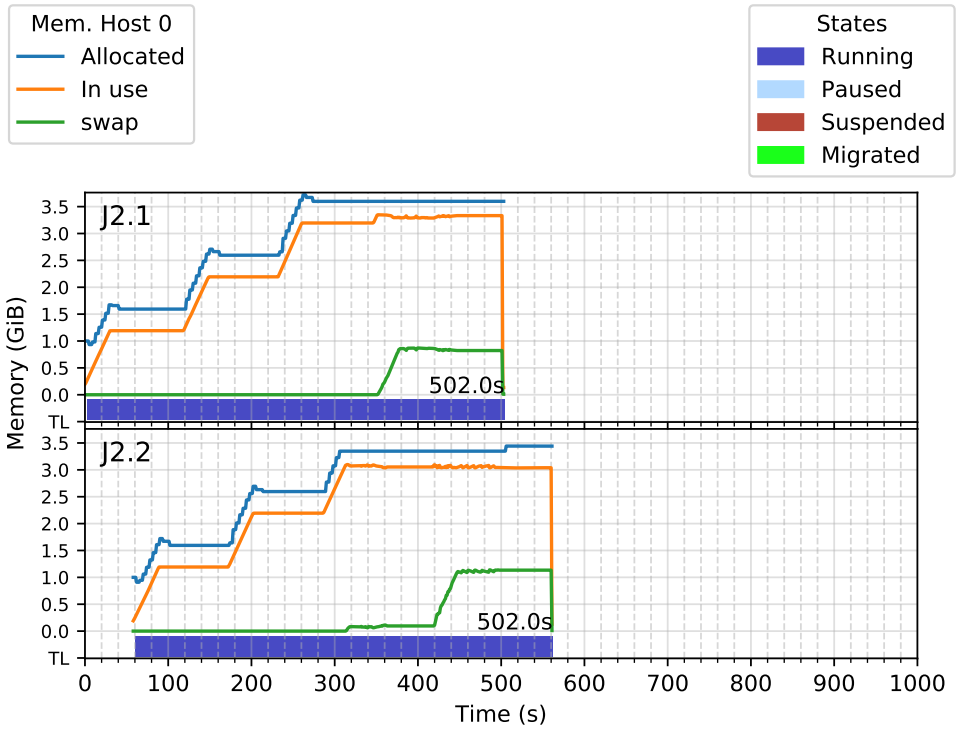

Figure 4.4: Execution time line for two concurrent $J2$ jobs under MEC.

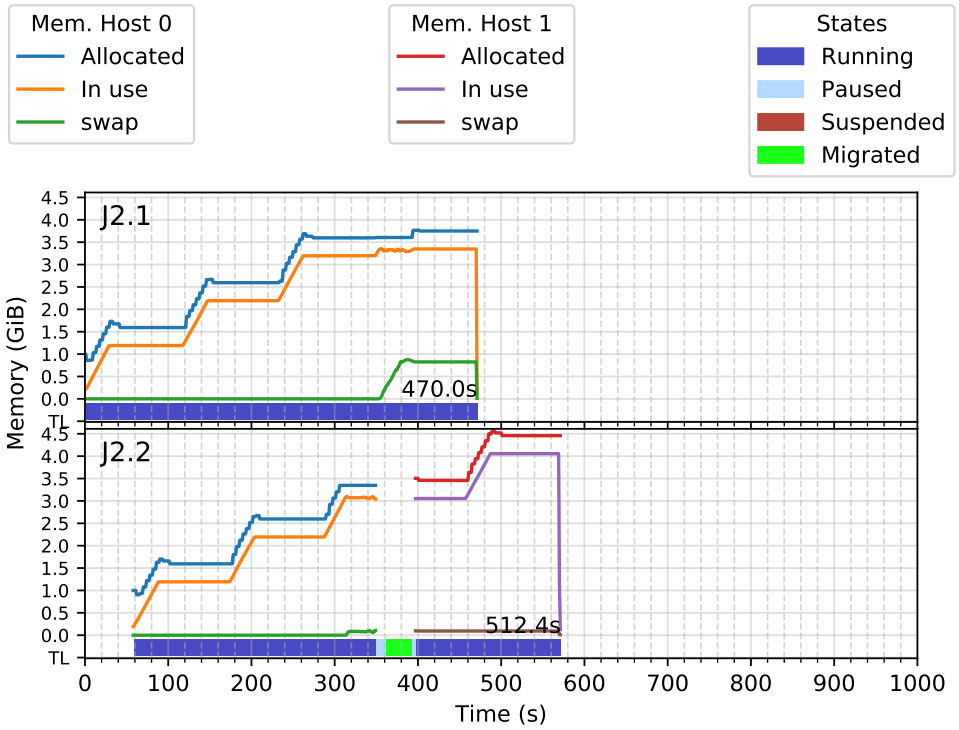

Figure 4.5: Execution time line for two concurrent $J2$ jobs with VM migration at time 360 seconds.

The experiments in the following two subsections only use jobs of type $J1$, in order to exemplify scenarios with fewer competing VMs. But first, in order to help justify: the behavior of MEC with respect to swap utilization, and; the performance difference between $J1$ and $J2$ jobs, as shown in Figures 4.1 and 4.3, some time lines of memory consumption are presented for the execution of two $J2$ jobs running under MEC's vertical elasticity management on a host server with 7 GiB of free memory, initially. These execution time lines also indicate the state of the VM and its swap consumption over time. As shown in Figure 4.4, the jobs are submitted with an interval of 60 seconds between them and consume memory in phases. There should appear four phases in all, but due to the memory limitation of 7 GiB, each job eventually has around 1 GiB less than its ideal allocation. Therefore, for the final phase, at around 360s for $J2.1$ and 60s later for $J2.2$, the jobs need to use swap. Note that, while the swap consumption is greater for $J2.2$ since it has less memory (in fact, $J2.2$ required a small amount of swap earlier to complete the third phase), the execution times of the two jobs were identical. As discussed in Section 4.1, this swap usage (no longer required memory pages are in fact only being swapped out to disk) has a relatively small impact on the job execution times. In comparison, Figure 4.5 presents the same execution of the two $J2$ jobs but now with the migration of the VM with the smaller memory allocation occurring at time 360s, the exact moment that the other job begins to use swap. Notice that although job $J2.2$ can now obtain the ideal quantity of memory, its effective execution time has been increased (but discounting the duration of the preemption, it would have had executed near its ideal time of around 450s). Furthermore, while this migration frees up memory on the original host, $J2.1$ no longer had any need for it at that moment (its execution time of 470s is in line with that indicated in Figure 4.1 for an execution with 3.5 GiB of memory). In this scenario, the solution with VM migration was more costly in terms of the total execution time. The main take away point is that while an increase in swap usage does have impact on performance, it is the way that swap is being used that is important in terms of execution times [28]. In the case of $J2$ jobs, the impact is an increase of 5%, or 10% with concurrent VM executions, in job execution times, whereas for $J1$, the same quantity of swap usage can lead to almost 80% increase due to the exchanging of pages between memory and swap (SISO). With this in mind, MEC has been designed to consider the performance impact of swap out as acceptable and of SISO to be avoided [28].

4.3.2 Preemption scheduling with some resource starved jobs

This subsection and the next highlight scenarios where combining preemption techniques can be more beneficial than only relying on live migration. Different from Section 4.3, the experiments in these subsections focus on the concurrent execution of $J1$ jobs in a more realistic scenario where an additional resource may not be available immediately.

In this subsection, the experiments model the scenario where some jobs have their ideal quantity of memory and other do not. To exemplify and make it easier to illustrate the outcomes, only two $J1$ jobs are used and are submitted to a server with 7 GiB of free memory, to be executed under a MRMV or MECv2 preemption strategy. While the first job is submitted alone, the second job arrives 120 seconds after the first, at which point the first $J1$ instance will have received its required memory (4.5 GiB). With not enough memory left, the second job will be forced to use swap. Under this scenario, by time 180s, all of the host's 7 GiB of memory will have been allocated, with the second $J1$ instance having received exactly half of its ideal memory requirement.

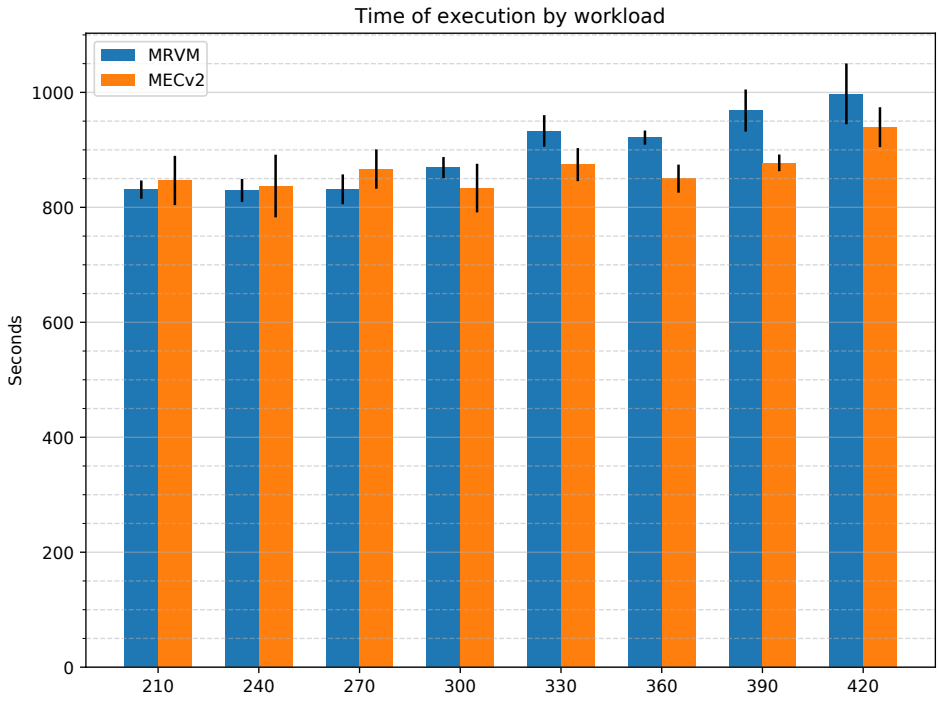

Figure 4.6: Comparison between MRVM and MECv2 managed execution times for two $J1$ jobs, with the second starting 120s after the first, and different values of t_h.

In this experiment, the moment t_h that the MEMiC cloud controller notifies MEC that an additional idle server is available is varied from time $t_h = 210, \ldots, 420$, in steps of 30 seconds. The total execution times for the two schemes are shown in Figure 4.6. For

$t_h = 210, 240, 270$, the MRVM approach appears to have a slightly better performance than MECv2 as it is able to migrate the VM almost immediately when required. However, when the additional host is only available later ($t_h > 300$), MECv2 manages to reduce the impact. For values of t_h closer to the execution time of the first job ($t_h > 420s$) , it will be best not to migrate the second job since the first $J1$ instance will release its memory around 450s and the migration overhead of around 30s can be avoided.

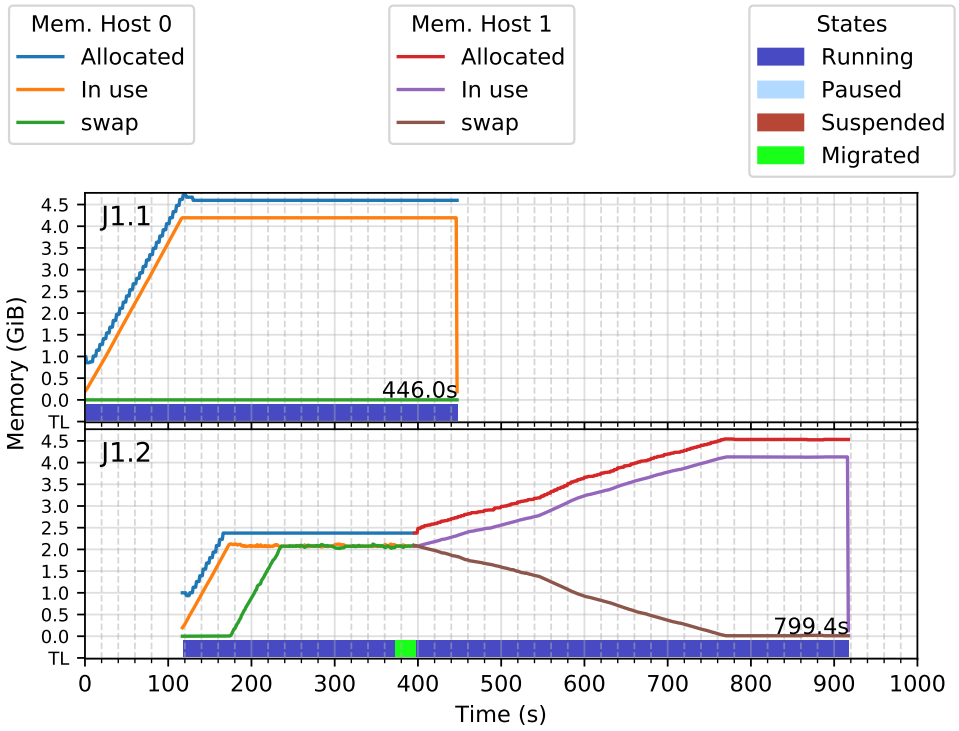

Figure 4.7: Execution time line under MRVM with $t_h = 360$ seconds.

Figures 4.7 and 4.8 present an execution time line of the job instances for $t_h = 360$, including the state of the VMs and the amount of memory allocated to each, the amount memory actually being used by each VM as well as the VM's respective swap consumption, over time, under the MRVM and MECv2 policies. In both scenarios, as the first instance of J1 has enough memory, it will complete its execution unaffected by the other job around 450s. As stated earlier, job $J1.2$ takes around 60s to consume the remaining host memory and then is forced to swap out pages to make room for its remaining data (from time 180s to 240s). Note that there is no apparent slowdown as the memory consumption rate during this period is the same as that between 120s and 180s and the same as job $J1.1$.

While in Figure 4.7, MRVM must let $J1.2$ continue to swap in and out pages between memory and disk from time 240s onwards until it can migrate the VM to another host, MECv2 is able to prevent further performance degradation by pausing the VM until there

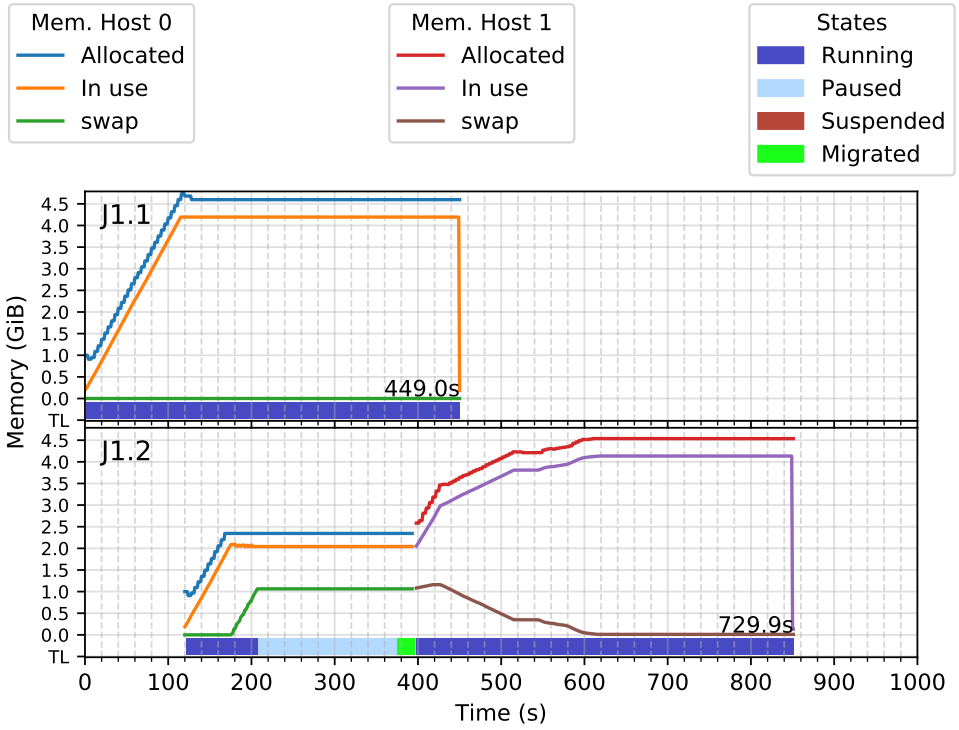

Figure 4.8: Execution time line under MECv2 with $t_h = 360$ seconds.

is enough memory. Since there is no other VM needing memory, $J1.2$ does not need to be suspended. At time 360s, MEMiC notifies MEC of the availability of a suitable server and both schemes migrates the second job. While MRVM employs live migration, MECv2 having paused the VM will use cold migration and then resume the execution of the VM on the target host. Both migrations operations take approximately the same amount of time. However, the fact that MECv2 paused the VM before the job started simultaneously swapping in and out pages meant it was able to make use of the free memory quicker. The reason for the relative slowness to copy pages from swap to memory (from 400s to 780s in Figure 4.7) is not all that clear and will be investigated in greater depth in future work. Two possible causes come to mind. First note that pages in swap are stored on disk and, in the case of virtualization, in order to support fast migrations, VM images are usually held on a remote server whose file system is visible to the host servers so the physical time consuming copying of VM disk images can be avoided. The overhead may also be impacted by the storage technology used to share the virtual disks between the hosts as well as its location in the environment. The second reason may be related to MEC's own behaviour when using pages in swap. Although, it appears that sufficient memory is being allocated to the VM, it may be that the job doesn't actually "ask for memory" when recovering pages from swap in the same way as it allocates memory at the beginning of its execution since its data has already been allocated. In any case, this

scenario shows that relying solely on live migration (MRVM version) may not always the best solution.

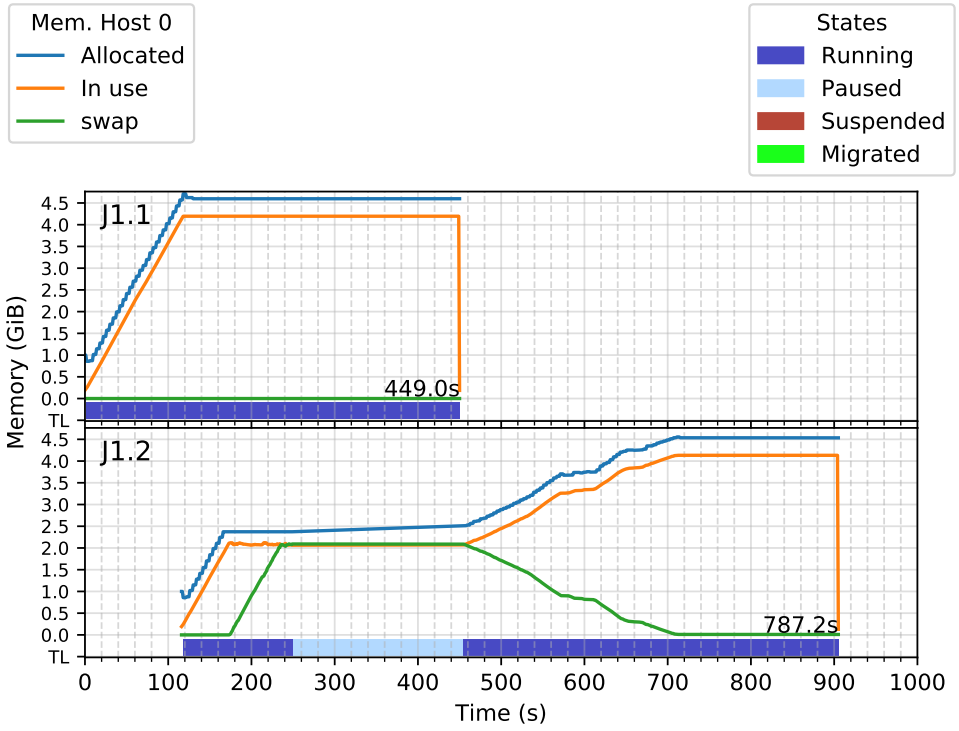

Figure 4.9: Execution time line under MEC management without migration (No MIG)

The motive for MEC to instinctively pause a VM when it begins to exhibit a particular swap usage characteristic comes from the dramatic performance degradation that occurs when there is a shortage of memory, e.g. see the Memory-Performance Profiles of $J1$ and $J2$ in Figure 4.1. Pausing a VM is a temporary preemptive action taken while the system looks for memory to satisfy the job's memory requirements. The next experiment follows the same setup described earlier in this experiment, but this time with $t_h > 450$ so that migration would not be beneficial since the additional server will only be available after the first job completes. Figure 4.9 presents the execution time line of the same two job workload, where, as expected, MEC chooses to pause the VM until $J1.1$ completes.

Now compare this to the execution time line presented in Figure 4.10, which is from a modified version of MEC where the preemption mechanisms were disabled (equivalent to the scheme No Preemp). As can be seen, this execution is around 77 seconds faster than No MIG, even though $J1.2$ was left to execute using swap. In Figure 4.9 , MEC detected SISO and paused $J1.2$ for 200s, while in this figure, $J1.1$ executed for 210 seconds thrashing memory pages. Nevertheless, the job made some non-insignificant progress and, albeit running at a slower rate, it can be estimated to be about 36% of its regular speed based

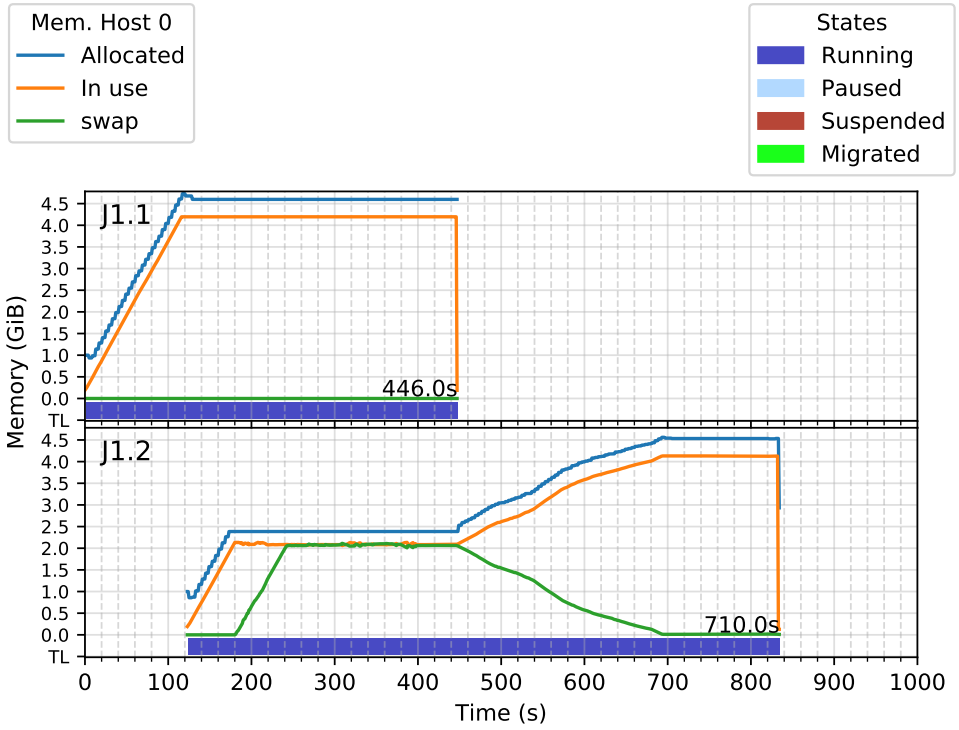

Figure 4.10: Execution time line under MEC management without any preemption mechanism (No Preemp)

on the profile shown in Figure 4.1, i.e. approximately 76 seconds of progress. What these results show is that preemption decisions should also be taken in the context of job scheduling. The previous experiment showed that if you don't have to wait for a spare resource, migrate directly, but if not, it is better to pause the VM first. However, in this experiment, it is better to not even pause the VM. All in all, the correct decision depends a series of scheduling details, e.g. job execution times, job start times, memory and host availability, etc.

For simplicity, the current MEC architecture is reactive, iteratively using recent history to predict the near future requirements of each job without any information regarding the jobs being managed. When manipulating multiple preemption options, the complexity of the scheduling problem grows. Recent approaches, like the one presented in [19], focus on the global scheduling of jobs using memory profiles to enable MEMiC to choose job allocations that avoid unnecessary preemption events. Future work will investigate what information should be forwarded by MEMiC when submitting a job to MEC.

Nevertheless, while it appears that MEC made an incorrect scheduling decision in choosing to migrate a VM, note that this subsection has focused on the scenario where only one job (or a subset of jobs) was starved of sufficient memory. By the time the second

job started, the first already had been allocated a sufficient amount of memory to execute normally. The next subsection presents a scenario where both jobs start at almost the same time and therefore will have to compete for the available memory between them.

4.3.3 Preemption scheduling with competing jobs

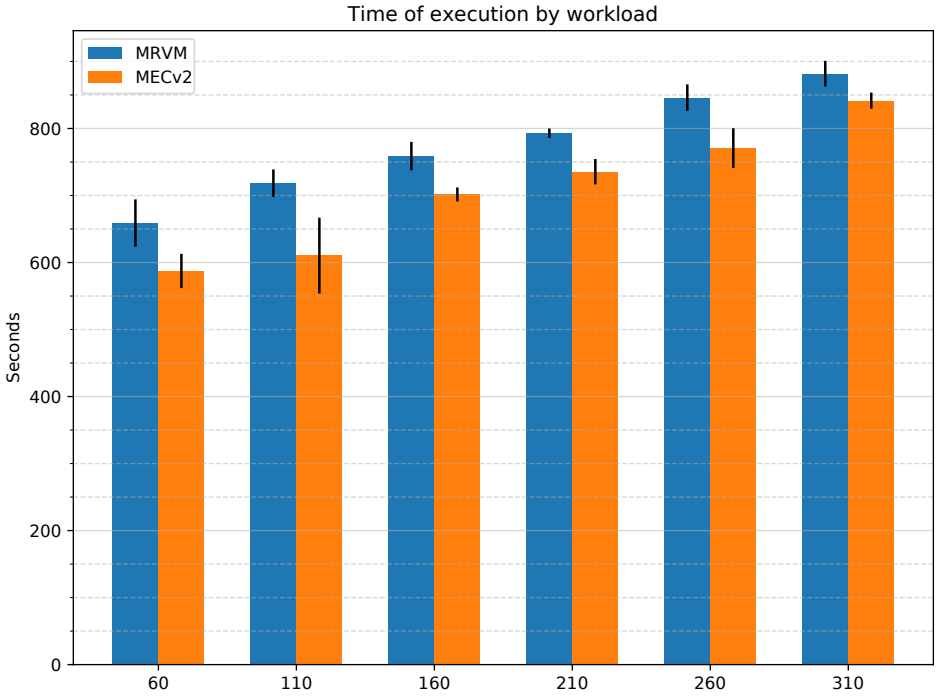

Figure 4.11: Comparison of the MRVM and MEC policies for two $J1$ job instances started concurrently, with different values of t_h

Given the performances presented in Figure 4.1, another issue related to VM scheduling is deciding how much memory should be allocated to each job, especially when resources are scarce. The setup of this experiment consists of executing two $J1$ jobs, launched concurrently, on a server without enough memory to execute both jobs simultaneously, so the $5GiB$ of free memory it does have will have to be shared between them. Figure 4.11 shows the execution times with values of $t_h = 60, \ldots, 310$ in steps of 30 seconds. In all cases, executions with MEC obtain better performance. These results are also quite close to the ideal makespan (apart from the swap usage) given that the first job executed alone on the first host and the second job executed in the second host as soon as it became available.

Figures 4.12 and 4.13 compare time lines with $t_h = 60$ seconds under MRVM and MEC respectively. Due to the hypervisor, the VMs are launched sequentially, the second trailing the initialization of the first by around 10s. In both scenarios, the two jobs each

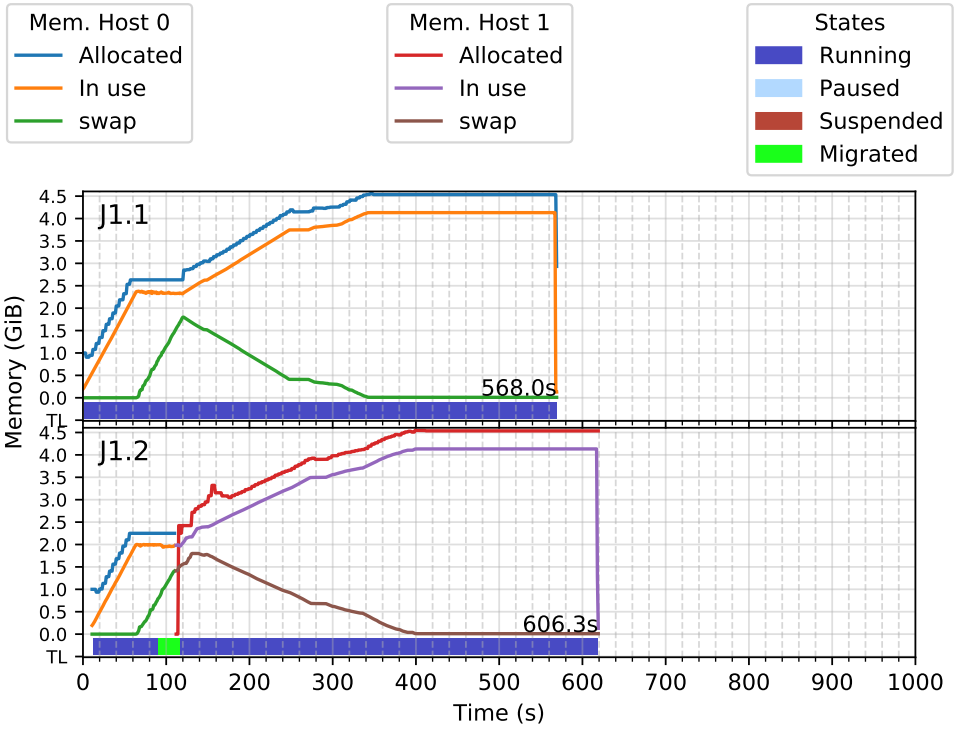

Figure 4.12: Execution time line under MRVM, $t_h = 60$.

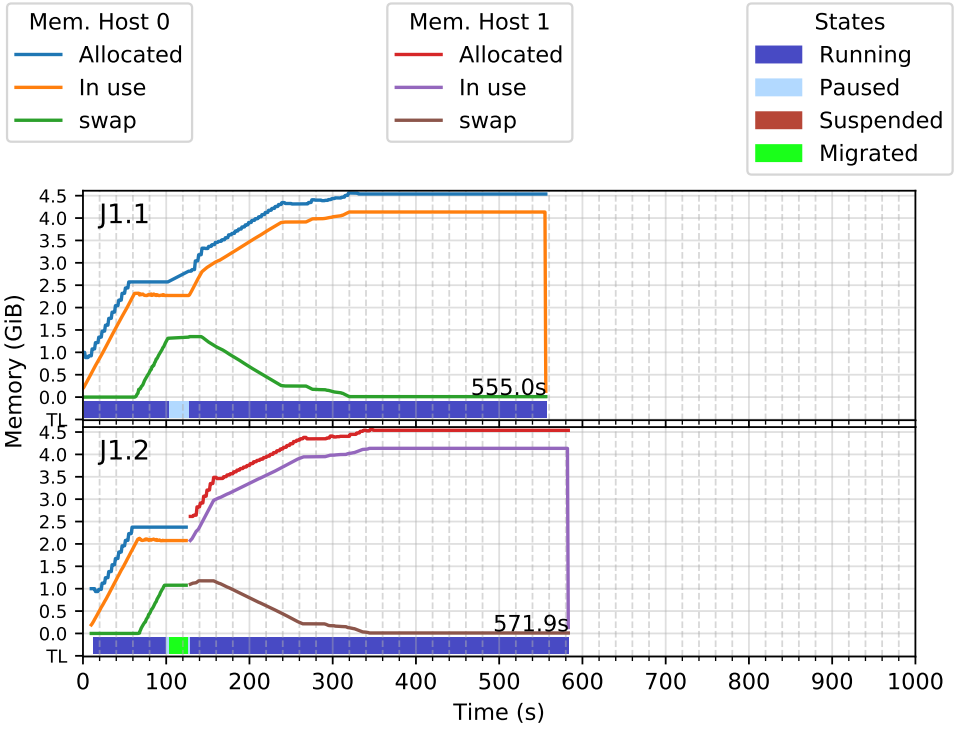

Figure 4.13: Execution time line under MEC, $t_h = 60$.

manage to consume around half the available memory and swap usage begins at around time 70s. While MRVM migrates the second VM immediately, MEC first pauses both

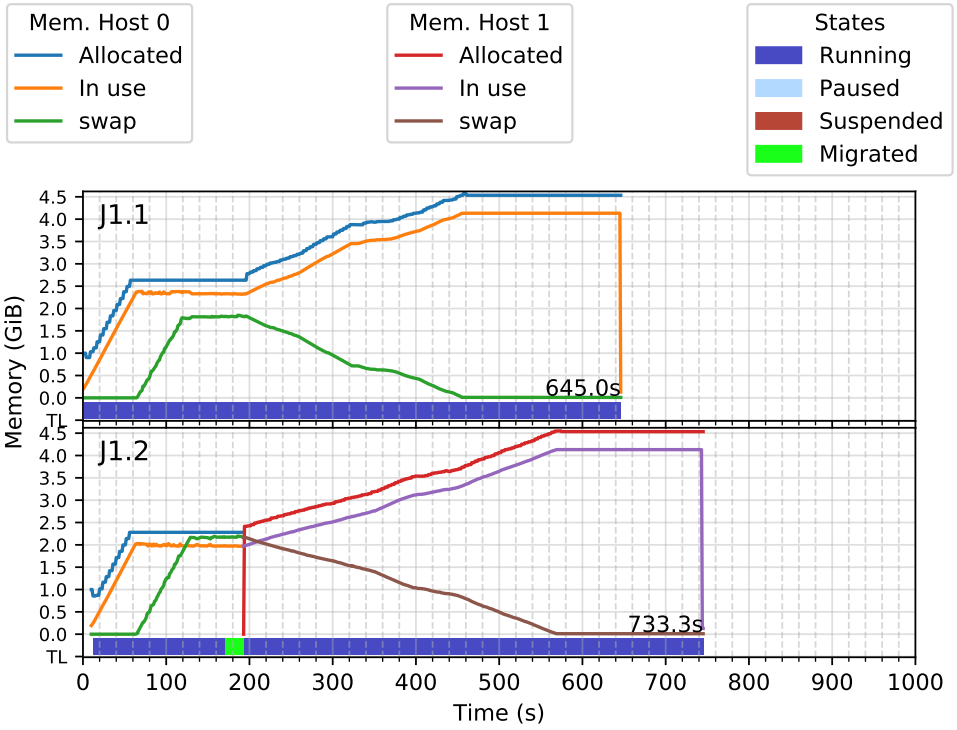

Figure 4.14: Execution time line under MRVM, $t_h = 160$.

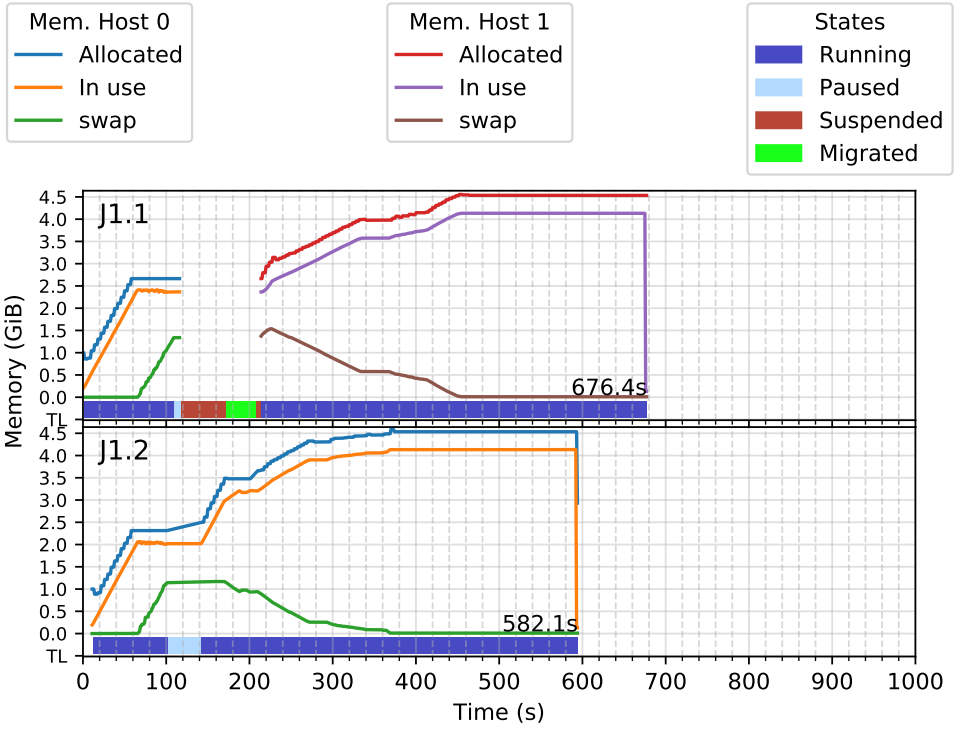

Figure 4.15: Execution time line under MEC, $t_h = 160$.

VMs and then migrates one of them, resuming it after the migration is completed. Pausing the VMs prevents further swap consumption by the VMs (less than 1.5 GiB compared to

2 GiB) which benefits their performances later on when more memory is available.

For values of t_h higher than 120s, MEC will suspend a paused VM to free up it's memory for the other VM. This also means that MEC will not only have to be able to migrate VMs in a running or paused state, but also VMs in suspended state, i.e. MEC supports three different forms of migration, as described in Section 3.3. This is quite effective, as shown in Figure 4.15, for $t_h = 160$s, where the first VM, after being paused and then suspended, is migrated off line to the other server at time 170s. The VM arrives at the new host in a suspended state and has to be restored to an executing state. While the first VM is suspended, $J1.2$ remains paused until $J1.1$'s memory has been returned to the host for reallocation at time 140s. MRVM, in comparison to Figure 4.14, behaves similarly to the execution with $t_h = 60$s, except that the migration can now only occur at time 170s. In the meantime, the two jobs continue to execute and consume more swap. So even when one of the VMs is later migrated, the job execution times under MRVM are longer than those under MEC management even though the MEC jobs were paused and suspended, effectively stalled, for over 150s against only 20s for MRVM's jobs.

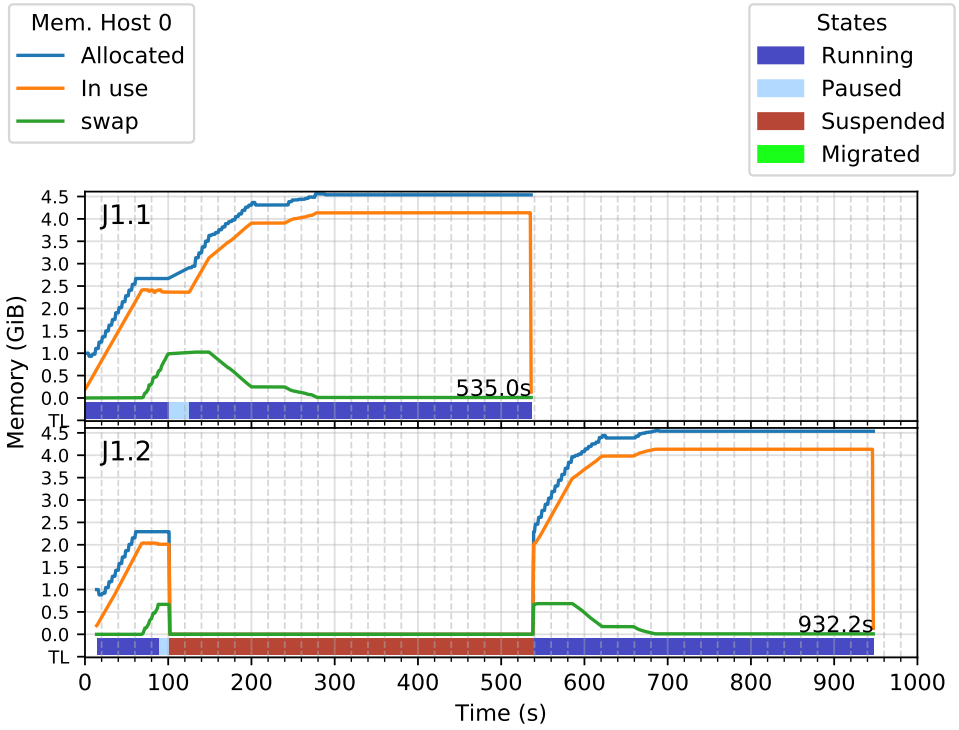

Figure 4.16: Execution time line under MEC management with $t_h > 1000s$, the equivalent of No MIG.

Time lines for the job executions under the policies of NO MIG (with MEC) and No Preemp, when no additional servers are available, are shown in Figures 4.16 and 4.17 respectively. For NO MIG, as seen previously, both jobs consume the available memory

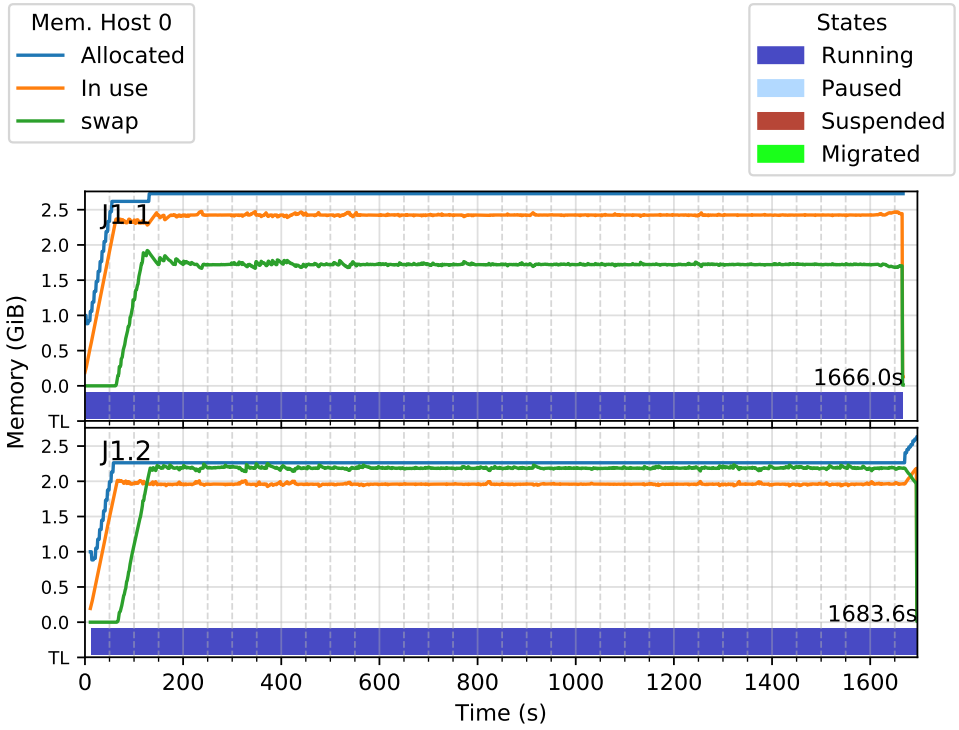

Figure 4.17: Execution time line with memory elasticity but without any preemption mechanisms (the equivalent to No Preemp).

and begin to use swap at around time 70s. At this point, MEC pauses the VM with least memory and after 10 seconds suspends and saves it to disk while the other VM is paused until the suspension process terminates. When the freed memory is available, MEC resumes the paused VM. Then, once $J1.1$ finishes its execution, the suspended VM is restored. The importance of MEC is highlighted by the comparison with time line for traditional migration schemes shown in Figure 4.17. Without any preemption mechanism and an additional host server being available , the two jobs will execute using swap with a significant degradation in performance. The execution time is 80% slower than MEC with non-migration preemption and can be as much as three times slower than migration-based preemption.

4.4 Summary

This chapter aimed to compare different preemption and migration schemes and motivate the need to consider more than one option in order to efficiently exploit vertical memory elasticity. The experiments in Section 4.3 consider scenarios where the job's increasing memory demands cannot eventually be met by their host server and different schemes that combine migration and other preemption mechanisms are evaluated. The first experiment

showed that when a spare host (or another host with sufficient spare capacity) is available when a host becomes overloaded, traditional live migration presents good results but those produced by MEC by combining VM pausing and migration (including cold migration) is equally competitive. Having a second host available immediately may not always be possible in real practical situations. The last two sets of experiments again consider similar conditions but this time focus on analyzing how the performance (execution time of the jobs and the workload as a whole) is affected by the time at which, if at all, a second host becomes available. The results showed that scenarios exist where it is better to not even use any form of preemption, even live migration. Furthermore, when multiple jobs have insufficient memory, combining preemption techniques including VM suspension to redistribute allocated memory can be significantly better than migration-only management approaches.

Chapter 5

Conclusions and future work

Maximizing resource utilization is a frequent goal of service providers in cloud and edge computing. Server consolidation and resource elasticity are two common features of resource management. The former attempts to improve resource utilization by reducing or minimizing the number of servers for a given workload. The latter aims to obtain utilization gains by trying to exploit the time-varying demands of applications during execution. Vertical memory elasticity can be used as a effective method to assign sufficient memory to VMs dynamically while allowing cloud providers to instantiate more VMs on a physical node without the risks of over-provisioning. However, when host memory becomes scarce, proactive resource management is required to avoid severe performance degradation.

Traditionally, VM migration has been used extensively as a mean to improve performance, and in the context of cloud computing, is a fundamental tool to support elasticity. This work presents the MEMiC framework and extend the original Memory Elasticity Controller (MEC) that was minded for single server memory management and provided the memory vertical elasticity by adjusting the amount memory allocated to each VM according to its demand and only two VM preemption schemes (pause and suspension), aiming to avoid performance loss from over-provisioning while improving utilization. The modified version of MEC developed in this work combines the use of three preemption schemes – VM pausing, suspension and migration – into a coordinated local scheduling policy for a more effective management of VMs with memory elasticity as part of a multiple server hierarchical framework. This dissertation discussed some of the design issues and evaluated the approach against alternatives within this common framework. The proposal is at least competitive with traditional live migration-only-based approaches, being significantly better in some scenarios.

Process migration has been used in various systems as a mean to improve perfor-

mance in spite of the known overheads. The main goal of this work has been to carefully incorporate VM migration in to the management of the execution of multiple VMs in a shared environment that supports vertical and horizontal elasticity with the aim of reducing the negative effects of overloaded resources on the overall execution time. For both VM migration or suspension experimental analysis guided us towards choosing the VM with the smallest amount of allocated memory. While, on one hand, this decision results in a smaller amount of memory being released per operation, on the other hand, it incurs a lower overhead in terms of latency and throughput by releasing memory on the host sooner. Different migration mechanisms were analyzed and while live migration has the least impact on the running VM, giving MEC the opportunity to also consider migrating either paused or suspended VMs can lead to better results.

VM scheduling with vertical memory elasticity, VM pausing, suspension and live migration could become a key tool as cloud service providers continue to strive to improve resource utilization for an ever diverse range of applications. Future work aims to refine the MEC scheduling policies and decisions in relation to multiple preemption options as well as study the multi-level scheduling problem where the predictive MEMiC Scheduler must interact with multiple reactive MEC controllers.

5.1 Future work

The outcomings of this work can be used to include energy-aware scheduling policies that can help to reduce energy consumption by improving server consolidation in virtualized data centers. Furthermore, in order to take advantage of idle resources when online jobs have low demand, online jobs and batch jobs can be distinguished and, depending on the load, dynamically re-prioritized to improve utilization. Supporting both online and batch jobs on the same infrastructure is particular useful in in fog and edge computing environments where resource capacities are general limited.

In the context of vertical memory elasticity, the global scheduling problem being address in this work is more complex than the problem where jobs are executed on dedicated resources. As such, many of the MEC scheduling decisions have yet to be studied fully, such as which VM should be chosen for a particular action, e.g. migration or suspension, or at what frequency actions should be taken, e.g. recover memory from running VMs. In the same manner that the VM migration policy was briefly analyzed in this work, a refinement of the policy to suspend paused VMs might also be merited. Would suspending

the paused VM(s) with smallest quantity memory be a better approach, since it would release the memory in a shorter period of time, than to suspend a single VM of sufficient size? These are just some of the pending issues related to MEC scheduling algorithm.

Note also, however, that preemption techniques can be costly and should be employed with care. In order to reduce the number of such events but still avoid overloading the host, it is important to find an effective initial VM allocation that takes into consideration the behavior of each job. To achieve this, any global scheduler must work in unison with the local host elasticity controllers to reduce interference (performance degradation). Existing approaches address the problems of elasticity and allocation separately. But recent work carried in parallel to the work of this dissertation has focused on the elaboration of the MEMiC (Memory Elasticity Management in Clouds) framework into a two-tier VM scheduler for batch jobs [19]. The global scheduler of the cloud manager attempts to predict the impact caused by competition for the memory of physical servers in shared Cloud-like environments in order to better harness VM allocation, suspension and migration. Future work should focus in refining the interaction between the schedulers in this two level hierarchy. One factor that could have a significant impact on local scheduling might the use of job specific information such as its ideal execution time. Given that MEMiC scheduler uses job profile information and simulations to predict interference when allocation jobs to servers, it should be possible to pass on job information to individual MEC controllers to help improve local decision making.

Although this work has focused on VM scheduling, the concepts presented here can equally be applied to container-based systems. Despite such systems have inbuilt mechanisms to handle the elastic allocation of resources, scheduling decisions still have to be made as to where containers should be allocated and how much of the limited resources should be allocated to a given container at a given time.

Another practical issue that affects performance, but was not treated in this work, is the choice of storage technology used to share the VM virtual disks. As disk files are located on a remote storage server to facilitate migration, the swap IO carried out over network may be subject to external interference. The computational environment used for the experimental analysis in this work was *NFS*, however, there exists other technologies like *glusterfs* that might be able to reduce the impact. This impact is more noticeable when a VM that was using swap is then given memory, the rate to recover swap to memory is significantly slower than writing to either memory or swap.

References

[1] A., S.; B., H. Pre-copy and post-copy vm live migration for memory intensive applications. In *Euro-Par 2012: Parallel Processing Workshops.*, vol. 7640 of *LNCS*. Springer, 2012, pp. 539–547.

[2] Ahmad, B.; Yazidi, A.; Haugerud, H.; Farokhi, S. Orchestrating resource allocation for interactive vs. batch services using a hybrid controller. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)* (May 2017), pp. 1195–1203.

[3] Ahmad, R. W.; Gani, A.; Hamid, S. H. A.; Shiraz, M.; Yousafzai, A.; Xia, F. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of Network and Computer Applications 52* (2015), 11 – 25.

[4] Al-Dhuraibi, Y.; Paraiso, F.; Djarallah, N.; Merle, P. Autonomic vertical elasticity of docker containers with ELASTICDOCKER. In *10th IEEE International Conference on Cloud Computing (CLOUD 2017)* (Jun 2017), pp. 472–479.

[5] 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 PP*, 99 (2017), 1–18.

[6] Al-Dhuraibi, Y.; Zalila, F.; Djarallah, N. B.; Merle, P. Coordinating Vertical Elasticity of both Containers and Virtual Machines. In *8th International Conference on Cloud Computing and Services Science - CLOSER 2018* (Funchal, Madeira, Portugal, Mar. 2018).

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

[8] Bradford, R.; Kotsovinos, E.; Feldmann, A.; Schiberg, H. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (2007), pp. 169–179.

[9] Burns, B.; Grant, B.; Oppenheimer, D.; Brewer, E.; Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (Jan. 2016), 10:70–10:93.

[10] Campbell, E. *Apache Mesos Basics*. CreateSpace Independent Publishing Platform, USA, 2017.

[11] Choudhary, A.; Govil, M. C.; Singh, G.; Awasthi, L. K.; Pilli, E. S.; Kapil, D. A critical survey of live virtual machine migration techniques. *Journal of Cloud Computing 6*, 1 (Nov 2017), 23.

[12] Clark, C.; Fraser, K.; Hand, S.; Hansen, J. G.; Jul, E.; Limpach, C.; Pratt, I.; Warfield, A. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 273–286.

[13] 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.

[14] Edler, J.; Lipkis, J.; Schonberg, E. *Process management for highly parallel UNIX systems*. Citeseer, 1988.

[15] 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 Systems 65* (2016), 57–72.

[16] Farokhi, S.; Jamshidi, P.; Lucanin, D.; Brandic, I. Performance-based vertical memory elasticity. In *IEEE International Conference on Autonomic Computing* (2015), pp. 151–152.

[17] Hines, M. R.; Deshpande, U.; Gopalan, K. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev. 43*, 3 (July 2009), 14–26.

[18] Jette, M. A.; Yoo, A. B.; Grondona, M. SLURM: Simple Linux Utility for Resource Management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003* (2002), Springer-Verlag, pp. 44–60.

[19] Klôh, H. *Job Scheduling in Elastic Virtualized Environments (in Portuguese)*. Tese de Doutorado, Pós-graduação em Computação, Univ. Federal Fluminense, 2018.

[20] Libvirt virtualization api. `https://libvirt.org/`. Accessed: 2018-06-29.

[21] Moltó, G.; Caballer, M.; Alfonso, C. Automatic memory-based vertical elasticity and oversubscription on cloud platforms. *Future Generation Computer Systems 56* (2016), 1 – 10.

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

[23] Moreno-Vozmediano, R.; Montero, R.; Llorente, I. M. IaaS cloud architecture: From virtualized datacenters to federated cloud infrastructures. *IEEE Computer 45*, 12 (2012), 65 – 72.

[24] Pahl, C.; Brogi, A.; Soldani, J.; Jamshidi, P. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* (2017), 1.

[25] 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.

[26] QEMU Features postcopy live migration. `https://wiki.qemu.org/Features/PostCopyLiveMigration`. Accessed: 2018-02-14.

[27] Sawamura, R. Improving application throughput by exploiting vertical memory elasticity in virtualized environments. Master's thesis, Pós-graduação em Computação, 2015.

[28] 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.

[29] 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.

[30] Sedaghat, M.; Hernandez-Rodriguez, F.; Elmroth, E. A virtual machine repacking approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference* (New York, NY, USA, 2013), CAC '13, ACM, pp. 6:1–6:10.

[31] Sotiriadis, S.; Bessis, N.; Amza, C.; Buyya, R. Vertical and horizontal elasticity for dynamic virtual machine reconfiguration. *IEEE Transactions on Services Computing PP*, 99 (2016), 1–1.

[32] Staples, G. TORQUE Resource Manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[33] Voorsluys, W.; Broberg, J.; Venugopal, S.; Buyya, R. Cost of virtual machine live migration in clouds: A performance evaluation. In *Cloud Computing* (Berlin, Heidelberg, 2009), M. G. Jaatun, G. Zhao, and C. Rong, Eds., Springer Berlin Heidelberg, pp. 254–265.

[34] Yi, S.; Li, C.; Li, Q. A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data* (New York, NY, USA, 2015), Mobidata '15, ACM, pp. 37–42.