Virtualization Techniques for Memory Resource Exploitation

Luis Angel Garrido Platero

Advisor: Paul Carpenter
Co-advisor: Rosa María Badía
Computer Architecture Department

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Doctor of Philosophy

Acta de qualificació de tesi doctoral **Curs acadèmic: 2018-2019**

Nom i cognoms: Luis Angel Garrido Platero
Programa de doctorat: Arquitectura de Computadors
Unitat estructural responsable del programa: Departament d'Arquitectura de Computadors

Resolució del Tribunal

Reunit el Tribunal designat a l'efecte, el doctorand exposa el tema de la seva tesi doctoral titulada Virtualization Techniques for Memory Resource Exploitation.

Acabada la lectura i després de donar resposta a les qüestions formulades pels membres titulars del tribunal, aquest atorga la qualificació:

☐ NO APTE ☐ APROVAT ☐ NOTABLE ☐ EXCEL·LENT

(Nom, cognoms i signatura) President/a	(Nom, cognoms i signatura) Secretari/ària

(Nom, cognoms i signatura) Vocal	(Nom, cognoms i signatura) Vocal	(Nom, cognoms i signatura) Vocal

Barcelona, _____ d'/de _____ de _____

El resultat de l'escrutini dels vots emesos pels membres titulars del tribunal, efectuat per l'Escola de Doctorat, a instància de la Comissió de Doctorat de la UPC, atorga la MENCIÓ CUM LAUDE:

☐ Sí ☐ NO

(Nom, cognoms i signatura) President de la Comissió Permanent de l'Escola de Doctorat	(Nom, cognoms i signatura) Secretari de la Comissió Permanent de l'Escola de Doctorat

Barcelona, _____ d'/de _____ de _____

Abstract

Cloud infrastructures have become indispensable in our daily lives with the rise of cloud-based services offered by companies like Facebook, Google, Amazon and many others. The e-mails we receive everyday, the files and pictures we upload/view to/from social media and all the different market-places through which we make purchases daily rely in some way on services offered by a cloud infrastructure somewhere.

These cloud infrastructures use a large numbers of servers provisioned with their own computing resources. Each of these servers use a piece of software, called the Hypervisor ("HV", for short), that allows them to create multiple virtual instances of a subset of their physical computing resources and abstract them into "Virtual Machines" (VMs). The hypervisor multiplexes among the VMs some of the physical resources of the server like CPU and I/O interfaces. But other resources, like memory, are allocated because they cannot be multiplexed over time in the same way.

A VM for cloud-based services runs an Operating System (OS), which in turn runs the applications used by the service consumers. The administrator of the cloud infrastructure is responsible for managing the VMs. The management tasks involve determining the resources that the VMs can use, in which physical node they can execute and any other constraint the administrator deems reasonable to enforce, such as Service Level Agreements (SLA) [50, 22].

The VMs within the computing nodes generate varying memory demand behavior. When memory utilization reaches its limits due to this demand, costly operations such as (virtual) disk accesses and/or VM migrations can occur. As a result, it is necessary to optimize the utilization of the local memory resources within a single computing node.

However, pressure on the memory resources can still increase, making it necessary to move (migrate) the VM to a different node with larger memory or add more memory to the same node. At this point, it is important to consider that some of the nodes in the cloud infrastructure might have memory resources that they are not using. So, this opens up the possibility of making this memory available to the nodes that actually need it, avoiding costly migrations or hardware memory enhancements.

Considering the possibility to have a node's memory made available to the rest of the nodes in the infrastructure, new architectures have been introduced that provide hardware support for a shared global address space across multiple computing nodes. Together with fast interconnects, these architectures enable the

sharing of memory resources across computing nodes while enforcing coherence locally within nodes. In this context, every node is a coherence island and the memory is a global resource.

This thesis presents multiple contributions to the memory management problem. First, it addresses the problem of optimizing memory resources in a virtualized node through different types of memory abstractions. Two full contributions are presented for managing memory within a single node called SmarTmem and CARLEMM (Continuous-Action Reinforcement Learning for Memory Management). In this respect, a third contribution is also presented, called CAVMem (Continuous-Action Algorithm for Virtualized Memory Management), that works as the foundation for CARLEMM.

SmarTmem is designed to optimize the memory made available through the VMs through the Transcendent Memory (Tmem) abstraction, while CARLEMM targets the optimization of memory made available through ballooning. The initiative behind CARLEMM is initially based on a proof-of-concept mechanism called CAVMem that provided insight on the use of Continuous Action Reinforcement Learning (RL) algorithms for memory management. To the best of our knowledge, CARLEMM and CAVMem are the first mechanisms to employ CARL for the problem of memory management within a virtualized node.

Second, this thesis presents two contributions for memory capacity aggregation across multiple nodes, offering two mechanisms called GV-Tmem (Globally Visible Transcendent Memory) and vMCA (Virtualized Memory Capacity Aggregation), this latter being based on GV-Tmem but with significant enhancements. These mechanisms focus primarily on distributing the memory pool of the infrastructure across the computing nodes using a user-space process with high-level memory management policies.

To summarize, the main contributions of this thesis work are:

- A mechanism called SmarTmem for the optimal allocation of Tmem in a single virtualized node.
- A mechanism called CAVMem (Continuous Action Algorithm for Virtualized Memory Management) developed as a proof-of-concept for the use of Continuous-Action Reinforcement Learning for RAM allocation in a *simulation* of a virtualized node
- A mechanism called CARLEMM (Continuous Action Reinforcement Learning for Memory Management), based on the proof-of-concept provided by CAVMem, but enhanced to solve the memory allocation problem in a *real* computing node.
- A mechanism called GV-Tmem (Globally Visible Tmem) for remote memory capacity sharing across multiple computing nodes in a cloud infrastructure
- A mechanism called vMCA (virtualized Memory Capacity Aggregation) for the sharing of memory capacity across nodes, but enhanced for resiliency and efficiency in comparison to GV-Tmem.

Acknowledgements

I would like to thank my advisor Paul Carpenter, which throughout this time has transmitted to me a lot of knowledge regarding our research field and a lot of knowledge on the methodology of research. I would like to thank him for his time and dedication to our work. I would also like to thank Rosa Badia, my co-advisor, for being accessible in par with Paul during all this time. I would also like to thank Rajiv Nishtala for his contributions during the development of this thesis, and for his thoughtful advice.

This research has received funding from the European Union's 7th Framework Programme (FP7/2007–2013) under grant agreement number 610456 (Euroserver) and the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 754337 (EuroEXA). The research was also supported by the Ministry of Economy and Competitiveness of Spain under the contract TIN2012-34557, the European UniHiPEAC-3 Network of Excellence (ICT-287759), and the FI-DGR Grant Program (file number 2016FI_B 00947) of the Government of Catalonia.

Contents

List of Figures

LIST OF FIGURES

List of Tables

Chapter 1

Introduction

This thesis starts off by explaining what Cloud Computing is, how it came to be and its current state of the art. The purpose of this chapter is to give the reader a clear vision of the context in which this research was developed, and the background to comprehend the scope of our contributions, which mainly focus on the problem of memory management.

Cloud Computing is increasingly becoming more relevant, despite its ups and down on the market and the over-hype that always comes when new technologies get mass adoption. It is a market that changes rapidly, as new services come and go and as new capabilities get exploited. However, the fundamental technologies that gave rise to Cloud Computing have been around for decades and they have steadily evolved to produce the computer landscape we see today.

In this research, we try to look beneath the surface without losing sight of it, we try to look past the hype of the markets and dive deeper into the fundamentals of the technology. This is why we use terms like "Virtualization Techniques", "Coherence Islands" and "Memory Resource", all of which is terminology that will be very unlikely to appear on a presentation from a motivational speaker or a marketing specialist. But using this terminology already gives the reader an idea of the issues we are addressing and how they relate to the underlying technology that enables Cloud Computing.

1.1 Cloud Computing in Context

Cloud Computing infrastructures have become indispensable in our daily lives with the rise of cloud-based services offered by companies like Facebook, Google, Amazon and many others. The e-mails and text messages we receive everyday, the files and pictures we upload/view to/from social media and all the different market-places through which we make purchases daily rely in some way on services offered by a cloud infrastructure. Figure 1.1 shows the way cloud computing services market has increased since 2009 until 2018, and it also shows growth estimations from 2019 until 2022.

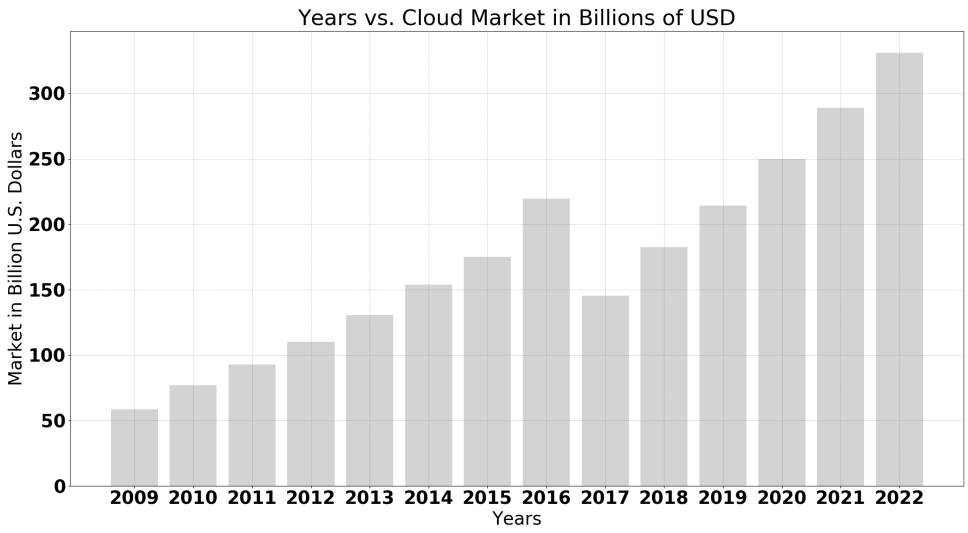


Figure 1.1: Yearly Cloud Market Growth since 2009 until 2018, and estimations of growth from 2019 until 2022. Source of data from [1].

The Cloud Computing market has been growing steadily for the last 10 years, and it is expected to continue growing in the years to come. Cloud Computing services provide many advantages, that basically translate in a cost reduction for online service providers and a better user experience for their consumers. It also makes management of this services and their data a lot easier, since it allows for a higher level of abstraction and automation.

But the questions remain: what is a Cloud? What is a Cloud Infrastructure? Basically, a cloud infrastructure is a group of computer servers (nodes) that *pool* together their hardware resources (CPU, memory, storage, communication interfaces, etc.) to execute applications on behalf of the users in a transparent way. The applications are executed as if they were running on a single computer with monolithic computing resources.

In the cloud infrastructure, every node has a software stack that provides the necessary software support, not just for the pooling of resources, but also to execute the applications required by the customers/users. This software stack (also called server stack) has some similarities with the software stacks used for nodes in other types of distributed systems, with one fundamental difference: the presence of the Virtualization Layer. Figure 1.2 shows a common software stack for a computing node in both cases.

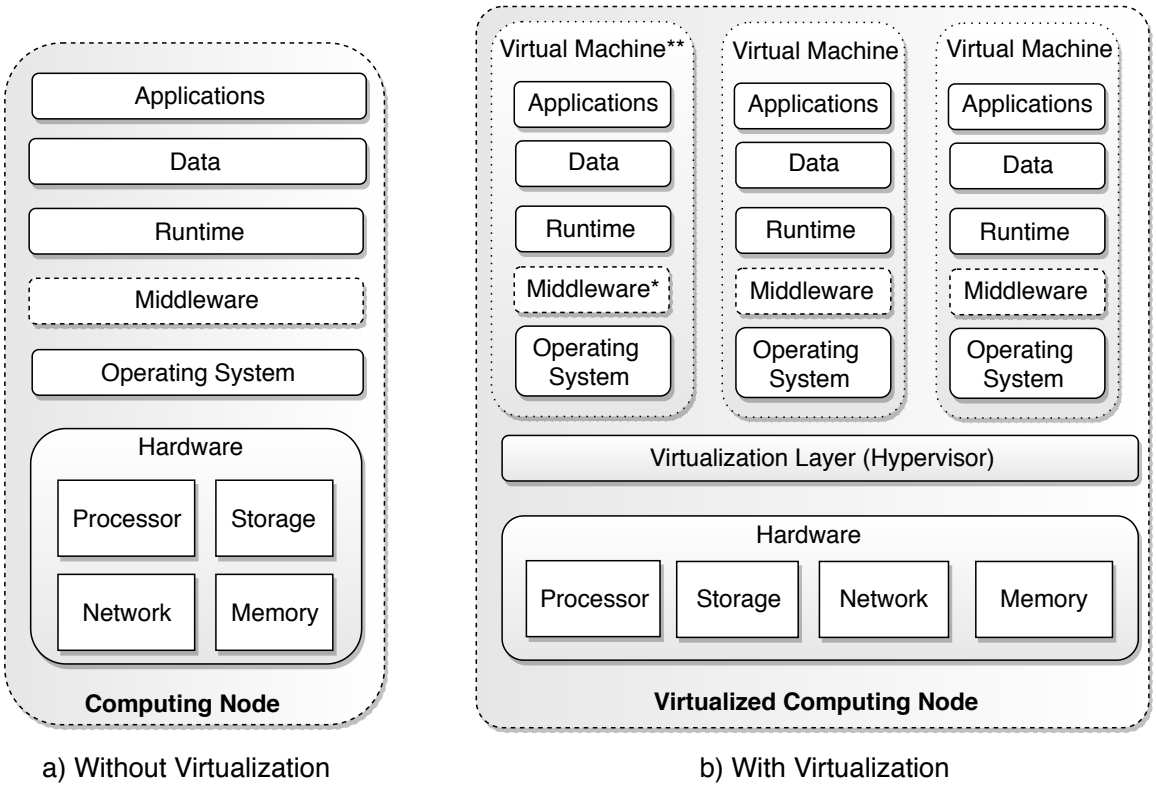


Figure 1.2: Server stack for the cases in which a) there's no virtualization layer, and b) with a virtualization layer.

When a node is not virtualized (Figure 1.2(a)), every application executed could potentially access all of the resources of the node. There are ways to manage the resource allocation among the applications from within the Operating System (OS), but such environment does not provide the same runtime flexibility and security that a virtualized node (Figure 1.2(b)) can. As it will be explained in Section 1.2.2, when the node is virtualized, its resources can be packed together into Virtual Machines or VMs, that are able to execute their own OSs and different runtime environments, all within the same physical node. Notice how every "Virtual Machine" in Figure 1.2(b) has the same internal software stack of the non-virtualized node, starting from the "Operating System"

In a virtualized node, the component tagged as "Middleware*" is singled out. This is because in many server stack implementations, one of the VMs may have a middleware with capabilities to interact with the virtualization layer in order to manage its utilities. This middleware is also able to interact with other nodes to pool and share resources across the cloud infrastructures, but this also depends on the type of server stack implemented.

The sharing of resources is achieved through the deployment of virtualization technology [113], which consists of a set of specialized hardware and software features present on each computing node. The computer node needs to have a processor with certain characteristics [85] in order to execute virtualization software, the most important of this software is commonly known as a Virtual Machine Manager or Hypervisor (HV), also shown in Fig-

ure 1.2(b).

Computing resource sharing thus occurs among the VMs within a single node of the infrastructure, as well as across the nodes. There are many mechanisms through which nodes can share their resources, depending on the hardware capabilities of the node and the details of the runtime and middleware. There are state-of-the-art server architectures that have been designed with a share-something design approach, enabling nodes to share computing resources through their memory address spaces. Such architectures, like Euroserver [26] and ExaNoDe [89], rely on fast interconnects and fast communication protocols among the nodes to achieve this. By leveraging the characteristics of these new hardware architectures, it is possible to *disaggregate* the memory-mapped resources of every computing node in a cloud, and make those resources accessible to every other computing node, by allowing the flexible *aggregation* of memory to the nodes.

This thesis presents a series of contributions to the memory management problem. First, this thesis addresses the problem of memory resource allocation in a single virtualized node through different types of memory abstractions. In this context, two different mechanisms for managing memory, SmarTmem and CARLEMM, are introduced, the latter using a reinforcement learning approach based on the foundation provided by CAVMem, which is another contribution developed in this thesis.

Second, this thesis goes on to provide contributions for the memory capacity aggregation problem across multiple virtualized nodes (coherence islands) by providing two different related mechanisms, called GV-Tmem and vMCA, for cloud environments based on Tmem. They both distribute and manage the system's total memory globally across computing nodes using a user-space process with high-level memory management policies. The difference between them is the enhancements of vMCA with respect to GV-Tmem for resiliency and more efficient use of the available memory resources.

The contributions to the memory management problem in this thesis center around the hardware virtualization layer. Targeting the memory management problem in such a way makes our solutions relevant for the cloud computing environments.

The rest of this introductory chapter will explain the background on virtualization technology up to its current state-of-the-art. It will also explain in more detail the problem of memory management at the virtualization layer and the problem of aggregating memory across nodes of a cloud infrastructures

1.2 Introduction to Virtualization

1.2.1 Antecedents

The term "Virtualization Technology" (VT) refers to a set of system features, both in hardware and in software, that allow for subsets of the physical resources of a node to be *virtualized* into multiple instances, effectively creating what are known as VMs.

The HV is the component that creates multiple instances of subsets of the computing resources of the node, and abstracts them in one or more VMs. Each VM is able to execute its own OS isolated from other VMs, and this OS uses and manages the subset of the resources made visible to the VM by the HV.

The first implementations of VT were introduced by IBM in the 1970s, when IBM developed the IBM System 360/67 which introduced the concept of virtual memory and time-

sharing [38], based on the architectures and concepts outlined by Arden et. al [7].

Subsequently, IBM introduced the CP-40 [73] and the CP-67/CMS [9], the latter being an OS that exploited the time-sharing and virtual memory capabilities of the System 360/67 to provide all users with a simulated System/360 computer of their own. Around this same time (1970s), Popek et. al [85] formally defined the features that a computer system should have in order to support virtualization.

Simultaneously, the MultiCS [20] OS was developed, which became the blueprint for multi-user OSs like Unix [90]. These OSs were highly successful in the consumer and mainframe market for a long time. Even though some virtualization technology concepts were being introduced into Unix-like OSs as time went by, the advances on virtualization technology kind of stagnated (1980s and 1990s).

As the computer market started growing, the amount of available computer architectures increased as well, together with the different OSs available. This presented a problem for the portability of programs, which usually were written and compiled for specific hardware and software environments. The software environments are commonly understood to be the software stack within the OS, including the OS itself, used to support a set of user-space applications.

Towards the end of 1980s, computer hardware emulators appeared, such as SoftPC. These emulators enabled the execution of applications developed for other OSs (such as DOS/Windows) to execute in Unix-based workstations. During the 1990s, computer emulators started to gain more traction, which resulted in the comeback of virtualization technology concepts and their subsequent refinement.

By the end of the 1990s different types of OSs and hardware architectures were well established, and virtualization solutions started to appear in order to solve the portability problem and allow users to run applications across different computer hardware and software stacks in a simplified and straightforward way.

During this time, the concepts of VT, first introduced by IBM, became relevant once again. Coupled with this, the widespread use of the Internet and the creation of online services established VT as one of the cornerstones for the computing landscape as we know it today. The following section will review the state-of-the-art virtualization concepts.

1.2.2 Fundamentals of Virtualization

In the context of the Cloud Computing industry, it is common to talk about Software Virtualization, which is the ability of a system to execute an OS within another host OS, or the ability to execute an application abstracted from the rest of the software stack of the node, as it is the case with the Java Virtual Machine [62]. Another common term is "Data Virtualization", which refers to a *service* that allows data manipulation by applications without having technical details on how data is stored or represented [29], and there's also a similar concept called Storage Virtualization [17], which refers to the pooling of persistent storage devices external to the node.

However, the types of virtualization that are more clearly defined are [86]:

- Application Virtualization: an application is virtualized when it is able to execute in a node abstracted from the rest of the software stack in that node.

- Desktop Virtualization: a type of virtualization in which a logical OS instance is isolated from the client that is used to access it.
- Network Virtualization: ability to present logical networking devices and services (logical ports, switches, routers, firewalls, etc) to connected workloads.
- Server or Hardware Virtualization: virtualization in the classical sense, in which the computing resources of a node are abstracted into one or more virtual instances called VMs.

Hardware Virtualization is one of the building blocks of Cloud Computing and associated services, and it is part of what we refer in this thesis as "Virtualization Technology". This is where the HV comes into place, of which there are two known types:

- Type 1 Hypervisor: the type that runs directly over the computing hardware and exploits its virtualization capabilities, first hand. This is the most important and critical for Cloud Computing support.
- Type 2 Hypervisor: runs inside a host OS, and it is used for a specific type of Software Virtualization.

When it comes to the Type 1 HV for Hardware Virtualization, currently there are three known sub-types:

- Full Virtualization: consists on a full binary interface between the VM and the underlying host computer to trap sensitive instructions into the HV and emulate them in software.
- Para-virtualization (PV): a technique in which the HV provides an API and the OS of each guest VM undergoes certain modifications to make use of the API.
- Hardware-assisted Virtualization or Hardware Virtual Machine (HVM): exploits specific virtualization features of the processor and requires (ideally) no modifications to the OS.

The HV (assume Type 1 HV, from now on) creates one or more logical (virtual) instances of a subset of the physical computing resources of the node, effectively creating VMs. Depending on how many VMs and the amount/type of specific resource, it is possible to either time-share (multiplex in time) the computing resources or have resources assigned to VMs. This is how CPU resources are managed, as well as I/O interfaces. The mechanisms of time-multiplexing and assignment of resources is available in most HVs available on the market (Xen [10], KVM [41, 23], ESXi [74]).

When speaking about time-multiplexing of resources, it refers to a mechanism in which, in very blunt terms, every VM will take turns on a temporal basis to access a specific resource. However, it is also possible to give a VM exclusive access to resources, by doing an explicit resource-to-VM assignment [65].

However, when it comes to managing memory among the VMs, it is not possible to time-multiplex memory in the same way as CPU or I/O interfaces. This is because it is in memory where the state of the VM is *stored*, even if the VM is idle or inactive. As VM

take turns to execute their instructions in the CPUs and access driver interfaces, all that input and output data used for the operations of the VM need to be stored somewhere in a persistent way.

Moving in-memory VMs' data into a long-term storage like a disk or a storage network and rotating their data between long-term storage and memory as a way to multiplex memory is not viable, due to the performance costs related to accesses to persistent storage [116]. Thus, it is necessary for the data to be in memory, which makes memory the main resource bottleneck in a virtualized computing node.

At this point, the necessity to manage memory becomes evident, and the management problem for memory is different when compared to the management of CPUs and other resources. The following section will explain more details about the memory management in the hypervisor and the core of the management problem.

1.3 Memory Management using Virtualization

The HV distributes the memory capacity among the VMs using a set of mechanisms that explicitly map memory to the VMs by adding an extra level of address translation. In a non-virtualized node, when a processor needs to access data from memory, it issues a virtual memory address, that is translated into a physical address by the page translation mechanism of the OS.

The addressing is done at a *page* granularity, where each page is of 4 KB in size. Sometimes, the physical *page* that results from the translation of a virtual address may not be found in memory, a situation that generates a page fault. In this case, the page needs to be fetched from disk and allocated in memory. This procedure, as costly as it is, *may* result in another page being deallocated from memory to release space for the new page, increasing the cost even further.

When virtualization is enabled, the HV adds another level of indirection to the address translation process. Now, the addresses issued by a processor are translated from a Virtual Address (VA) to a Physical Address (PA), just as before, but then the PA is translated into a Machine Physical Address or simply a Machine Address (MA). This process is illustrated in Figure 1.3. The PAs and the MAs are mapped together when the translation process occurs, binding the memory to the VM until the memory addressed with the MA becomes deallocated. When deallocation occurs, it can be remapped to a different PA or to a different VM altogether.

Depending on the type of virtualization used and the memory abstraction used by the HV, the MAs are exposed to the VM in order for the VM to access memory directly. Then, the OS will manage the memory amongst it processes according to its own constraints.

The HV is able to abstract memory in different ways and add/reclaim it to/from the VM through different mechanisms. Two of the most common mechanisms to add/reclaim memory from the VM are Memory Ballooning [67, 112] and Memory Hotplug [97, 63]. Memory Hotplug works using a SPARSEMEM memory model, which is a memory abstraction of discontinuous memory mapping.

1.3.1 Memory Hotplug

The process of adding memory (*hot-add*) with Memory Hotplug consists of two phases: 1)

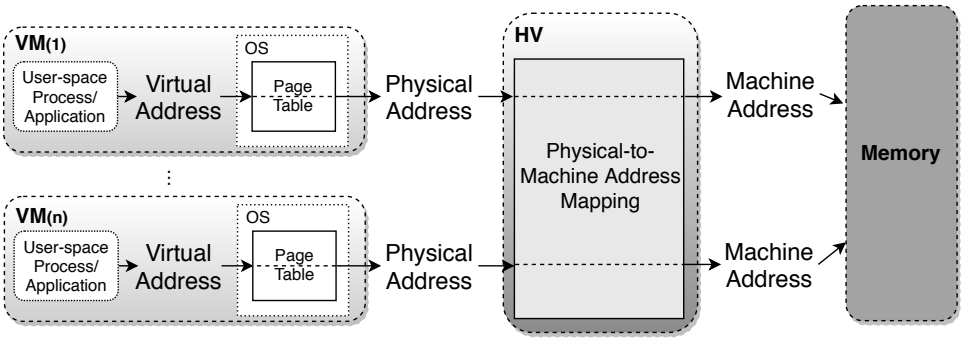


Figure 1.3: Address translation process in a virtualized node.

notifying the hardware/firmware that new memory will be added, and 2) making the memory available to different users. The kernel adds then physical pages into its address translation mechanism and into the memory allocator. Removing pages using Memory Hotplug (*hot-remove*) is known to be a complicated process [63], and it usually relies on Memory Ballooning to carry out memory removal.

1.3.2 Memory Ballooning

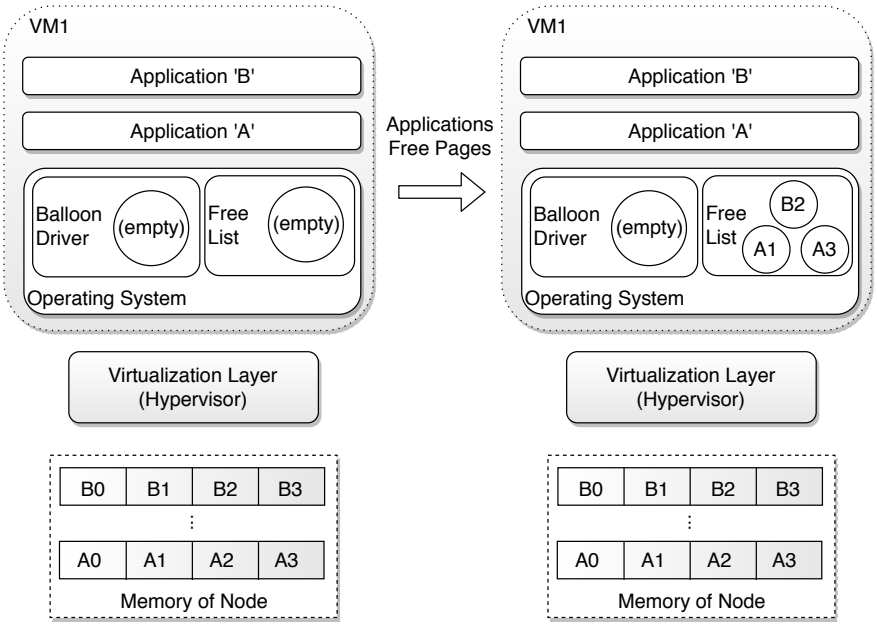
Memory Ballooning, on the other hand, relies on a special driver that resides in each guest OS and cooperates with the HV to dynamically change a VM's memory capacity, effectively moving memory back and forth between the HV and the VM [63, 97]. Figure 1.4 illustrates the way memory ballooning works.

When applications that are running inside a VM (VM1 in Figure 1.4) free up pages they previously allocated, the OS includes them in a list of free pages, as shown in Figure 1.4(a). However, the pages still remain allocated by the HV, which is unable to reclaim them since it has no access to the free list in the OS.

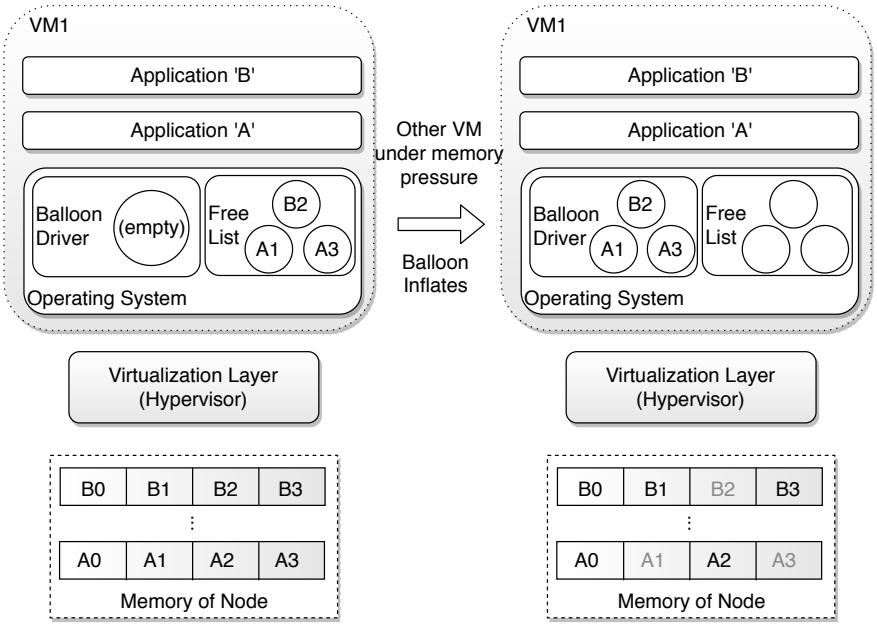
When another VM in the node starts to experience memory pressure, the HV communicates with the balloon driver in VM1 and tells it to kick back some pages to the HV, preferrably from the list of free pages. The balloon driver pins these pages inside the VM in order to prevent them from getting evicted by the VM. This is when the balloon driver is said to inflate, as shown in Figure 1.4(b). The balloon driver then communicates with the HV telling it which pages have been pinned, allowing the HV to unmap them from VM1 and to re-map them to a VM that needs more memory.

These mechanisms to manage and manipulate memory, in particular Memory Ballooning, make it possible for the HV to *overcommit* memory, that is, to give the VMs more memory than the amount available on the node [112]. This allows for a higher degree of server consolidation (the amount of VMs a single node can handle) than doing static partitioning.

However, Memory Ballooning presents some limitations of its own, some of which are: 1) a maximum limit is set when configuring the balloon driver at boot time (requires rebooting if the limit needs to be modified), and 2) the allocation/deallocation process can be time consuming when large amounts of memory need to be allocated, especially when the memory is fragmented. Because of the latter case, Memory Ballooning can present some limitations when memory demand changes in irregular ways: either in spikes or too rapidly.



(a) Free list of the OS getting populated.



(b) Balloon driver getting inflated because the HV needs memory.

Figure 1.4: Graphic illustration for Memory Ballooning, when (a) the free list of the OS gets populated, and (b) when the balloon driver inflates.

1.3.3 Transcendent Memory

In order to overcome these two limitations of Memory Ballooning, another mechanism called Transcendent Memory (Tmem) [69] was introduced to add/reclaim memory to a VM. Figure 5.2 illustrates the way Tmem works.

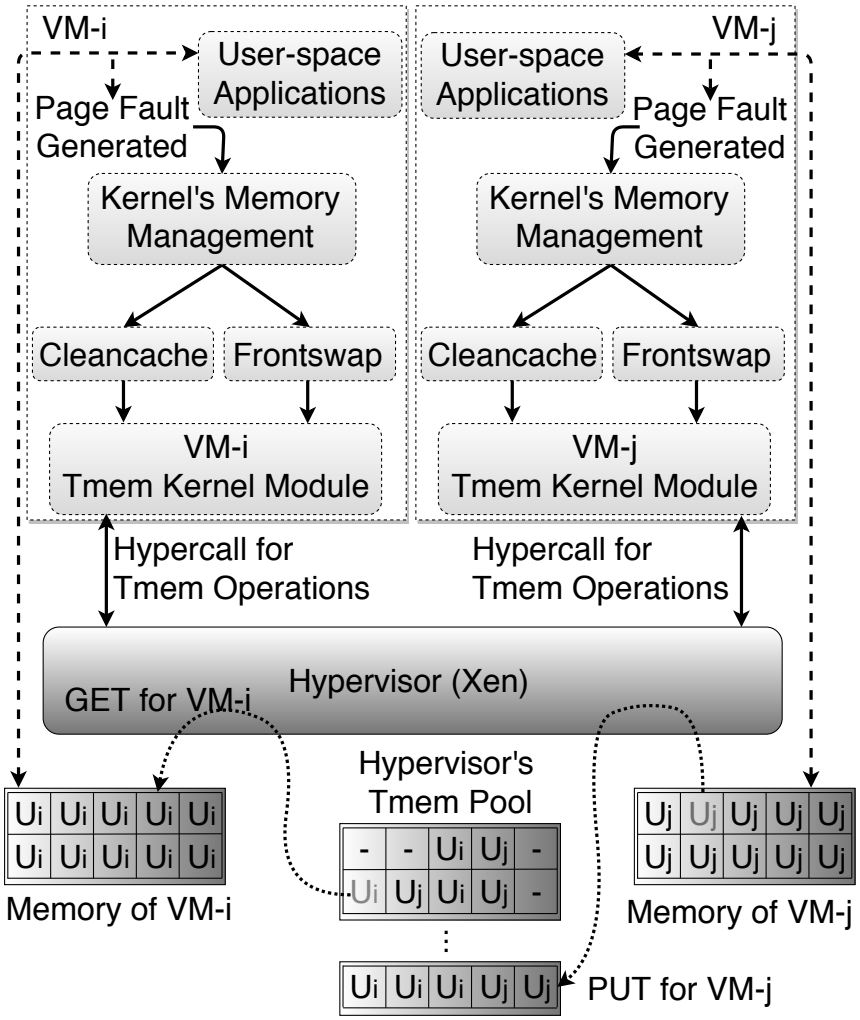


Figure 1.5: Graphic illustration of Tmem.

Tmem works by pooling together the fallow and idle memory in the node, and uses this memory pool as a cache between the main memory and the external/persistent storage (disk or other). This pool of memory is shown as the "Hypervisor's Tmem Pool" in Figure 5.2. When the OS attempts to evict a page from memory, it generates a page fault, a kernel module traps this fault and generates a hypercall for the HV to handle the eviction attempt. The hypercall generated corresponds to a *PUT* operation, as shown in Figure 5.2. The HV will allocate a new page or set of pages from the Tmem pool, copying the data from the evicted page(s) into the newly allocated Tmem page(s), and storing it in a key-value store. In this scheme, the OS "believes" that the memory it evicted is on a disk device, but it is fact allocated in a memory section only visible to the HV.

A read from the Tmem pool is triggered when the VM attempts to read a page from disk that it previously evicted from memory. The read attempt will generate a page fault as well, but this time the kernel module generates a hypercall that corresponds to a *GET* operation, instead of a *PUT*. The *GET* operation will copy the data from the Tmem page

into a page allocated by the OS inside the VM. Depending on the way the Tmem was initially configured, the *GET* operation will free up the Tmem page after copying its content back to the VM, and will make it available once again.

Tmem also supports a *FLUSH* operation that the OS in the VM uses to maintain coherence between the data in Tmem, the internal data structures of the OS in the VM and the data stored in disk. A *FLUSH* operation frees up a Tmem page from the pool, and allows it to be reallocated when another *PUT* is issued.

Figure 5.2 shows that Tmem can be used in two modes of operations: Cleancache, which is used to stored clean pages (not modified) that are backed by the filesystem; and Frontswap, which is used to store dirty pages that would be otherwise be written to the swap space. However, in both cases, the idea is for Tmem to act as a cache between memory and persistent storage, in order to prevent costly accesses to the latter.

One advantage that Tmem possesses over Memory Ballooning is that it can allocate and reclaim pages at a faster rate than Ballooning, resulting in Tmem being more responsive to fast changes in memory demand [69]. Another advantage of Tmem is that, given that it works as a cache between memory and persistent storage, the OS inside the VM really does not need to know the capacity of the underlying Tmem nor its location. This feature can be exploited to allow for memory to be disaggregated from the node.

Even though the cost of allocating/deallocating a page is smaller with Tmem when compared to Ballooning, one important limitation of Tmem is that the cost of accessing a page through Tmem is higher than the cost of accessing pages that have been previously allocated through Ballooning. Thus, it is not viable to allocate memory to a VM exclusively through Tmem.

Given all these mechanisms that allow to move memory among the VMs in the node, then the problem is to leverage these mechanisms to dynamically reallocate the memory in order to make efficient use of the available memory resources, first, within one single node and then, across multiple nodes in the cloud infrastructure.

1.4 The Memory Management Problem with Virtualization and State-of-the-Art Solutions

The problem of dynamic memory management in a virtualized node appears when different active VMs executing different applications present different memory utilization behavior that also changes over time. Then, the problem boils down to the reallocation of memory dynamically among the VMs such that each VM has all the memory it needs (assuming there's enough memory in the node) or that the available memory is dynamically reallocated proportionally to ensure a performance target (assuming memory pressure in one node). The reallocation of memory could be made possible through Memory Ballooning, Memory Hotplug and/or Tmem, but still it is necessary to *control* them by designing and implementing algorithms/mechanisms built for this purpose.

When considering the problem of memory management across multiple nodes, the source of the problem is the same, but the scope of the possible solutions will include the reallocation of memory capacity across nodes using an appropriate memory abstraction (like Tmem), and also might include solutions that involve more complex memory architectures and management schemes.

When a new VM is created, it is allocated a portion of the physical memory. If the VM later starts executing applications that need more memory to execute without accessing persistent storage, then the VM is said to be *under-provisioned* of memory capacity. In other cases, some physical memory may still remain in the system, either unallocated by the hypervisor (fallow) or allocated to another VM that does not need it (idle). A VM with idle memory is said to be *over-provisioned* of memory. The occurrence of either or both of these cases (under- and over-provisioning) implies that the utilization of memory is sub-optimal.

The adequate thing to do in this cases would be to reclaim memory from over-provisioned VMs and make it available to the VMs that are under-provisioned, by readjusting the memory allocation of VMs dynamically and improving memory utilization.

1.4.1 Memory Management in a Virtualized Node

Many research efforts seek to improve main memory management in a single node [120, 12, 13, 15, 27, 51, 63, 72, 80, 95, 101, 105, 125]. For example, Zhao et al. [125] proposed a mechanism to dynamically adjust the memory allocation by implementing a user-space program in the Dom0 (using Xen) to trigger inflation and deflation of the balloon driver in the VMs. This work is particularly interesting because they include predictive capabilities based on an enhanced reuse distance approach [71]. They design a memory predictor that predicts the VM's working set size (WSS) based on Least Recently Used (LRU) histograms. The predicted working set size determines the target memory allocation, which is enforced, in each VM by the respective balloon driver. Both Zhao et al. [125] and Liu et al. [63] predict the memory demands of the VMs. Whereas, Zhao et al. [125] predicts the working set size of the VMs, Liu et al. [63] predicts the total memory footprint including swap. The approach of estimating WSS to drive the memory allocation of the VMs has been very common in other works as well [15, 47, 84]. When estimating the WSS, the objective is to allocate enough memory to the VMs in order for their working set to fit in.

One problem with WSS estimation is that when applications do not use the whole WSS during different phases of their execution, it results in memory being overallocated (that is, misusing memory) during those times when part of the memory is not used. This problem is exacerbated when there are multiple applications running inside the VMs and variation on the applications behavior over time start affecting the measurements of the WSS.

In [120], Zhang et al. opted to avoid a WSS estimation approach due to the difficulties of accurately estimating it at runtime for changing workloads, and instead decided to dynamically allocate memory through a pool of shared memory across the VMs that is managed by the host. In this scheme, memory is reclaimed from the host in an anticipated way, and made readily available to the VMs that come under memory pressure in a sudden way. In this way, the delays associated to the reclaiming of memory by the balloon driver are avoided, preventing the VMs from evicting pages from memory. This avoids further accesses to persistent storage, since the data would not have to be fetched back from persistent storage again.

Approaches based on control engineering techniques [45, 83] have also been used to solve the memory management problem. The main issue with these approaches is their necessity to develop models of the quantitative relationship between an application's performance and its memory allocation. This is very difficult to do accurately, given all the factors that have an impact on this relationship. The use of reinforcement learning (RL) approaches become

thus justified when grasping the complexity of the factors involved in this relationship.

Many learning-based solutions have been proposed to solve the resource allocation problem in cloud infrastructures [13, 87, 124, 21] but so far, there have not been too many efforts that use RL to exclusive solve the memory management problem. There are some notable RL-based solutions for the resource allocation problem that targets CPU and memory (and other resources) within a virtualized node. In [87], the authors implemented a model-based RL algorithm for VM resource configuration, which included CPU time, virtual CPU and memory. This algorithm is a Deep Q-learning (DQL) [79] algorithm using a neural network to approximate the value function.

Motivation for Contributions for the Single-Node Memory Management Problem

The first three chapters of this thesis present solutions to improve the utilization of memory within a single node. The first solution to achieve this, called SmarTmem, focuses on improving memory resource utilization through Tmem, by providing a software stack that allows its user-space control and deploys high-level Tmem management policies.

The motivation behind SmarTmem comes from the remote memory aggregation case, because increasing the efficiency of Tmem utilization within a single node was a necessary step in order to make a memory aggregation case work more efficiently.

The results of SmarTmem show that it is possible to improve Tmem utilization for the cases in which Tmem is deployed in its frontswap/private mode. Future work could establish whether the approach of SmarTmem could also improve the performance of other settings in which Tmem is deployed. Settings of this type include cases where Tmem is used to manage memory in systems with heterogeneous memory hierarchies [110], systems that expose Tmem to user-space processes directly [107], and virtualized systems that use Tmem as a hypervisor-level secondary cache [52, 111, 75, 46].

The other two solutions for the memory management problem within a single node are closely related and are based on Continuous Action RL algorithms. One of these is called CAVMem (Continuous-Action Algorithm for Virtualized Memory Management) and it is a proof-of-concept mechanism to solve the memory allocation problem for memory exposed as RAM, while the other solution, called CARLEMM (Continuous-Action Reinforcement Learning for Memory Management), builds on the proof-of-concept by CAVMem to provide a memory allocation solution in an actual virtualized node.

CARLEMM focuses on management of memory through the memory ballooning interface, which is a standard way to manage memory in cloud infrastructures. Currently, there are multiple solutions that target the RAM allocation problem, but this problem is difficult to solve because the memory demand of VMs change continuously. Many solutions for this problem are limited by the complex relationship between the memory allocated to a VM, the applications' behavior and their performance, for which RL provides a good alternative. By using RL to solve the memory allocation policy, we seek to increase the efficiency of memory resource utilization in a single node, which would allow for better VM consolidation and better strategies for the overcommitment of resources [19, 4, 21].

All of these issues (memory utilization efficiency, consolidation, overcommitment) are important for cloud service providers, because good solutions on memory management provides them with multiple opportunities to increase their revenue while still providing the desired QoS for their customers. It has been shown that large production cloud infrastructures of

service providers, like Google, only use 50% of the memory resources that are allocated [88]. In other cases, the utilization of memory resources can be slightly more than 50% [14], as the case of Alibaba. This means that there is significant opportunity to increase efficiency regarding the memory resource utilization. Improving memory management is necessary to achieve this efficiency, and it also opens the door for better overcommitment policies [52, 21].

1.4.2 Memory Management across Virtualized Nodes

Computing is moving increasingly towards the cloud and to infrastructures that provide Software-as-a-Service (SaaS) [48], with a larger presence of more data-intensive workloads. This has caused a shift in the design approach of computing servers, from a "share nothing" self-contained design, into a "share something" approach [30]. This implies that the computing resources of the data center can be pooled across its nodes (totally or partially), and that different forms of resource disaggregation can take place, allowing for the scaling process to go at different paces depending on the specific resource [6].

This approach allows for the memory to be scaled out independently of the processing resources, because scaling up the memory within a node is becoming increasingly prohibitive [108]. This "share something" approach for remote memory targets some of the challenges that are very relevant in today's data centers, some of which that are:

- The increasing demand for memory due to the presence of more workloads that use a large amount of data
- The need to improve memory resource efficiency in the data center [40]
- The need to reduce the data center's Total Cost of Ownership [55]
- Latency of networks is being reduced as new hardware for networking becomes present, bringing them closer to the latency of DRAM technology [5, 33, 26, 49]

As more data intensive workloads i.e. applications with large memory footprint, are deployed in the cloud infrastructure, the computing nodes need to be provisioned with more cores and more memory capacity to perform their computations and keep performance. Thus, the core count in the processor of the nodes keeps scaling up, but the memory is unable to scale-up at the same pace, resulting in a drop of the capacity-per-core of 30% every year, yielding a compute-to-memory imbalance [61, 117].

However, provisioning nodes in the cloud infrastructure with enough memory to fit the applications with the largest memory footprints (the worst-case) results in memory being under-utilized (over-provisioned), because most of the time the nodes execute workloads that have a considerable smaller memory footprint that would sub-utilize the large memory capacity [88, 14, 64]. Moreover, provisioning nodes with large amounts of memory implies a large investment cost, in order to acquire nodes with such large memory capacity. This over-provisioned nodes result memory being used in an imbalanced [40] way across nodes, because there will be nodes that have a large amount of memory pressure while the memory of other nodes remains idle.

In addition to this memory resource inefficiency, it also increases the Total Cost of Ownership (TCO) of keeping the cloud data center running with nodes that have a lot of cores and a lot of memory, a huge amount of memory that is not being used [55]. This means

more money loss: in paying for the energy to keep the nodes running while they are not performing any useful work.

Thus, the use of remote memory becomes attractive, because it allows for the memory to become external to the node, which avoids the presence of memory over-provisioned nodes and reduces the compute-to-memory imbalance. Even though the idea of using remote memory has been around for more than 20 years, it was not feasible to deploy because of the high-latency associated with the communication links of TCP/IP networks over which the first solutions were tested on: the performance losses were too high [5].

But advances in networking technology have managed to reduce the latency of the communication links across the computing nodes in the cloud data center[33, 5, 26, 49, 6], by using technologies like Infiniband [40] or other interconnection solutions (backplane-based). As a result of all of this, remote memory has become attractive once again.

In a cloud data center, remote memory can be implemented either as software-based or as hardware-based. Sofware-based approaches are simply known as "remote memory" in the literature [5], and they are related to a long-line of efforts for distributed shared memory. These approaches enable the access to remote memory over high-latency off the shelf network technology, with no hardware extensions, but extending the software stack at every level to enable remote memory access.

Hardware-based approaches, on the other hand, are referred to as "disaggregated memory" and these approaches involves additional hardware support at different architectural layers. In this thesis work, we focus on this type of approaches. Disaggregated memory can be implemented as [61]:

- Byte-addressable: requires the remote memory to be attached to the system bus of the processor.
- Block-device: it is accessed on a per-page basis, and can be accessed using a device interface like PCIe or similar.

In both cases, it is required that the physical memory address space of the processors is partitioned in such a way that a subset of the physical addresses maps to regions of remote memory. The communication mechanisms between the node and the location of the remote memory can be customizable and is up to the system designers.

When memory is disaggregated and becomes a pooled resource, it is possible to use different types of memory technologies besides the traditional DRAM, such as Storage-Class Memories (SCMs) [108], a term that refers generally to Non-Volatile Memories (NVMs) [77]. NVMs provide a good option for pooled disaggregated memory because it is able to provide higher density and higher capacity than traditional DRAMs, but at a much lower cost [28]. However, NVMs have higher latency than DRAM (but still less latency than Flash or disk devices), less bandwidth and has endurance limitations [108, 126, 28]. NVMs can be used as an additional level in the memory hierarchy functioning as a cache between DRAM and permanent storage (even in cases when it is not pooled) [108, 110], as block-device for swapped pages [126, 40], or for increasing system resiliency through data replication [122] or as back-up for system memory [96].

Remote memory capacity aggregation is also relevant for cases in which High-Performance Computing (HPC) applications are deployed [127], because many of these applications have a memory footprint well under the amount of memory available in server nodes. By allowing

remote memory aggregation, it is possible to increase the utilization of memory in nodes where low-memory footprint HPC applications are running.

Motivation for Contributions for the Muti-Node Memory Capacity Aggregation

The solutions presented in this thesis work for remote memory support, namely GV-Tmem and vMCA, belong to the disaggregated memory category. In this context, our contributions have the following characteristics:

- Rely on specialized hardware support that includes an specialized interface on the system bus as the gateway for remote memory regions.
- The communication fabric consists of a customized high-bandwidth low-latency Network-on-Chip (NoC) that addresses remote memory regions through a partial global physical address space
- The nodes make use of the disaggregated memory through the kernel's swap interface i.e. disaggregated memory used as a swap device, but byte-addressable features are still exploited.
- In our solutions, OSs and applications make use of the disaggregated memory in a transparent way, through a layer of virtualization.

Our contributions for disaggregated memory support use the remote memory regions as a cache between DRAM and permanent storage for pages what have been swapped out from a node's main memory, making use of the remote memory in a similar way to a swap device (but not entirely, as it will be explained shortly). The kernel decides which pages to swap out and, since we are using a virtualization layer, every page that gets swapped out belongs to one VM, which has exclusive access to it.

When a VM swaps out a page, the VM calls on the page fault mechanism of its respective kernel, which results in the kernel attempting to write the page into permanent storage before deallocating it. As it will be explained in more detail in Chapters 5 and 6, in our approaches GV-Tmem and vMCA, the hypervisor intercepts the kernel's attempts to swap out pages to permanent storage. The hypervisor then allocates a page in a local or disaggregated memory region (depending on availability) using hypervisor-level memory allocation constructs which exploit the byte-addressable capability of the remote memory system architecture, relying on the presence of a partially global physical address space. In this way, the node is able to effectively *aggregate* more memory capacity for itself.

By using disaggregated memory in GV-Tmem and vMCA as a swap device from the VM/kernel perspective, the disaggregated memory becomes transparent to the OS of the VMs. It is not necessary for the OSs to know the capacity nor the location of the remote memory. This means that expensive modifications to the OS or to the applications are not required to support the disaggregated remote memory regions, that may even include NVMs [57].

In our contributions, Tmem is the kernel interface used to access the remote memory regions, and different approaches that exploit Tmem capabilities can be implemented. For example, Lorrillere et al. [66] proposed a mechanism, called PUMA, to improve memory utilization based on remote caching to pool VMs memory across a data center. PUMA pools

VMs' unused memory for the benefit of other VMs having I/O intensive workloads. PUMA makes use of an interface very similar to Tmem to access the cache present in a remote server.

Zcache [68] is a backend for frontswap and cleancache that provides a compressed cache for swap and clean filesystem pages. Another related mechanism called Zswap has been used for other state-of-the-art approaches to exploit the presence of remote memory regions in data centers [55], which is roughly equivalent to using Zswap in frontswap and exclusive mode, by-passing the virtualization support. RAMster [68] is an extension of zcache that uses kernel sockets to store pages in the RAM of remote nodes. Similarly, RAMCloud [82] is a POSIX-like filesystem in which all data is stored in RAM across the nodes.

For GV-Tmem and vMCA, we use Tmem's interface to provide a new software-stack to enable a node to aggregate memory capacity to itself, while at the same time providing mechanisms to make more efficient use of the available memory. In these contributions, the work is specifically motivated into:

- Improve memory resource efficiency across nodes by aggregating memory dynamically to the nodes
- Leverage the capabilities of new hardware architectures designed to support partial global physical address spaces across multiple nodes and fast interconnects [26, 49]

1.5 Contributions and publications

This thesis presents contributions for the memory management problem in a single node and then across multiple nodes. To summarize, the contributions in this thesis are:

- A mechanism called SmarTmem for more efficient allocation of Tmem in a single virtualized node.
  - A demonstration that the default way of allocating Tmem at the single node level, used in state-of-the-art hypervisors, is not optimal.
  - Development of a high-level memory management policies to manage Tmem through SmarTmem in a single node.

  Luis Garrido, Rajiv Nishtala, and Paul Carpenter. SmarTmem: Intelligent management of transcendent memory in a virtualized server. In *RADR 2019: 1st Workshop on Resource Arbitration for Dynamic Runtimes*, 2019 [35].

- A mechanism called CAVMem (Continuous Action Algorithm for Virtualized Memory Management) developed as a proof-of-concept for the use of continuous action Reinforcement Learning for RAM allocation in a *simulation* of a virtualized node
  - Formulation of the RAM management problem as a Markov Decision Process in a simulated environment that models memory management constraints.

  - Development of a continuous action off-policy model-free reinforcement learning algorithm for memory allocation in a simulated environment.
  - Comparison of the CARL algorithm in CAVMem with non-continuous action reinforcement learning algorithms like Q-Learning and Deep Q-Learning in the context of RAM management within the simulated environment

Luis Garrido, Rajiv Nishtala, and Paul Carpenter. Continuous-Action Reinforcement Learning for Memory Allocation in Virtualized Servers, In *14th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'19), International Conference on High Performance Computing*, 2019 [37].

- A mechanism called CARLEMM (continuous action Reinforcement Learning for Memory Management), based on the proof-of-concept provided by CAVMem, but enhanced to solve the memory allocation problem in a *real* computing node.
  - Formulating the RAM management problem as an MDP for a real virtualized computing node.
  - A continuous action reinforcement algorithm for RAM allocation in a real virtualized node
  - A flexible software stack for the implementation of CARLEMM and performance evaluation.

- A mechanism called GV-Tmem (Globally Visible Tmem) for remote memory capacity sharing across multiple computing nodes in a cloud infrastructure
  - A software architecture to share memory capacity across nodes using Tmem.
  - A two-tier mechanism for allocation and management of dis-aggregated memory.

Luis A. Garrido and Paul Carpenter. Aggregating and managing memory across computing nodes in cloud environments. In *12th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'17), International Conference on High Performance Computing*, pages 642–652. Springer International Publishing [36].

- A mechanism called vMCA (virtualized Memory Capacity Aggregation) for the sharing of memory capacity across nodes, but enhanced for resiliency and efficiency in comparison to GV-Tmem.
  - A software stack to aggregate memory capacity across multiple nodes, focused on resiliency and efficiency.

Date	Description
September 2014	PhD Matriculation Working with Euroserver Prototype as the experimental platform (ultimately unsuccessful) and development of our research proposal
20 July 2015	Presentation of thesis plan Development of a new experimental platform (successful) and the first mechanism for support of remote memory capacity
22 June 2017	Presentation of GV-Tmem (Chapter 5) at VHPC 2017
16 December 2017	Presentation of vMCA (Chapter 6) at ICPADS 2017
24 May 2019	Presentation of at SmarTmem (Chapter 2) at RADR Workshop
20 June 2019	Presentation of CAVMEM (Chapter 3) at VHPC 2019
17 June 2019	PhD predefence
September 2019	PhD defence

Table 1.1: Thesis timeline

- A multi-level user-space mechanism for allocation management of aggregated memory capacity, with special considerations for memory allocation within a node and memory distribution across nodes.
- A more complete analysis of high-level policies for memory aggregation and allocation.

Luis A. Garrido and Paul Carpenter. vMCA: Memory capacity aggregation and management in cloud environments. In *IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, December 2017 [34].

1.5.1 Thesis outline and Timeline

Table 1.1 shows a rough timeline of the stages of the thesis project. Initially, the scope of our research was on aggregating memory capacity across nodes. But during the course of this, it also became evident that to provide an integral solution for memory management, new approaches for managing memory locally through Tmem and Ballooning would increase the effectiveness of our remote memory aggregation solutions.

Figure 1.6 shows the structure of the rest of this thesis. Chapter 2 describes SmarTmem, which uses high-level policies to allocate Tmem inside a virtualized node. Chapter 3 develops CAVMem, a proof-of-concept Continuous-Action Reinforcement Learning approach for allocation of RAM via ballooning, again inside a single node. Chapter 4 provides the initial steps towards a complete implementation of the reinforcement-learning approach on a real system. Next, Chapter 5 develops GV-Tmem, the first mechanism for the control of memory capacity sharing across multiple nodes. This is extended, in Chapter 6, to vMCA, which enhances GV-Tmem for resiliency and efficiency, and solves some the issues that come with sharing memory across remote nodes. Finally, Chapter 7 concludes the thesis.

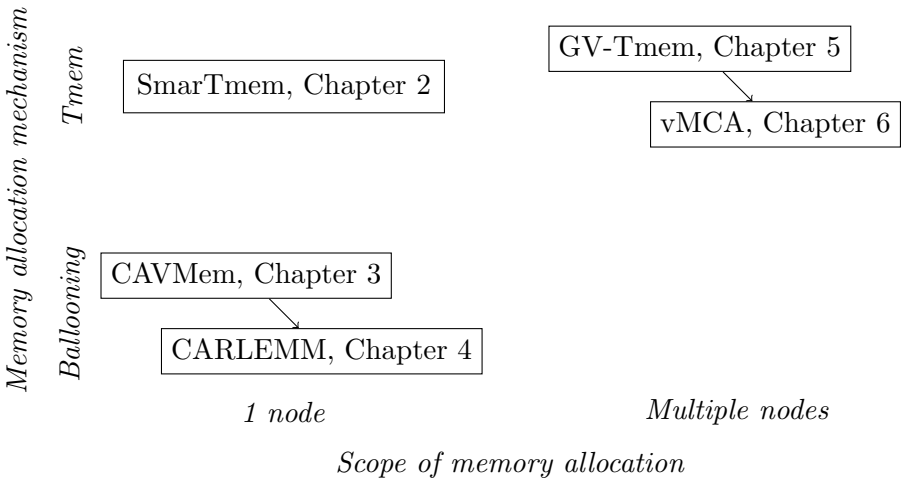


Figure 1.6: Thesis structure

Chapter 2

SmarTmem: Tmem Management in a Virtualized Node

Managing memory capacity in virtualized environments is still a challenging problem. Many solutions have been proposed and implemented that are based on memory ballooning and memory hotplug. But these mechanisms are slow to respond to changes of the memory demands of VMs. As explained in Section 1.3, Tmem was introduced to improve responsiveness in memory provisioning, by pooling idle and fallow memory in the HV, and making these physical pages available as additional memory for the VMs through a key–value store.

However, Tmem presents some limitations of its own. State-of-the-art hypervisors do not implement any efficient way to manage Tmem capacity, letting VMs compete for it in a greedy way by default, regardless of their actual memory demand.

In this paper, we demonstrate the need for intelligent memory capacity management for Tmem, and we present the design and implementation of SmarTmem, a mechanism that integrates coarse-grained user-space memory management with fine-grain allocation and enforcement at the virtualization layer. Our results show that our solution can improve the running time of applications from the CloudSuite benchmarks by up to 35% compared to the default Tmem allocation mechanism.

2.1 Introduction

Virtualization technology is prevalent across cloud service providers as it reduces capital and operational costs [8, 119, 18, 81, 123, 39]. As explained before, the HV [85, 94] in every node of the cloud infrastructure creates VMs within each node, and manages the physical resources allocated to the VMs. The HV multiplexes the CPUs and I/O devices and it controls the allocation of the physical memory capacity. In such environments, memory is often one of the most critical and scarce resources [99, 63].

As it was mentioned in Chapter 1.1, state-of-the-art HVs have multiple mechanisms to dynamically manage memory capacity [112, 97, 69], and Tmem [69] was introduced to improve memory reallocation responsiveness, since other mechanisms like memory ballooning are relatively slow to reallocate memory upon changes in memory demand [69]. Tmem has two modes of operation: 1) *frontswap*, which serves as a page cache for pages swapped out by the VMs, and 2) *cleancache*, which serves as a page cache for clean pages that were fetched from disk. In either mode of operation, Tmem pools all the idle and fallow memory pages in the node, and the pages get assigned to the VMs as Tmem when they need it, which then become accessible through a key-value store. If there are no free pages, any write to a Tmem page will fail, causing an access to the (virtual) disk device.

When multiple VMs use Tmem, the hypervisor will assign Tmem pages to the VMs in a greedy manner. By default, the VMs compete for the Tmem capacity, so if some VMs take a large amount of the available Tmem, the rest of VMs will starve and will generate a large number of disk accesses, degrading performance across the VMs and the computing node [59].

This chapter addresses the issue of optimizing VM memory allocation using Tmem in a single computing node. SmarTmem is introduced here, which is a software stack for intelligent Tmem allocation in a single node. This chapter analyzes in depth the problem of Tmem optimization at the single node level. To the best of our knowledge, this is the first effort to analyze Tmem capacity optimization in a single virtualized computing node.

SmarTmem employs a user-space process executing in Xen's privileged domain that implements intelligent Tmem management policies based on the memory utilization behavior of the VMs. Our results show that using high-level policies improves the running time of applications compared to the baseline implementation of Tmem.

To summarize, the main contributions of this chapter are:

- We demonstrate that the default way of allocating Tmem in a single node, used in state-of-the-art HVs, is unable to adapt to changes in memory demand and to ensure fair and proportional allocation of Tmem when multiple VMs are active.

- We propose a software architecture called SmarTmem, for intelligent memory management of Tmem in a single virtualized computing node.

- We implement high-level Tmem management policies in SmarTmem and evaluate them using benchmarks from CloudSuite [31], and our results show up to 35% improvement over the default greedy approach of allocating Tmem.

The rest of this chapter is organized as follows. Section 2.2 discusses the SmarTmem architecture and the high-level Tmem management policies. Section 2.3 describes our experimental framework. Section 2.4 presents the evaluation results. Section 2.5 expands on the related work. Finally, Section 2.6 states our conclusions and future work.

2.2 SmarTmem: memory management policies and architecture

This section describes the architecture of SmarTmem, which is a software stack designed for the intelligent Tmem management in a single virtualized computing node. SmarTmem divides Tmem management between coarse-grain level memory management in a user-space process running in a privileged domain and fine-grain allocation and enforcement in the HV.

2.2.1 SmarTmem Architecture

Figure 2.1 shows the architecture of SmarTmem, consisting of three components:

- HV support for SmarTmem
- Tmem Kernel Module (TKM)
- Memory Manager (MM) User-space Process for Tmem Allocation in a Single Virtualized Computing Node

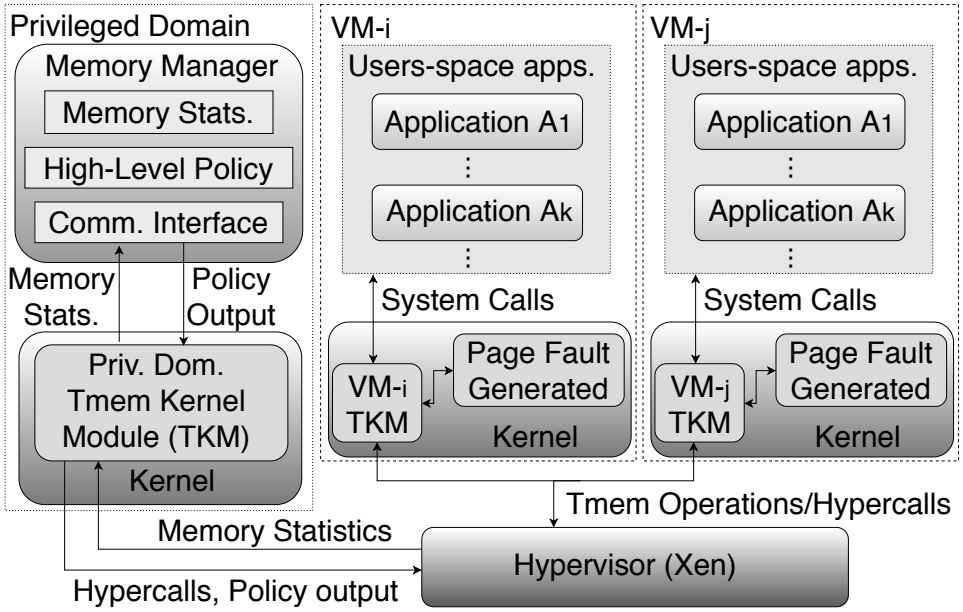


Figure 2.1: SmarTmem architecture. The Hypervisor layer and the Tmem Kernel Module have been extended from state-of-the-art implementations, whereas the Memory Manager (MM) was created from scratch. The MM runs in Xen's privileged domain.

Table 2.1: Memory statistics used in SmarTmem. The sampling interval is one second.

Memory Statistics	Description
E_TMEM	Value used in the HV indicating that a *put* (or other Tmem op.) cannot succeed.
S_TMEM	Value used in the HV indicating that a *put* (or other Tmem op.) has succeeded.
$node\_info.free\_tmem$	Number of free pages available for Tmem. Constant throughout the lifetime of the node.
$node\_info.vm\_count$	Number of VMs registered.
$vm\_data_{\text{hyp}}[id].vm\_id$	Identifier of the VM within the HV.
$vm\_data_{\text{hyp}}[id].tmem\_used$	Number of pages of Tmem memory currently used by the VM.
$vm\_data_{\text{hyp}}[id].mm\_target$	Target number of pages allocated the VM.
$vm\_data_{\text{hyp}}[id].puts\_total$	Total number of *puts* issued by the VM in the current sampling interval.
$vm\_data_{\text{hyp}}[id].puts\_succ$	Total number of successful *puts* issued by the VM in the current sampling interval.
$memstats$	Variable storing the last sampled statistics that the HV sent to the MM.
$memstats.vm\_count$	Amount of active VMs as seen by the MM.
$memstats.vm[i].vm\_id$	Identifier of the VM within the MM.
$memstats.vm[i].puts\_total$	Number of *puts* issued by a VM in the sampling interval.
$memstats.vm[i].puts\_succ$	Number of *puts* of a VM that succeeded in the current sampling interval.
mm_out	Data structure that holds the output parameters of the MM policy.
$mm\_out[i].vm\_id$	VM identifier that maps a VM to its target allocation as calculated by the MM.
$mm\_out[i].mm\_target$	Memory allocation target as calculated by the policy in the MM.

2.2.2 HV support for SmarTmem

This section describes the HV support for SmarTmem. A state-of-the-art HV (Xen, in this case) was extended in order to enable the functionalities for SmarTmem. The extensions done to the HV have many roles, among these roles are to generate the metrics used by the high-level policy, enforce target allocations and make the fine-grained Tmem page allocation. This separation means that only a small amount of common development is needed at the HV level, which is specialized and intrusive.

It is necessary for the HV to gather information about the Tmem capacity utilization by the VMs. This information is used by the MM to determine the Tmem dynamic reallocation. Table 2.1 summarizes the data collected by the HV.

When a VM maxes out its memory, any future access that attempts a swap to disk will generate a *put* operation to Tmem. The HV monitors the amount of *puts* of each VM, as well as other Tmem operations. The HV also keeps track whenever *puts* fail, and the amount of memory used by the VMs. When a *put* fails, it means that there is no Tmem capacity to satisfy the request, or that the VM has exceeded the amount of Tmem it can take. Everytime this happens, the VM that generated the *put* will swap the page to disk.

The HV gathers and monitors all the memory utilization behavior and sends it to the TKM in the privileged domain via a virtual interrupt request (VIRQ). The HV gathers this information through a set of data structures that were added to the section of the HV that provides Tmem support, and are passed to the user-space process through page-copy operations that copy the data from the corresponding memory page in the HV into a user-

space page.

The VIRQ is triggered using a timer in the HV that calls it up every time a fixed amount of time has passed. Currently, this fixed amount is one second. The timer is only updated when a Tmem operation takes place (*put*, *get*, *flush*), otherwise no transmission of data from the HV to the user-space process will occur.

With this information, the MM calculates new target Tmem capacities for every VM and sends these target capacities or allocations back to the HV again through the TKM, which generates a specific hypercall for this purpose. The data structures that stored the memory utilization information in the HV and the data structures that store it in the user-space process are similar, as well as the data structures that store the memory capacity targets for each VM.

Algorithm 1 presents the pseudo-code to illustrate the tasks carried out by the HV. When the target allocations reach the HV, it stores them and keeps them until the MM modifies them (line 3). Everytime a VM attempts to get a Tmem page (through a *put*, line 4), the HV checks if the current amount of Tmem used by the VM is less than its target (line 5). If so, the VM obtains a page from the Tmem pool, the HV allocates a Tmem page and copies the data contained in the VM's page into the Tmem page frame (line 10). SmarTmem only requires one single allocator, since all the Tmem is owned by the current node and no inter-node transfer of pages takes place. This is an important difference that needs to be taken into account when explaining the architecture for inter-node memory capacity sharing presented in Chapters 5 and 6. SmarTmem does not require any special hardware support and can be deployed in any computer server available on the market as long as it provides basic virtualization support, whereas the approaches presented in Chapters 5 and 6 do require specialized hardware support.

In case there's no free Tmem or if the amount of Tmem used by the VM is equal or exceeds its target, then any *put* will fail (lines 5–8), forcing the VM to swap to disk. Every *put* issued by a VM will fail as long as the target is equal or smaller to the current amount of Tmem in use. The HV can reclaim Tmem pages from a VM very slowly, but pages can also be released when a VM explicitly flushes a page (line 16). While the target remains smaller than the Tmem capacity in use, the VM will be unable to acquire more Tmem pages.

The values of $vm\_data_{\text{hyp}}[id].tmem\_used$ and $vm\_data_{\text{hyp}}[id].puts\_succ$ are incremented when a *put* succeeds (lines 10–13), and a page is allocated for the VM. The parameter $vm\_data_{\text{hyp}}[id].tmem\_used$ is decremented when a VM releases pages (*flush*, lines 16–19), deallocating a Tmem page. The parameter $vm\_data_{\text{hyp}}[id].puts\_total$ (line 15) is incremented when a *put* occurs, regardless if it succeeds or not.

A similar thing happens to $vm\_data_{\text{hyp}}[id].gets\_total$, which is incremented every time a *get* occurs. A *get* attempts to retrieve a page that was previously *put* into Tmem, but it can fail to do so, depending on the way Tmem was configured. When using Tmem in its frontswap mode, a *get* cannot fail, since it is treated as persistent storage, but when Tmem is used in its clean-cache mode, pages previously *put* can be reclaimed by the HV for a different purpose without any notification. In this case, pages are ephemeral.

It is possible for a VM to use more Tmem than its VM target. This can happen because the targets are continuously modified, and the target for a VM might be reduced below the capacity it is using. This might occur because the VM might be temporarily idle, or executing a phase of the application with reduced memory pressure with respect to previous

Algorithm 1 Tmem allocation in the HV

```
1: function HYPERVISOR_OP(vm_data_hyp, id, op)
2:     tmem_used ← vm_data_hyp[id].tmem_used
3:     mm_target ← vm_data_hyp[id].mm_target
4:     if op == PUT then
5:         if tmem_used ≥ mm_target then
6:             return_value ← E_TMEM
7:         else if node_info.free_tmem == 0 then
8:             return_value ← E_TMEM
9:         else
10:            allocate_tmem_page(id)
11:            vm_data_hyp[id].tmem_used ← tmem_used + 1
12:            vm_data_hyp[id].puts_succ ← puts_succ + 1
13:            return_value ← S_TMEM
14:        end if
15:        vm_data_hyp[id].puts_total ← puts_total + 1
16:    else if op == GET then
17:        get_succ, page_got ← get_page_from_store()
18:        if get_succ == 1 then
19:            vm_data_hyp[id].gets_succ ← gets_succ + 1 return_value ← S_TMEM
20:        else if get_succ == 0 then return_value ← E_TMEM
21:        end if
22:        vm_data_hyp[id].gets_total ← gets_total + 1
23:    else if op == FLUSH then
24:        deallocate_tmem_page(id)
25:        vm_data_hyp[id].tmem_used ← tmem_used − 1
26:        return_value ← S_TMEM
27:    end if
28:    return return_value
29: end function
```

ones. However, this VM won't be able to obtain additional pages until it releases enough pages below its target or until its target is increased.

2.2.3 Tmem Kernel Module (TKM)

The TKM used in SmarTmem is an extension from the baseline state-of-the-art present in version of the Linux kernel used (version 3.19). This extended TKM functions as an interface between user-space processes and the Tmem implementation of the HV. The TKM provides support for the baseline Tmem interface through a series of hypercalls. It also provides additional support for special interrupts generated by the HV to initiate communication with user-space processes, which requires a series of custom-made hypercalls.

The TKM forwards the memory statistics sent by the HV to a user-space process (MM) through a netlink socket interface. As mentioned before, the MM uses these statistics to calculate Tmem target allocations, and the TKM forwards this information from the MM back to the HV, for which a series of custom-made hypercalls were also developed.

2.2.4 Memory Manager Process for Tmem Allocation in a Virtualized Computing Node

The MM was created from scratch, and holds the core of the functionality of SmarTmem. As mentioned in Section 2.2.2, the MM receives information from the HV regarding the way the VMs make use of their memory. The MM keeps track of this information across time, generating a historial of how the VMs use Tmem, their Tmem operations and overall system memory status. The MM uses this information to calculate a Tmem capacity target per VM according to custom-made high-level policies.

The complexities associated to determine the Tmem allocations for every VM and the storage of the VM utilization metrics are present in the MM, and not the HV, for the following reasons:

- The HV has to be kept lightweight, and any modifications to the HV layer have to be localized in order not to affect its functionality in other respects.
- Maintain the flexibility on implementing high-level tmem management policies. This allows for more ambitious data analysis strategies on the MM in order to implement more sophisticated policies, and allows the deployment of the policies without requiring frequent HV compilations.
- It allows for policies to be deployed into production with ease, either after tuning policies already implemented or when introducing new ones.

2.2.5 High-Level Tmem Management Policies

Currently, we have implemented three policies in the MM besides the greedy approach used by default (*greedy*), which are: 1) single-node static memory capacity allocation, 2) single-node reconfigurable static capacity allocation, and 3) single-node smart allocation policy. We will proceed to explain and analyze each of the policies.

Single-node Static Memory Capacity Allocation (sn-static-alloc)

This policy divides the available Tmem capacity equally across all Tmem-capable VMs, as described by Algorithm 2. This policy secures a fair share of Tmem for every VM, when each VM has a similar demand for memory.

The MM sends target allocations to the HV when it has calculated a new Tmem target. This feature is implemented within the *send_to_hypervisor*() function. If no changes are detected, then no transmission takes place, avoiding unnecessary communication overhead.

For the static allocation policy, the targets are only modified when a new VM is created and registers itself with Tmem or when a VM is destroyed. After that, the targets will remain the same as long as the number of active VMs does not change, regardless of the applications they are running. This policy is designed to avoid starvation on the available Tmem capacity, but it might allocate pages unnecessarily to a VM that does not need them.

Single-node Reconfigurable Static Allocation (sn-reconf-static)

This policy divides the available Tmem capacity equally among the VMs that are actively using Tmem. That is, the policy monitors the activity of each Tmem-capable VM, and

Algorithm 2 Single-node Static Allocation Policy

```
1: function STATIC_POLICY(memstats, node_info)
2:     num_vms ← memstats.vm_count
3:     ms_prev ← memstats.prev
4:     local_tmem ← node_info.total_tmem
5:     mm_target ← local_tmem/num_vms
6:     for i ← 1, num_vms do
7:         mm_out[i].vm_id ← memstats.vm[i].vm_id
8:         mm_out[i].mm_target ← mm_target
9:     end for
10:    send_to_hypervisor(mm_out)
11: end function
```

Algorithm 3 Single-node Reconfigurable Static Allocation Policy

```
1: function RECONF_STATIC(memstats, node_info)
2:     num_vms ← memstats.vm_count
3:     num_active_vms ← 0
4:     for i ← 1, num_vms do
5:         puts_failed ← memstats.vm[i].cumul_puts_failed
6:         if puts_failed > 0 then
7:             num_active_vms ← num_active_vms + 1
8:         end if
9:     end for
10:    local_tmem ← node_info.total_tmem
11:    for i ← 1, num_vms do
12:        mm_target ← local_tmem/num_active_vms
13:        mm_out[i].vm_id ← memstats.vm[i].vm_id
14:        mm_out[i].mm_target ← mm_target
15:    end for
16:    send_to_hypervisor(mm_out)
17: end function
```

allocates an equal share of the Tmem capacity to each VM that has performed at least one Tmem *put*, initially allocating no Tmem capacity to any VM. The pseudocode is given in Algorithm 3.

If additional VMs start issuing *puts*, then the Tmem capacity allocation is reconfigured, and every VM that *puts* at least once will get an equal amount of the Tmem capacity. This will remain so during the lifetime of the VMs. The main drawback of this approach is that it requires for the VM to swap a number of times before getting any Tmem capacity, since their initial target allocation is equal to zero. This is because the latency between the time a VM performs its first Tmem operation until the time that the targets are reset is roughly one second. On the other hand, this policy prevents the HV from allocating memory to a VM unnecessarily as it might occur for *static-alloc*.

Single-node Smart Allocation (smart-alloc)

This policy assigns Tmem capacity to each VM depending on the memory demand that is measured by the HV. It monitors every VM, and when it detects that a VM cannot acquire

more Tmem because it exceeded its target, the policy increases its target by a percentage P of the total *local* Tmem capacity, which is constant. It is possible that all VMs have swapped during the last interval, needing a target increase by a percentage P to avoid performance loss. Algorithm 4 shows the pseudo-code for *smart-alloc*.

Algorithm 4 uses the $failed\_puts$ that have occurred in the last sampling interval as a measure of a VM's swap activity, instead of calculating the corresponding rate, as in [34]. The sampling interval is fixed at one second. In this case, it is not necessary to establish communication with other nodes, nor to keep track of node IDs, and the total Tmem allocation of the node does not change (as opposed to [34]). Also, *smart-alloc* refrains from sending targets to the HV if they do not change since the last modification, a condition evaluated by the function *send_to_hypervisor* in line 34.

If all the VMs have their allocation increased by a percentage P at every interval, eventually the Tmem pages could end up being overallocated i.e. the sum of the targets of every VM ($sum\_targets$, in lines 4 and 25) is larger than the amount of local Tmem pages (line 27). If this happens, it is unlikely that the VMs will be able to meet their targets. To avoid this, we make sure that the following condition is met:

$$\sum_{i=0}^{i=n-1} vm\_data_{\mathrm{MM}}[i].mm\_target = local\_tmem \tag{2.1}$$

When the pages are over-allocated, we reduce the target of every VM according to the following equation:

$$target_{vm_i} = \frac{local\_tmem \times vm\_data_{\mathrm{MM}}[i].mm\_target}{\sum_{i=0}^{i=n-1} vm\_data_{\mathrm{MM}}[i].mm\_target} \tag{2.2}$$

By enforcing Equations 2.1 and 2.2, the policy ensures that: 1) *all local Tmem pages* are always assigned to a VM i.e. there are no Tmem pages that will remain unallocated, and 2) the total amount of *local Tmem pages* that are assigned to the VMs does not exceed the amount of Tmem pages available in the node. If these conditions are not enforced, and Tmem overallocation occurs, the VMs will compete for the excess pages, overriding the target allocations and unable to meet their targets. Equation 2.2 is implemented in lines 27–33.

The policy decreases the target of a VM by a percentage P of the amount of Tmem *it has* if the policy detects that a VM is using less pages than its target plus a *threshold* value (lines 16–21). This avoids premature target decrements which might cause the targets to oscillate resulting in an unstable policy.

2.3 Benchmarking and Experimental Framework

We tested SmarTmem using nested virtualization. We used a VirtualBox image with Xen 4.5.1, with all VMs running Ubuntu 14.04 with kernel 3.19. The VirtualBox environment was executed with two processor cores, 6 GB of RAM, 2 GB of swap and a 32 GB hard drive. The physical system in which VirtualBox was executed had a 4-core Intel Core i7 processor running at 2.1 GHz with 8 GB of RAM, 4 GB of swap and a 320 GB hard drive. Even though this is not a server class hardware, it was readily available at the time of the experiments

and it proved to be enough to test the functionality and effectiveness of SmarTmem for the workloads and datasets used. The MM was implemented in C language and running in user space in the privileged domain.

We used the CloudSuite Benchmarks [31] and a micro-benchmark called Usemem to evaluate SmarTmem. Usemem is a synthetic micro-benchmark that allocates an incremental amount of memory as it executes, starting from 128 MB and increasing it by 128 MB increments. Once it allocates a region of memory, it traverses it linearly performing write/read operations. Once it completes a run through a region, it then allocates a larger block, until it reaches 1 GB. Once there, Usemem stops increasing the allocation but continues to write/read on the 1 GB of memory allocated until stopped. All of these benchmarks were chosen because they generate an important amount of variable memory pressure on the VMs, which allows to see the evaluation of SmarTmem and the policies implemented.

To demonstrate the effectiveness of SmarTmem and the policies, we ran multiple VMs in different execution scenarios, described in Table 2.2. This table shows the scenario names, the VM parameters (CPU and RAM) and a description of the benchmark execution, together with the datasets used. Every scenario is executed five times with every policy. In every scenario, the amount of tmem enable was 1 GB, except for the *Usemem* Scenario in which only 384 MB was enabled.

The scenarios created, the configurations of the VMs and the datasets were chosen in order to create a reasonable amount of memory pressure over the available hardware, usually within two or three times the initial memory capacity of the VMs [100]. The use of microbenchmarks also provides important insight on the allocation capabilities of SmarTmem, because it provides us with a set of applications with regular access patterns and working set sizes that create a memory pressure profile that is possible to analyze and correlate with the allocation behavior of SmarTmem. In this way, it is possible to observe the effectiveness of SmarTmem in managing the available Tmem.

2.4 Results and Discussion for SmarTmem

2.4.1 Results for Scenario 1

Figure 2.2 shows the average running times (less is better) and standard deviations for Scenario 1, with the different policies. We vary P for *smart-alloc* and compare the running times against the case without Tmem support (*no-tmem*).

Figure 2.3 shows the amount of Tmem capacity of each VM for (a) *greedy* and (b) *smart-alloc* with $P = 0.75\%$. In Figure 2.3(a), VM3 reaches a Tmem capacity peak during the first run, while VM1 and VM2 can't reach a fair share. During the second run, VM2 is unable to reach a fair share, while the other two VMs take a higher portion of Tmem. Figure 2.3(b) shows that *smart-alloc* maintains similar Tmem capacity across VMs with some adaptability, and it also shows the allocation target for VM3 and the way it is enforced.

In this context, fairness is define qualitatively as being equivalent to a fair proportion of the available Tmem capacity with respect the memory demand of the application running in the VM. In the cases analyzed so far, the memory demand of all the VMs exceed the capacity of the node, which means that is necessary for all the VMs to have an equal proportion of the available Tmem (or approximate the equality) in order to reach fairness. The better the equal proportion condition is met, the better fairness is reached.

Table 2.2: List of scenarios used for benchmarking. In all cases, we deploy 3 VMs.

Scenario Name	VM Parameters	Comments
Scenario 1	VM1, VM2, VM3: 1 GB RAM, 1 CPU	All VMs execute in-memory-analytics once simultaneously, sleep for 5 seconds and execute it again. The data set was taken from [43].
Scenario 2	VM1, VM2, VM3: 512 MB RAM, 1 CPU	All VMs execute graph-analytics once. The first two VMs launch the benchmarks simultaneously, and the third one launches it 30 seconds later. They all use the same dataset provided by [93], [92], [91].
Usemem Scenario	VM1, VM2, VM3: 512 MB RAM, 1 CPU	All VMs execute *usemem*. VM1 and VM2 start executing *usemem* simultaneously, and VM3 starts when VM1 and VM2 attempt to allocate 640 MB of memory. From this point on, all VMs run concurrently and they are stopped simultaneously when VM3 attempts to allocate 768 MB.
Scenario 3	VM1, VM2: 512 MB RAM, 1 CPU VM3: 1 GB RAM, 1 CPU	VM1 and VM2 execute graph-analytics and VM3 execute in-memory analytics. VM1 and VM2 launch execution simultaneously, and VM3 launches it 30 seconds later. All VMs use the dataset from [93, 92, 91].

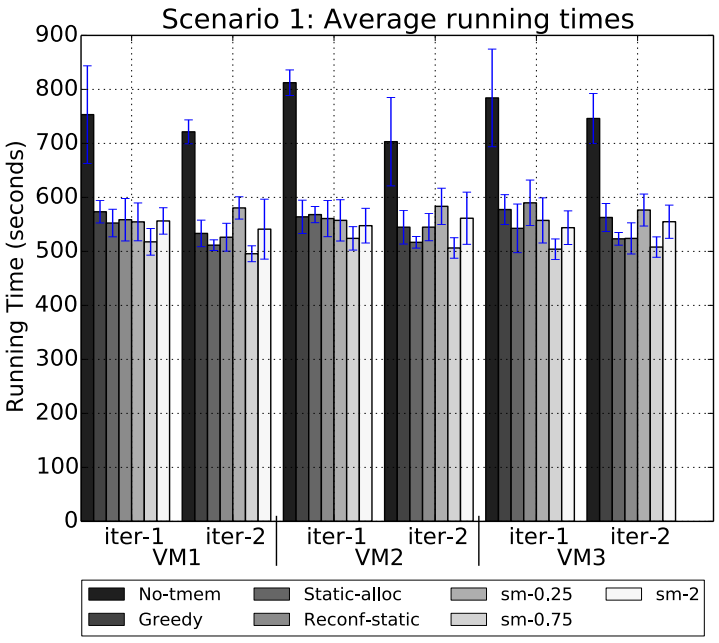


Figure 2.2: Running times for Scenario 1. SM refers to *smart-alloc*.

Some policies were not able to obtain performance benefits. *reconf-static* showed almost no improvement in most of the VMs. Something similar happens to *static-alloc* in the first

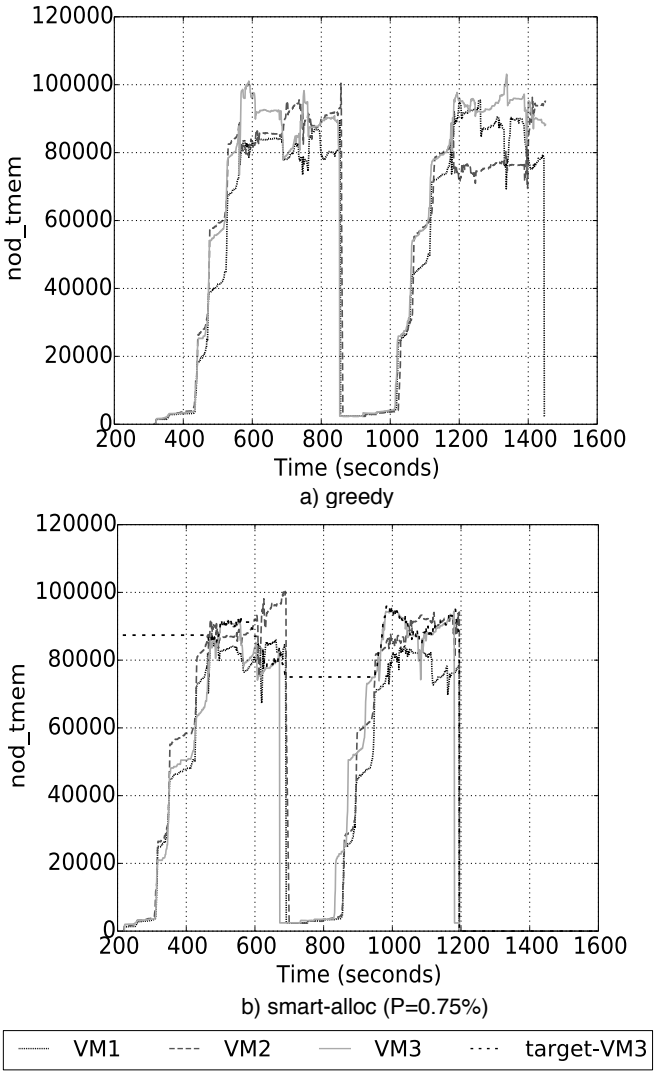


Figure 2.3: Utilization of the Tmem capacity (nod-tmem) by every VM in number of pages for Scenario 1. The label *target-VM3* refers to the target allocation of the third VM.

run of VM2. The case of *smart-alloc* with $P = 0.25\%$ performed poorly for almost every case. The lack of improvement shown by some policies can be attributed to their lack of adaptability to the changes on the memory demand. For example, when using *smart-alloc* with $P = 0.25\%$, the allocation targets increase at a slower pace, causing the VMs to swap more, demonstrating the need for the policy to adapt quickly to the VM's memory demand, for which it is necessary to tune the parameters of *smart-alloc*.

The best runtime performance (fastest) is obtained for *smart-alloc* (referred to as *sm* in Figure 2.2) with $P = 0.75\%$. Its standard deviation barely overlaps with *greedy*, demonstrating its clear benefits. *smart-alloc* with $P = 0.75\%$ runs faster than *no-tmem* by a maximum of 35.7% (first run of VM3) and by a minimum of 28% (second run of VM2). It also runs faster than *greedy* by a maximum of 12.7%, corresponding to the first run of VM3, and by a

minimum of 7.1%. These results demonstrate the need to manage Tmem with an adequate management policy.

2.4.2 Results for Scenario 2

The average running times for Scenario 2 are shown in Figure 2.4 and Figure 2.5 shows the use of Tmem capacity for the VMs. In this case, VM1 and VM2 take a lot of Tmem as they start executing since their memory demand rapidly increases, putting significant pressure on the Tmem capacity. This is shown in Figures 2.5(a) and (b), for the case of *greedy* and *smart-alloc* with $P = 6\%$, respectively. In Figure 2.5(a), the third VM is unable to obtain a fair share of Tmem. But in Figure 2.5(b), when using *smart-alloc*, despite the fact that the first two VMs initially take up a large amount of Tmem capacity really fast, the third VM is able to eventually obtain a fair amount. This shows that *smart-alloc* is at the same time adaptive and fair on how it allocates Tmem.

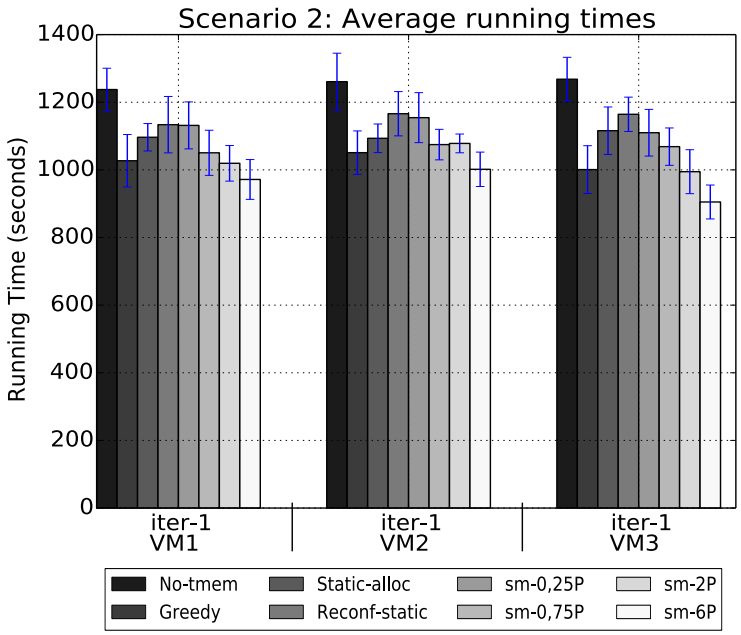


Figure 2.4: Running times for Scenario 2.

In this case, the best performance is obtained with *smart-alloc* with $P = 6\%$, performing better than *no-tmem* by a minimum and a maximum of 21% (for VM3) and 28% (for VM1), respectively. It performs better than *greedy* by minimum and a maximum of 4.7% (for VM2) and 9.6% (for VM3), respectively. This is in contrast to Scenario 1, for which the best benefit for *smart-alloc* was obtained with $P = 0.75\%$. The static policies do not present any improvement in this scenario. These results demonstrate that fairness in Tmem allocation among VMs and quick adaptiveness to memory demand spikes are necessary to improve performance. In this scenario, *smart-alloc* is able to achieve both.

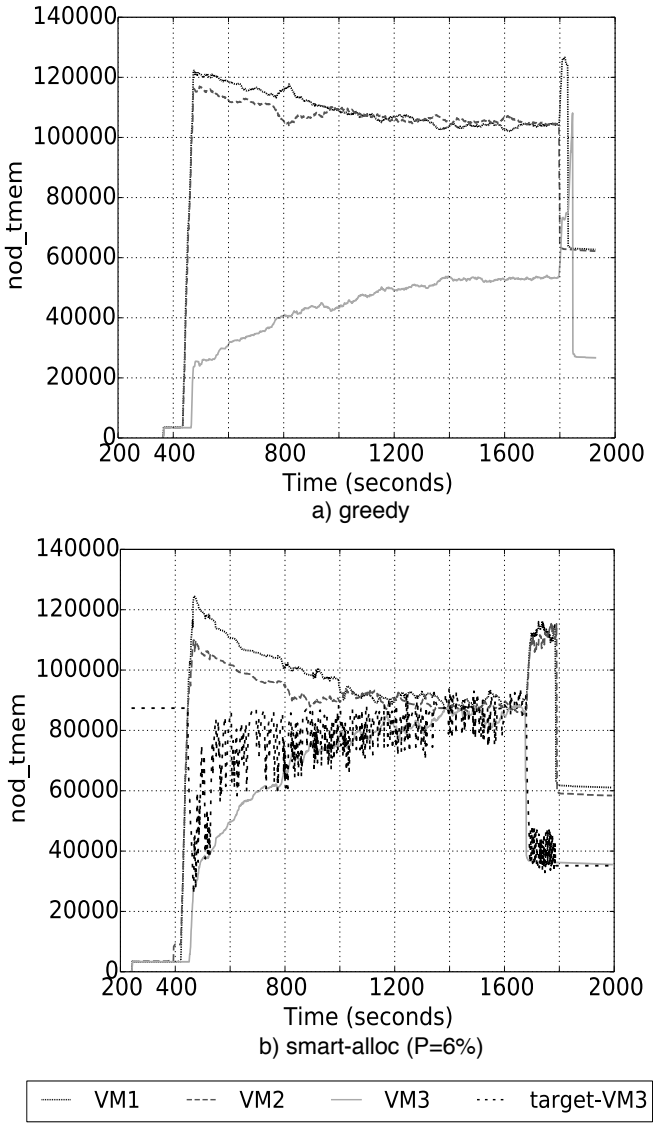


Figure 2.5: Tmem use of all VMs in Scenario 2 for a) greedy, and b) *smart-alloc* with $P = 6\%$

2.4.3 Results for the Usemem Scenario

The average running times for the *Usemem* Scenario are shown in Figure 2.6. The *usemem* micro-benchmark is designed to generate a similar memory demand in each VM, giving insight on how the policies behave.

The running times were approximately equal for all VMs when using *static-alloc* for the same amount of memory allocated. Nevertheless, improvements were observed for *reconf-static* when allocating 640 MB for VM2 and consistently across all allocations for VM3. In this case, *static-alloc* policy ensures fairness (not adaptiveness) on how Tmem is allocated, regardless of how much a VM is swapping, and *reconf-static* follows the same behavior depending how many VMs are actually swapping (limited adaptiveness).

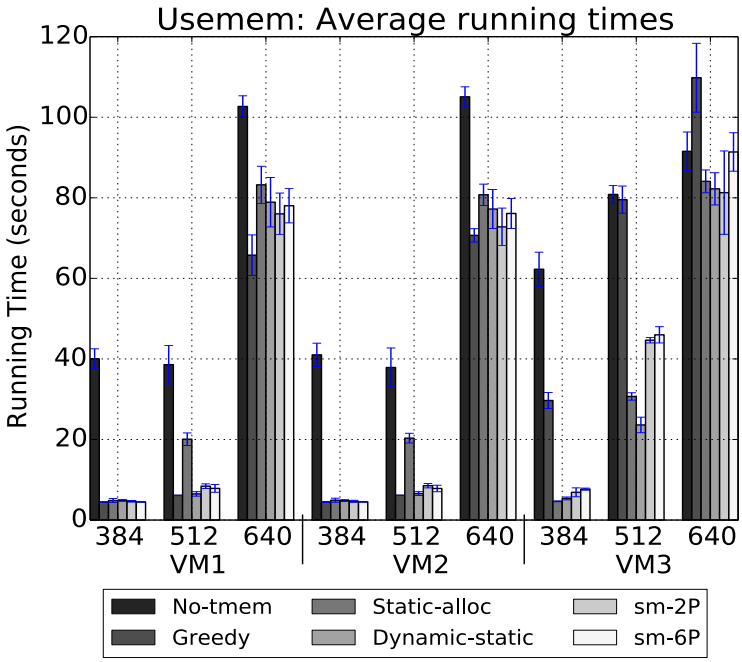


Figure 2.6: Running times for *usemem* scenario.

Notice how *static-alloc* and *reconf-static* perform worse than *greedy* for VM1 and VM2, but performing better for the third VM across all memory allocations. This is because when VM1 and VM2 are executing, the Tmem capacity is not under pressure, but the policies are still enforced restricting the access to Tmem unnecessarily. Remarkably, *greedy* performs significantly worse than *no-tmem* for the last allocation of *usemem* in VM3 (640 MB). However, all the other policies perform better than *no-tmem* for VM3.

Figure 2.7(a) shows that VM3 struggles to obtain Tmem pages when using *greedy* as the Tmem capacity is under pressure. This is similar to the case presented in Figure 2.5(a) for Scenario 2. However, in Figure 2.7(b) and Figure 2.7(c), corresponding to *reconf-static* and *smart-alloc* with $P = 2\%$, every VM is able to obtain a fair share of memory.

The case of *smart-alloc* allows for the VM1 and VM2 to take much more memory than with *reconf-static*, but still less than what *greedy* allows, as seen on Figures 2.7(a) and 2.7(c). In this case, *smart-alloc* exhibits more *adaptiveness*. *reconf-static* performs better for when VM3 allocates 384 MB and 512 MB, as previously seen. These results demonstrate that *reconf-static* and *static-alloc*, both of which are more oriented towards fairness, despite not showing the best performance in Scenarios 1 and 2, still present better performance than *smart-alloc* for these *Usemem* cases.

When the third VM starts executing *Usemem*, *smart-alloc* achieves fair allocation at a slower pace. In Figure 2.7(c), the three VMs take more Tmem than the limit imposed by *reconf-static*, demonstrating the better adaptiveness of *smart-alloc*.

These results are interesting because they highlight a trade-off between the *adaptiveness* of a policy to quickly respond to changes in memory demand (as *smart-alloc*) and its *ability* to fairly allocate Tmem (as *static-alloc* and *reconf-static*). The more responsive a policy

is, the more Tmem it will let a VM take, but as other VMs come under pressure, it will be harder to reach fairness.

2.4.4 Results for Scenario 3

The average running times for Scenario 3 are shown in Figure 2.8, and Figure 2.9 show the Tmem capacity used by the three VMs for *greedy*, *static-alloc*, *reconf-static* and *smart-alloc* with $P = 4\%$.

The *graph-analytics* benchmark starts by making use of a large amount of Tmem. For *greedy*, VM1 and VM2 take up half of the available memory each, leaving almost no memory available for when VM3 increases its demand. This helps explain why VM3 executes very slow with *greedy*. Notice the way *static-alloc* is very rigid (not adaptive), setting an upper-bound for all three VMs. This seems to benefit VM3 very much and, *surprisingly*, VM1 and VM2 also improved.

In Figure 2.9(c), we see the case with *reconf-static*. This policy allows for VM1 and VM2 to share half of the available capacity until VM3 starts swapping. By this time, the targets are reconfigured but VM3 is still unable to reach a fair share. because pages are released by the VM at a slower pace, despite the targets already being modified.

Notice how *smart-alloc* allows for VM1 and VM2 to take up a similar amount of Tmem when compared to *greedy*, but reduces their share as soon as VM3 swaps, but VM3 is unable to reach a fair share, allowing VM1 and VM2 to have more Tmem. This helps explain the results observed in Figure 2.8, where VM1 and VM2 run faster for *smart-alloc* but slower for *static-alloc*, while it is the opposite for VM3. Again, this highlights the trade-off between adaptiveness and fairness.

All the policies improve over *greedy* consistently, except for *reconf-static* which fails for the first two VMs. For VM1 and VM2, *smart-alloc* with $P = 4\%$ performs better than the other policies. For VM3, the best performance is obtained with *static-alloc* by a very significant margin. The maximum and minimum improvements over *no-tmem* are 40% (in VM3 for *static-alloc*) and 22% (in VM for *smart-alloc* with $P = 4\%$), respectively, while the maximum and mininum improvements over *greedy* are 35% (in VM3 for *static-alloc*) and 10.8% (in VM2 for *smart-alloc* with $P = 4\%$), respectively.

2.5 Related Work

Many research efforts seek to optimize memory allocation by implementing better control strategies for memory ballooning [120, 100, 95, 125, 51, 15, 80, 63]. One similarity we have with respect to Liu et al. [63] is that we also implement a user-space process to manage the allocation of memory capacity to every VM, albeit their communication mechanisms are slightly more complicated, since they need to communicate the allocation targets to every VM. In our case, it is the hypervisor in charge of doing so, which reduces significantly the memory and communication overheads.

Both Zhao et al. [125] and Liu et al. [63] predict the memory demands of the VMs. Whereas, Zhao et al. [125] predicts the working set size of the VMs to dictate target memory allocations to the balloon driver, Liu et al. [63] predicts the total memory footprint including swap. In contrast, rather than predicting memory requirements, we shift the core of the

problem into directly reducing the amount of swapping to disk, thus keeping the data in system memory. In addition, we focus entirely on Tmem instead of memory ballooning.

Smith et. al [100] designed a system for memory resource management that includes an OS daemon that implements memory management policies, that includes important modifications to the balloon driver in order for the memory allocations dictated by the daemon are executed in an efficient way, and also includes modifications to the paging mechanism of the hypervisor. SmarTmem uses a similar evaluation methodology used in [100], in the types of benchmarks we use and the amount of memory pressure we generate in the node in order to evaluate the effectiveness of our mechanism. However, our work differs from the work in [100] because SmarTmem focuses on memory optimization through Tmem, which is a mechanism for hypervisor-level caching, and the policies in [100] do not focus in hypervisor-level caching.

Zhang et al. [120] also exploit the capabilities of ballooning and they enhance some of its functionalities in order to prevent paging by the VMs. They note the significant time delays necessary to trigger the balloon driver and to reclaim idle memory before reallocation takes place, during which swap operations occur. In order to solve this situation, they introduce the presence of a host memory region shared across the VMs within the node, and they provide a mechanism to allocate and deallocate memory that takes or releases memory from the shared memory pool. In our work, we leverage the concept of a shared memory pools through the use of Tmem, which is designed to pool all idle and fallow memory in the computing node. However, we differ from [120] because we explicitly leverage the available Tmem capacity as a cache for pages that have been swapped out once the VM becomes under memory pressure, effectively using Tmem as a hypervisor-level cache. Our approach also prevents swap operations from taken place, but still leveraging the paging mechanism of the kernel, whereas the mechanism in [120] avoids it. Our approach is particularly relevant for Chapters 5 and 6, in which Tmem is used to access remote memory resources [36, 34] in cases which the remote memory is not available to the VMs as traditional DRAM.

Venkatesan et al. [110] use Tmem, in both cleancache and frontswap mode, in a computer node with non-volatile memory (NVM) alongside traditional DRAM. They use Tmem to access the NVM where the disk device would have normally been used. They divide the NVM space into a clean region (for clean cache pages) and a swap region (for processes that attempt to swap to disk). They implement a two-level cache for clean pages: the first level cache is implemented in DRAM and the second-level cache is located in the NVM space, and they implement cache strategies to manage pages across these two levels.

One difference between SmarTmem and Venkatesan et al. [110] is that we only make use of Tmem on its frontswap mode. The type of applications we run from CloudSuite are memory intensive, and the processes of these applications dynamically allocate memory pages that are not backed by the filesystem. Thus, when data is evicted from memory, those pages have to be stored in frontswap.

2.6 Conclusions and Future Work

This chapter introduced a mechanism to manage efficiently memory resources through Tmem in a virtualized computing node executing multiple applications and VMs. Our results demonstrate the need to intelligently and efficiently allocate Tmem capacity to VMs in a

single node, showing significant performance improvements, of up to 35%, over the default greedy policy with which Tmem is currently managed. At the same time, we identified a trade-off between the policy's adaptiveness and its fairness. These results will be relevant also for Tmem-based memory disaggregation solutions and for other systems in which Tmem is deployed. [110, 34].

We demonstrated that the policies implemented in SmarTmem ensure *fair* capacity allocation in different scenarios, while being able to adapt to the changing memory demand of the VMs. The main and singular contribution of this work is to provide a framework (architecture of the software stack) to optimize Tmem capacity, and the management policies implemented so far were designed to demonstrate the performance improvements that can be obtained when optimizing Tmem.

Considering this, for future work it would be necessary to develop policies that can have a better balance between adaptiveness and fairness in order to better balance the performance of the applications inside VMs. This implies using policies from other research works that implement some form of memory management, even for cases in which other memory abstractions different to Tmem are implemented (ballooning, for example). However, porting policies in this way makes it necessary to make a thorough analysis on the dependency of those policies in the specific memory abstraction. Moreover, it is also of great interest to enhance SmarTmem to include the management of memory exposed to the VM as RAM.

It will also be necessary to analyze its effectiveness with a larger set of workloads in cloud data centers such as data caching frameworks and web search. In this context, it will be necessary to implement ways for SmarTmem to optimize other performance metrics such as energy costs, cost of ownership, to define the concept of fair allocation in a quantitative way and to evaluate tail latency, which is particularly relevant for workloads such as Web Search.

Algorithm 4 Smart Allocation Policy

```
1: function SMART-ALLOC_POLICY(memstats, node_info, P)
2:     local_tmem ← node_info.total_tmem
3:     num_vms ← memstats.vm_count
4:     sum_targets ← 0
5:     for i ← 1, num_vms do
6:         put_total ← memstats.vm[i].puts_total
7:         put_succ ← memstats.vm[i].puts_succ
8:         failed_puts ← put_total − put_succ
9:         if failed_puts > 0 then
10:            curr_tgt ← memstats.vm[i].mm_target
11:            incr ← (P × local_tmem)/100
12:            mm_target ← curr_tgt + incr
13:        else
14:            curr_tgt ← memstats.vm[i].mm_target
15:            curr_use ← memstats.vm[i].tmem_used
16:            difference ← curr_tgt − curr_use
17:            if difference > threshold then
18:                mm_target ← ((100 − P) × curr_tgt) /100
19:            else
20:                mm_target ← curr_tgt
21:            end if
22:        end if
23:        mm_out[i].vm_id ← memstats.vm[i].vm_id
24:        mm_out[i].mm_target ← mm_target
25:        sum_targets ← sum_targets + mm_target
26:    end for
27:    if sum_targets > local_tmem then
28:        factor ← local_tmem/sum_targets
29:        for i ← 1, num_vms do
30:            new ← factor × mm_out[i].mm_target
31:            mm_out[i].mm_target ← new
32:        end for
33:    end if
34:    send_to_hypervisor(mm_out)
35: end function
```

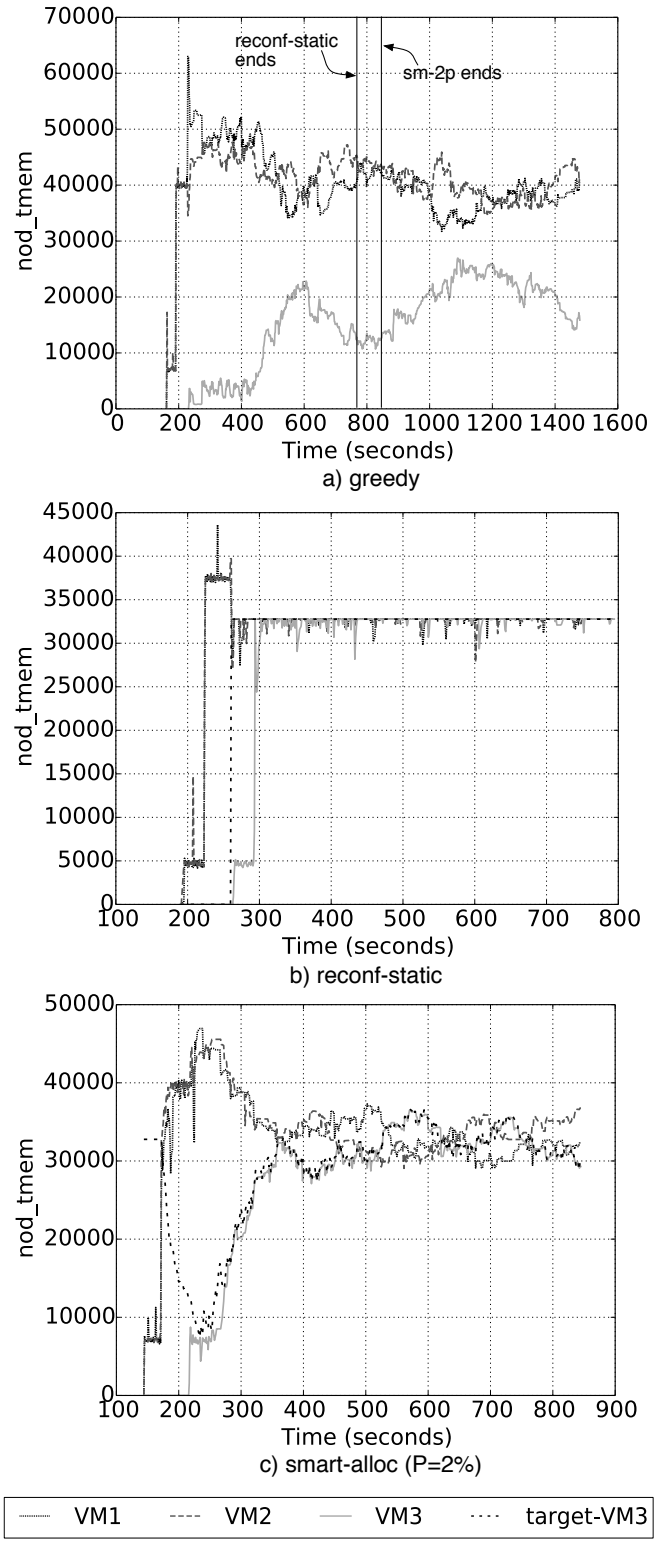


Figure 2.7: Tmem use of all VMs in *usemem* for a) greedy, b) *reconf-static* and c) *smart-alloc* with $P = 2\%$. The vertical lines in a) show the time when *usemem* completes execution.

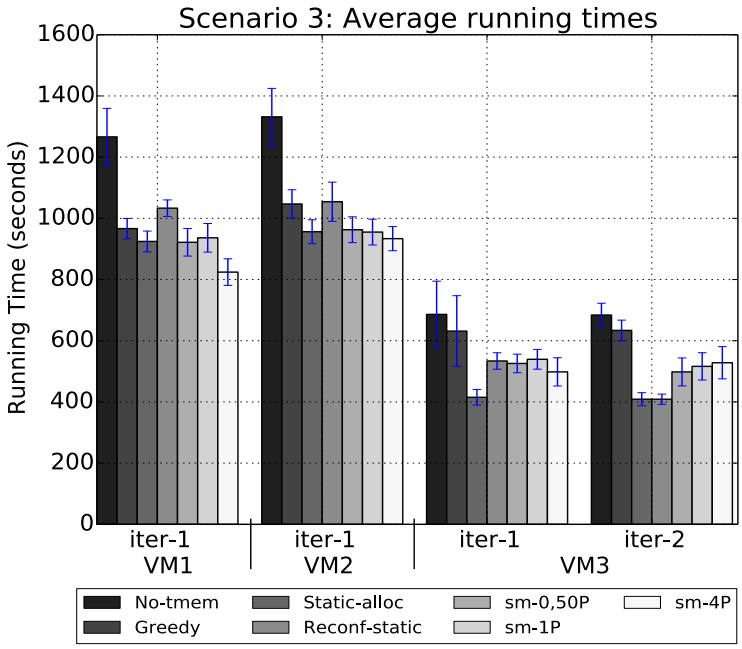


Figure 2.8: Running times for Scenario 3.

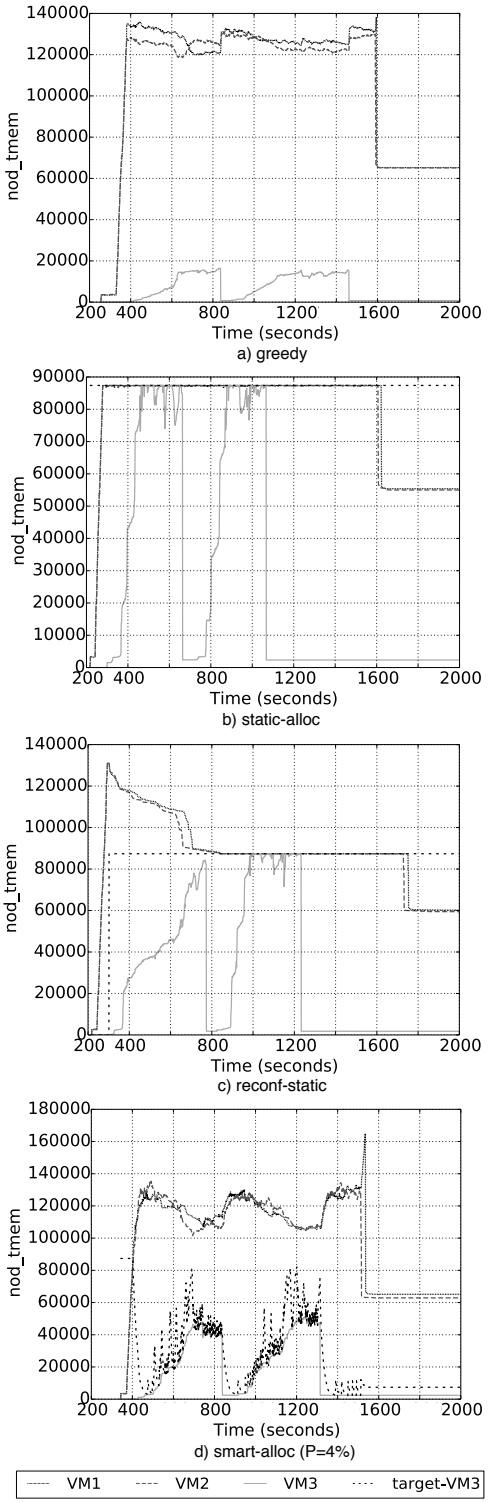


Figure 2.9: Tmem use of all VMs in Scenario 3 for a) *greedy*, b) *static-alloc*, c) *reconf-static* and d) *smart-alloc* with $P = 4\%$.

Chapter 3

Using continuous action Reinforcement Learning for Dynamic RAM allocation

The main focus of the previous chapter was to optimize memory through Tmem within a single node. This is a relevant solution for systems in which Tmem is deployed, as in cases in which Tmem is used to aggregate memory across nodes, as we shall see in Chapters 5 and 6, or when it is deployed in systems with heterogeneous memory hierarchies [110].

But optimizing memory in a single node also requires the optimization of the memory given to the VMs as RAM. The memory that becomes accessible to the VMs in this way has a different set of constraints and thus it is managed using a different set of mechanisms and variables. In these cases, it is very common to drive the management policy by running a time-consuming memory footprint analysis or by wastefully overallocating memory and running fewer VMs. In many cases, this is done only considering the memory demand of each VM without having a node-wide view.

Ideally, a high-level control mechanism would dynamically find the best memory allocation for the VMs. There are many solutions for the dynamic memory allocation problem, some of which use machine learning in some form.

This chapter introduces CAVMem (continuous action Algorithm for Virtualized Memory Management), a proof-of-concept mechanism for a decentralized dynamic memory allocation solution in virtualized nodes that applies a *continuous action* RL algorithm called Deep Deterministic Policy Gradient (DDPG).

3.1 Introducing CAVMem

In a virtualized node inside a cloud infrastructure, resource manamagent is critical to drive performance. When multiple VMs are active in a node, its physical memory is allocated among the VMs in order to optimize throughput and prevent memory starvation. This memory allocation problem is difficult to solve because the memory demand of a VM changes continuously. Many solutions for this problem are limited by the complex relationship between the memory allocated to a VM, the applications' behavior and their performance, for which RL provides a good alternative.

Different approaches of RL have been applied to the resource management problem in virtualized nodes [87] which rely on both the discretization of states and actions. Discretization introduces some limitations, mainly combinatorial explosion: as the number of states/action grows, the problem becomes unsolvable [60]. In the context of memory allocation, discretization restricts the granularity at which memory can be allocated, limiting the opportunities for better optimization. Another limitation of current RL approaches for resource management is that they deploy a single agent responsible for re-allocating memory. Such a centralized approach has scalability and flexibility restrictions, since it introduces a traffic bottleneck and a single point of failure.

CAVMem, presented in this chapter, provides a distributed continuous action RL formulation, avoiding the limitations of discretization and centralization. To the best of our knowledge, this is the first such formulation of the memory allocation problem.

CAVMem is initially designed with DDPG, but other RL algorithms were also implemented, namely Q-Learning [114] (QL) and Deep Q-Learning [79] (DQL). These are compared in a model environment that simulates certain aspects of a virtualized computing node. Application traces taken from SPEC benchmarks were used to model the memory demand of applications, assuming uniform distribution of memory access across the memory capacity used by the application model.

In summary, the contributions of this chapter are:

1. Formulation of the memory management problem as a distributed continuous action MDP. This formulation supports an unlimited and variable number of VMs.

2. Development of a continuous action off-policy model-free reinforcement learning algorithm for dynamic memory allocation

3. Comparison between three reinforcement learning approaches: a) CAVMem with DDPG (continuous action space), b) CAVMem with QL (tabular, non-continuous action) and c) CAVMem with DQL (non-continuous action). We also compare against the static policy that divides memory equally.

In order to obtain linear scaling and be able to dynamically add and remove VMs, CAVMem has one agent per VM connected via a lightweight coordination mechanism. The agents learn how much memory to bid for or return, in a given state, so that each VM obtains a fair level of performance subject to the available memory resources. Our results in this chapter show that CAVMem with DDPG performs better than QL and a static allocation case, but it is competitive with DQL. However, CAVMem incurs significant less training overheads than DQL, making the continuous action approach a more cost-effective solution.

3.2 The Context for CAVMem: Memory Management and Reinforcement Learning

3.2.1 Memory Management in Virtualized Nodes

The dependence of Cloud services on virtualization technology and the way virtualization is used to share a node's resources among VMs were clearly explained in Chapter 1. For the particular case of memory, when a VM is created, it is allocated a portion of the node's memory. If the applications executing inside the VM increase their aggregated memory demand, it is very possible that this changing and increasing memory demand exceeds the VMs initial memory allocation. In this case, the VM may swap data to its (virtual) disk device(s), suffering a significant performance loss. Thus, it is necessary to rebalance the memory allocation and optimize overall system performance, minimizing the overall amount of swap operations that take place. This can be achieved by observing application performance in each VM and re-allocate memory for them to reach a performance objective.

Analytical solutions for this problem require the analysis of many VM–application sets and combinations. The number of possibilities is huge, making it intractable to obtain solutions in this way. Instead, heuristics are designed to exploit optimal allocations for known sets. Heuristics are also limited since sometimes they do not generalize well for all possible combinations.

3.2.2 Reinforcement Learning Concepts

Reinforcement learning problems are those in which an agent acts in an environment (usually stochastic) by choosing one or more actions in a sequences of time steps. The objective of the agent is to maximize the reward it accumulates over time. These problems can be modeled as a MDP, which have the following elements:

- A state space, denoted by set S, which are the possible states in which the environment/agent can be in.
- An action space, denoted by the set A, which are the possible actions that the agent can take.
- A transition probability function, denoted by $P : S \times A \times S \rightarrow \mathbb{R}$. The number $P_{i \rightarrow i+1} \in [0, 1]$ represents the probability of an agent transitioning from a state s_i, by doing an action a_i in the current timestep i, to a state s_{i+1} in the following timestep $(i + 1)$. The probability function is totally dependent on the environment and satisfies the Markov property [115].
- A reward function, denoted by $R : S \times A \times S \rightarrow \mathbb{R}$. This function serves to offer an inmediate evaluation of how good or bad is the action a_i in timestep i. The source of the reward function, as the probability function, is the environment in which the agent is in.

The objective of the agent is to learn an optimal policy $\pi : S \rightarrow A$ for which the discounted reward $r_t^\gamma = \Sigma_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$, starting from timestep t with discount factor γ, is maximized. There are many different ways for the agent to learn the optimal policy to

take the right actions (value-based or policy-based) [109], one of the most widely used value-based approaches rely on maximizing the (expected) state-action value function $Q^{\pi}(s,a)$, expressed by the Bellman equation [109, 25]:

$$Q^{\pi}(s_t,a_t) = \mathbb{E}[r(s_t,a_t) + \gamma Q^{\pi}(s_{t+1},\pi(s_{t+1}))] \tag{3.1}$$

Equation 3.1 is the expected reward starting from a given state s_t and taking an action a_t, and then continue to accumulate the rewards choosing actions following the policy π from the next time step. The idea is to estimate the state-action value function (or Q-function for short) through Equation 3.1 as an iterative update, as shown in equation 3.2, which will drive the sequential decision-making process.

$$Q_{i+1}(s_t,a_t) = \mathbb{E}[r(s_t,a_t) + \gamma Q_i(s_{t+1},\pi(s_{t+1}))] \tag{3.2}$$

In [78], the Q-function is estimated through a parametrized neural network (NN) function approximator because the original approach [114] of updating the values of the Q-function iteratively is impractical. This is because in many cases it takes too much time for the Q-function to converge to the optimal values. As a result, the Q-function can then be represented as $Q(s,a;\theta) \approx Q^{*}(s,a)$, where $Q^{*}(s,a)$ is the optimal Q-function.

By approximating the Q-function through a neural network, it is possible to train the neural network by minimising a sequence of loss functions given by [60, 78, 79]:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s_t,a_t;\theta_i))^2] \tag{3.3}$$

in which s_t and a_t denote the state and action at the current timestep t, $\rho(s,a)$ is a probability distribution over sequences of states s and actions a, and y_i is given by:

$$y_i = \mathbb{E}[r(s_t,a_t) + \gamma Q^{\pi}(s_{t+1},\pi(s_{t+1});\theta_{i-1})] \tag{3.4}$$

The approach of using function approximators also presented some limitations, and was generally avoided because of the lack of theoretical performance guarantees and instability issues [60].

In order to overcome these limitations (instability and performance guarantees), the Q-learning algorithm in [79] was further extended with two additional features: 1) experience replay buffer and, 2) iterative update to adjust the values of the Q-function to target values that are updated periodically i.e. a separate target network for y_i. This is represented in equations 3.3 and 3.4, where the NN function approximator is trained (updated) every B timesteps.

This value-based method of learning has been successfully used for problems in which the action space of the agent is discretized [79, 60]. By using the optimal values of the Q-function, it is then possible to have an agent that chooses its actions (use a policy π) in a greedy way relative to the value function. Such policy π would be represented by:

$$\pi_Q = \arg\max_{a\in A} Q(s,a) \tag{3.5}$$

But when applied to continuous action spaces, equation 3.5 cannot be applied in a straightforward way and the whole value-based approach requires additional modifications. This consideration is particularly important for our RAM allocation problem because the

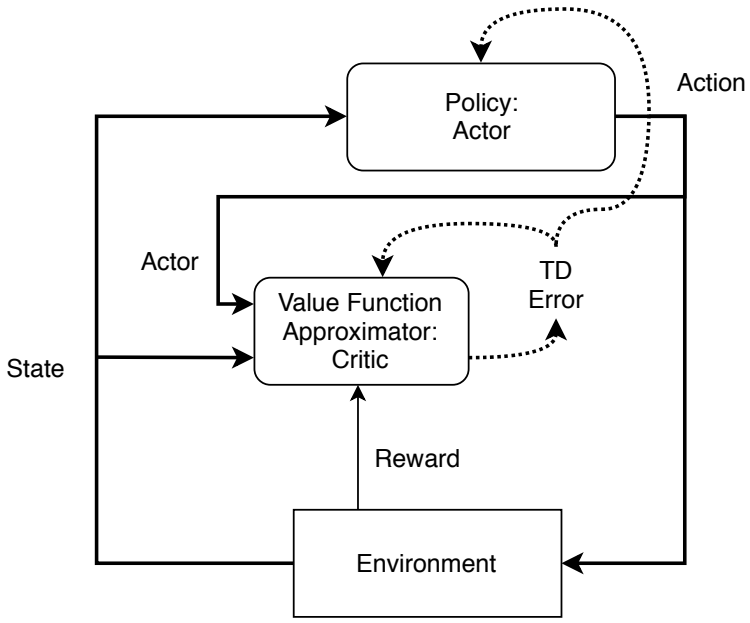


Figure 3.1: Diagram of the actor–critic architecture for the DDPG.

allocation of memory pages in computing nodes using virtualization software needs to be done continuously.

Deep Reinforcement Learning for Continuous Action spaces

Some solutions for continuous action spaces that use RL algorithms rely on some form of discretization of the action space [87, 118] an approach that can sometimes be justified. But in general, discretization reduces greatly the applicability of these algorithms [56, 60]. In the current work, we have not to discretized the action space by allowing the MM to allocate memory continuously at the page granularity.

In our case, we implemented Deep Reinforcement Learning (*DRL*) algorithms for continuous action spaces. One of the most popular of algorithms for these type of problems are the *policy gradient* algorithms. Concretely, we implemented the Deep Deterministic Policy Gradient (*DDPG*) [60] algorithm which uses an actor–critic architecture [54, 53] as the basis of the Deterministic Policy Gradient (DPG) [98] algorithm. Figure 3.1 shows the architecture of a single DDPG learning agent.

The DPG algorithm [98] makes use of a parametrized actor function $\mu(s; \vartheta^{\mu})$ which specifies the current *policy by deterministically mapping states to specific actions* to drive the decision making process, instead of using equation 3.5. A performance objective is defined as in the discrete action space for Q-learning, in this case denoted by [103]:

$$J(\mu_{\vartheta}) = Q(s_t, \mu_{\vartheta}) \tag{3.6}$$

The main idea is then to adjust the parameters μ_{ϑ} of the deterministic policy in the

direction of the performance gradient $\nabla_{\vartheta} J(\mu_{\vartheta})$ expressed as [103], [60], [98]:

$$\vartheta^{k+1} = \vartheta^{k} + \alpha \nabla_{\vartheta} J(\mu_{\vartheta}) \tag{3.7}$$

where α is a real number such that $\alpha \in [0, 1]$. In this approach, the state-action value function $Q(s, a)$ is also approximated through a neural network function approximator ($Q(s, a; \theta)$) in a similar way to [79], represented in the critic network in the actor-critic architecture (Figure 3.1).

However, the critic network are updated in a different way, by making use of "soft" target updates [60], rather than doing a straight-forward copy of the weights. It is important to note that the target networks have the same architecture identical to their correspondent original ones. Thus, the update of the target networks is performed as:

$$\vartheta_{target} = \tau \vartheta_{current} + (1 - \tau) \vartheta_{target} \tag{3.8}$$

$$\theta_{target} = \tau \theta_{current} + (1 - \tau) \theta_{target} \tag{3.9}$$

where θ and ϑ are the parameters of the critic (Q-function) and the actor (policy) networks, respectively.

To the best of our knowledge, there is not a published work that uses RL algorithms for continuous action spaces to solve memory allocation problems in this way.

3.2.3 Employing DDPG for Memory Capacity Management in Virtualized Environments

Designing a high-level memory management policy that is both optimal and responsive is a challenging task. This is because choosing general heuristics or analytical models to design a policy in a node that runs a very complicated software stack, with different VMs, with different users, applications, datasets and performance metrics requires analyzing a huge amount of data. By using a RL algorithm, we expect that an optimal policy for memory management can be obtained (or approximated) by the learning agent(s).

3.3 CAVMem: Algorithm for Virtualized Memory Management

In this section, we introduce CAVMem, a mechanism using continuous action RL for memory management in virtualized nodes. We formulate the problem as a Markov Decision Process. The MDP is formulated such that it does not impose limits on the number of VMs the system can manage. It creates a RL agent per VM which has both a local view of the VM state and a node-wide view of the node's state, indirectly passing to a VM information on the behavior of other VMs. The reward is set to optimize system-wide performance while ensuring fairness with respect to the memory usage among the VMs. In the following subsections, we detail how the MDP is formulated, its advantages and the learning mechanism.

3.3.1 Decentralized Strategy for Memory Management

Determining a priori the memory demands of VMs in virtualized servers is complicated because the nodes seldom have prior knowledge of applications on each VM or the number of concurrently active VMs. Memory demand can be estimated by carrying out analyses of the resource utilization in the node over time, but it is difficult to rely on this estimations because:

1. Traffic pattern across data centers implies VMs are added and removed continuously, making it is difficult to anticipate the number of active VMs and the respective memory signatures of their applications.

2. Applications have different execution phases at runtime and obtaining precise application signatures is difficult;

3. Combinatorial complexity of aggregated signatures of different (or same) application complicates the matter even more;

Instead, another approach is to let the VMs ask for the memory they need. In this case, each VM monitors its own resource utilization and bids for memory independently, resulting in a decentralized solution.

Pros and Cons of Decentralized vs. Centralized

CAVMem introduces a de-centralized approach to the dynamic memory allocation. The de-centralized approach has a learning agent for each VM on the node allowing for a VM to have its own individual state-space. The state space for each VM includes state of the node to determine memory availability and some other variables that summarize the state of the node itself (see table 3.1).

The singular advantage that the decentralized approach provides over a centralized approach is *scalability*, that expresses itself in multiple ways. First, CAVMem is designed to adjust itself to the dynamic behavior related to the activation and deactivation of the VMs, since it enforces no constraints on the amount of VMs that can be present in the node. In most centralized approaches, the state space is defined in such a way that the addition of VMs becomes problematic after a certain amount. But in reality, this depends on how the MDP model is defined. Second, CAVMem removes a single point of control, which allows to scale even beyond a single computing node.

3.3.2 Formulating the problem as an MDP

Defining the State space, S

Table 3.1 lists the five state variables for each agent, two of which belong to the VM and the other three are common to all VMs on the node. This state space definition allows the agent to have *some* information about the other active VMs through the information related to the node.

State information from computing node and VMs

	Name	Range	Description
perVM	$msr_t^{VM_j}$	[-1, 1]	RAM miss rate of VM_j, rescaled to range [-1, 1]; i.e. a value of -1 means zero miss rate.
perVM	$M_t^{VM_j}$	[0, 1]	Fraction of total RAM allocated to VM_j.
Node	$avgMsr_t^{node}$	[-1, 1]	Average RAM miss rate across all VMs, equal to $\frac{1}{N}\sum_j msr_t^{VM_j}$.
Node	$msrTgt_t^{node}$	[-1, 1]	Target RAM miss rate for all VMs given by Equation 3.11.
Node	$totalMemUse_t^{node}$	[0, 1]	Fraction of the node's physical RAM that is allocated to VMs, equal to $\sum_j M_t^{VM_j}$.

Table 3.1: State inputs to the actor and critic networks of the DDPG learning agents.

Defining the Action space, A

The action $a_t^{VM_j}$ chosen for VM_j in timestep t is referred to as the VM's **memory bid**, which ranges from -1.0 to 1.0. A positive bid is a request for more memory and a negative bid is an action to release memory. Concretely, the VM agent requests a total memory allocation equal to $(1 + a_t^{VM_j}) \times M_t^{VM_j}$. As discussed later, it may not be possible to fully satisfy the request, for instance when there is insufficient memory for all requests or the memory allocation for the VM would become too small.

Defining the Reward function, R

The reward function encourages a fair level of performance across the VMs by encouraging all VMs to suffer the same level of swapping over the time step. This is done through a penalty (negative reward) proportional to the absolute difference between the RAM miss rate of the VM, given by $msr_{t+1}^{VM_j}$ and the miss rate target, $msrTgt_{t+1}^{node}$:

$$r_t^{VM_j} = -|msr_t^{VM_j} - msrTgt_t^{node}| \times k \tag{3.10}$$

The parameter k, which we set to 1, affects the learning rate. The miss rate target for the next timestep is given by:

$$msrTgt_{t+1}^{node} = avgMsr_t^{node} \times \sqrt{totalMemUse_t^{node}} \tag{3.11}$$

Here, $avgMsr_{node}$ is the average RAM miss rate across all active VMs, defined in Table 3.1. When it is non-zero and there is free memory, the factor $totalMemUse_{node}$ encourages the VMs to use more memory, via a well-known square-root rule of thumb.

Environment Constraints for Decentralized Control

Figure 3.2 shows a high-level view of our system, which include the VMs, the learning agents of each VM and an additional module called the Bid Analyzer (BA). The BA performs a

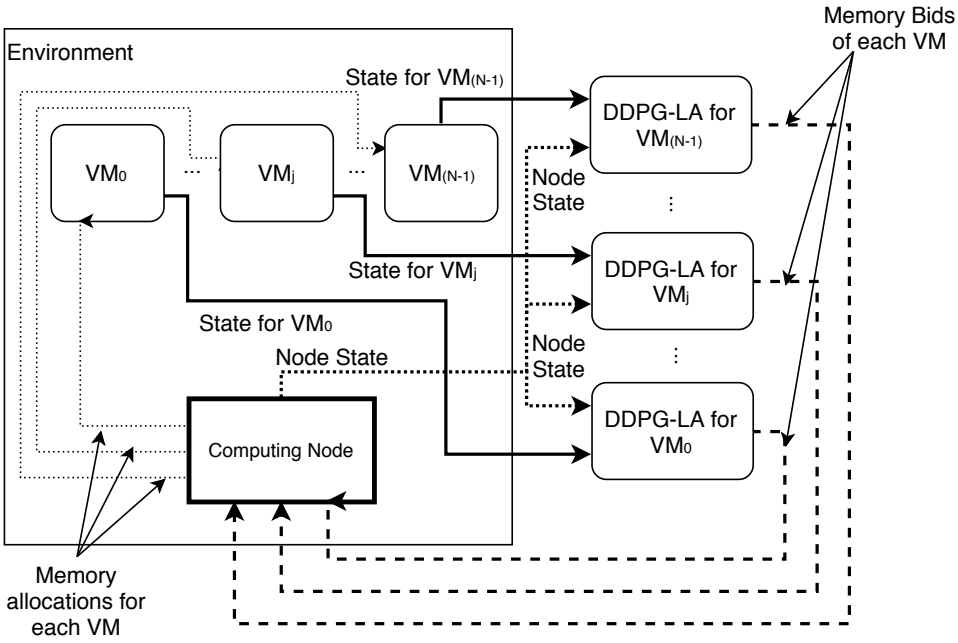


Figure 3.2: Diagram of the DDPG learning Agents, one per VM.

series of high-level control tasks. First, it prevents VMs for bidding less than the minimum amount they need to function (set manually at 384 MB), as well as preventing the VMs to bid more than the actual memory available in the node. Additionally, it also performs a bid arbitration process to prevent aggregated overallocation of memory. The BA also enforces bid prioritization for VMs that issue negative bids.

Action Space Exploration

A classic problem in RL is the *exploration–exploitation dilemma* [102], which captures the need to exploit the "optimal" solution obtained so far but also explore potential better actions. To explore in continuous action domains, the common approach is to use an Ornstein–Uhlenbeck process [11, 58]. This process is a noise signal that creates a Brownian motion around the deterministic action generated by the DDPG agent. At every timestep, a noise signal is added to the action with a probability of ϵ. In our implementation, initially there's a phase of aggressive exploration in which $\epsilon = 1$. This probability reduces over time until it reaches $\epsilon = 0.001$. This process is called epsilon annealing, and it is used to transition from explorative policy to an exploitative one [102].

3.4 Experimental Framework

CAVMem was implemented in Python 3.6 with Tensorflow 1.12 [3] on a system with a 2.3 GHz Intel Core i7 processor and 16 GB of RAM. CAVMem is designed with an underlying DDPG implementation, but we also tested DQL and QL implementations for comparison, and a static allocation policy that divides memory equally among VMs. Whenever CAVMem is

mentioned, it is assumed that the agents are DDPG implementations (continuous action).

The neural networks for DDPG are fully-connected with two hidden layers of 64 units with ReLU activation functions, while the output layers use Tanh. The learning rates for the actor and critic networks are 0.0001 and 0.001, respectively, using the Adam optimizer. The exploration phase lasts for 200 episodes, where each episode has 100 steps. All experiments run for 1000 episodes.

DQL has similar parameters, but it only has one learning rate of 0.001 for the Q-function approximator for 3 discrete actions. The QL implementation also consists of 3 actions, and it has each state space variable quantized in 10 discrete values. This results in a state-action matrix of 300,000 entries.

The learning agents are deployed in an environment that simulates a virtualized node, which include models of VMs and some aspects of system memory (capacity, RAM misses, utilization). This evaluation methodology allows to see the effectiveness of the MDP formulation, without intervention of specific hardware.

We evaluate CAVMem in runtime scenarios consisting of different combinations of SPEC CPU 2006 benchmarks [44] and benchmarks from Cloudsuite [31]. All of these scenarios are summarized in Table 3.2. Every scenario that uses SPEC benchmarks runs two VMs with 6 GB of RAM, and each VM starts with 1 GB of RAM. We decided to use SPEC and Cloudsuite benchmarks in this case in order to expose CAVMem to a wider set of workloads that present an increased variability in their memory demand profiles. This allows us to evaluate the effectivness of CAVMem in more different scenarios.

In every scenario, three metrics are evaluated: 1) the average miss rate (pages/second), 2) the average miss rate deviation, and 3) the overhead (time spent in seconds) associated to the training of each agent. Every metric is measured over the last 100 episodes of each experiment. It is desirable that the average miss rate is minimized by the agent, since this would avoid disk accesses on a real system. The simulated environment does not model execution time, but it does model miss rates on RAM, which on a real system translates to swap operations. In current systems, as the swap operations increase, it implies a performance degradation on the use of memory resources [42], which have an adverse impact on the execution time on the applications running. Since the agents are constrained to reduce the memory misses/swap operations, it would imply an improvement on the execution time of the applications [120]. The miss rate deviation is to compare how well the agent learns the desired behavior according to the reward function. And the overhead allows us to estimate the cost to the deployment of each agent.

3.5 Results for Evaluation

3.5.1 Results for Scenario 1

Figure 3.3 shows the average miss rates for Scenario 1. CAVMem with *DDPG* has a miss rate 10% larger than *static*, while *DQL* and *QL* are higher by 14.7% and 13.9% respectively. Since the VMs are executing the same benchmark, an equal memory allocation is optimal resulting in *static* being the best in this case. Nevertheless, we see that CAVMem performs the best.

Figure 3.4 shows the average miss rate deviations for Scenario 1. CAVMem tracks the target better than all other approaches. CAVMem has a miss rate deviation 82.2% smaller

Label	Benchmark Name	Description
Scenario 1	VM1, VM2: *perlbench*	Both VMs run *perlbench* repeatedly.
Scenario 2	VM1, VM2: *gcc*	Both VMs run *gcc* repeatedly.
Scenario 3	VM1: *perlbench*; VM2: *gcc*	VM1 runs *perlbench* repeatedly, VM2 runs *gcc*.
Scenario 4	VM1: *gcc*; VM2: *bzip2*	VM1 runs *gcc* repeatedly, VM2 runs *bzip2*.
Scenario 5	VM1:*soplex*; VM2:*perlbench*	VM1 runs *soplex* repeatedly while VM2 runs *perlbench*.
Scenario 6	VM1:*web-search client*; VM2:*in-memory analytics*; VM3: *web-search server*	VM1 runs the Cloudsuite's *web-search client*, VM2 runs Cloudsuite's *in-memory analytics* and VM3 runs *web-search server*. All VMs run the benchmark trace repeatedly.
Scenario 7	VM1:*web-search client*; VM2:*graph analytics*; VM3: *web-search server*	VM1 runs the Cloudsuite's *web-search client*, VM2 runs Cloudsuite's *graph analytics* and VM3 runs *web-search server*. All VMs run the benchmark trace repeatedly.
Scenario 8	VM1:*data-caching client*; VM2:*graph analytics*; VM3: *data-caching server*	VM1 runs the Cloudsuite's *data-caching client*, VM2 runs Cloudsuite's *graph analytics* and VM3 runs *data-caching server*. All VMs run the benchmark trace repeatedly.
Scenario 9	VM1:*graph analytics*; VM2:*in-memory analytics*	VM1 runs the Cloudsuite's *graph analytics* and VM2 runs *in-memory analytics*. Both VMs run the benchmark trace repeatedly.
Scenario 10	VM1:*in-memory analytics*; VM2:*in-memory analytics*	VM1 and VM2 run *in-memory analytics*. Both VMs run the benchmark trace repeatedly.
Scenario 11	VM1:*graph analytics*; VM2:*graph analytics*	VM1 and VM2 run *graph analytics*. Both VMs run the benchmark trace repeatedly.

Table 3.2: List of scenarios used for evaluation

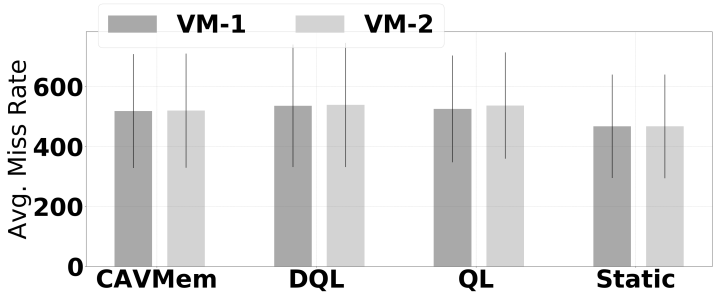


Figure 3.3: Average Miss Rate for Scenario 1. Less is better.

than *static*, and 66.1% and 75.4% smaller than *DQL* and *QL*, respectively.

Figure 3.5 shows the overheads for Scenario 1. *DQL* presents an overhead 10.64 and 15.0 times larger than CAVMem and *QL* respectively, a major consideration when deploying *DQL*. *QL* presents the smallest overhead, while CAVMem's is 37% larger than *QL*'s.

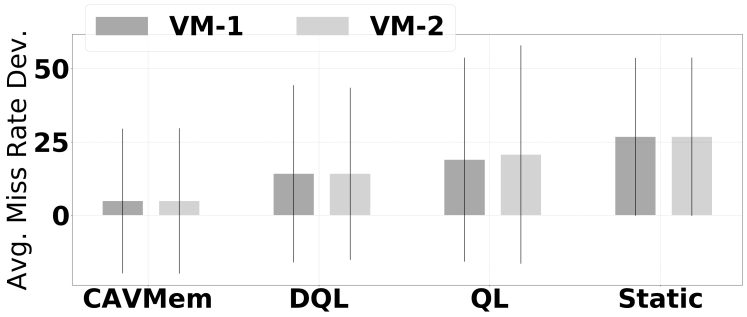


Figure 3.4: Average Miss Rate Deviation for Scenario 1. Less is better.

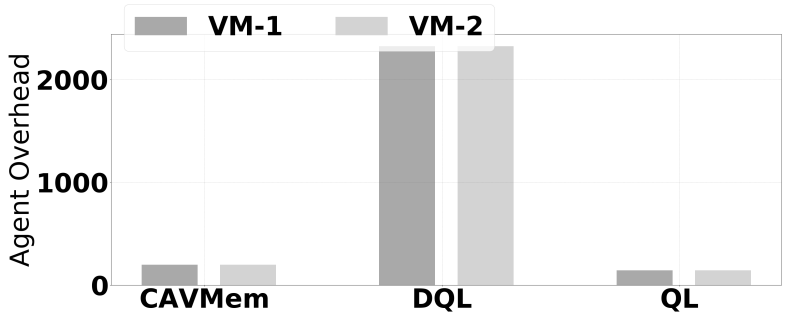


Figure 3.5: Overhead (in seconds) for each agent in Scenario 1. Less is better.

3.5.2 Results for Scenario 2

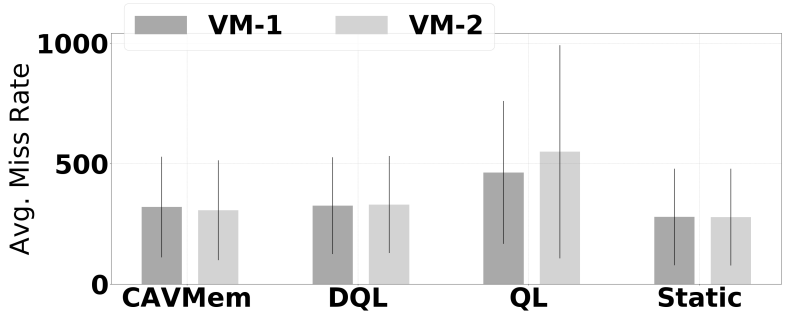


Figure 3.6: Average Miss Rate for Scenario 2

Figure 3.6 shows the average miss rate for each agent in Scenario 2. The results here are similar to Scenario 1. CAVMem has a miss rate 12.55% than *static*, while *DQL* and *QL* are higher by 17.8% and 82.04% respectively. CAVMem performs the best among the learning agents, but still *static* outperforms them all.

Figure 3.7 shows the miss rate deviation for Scenario 2. *QL* performs the worst, while CAVMem performs slightly better than all of them but not by muc, except for *QL*, that performs badly.

Figure 3.8 shows the overhead of the agents for Scenario 2. The results are very similar

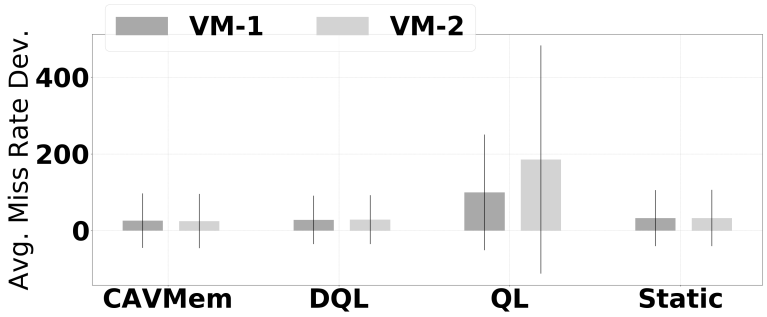


Figure 3.7: Average Miss Rate Deviation for Scenario 2. Less is better

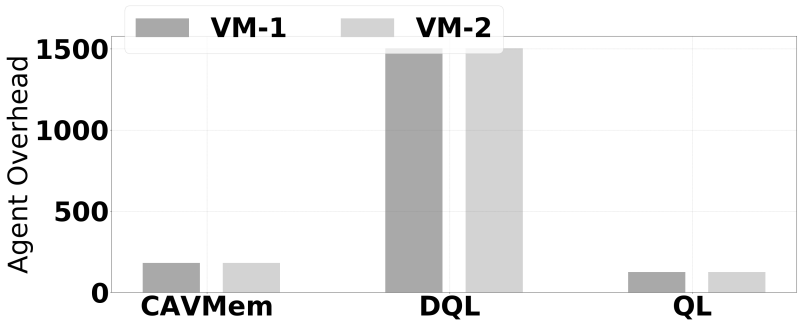


Figure 3.8: CAVMem: Average Overhead (in seconds) for each agent in Scenario 2. Less is better

to those of Scenario 1.

3.5.3 Results for Scenario 3

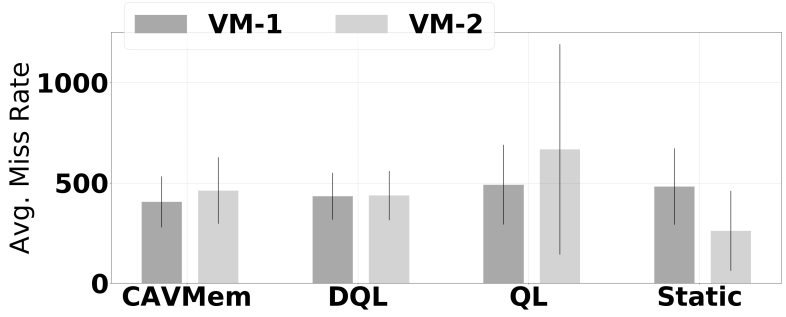


Figure 3.9: Average Miss Rate for Scenario 3. Less and equal is better.

Figure 3.9 shows the average miss rates for Scenario 3. *QL* and *static* fail to balance the miss rates, which is an undesirable result, since they both benefit one VM while harming the other. CAVMem and *DQL* maintain the miss rate balanced to a better degree by increasing the miss rate of both VMs. CAVMem minimizes the miss rate 6.41% below *DQL* for VM1 and 5.81% above *DQL* for VM2. Seemingly, CAVMem and *DQL* are competitive in performance.

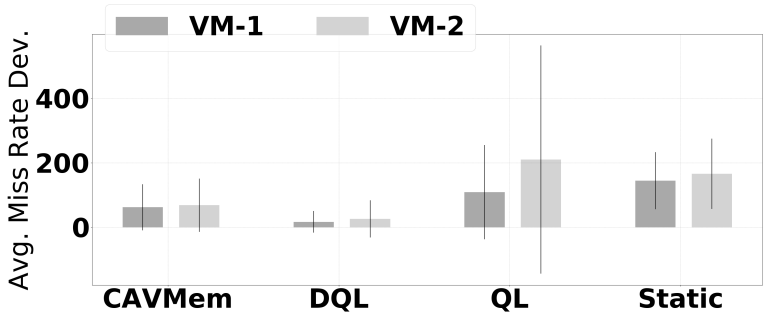


Figure 3.10: Average Miss Rate Deviation for Scenario 3. Less is better.

Figure 3.10 shows the miss rate deviations for Scenario 3. Here, *static* is further away from the target, while CAVMem deviates by 67.6% more than *DQL*, even though they both minimize the miss rate to similar values.

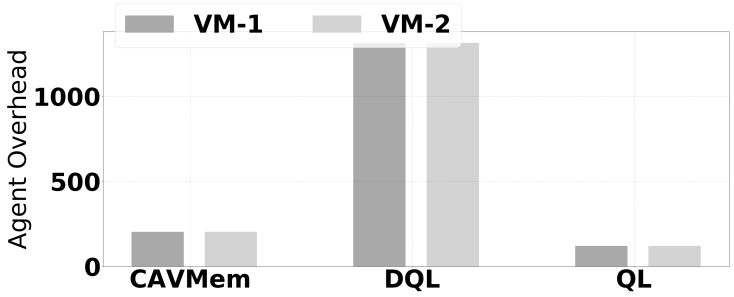


Figure 3.11: Overhead (in seconds) for each agent in Scenario 3. Less is better.

Figure 3.11 shows the overhead of the agents for Scenario 4. *DQL* maintains an overhead 5.44 and 9.96 times larger than CAVMem and *QL*, respectively. CAVMem's overhead is 70% larger than *QL*'s but performs similar to *DQL*. Thus far, CAVMem provides the best performance-overhead trade-off.

3.5.4 Results for Scenario 4

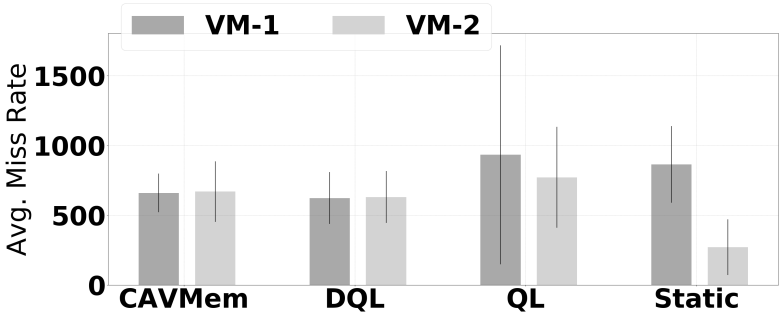


Figure 3.12: Average Miss Rate for Scenario 4. Less and equal is better.

Figure 3.12 shows the average miss rates for Scenario 4. *DQL* and *static DQL* present imbalanced miss rates for the VMs, while CAVMem and *DQL* yield similar values, with CAVMem being 5.96% larger than *DQL*.

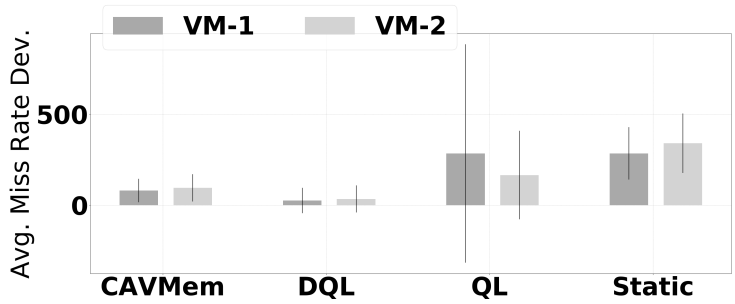


Figure 3.13: Average Miss Rate Deviation for Scenario 4. Less is better.

Figure 3.13 shows the miss rate deviations for Scenario 4. *DQL* tracks the miss rate targets better being 65.92% more accurate than CAVMem, while *QL* and *static* fail to do so.

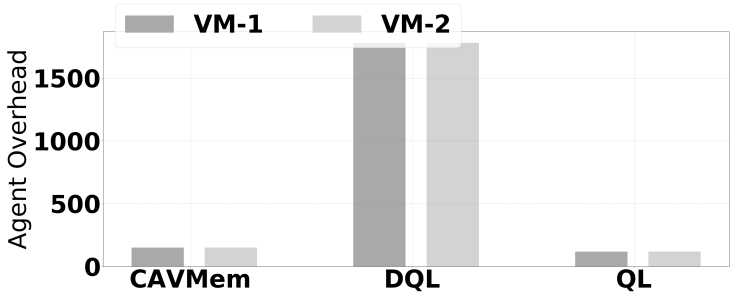


Figure 3.14: Overhead (in seconds) for each agent in Scenario 4. Less is better.

Figure 3.14 shows the overhead of the agents for Scenario 4. The experiments show that *DQL*'s overhead is 10.9 and 14.2 times larger than CAVMem's and *QL*'s, respectively. CAVMem's overhead is larger than *QL*'s by 28%, still maintaining the cost-effectiveness of CAVMem.

3.5.5 Results for Scenario 5

Figure 3.15 shows the average miss rate achieved for each agent in Scenario 5. This results show a similar behavior to Scenarios 3 and 4: *static* maintains unbalanced miss rates among the VMs, CAVMem and *DQL* have similar performance with *DQL* being better by a small percentage (CAVMem achieves 3.62% higher miss rate).

Figure 3.16 shows the miss rate deviation for Scenario 5. The behavior is very similar to Scenarios 3 and 4, with *DQL* still managing to track the miss rate target better than CAVMem. The mean deviation of CAVMem is 62.2% larger when compared to *DQL*, while *QL* and *static* present high inaccuracies with respect to the desired target.

Figure 3.17 shows the overhead of the agents for Scenario 5. These experiments show consistently the enormous overhead incurred by *DQL*, and the cost-effectiveness of CAVMem.

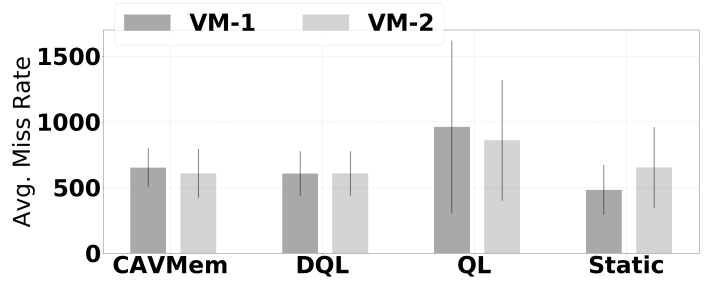


Figure 3.15: Average Miss Rate for Scenario 5. Less is better

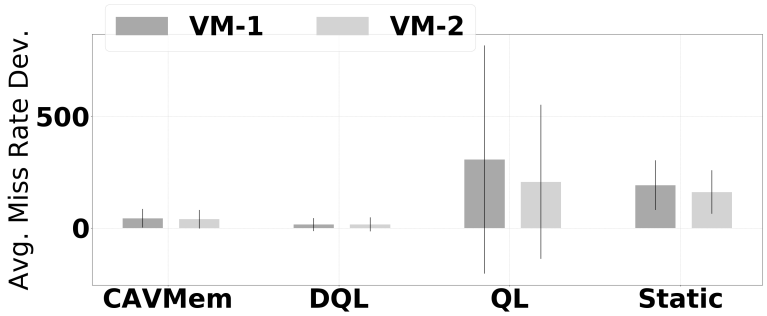


Figure 3.16: Average Miss Rate Deviation for Scenario 5. Less is better

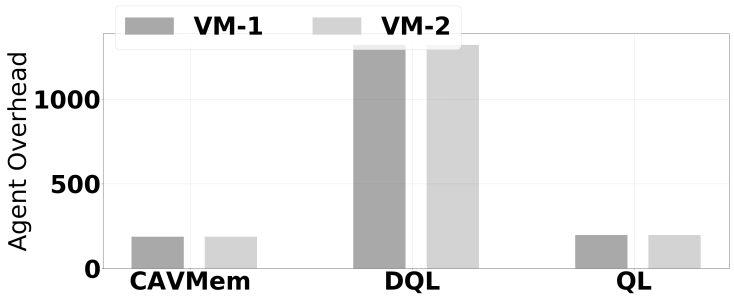


Figure 3.17: Overhead (in seconds) for each agent in Scenario 5. Less is better.

3.5.6 Results for Scenario 6

Figure 3.18 shows the average miss rate achieved for each agent in Scenario 6. This results show a similar behavior to Scenarios 4 and 5: *static* maintains unbalanced miss rates among the VMs, CAVMem and *DQL* have similar performance. But in this case, it is harder to establish which one of the two performs better, because VM1 and VM3 with CAVMem have a 3.2% and 2.5% smaller average miss rate when compared to *DQL*, respectively, while VM2 has a larger average miss rate by 4.8% when compared to *DQL*.

Figure 3.19 shows the miss rate deviation for Scenario 6. The behavior is very similar to Scenarios 4 and 5, with *DQL* still managing to track the miss rate target better than CAVMem. The mean deviation of CAVMem is 57.9% larger when compared to *DQL* across all three VMs, while *QL* and *static* present high inaccuracies with respect to the desired

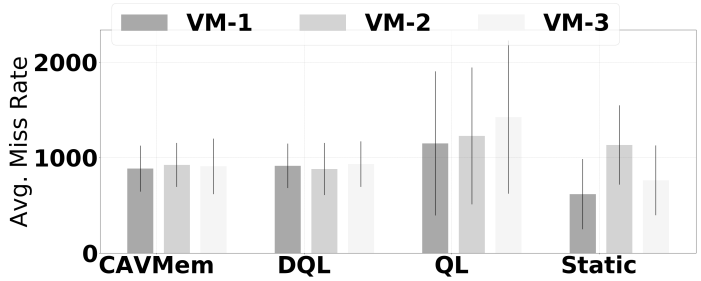


Figure 3.18: Average Miss Rate for Scenario 6. Less is better

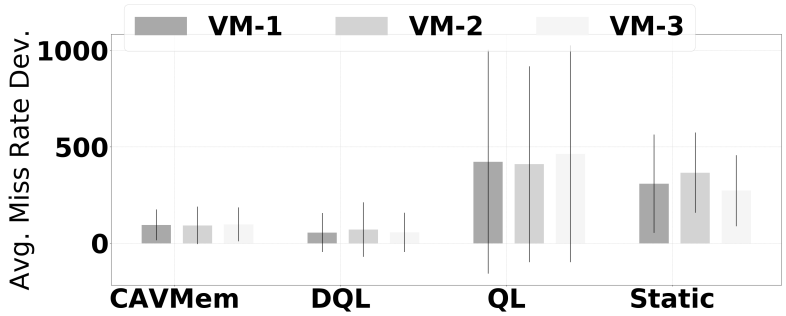


Figure 3.19: Average Miss Rate Deviation for Scenario 6. Less is better

target.

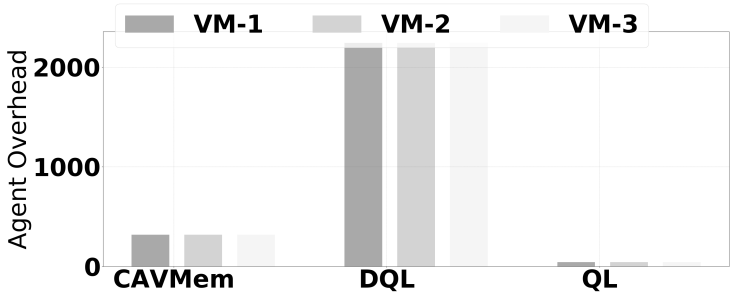


Figure 3.20: Overhead (in seconds) for each agent in Scenario 6. Less is better.

Figure 3.20 shows the overhead of the agents for Scenario 6. The overhead associated to *DQL* is 6.1 times larger than the one associated to CAVMem.

3.5.7 Results for Scenario 7

Figure 3.21 shows the average miss rate achieved for each agent in Scenario 7. This results show again the inability of *QL* and *static* to minimize the miss rates in a balanced way. These results also show the competitiveness between *DQL* and CAVMem: for VM1 and VM2 *DQL* is able to minimize the miss rate by 2.3% and 8.1% lower than CAVMem, respectively, while for VM3 CAVMem minimizes the miss rate by 1.3% less than *DQL*.

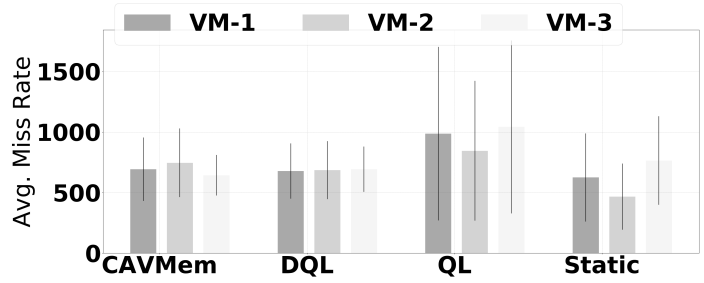


Figure 3.21: Average Miss Rate for Scenario 7. Less is better

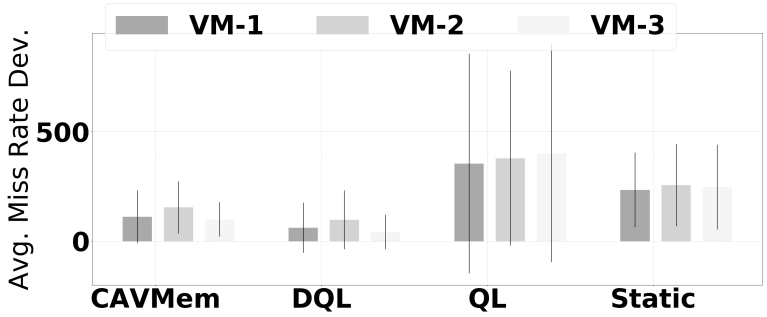


Figure 3.22: Average Miss Rate Deviation for Scenario 7. Less is better

Figure 3.22 shows the miss rate deviation for Scenario 7. *QL* and *static* deviate very significantly from the desired targets during the execution, but *DQL* and CAVMem are able to approximate it much better. *DQL* gets the better part, by keeping the miss rate deviation 46.2% smaller than CAVMem across the three VMs.

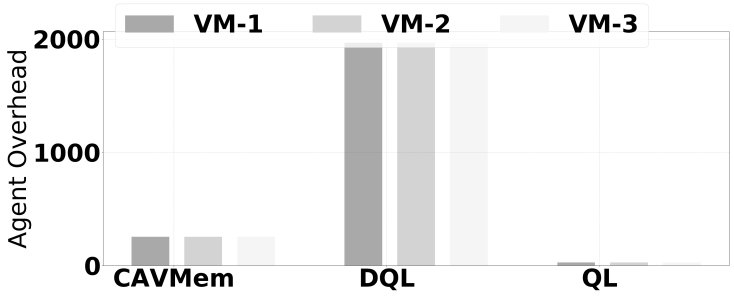


Figure 3.23: Overhead (in seconds) for each agent in Scenario 7. Less is better.

Figure 3.23 shows the overhead of the agents for Scenario 7, which shows that *DQL* has an overhead that is 6.8% times larger than the one associated to CAVMem, consistently high-lighting the cost-effectiveness of CAVMem.

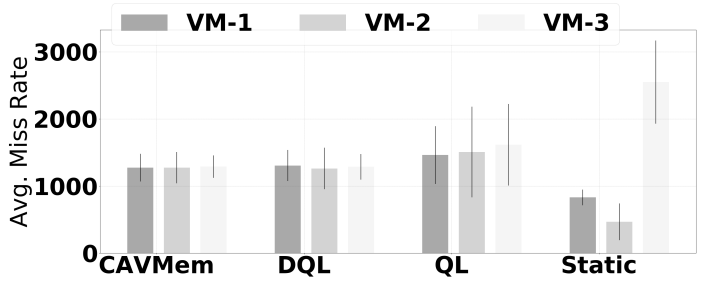


Figure 3.24: Average Miss Rate for Scenario 8. Less is better

3.5.8 Results for Scenario 8

Figure 3.24 shows the average miss rate achieved for each agent in Scenario 8. In this case, CAVMem performs slightly better than *DQL* by a 0.45% in average across the three VMs. Moreover, *QL* is in fact more competitive in this case, and it is able to balance the miss rates of the three VMs, but it is unable to minimize it in a similar way to CAVMem or *DQL*. Quantitatively, *QL* has a miss rate above that of CAVMem by 14,8%, 18,3% and 25,3% for VM1, VM2 and VM3, respectively.

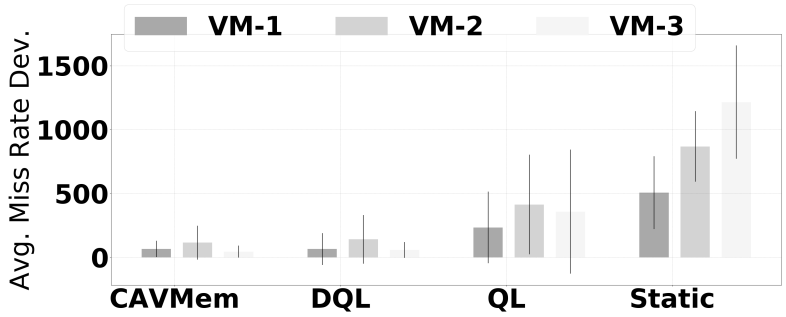


Figure 3.25: Average Miss Rate Deviation for Scenario 8. Less is better

Figure 3.25 shows the miss rate deviation for Scenario 8. In this case, CAVMem is able to track the miss rate targets better than *DQL*, by an average of 17.1% across all VMs: 23,2% and 28.7% smaller for CAVMem in VM2 and VM3, but in VM1 the miss rate deviation is 0.6% smaller for *DQL*. The deviation of *QL* is significantly higher, by 77.2% in average, when compared to CAVMem.

Figure 3.26 shows the overhead of the agents for Scenario 8. In this case, the results show that *DQL*'s overhead is 6,9 and 50,2 times larger than CAVMem's and *QL*'s, respectively, while CAVMem's overhead is 5.5 times larger than *QL*'s. Even though *QL* is able to balance out the miss rates in this case with some limitations, still is not able to obtain the same performance than CAVMem or *DQL*. Even though CAVMem has a higher overhead than *QL* in this case, overall CAVMem remains potentially the best cost-effective solution.

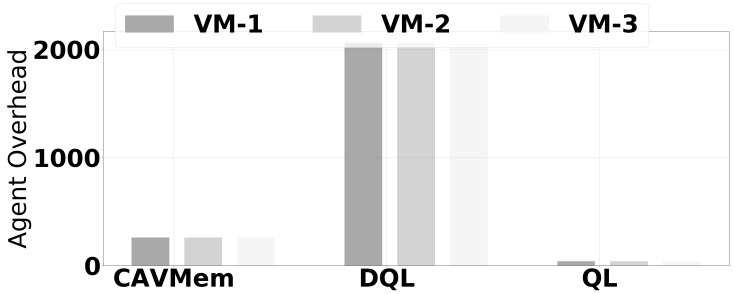


Figure 3.26: Overhead (in seconds) for each agent in Scenario 8. Less is better.

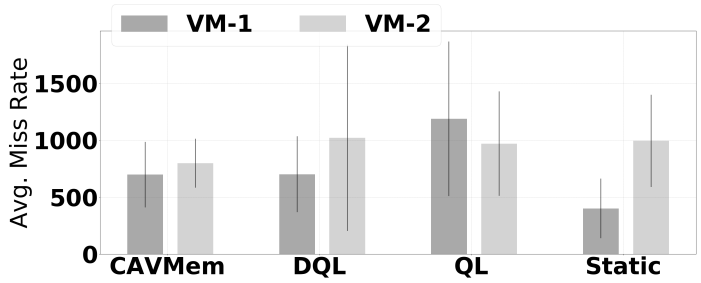


Figure 3.27: Average Miss Rate for Scenario 9. Less is better

3.5.9 Results for Scenario 9

Figure 3.27 shows the average miss rate achieved for each agent in Scenario 9. Interestingly, in this case *DQL* is unable to balance the performance of the VMs and maintains a miss rate difference of 45.6% between VM2 and VM1, which is very undesirable. On the other hand, CAVMem is able to keep the performance difference among the VMs within a 14%, both miss rate minimization and balancing.

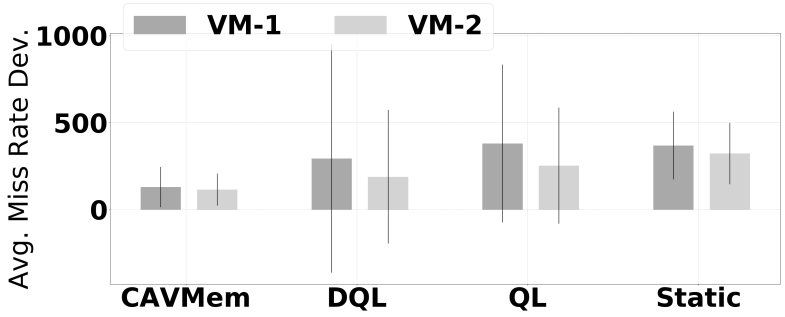


Figure 3.28: Average Miss Rate Deviation for Scenario 9. Less is better

Figure 3.28 shows the miss rate deviation for Scenario 9. Considering already that *DQL* is unable to balance the miss rates (performance), figure 3.28 shows that it is not as effective as CAVMem to track the miss rate targets. In this case, *DQL* has a miss rate target deviation of 55.8% and 38.9% for VM1 and VM2, respectively. So far, CAVMem has out-performed

significantly the other agents and *static* for Scenario 9.

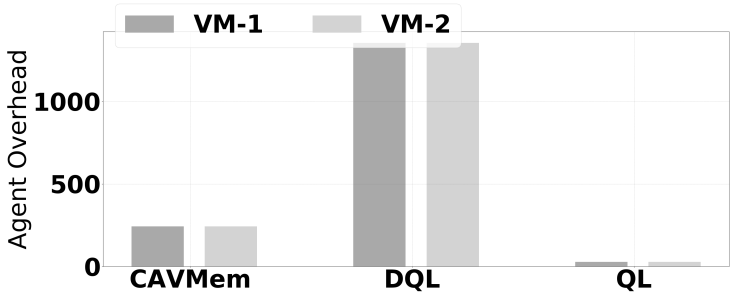


Figure 3.29: Overhead (in seconds) for each agent in Scenario 9. Less is better.

Figure 3.29 shows the overhead of the agents for Scenario 9. This figure shows that. in addition to the under-performance presented by *DQL* for miss rate balancing and minimization, and miss rate target tracking, *DQL* also presents an overhead that is 4.6 times larger than the overhead associated to CAVMem.

3.5.10 Results for Scenario 10

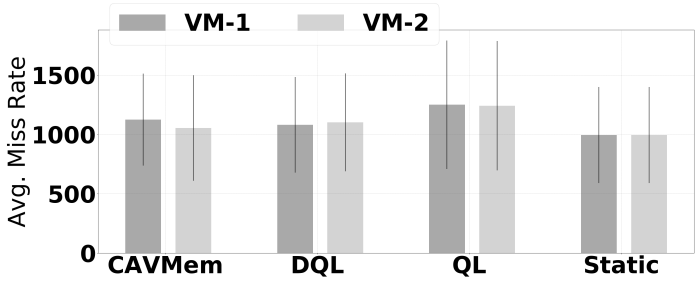


Figure 3.30: Average Miss Rate for Scenario 10. Less is better

Figure 3.30 shows the average miss rate achieved for each agent in Scenario 10. The best performing allocation strategy is *static*. All of the other agents, namely CAVMem, *DQL* and *QL*, generate average miss rates larger than *static* by 9.5%, 9.7% and 25.3%, respectively, across all of the VMs. In this case, CAVMem performs slightly better among the learning agents.

However, the VM1 and VM2 have unbalanced miss rates when running with CAVMem even though their average is slightly better (lower). With CAVMem, VM1 and VM2 has miss rates larger than *static* by 13.1% and 5.9%, whereas with *DQL* they have 8.7% and 10.8%. In this case, *DQL* is able to keep the performance balanced among the VMs better than CAVMem in this scenario.

Figure 3.31 shows the miss rate deviation for Scenario 10. In this case *DQL* is able to track the miss rate targets better than all the allocation strategies. As expected, *static* is not designed to track miss rate targets, but it still able to minimize the miss rates better than *DQL* and CAVMem for this scenario. However, *DQL* is able to track the miss rate targets by 14.1% in average across all the VMs when compared to *static*. On the other hand, the

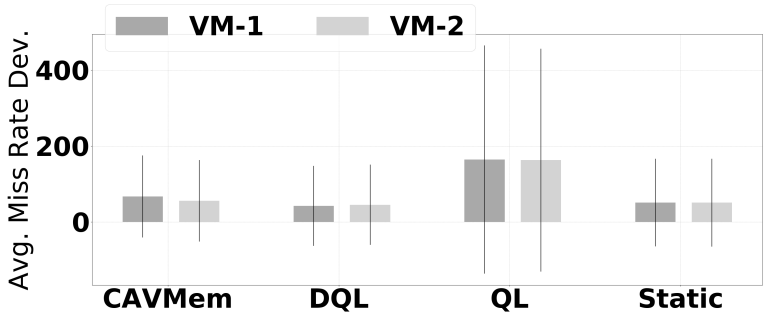


Figure 3.31: Average Miss Rate Deviation for Scenario 10. Less is better

deviation of CAVMem is larger than *static* by 21.0% in average across all the VMs, showing the effectivness of *DQL* in this case. As can be seen in figure 3.31, *QL* performs the worst out of all the learning agents.

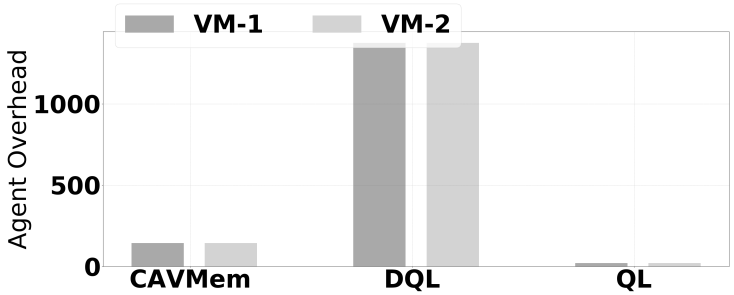


Figure 3.32: Overhead (in seconds) for each agent in Scenario 10. Less is better.

Figure 3.32 shows the overhead of the agents for Scenario 10. As it has been consistently shown so far, the performance overhead associated to *DQL* is extremely large when compared to CAVMem and the other approaches, an important factor that has to be considered when deploying *DQL*. In this case, the overhead associated to *DQL* is 8.6 and 63.9 times larger than the one associated to CAVMem and *QL*, respectively. Any deployment of *DQL* has to consider the trade-off between performance gains and its significant overhead.

3.5.11 Results for Scenario 11

Figure 3.33 shows the average miss rate achieved for each agent in Scenario 11. Similarly as with Scenario 10, the best performing allocation strategy is *static* again. All of the other agents, namely CAVMem, *DQL* and *QL*, generate average miss rates larger than *static* by 11.8%, 12.0% and 39.9%, respectively, across all of the VMs. In this case, CAVMem performs slightly better among the learning agents.

However, the VMs slightly more unbalanced miss rates when CAVMem compared to *DQL*. For example, VM1 with CAVMem has a miss rate larger than the one corresponding to VM1 when running *static* by 6,9%, while VM2's miss rate is 16.7% larger. In the case of *DQL*, the miss rates for VM1 and VM2 are 12.5% and 11.5%, respectively. Notice that in average CAVMem reduces the miss rate better across all the VMs, but *DQL* keeps them

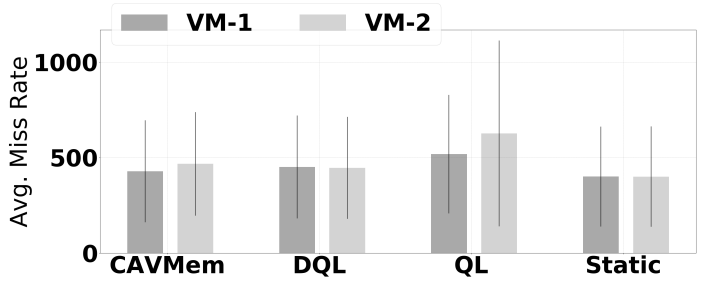


Figure 3.33: Average Miss Rate for Scenario 11. Less is better

better balanced.

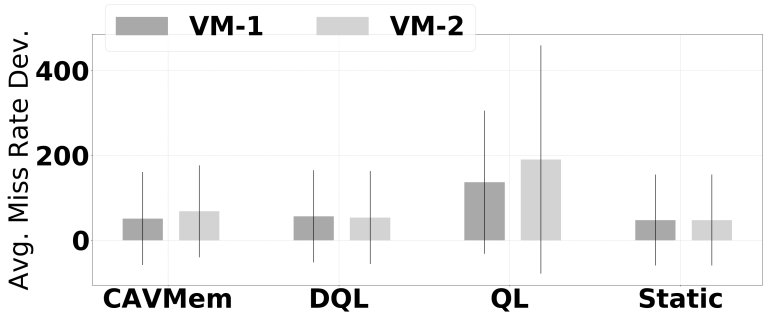


Figure 3.34: Average Miss Rate Deviation for Scenario 11. Less is better

Figure 3.34 shows the miss rate deviation for Scenario 11. In this case, the learning agents are not able to track the miss rate targets any better than *static*, despite the fact that *static* is not really attempting to do so. In this scenario, *QL* have a very significant miss rate target deviation of 188.4% and 299.8% with respect to *static* for VM1 and VM2, respectively.

DQL and CAVMem perform better than *QL* in this case. VM1 and VM2 for *DQL* have a larger miss rate target deviation of 18.8% and 12.9%, respectively, when compared to *static*. On the other hand, VM1 and VM2 for CAVMem have a larger miss rate target deviation of 7.9% and 43.7%, respectively, when compared to *static*. Notice that *DQL* is able to keep a better balancing on the miss rate deviation for the VMs when compared to CAVMem, but still performs slightly worst than *static*.

Figure 3.35 shows the overhead of the agents for Scenario 11. Yet again, the overhead for *DQL* is massive, being at 4.9 times the overhead associated to CAVMem and 67.5 when compared to *QL*.

3.5.12 Discussion

We can summarize the findings as follows: 1) CAVMem with *DDPG* and *DQL* are able to minimize and balance the miss rates in a comparable way, better than *static* in most cases, while *QL* consistently fails to do so, 2) *DQL* tracks the performance target better than CAVMem with *DDPG*, while *static* and *QL* fail in most instances, 3) *DQL* has extremely

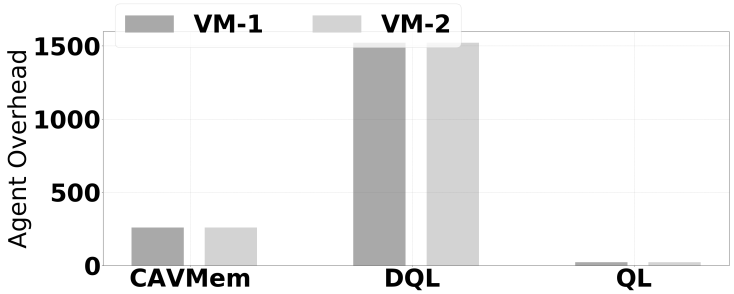


Figure 3.35: Overhead (in seconds) for each agent in Scenario 10. Less is better.

high learning overheads when compared to CAVMem with *DDPG* and *DQL*, highlighting a performance-cost trade-off.

3.6 Related Work

Many solutions have been proposed to solve the resource allocation problem for cloud infrastructures [105, 101, 72, 27, 13]. However, the memory management problem has different constraints than a general resource allocation problem, due to the way memory is shared among applications/VMs. Whereas CPU can be time-multiplexed and scheduled, memory cannot be scheduled in a fine-grained way [76]. Instead it has to be allocated over a large period of time ensuring the integration of applications' data. Resource allocation schemes for the cloud need a different approach for memory allocation within each individual node.

Some common approaches for the memory allocation problem in virtualized computing servers include working set size (WSS) estimation [15, 47, 125, 84]. When estimating the WSS, the objective is to allocate enough memory to the VMs in order for their working set to fit in.

The main problem with WSS estimation becomes evident when applications do not use the whole WSS during different phases of their execution, therefore overallocating memory (that is, misusing memory) during those times when part of the memory is not used. This problem is exacerbated when there are multiple applications running inside the VMs and temporal variation on the applications behavior start affecting the measurements of the WSS.

The authors in [121] took a different approach by implementing a tax system that would increase the cost of a VM with idle memory, in order for this VM to release the memory is not using. This approach works by calculating the tax based on the measure of idle memory, which is a measure of the memory unallocated in the VM. The problem here is that once the application allocates all the memory it needs (its WSS), the memory will not be idle as long as it remains allocated even if it is unused. Thus, it will still overallocate memory unnecessarily at some phases of execution. One advantage of this approach is that it is very adaptable, since it does not try to predict the WSS before hand.

Approaches based on control engineering techniques [45, 83] have also been used to solve the memory management problem. The main issue with these approaches is their need to develop models of the quantitative relationship between an application's performance and its memory allocation. This very difficult to do accurately, given all the factors that have an

impact on this relationship. The use of RL approaches become thus justified when grasping the complexity of the factors involved in this relationship.

Many RL-based solutions have been proposed to solve the resource allocation problem in cloud infrastructures [13, 87] but so far, there have not been too many efforts that use RL to exclusive solve the memory management problem, and much less continous-action RL solutions. But there are some notable RL-based solutions for the resource allocation problem that targets CPU and memory (and other resources) within a virtualized node. In [87], the authors implemented a model-based RL algorithm for VM resource configuration, which included CPU time, virtual CPU and memory. This algorithm is a DQL algorithm using a neural network to approximate the value function.

CAVMem differs from these RL-based solutions in three ways: 1) CAVMem uses continuous action RL exclusively for memory management, 2) CAVMem avoids discretization, and 3) it is decentralized.

3.7 Conclusions for this Chapter and Future Work

This paper proposes CAVMem as a proof-of-concept of a distributed MDP formulation for the memory allocation problem in virtualized nodes. Moroever, CAVMem also offers a continuous action RL agent to solve the allocation problem, avoiding discretization and exploiting de-centralization. Our results show that the DDPG agent of CAVMem performs similar to the well-known DQL but with much less learning overhead, making it a very cost effective solution.

For future work, CAVMem should be deployed in a real computing node, and an exhaustive search of the parameter space of the learning agents is also necessary. Likewise, it is necessary to test CAVMem with more combinations of benchmarks and scenarios.

Chapter 4

Deploying a Reinforcement Learning in a Virtualized Node for Dynamic Memory Management

The previous chapter gave us a strong footing to use a continuous action RL approach to solve the memory allocation problem in a distributed way, even though it was tested in a simulation environment that only modeled certain memory behavior of a virtualized node.

The most important conclusion from CAVMem is that the formulation of the memory allocation problem as a distributed MDP is actually capable of learning and achieving the desired performance for all the learning agents we tried. In addition, CAVMem also demonstrated that using the continuous action RL that we used (DDPG) is able to perform well with a significant less amount of computational overhead.

Taking into account all the conclusions from CAVMem, the next step is to deploy CAVMem in a real virtualized node, and do the necessary enhancements and modifications to adapt it to the real environment.

This chapter introduces CARLEMM, which is a continuous action RL approach based largely on CAVMem, but with significant enhancements for it to be deployed in a real virtualized node. So far, we have been only able to get it to work for synthetic workloads and for very specific use cases, so it is difficult to assess at this point its effectiveness to solve the RAM allocation problem in a more definite way. At the end of the chapter, we show the results we have obtained so far and demonstrate its usability and limitations for the cases analyzed.

4.1 Introduction to CARLEMM

A cloud infrastructure will have virtualized nodes running multiple virtual machines (VMs). The physical memory of each node must be dynamically reallocated among the VMs it is running in order to optimize throughput and maintain fairness. This is a difficult problem because VMs are continually being started, migrated or stopped, and because the applications within them are also changing over time.

The most straightforward way to manage memory capacity is for the users/managers of the infrastructure to specify in advance the amount of memory to allocate to each VM. However, users/managers seldom know exactly how much memory to request, which results either in over-provisioning of memory, which wastes resources, or under-provisioning it, resulting in poor application performance. Moreover, such a static request-based policy is unable to adapt to changes in memory demand over time.

There are previous research efforts that have provided solutions for this. For instance, the proposal of Chiang et al. estimates the working set size of the applications in each VM [15]. Much of the prior work along these lines is limited by the complexity of the relationship between the allocation of physical memory to a VM and is performance.

Major advances have recently been made in RL, which demonstrate RL to be a powerful method to solve many types of complex problems without having to build a precise model of the problem to solve. In Chapter 3, we explained the main two limitations with these current state-of-the-art approaches, namely discretization and centralization, and why the continuous action RL approach formulated in this chapter and the previous one provide a good solution to overcome those limitations.

This chapter presents CARLEMM (Continuous Action Reinforcement LEarning for Memory Management), which addresses once again both of these limitations by formulating the memory allocation problem as a distributed continuous action RL problem, following up the work presented in Chapter 3, but this time deploying the learning agents in a real virtualized node. Similar to Chapter 3, the learning agent at the core of CARLEMM is a DDPG agent [60]. To the best of our knowledge, this is the first time that a continuous action algorithm has been deployed in a real virtualized node for dynamic memory management. In summary, this chapter makes the following contributions:

1. We deploy a continuous action off-policy model-free RL algorithm in a real virtualized node to solve the memory allocation problem.

2. We develop a flexible software stack to implement CARLEMM and evaluate its performance and fairness using synthetic benchmarks.

4.2 Understanding the Context for CARLEMM

As it has been explained in previous chapters, the HV running in every virtualized node of the cloud infrastructure distributes the memory capacity among the VMs it executes, but often tends to over-commit memory. Potentially, this can lead to a misuse of memory by a single VM and thereby, leading to memory pressure for the other VMs [67]. For this reason, memory management becomes an important problem, making research on dynamic memory management still a relevant field. The most commonly deployed mechanisms to manage

and reallocate memory are memory ballooning [67], memory hotplug [97] and Transcendent memory [69], in which the latter allows the aggregation of capacity across multiple computing nodes [36, 34]. In this chapter, we focus on memory reallocation via memory ballooning.

As explained in Section 1.3, memory ballooning [67] is a technology to enable the hypervisor to inflate or deflate the memory on a per-page basis across VMs using a balloon driver. However, the balloon driver makes the reallocation considering only local (per-VM) information, and it is unable by itself to make optimal decisions on when and for how much to reallocate memory in a system with multiple VMs with different memory requirements.

As we explained in Section 3.2.1 the problem really becomes how to drive the inflation/deflation mechanism i.e. allocation/deallocation, of ballooning to allocate memory to the VMs in order to prevent misuse of memory resources (over- or under-provisioning of memory). As we explained, this can be done by estimating the working set size of the applications running inside a VM, by overallocating memory, by modifying the allocations manually of the balloon driver or using automated high-level policies to drive the balloon driver.

Many automated solutions have been proposed, but there are very little (close to none) efforts that have focused on the memory allocation problem using RL. As explained in Section 3.2.1, the RL approaches becomes particularly attractive due to complexity associated to develop analytical models of performance to drive the memory ballooning mechanism in a node with multiple VMs.

In Section 3.2.2, the theoretical work and background necessary to understand the RL concepts (MDPs, continuous action, DDPG) employed for the development of CARLEMM were already explained. In general, CARLEMM builds on the RL concepts used to develop CAVMem, but with some important modifications on the definition of the state space, the reward function and the control flow of the memory allocation mechanism. All these modifications are introduced in order to make it work in a real virtualized node.

4.3 CARLEMM: Continuous Action Reinforcement Learning for Memory Management

In this section, we introduce CARLEMM, a continuous action RL algorithm for memory management in virtualized computing servers. We formulated the memory allocation problem as an MDP that allows for one learning agent to be bounded to a VM, where the VM instances can be spawned at arbitrary time frames. Specifically, we instantiate a neural network per VM and allow it to have both a local view (itself) and a view of the node (based on an approximate understanding of the behavior of other VMs in the system).

As with CAVMem, each learning agent bound to a VM monitors its resource utilization, the state of the VM and bids for memory independently of the rest of the VMs, resulting in a decentralized solution for memory management. The advantages of having such a distributed and decentralized formulation where analyzed in Section 3.3.1.

In the following subsections, we explain how CARLEMM builds on CAVMem, how the MDP is formulated for this case and how the RL mechanism works in a real virtualized node.

4.3.1 Formulating the Memory Management Problem in Virtualized Node as an MDP

Every problem formulated as an MDP requires the definition of the state space together with their respective variables and domains. The state space is defined in such way that part of the variables define the state of the VM (local) and another part define the state of the node (node, global). The partitioning of the state space in local and global variables follows the same scheme used for CAVMem, but for CARLEMM we have used a set of different variables to define the state space.

State information from computing node and VMs

	Variable Name	**Range**	**Description**
perVM	$usrtime^{VM_j}$	[0, 1]	The percentage of time spent by the processor of VM_j executing user space processes in relation to the sampling interval.
perVM	$M_t^{VM_j}$	[0, 1]	Fraction of total RAM allocated to VM_j.
perVM	$swap\_used^{VM_j}$	[0, 1]	The amount of swap space used by VM_j.
Node	$avgUsrtime$	[0, 1]	The average percentage of time spent by all VMs in the node executing user user space processes.
Node	$usrtimeTgt$	[0, 1]	Target percentage of time spent executing user space processes for all VMs given by Equation 4.2.
Node	$totalMemUse$	[0, 1]	Fraction of the node's physical RAM that is allocated to all VMs, equal to $\sum_j M_t^{VM_j}$

Table 4.1: State inputs to the actor and critic networks of the DDPG learning agents.

State space, S

Table 3.1 lists the six state variables for each learning agent, three of which belong to the VM and the other three are common to all VMs on the node. This state space definition allows the agent to have *some* information about the other active VMs through the information related to the node.

Notice that the variables used in this case are different than the ones used for CAVMem, even though the MDP formulation follows the same general principle. When comparing Table 4.1 to Table 3.1, it is clearly shown that some variables have changed in Table 4.1 for CARLEMM. For the current case, the variable $msr_t^{VM_j}$ was replaced by $usrtime^{VM_j}$, $avgMsr_t^{node}$ was replaced by $avgUsrtime$, $msrTgt_t^{node}$ was replaced with $usrtimeTgt$ and $swap\_used^{VM_j}$ was added as a new state variable.

The variables used to define the state of CAVMem were defined around the miss rate in RAM of a specific VM. Even though we still believe this is an attractive approach, it presented some limitations once we started developing CARLEMM. When attempting to measure the $msr_t^{VM_j}$ on RAM in real virtualized node, we find that this measurement is analogous to measuring the swap out rate and the swap in rate of the VM.

For example, in the case where a VM does not have enough memory to fit its data, it will start swapping out data from memory to the disk device, increasing the swap out rate of the VM. In this situation, the swap in rate increases if new data is brought in from disk, either because it is reading data from the filesystem or bringing in data that was previously swapped out. Therefore, the VM is unable to use its memory capacity to allocate the data that it will swap out, thus *missing* on a write access to memory, and then *missing* to find the data in memory when it tries to read it, resulting in a read from disk i.e. swap in. Thus, the $msr_t^{VM_j}$ variable used in CAVMem can be analogous to the swap out rate and/or swap in rates of the VM.

The main problem with using $msr_t^{VM_j}$ or a related swap out or swap in rate relates to their extremely noisy characteristic. Figure 4.1 shows the way the swap out rate looks for a VM in a system running a synthetic benchmark (explained in Section 4.7) that needs 1.2GB of memory to fit its data set in the VM. The benchmark reads and writes to this chunk of memory, while the VM is only given 1GB of memory.

Notice that the in the plot for the Swap Out Rate in Figure 4.1, the measurements go across the whole domain of possible values for a static allocation of memory, meaning that the standard deviation for the noise of this signal is extremely high. This behavior shown for the Swap Out Rate is consistent with different memory allocations and different benchmark/memory allocations as well. On the other hand, for the Percentage of User Space Time the standard deviation is way more controlled for a given allocation of memory, making it possible to correlate easier the values of a given Percentage of User Space time to memory allocations using a specific benchmark.

Using a signal with noisy characteristics makes it problematic for a RL agent to generate a state-action mapping i.e. an action policy, that is stable and meets the desired objectives specified in the reward function. In order to solve this problem during the development of CARLEMM, we evaluated the implementation of digital filters, starting with an straightforward implementation of a *moving average* that used a window of variable samples.

The moving average improved the quality of the signal, but still not enougt to cancel out the noise or even to generate a stable policy. Using the moving average approach presented some limitations as well, because its window size could not increase too much because the responsiveness of the state variable to the actions issued by the agent would be significantly delayed (as well as the reward), introducing a whole new set of problems. Opting to use more sophisticated filtering, like FIR filters, also entailed similar problems.

It became evident that using $msr_t^{VM_j}$ as a state variable in a virtualized node had significant limitations. For this reason, it was necessary to introduce $usrtime^{VM_j}$, which presented much less noise when measured and presented a better correlation with the memory allocation.

In order to support this view, we ran several experiments to understand the relationship between the memory allocation and $usrtime^{VM_j}$, which resulted in Figure 4.2. This figure was obtained by running a synthethic benchmark (explained in Section 4.7) with constant memory allocation and evaluate the Percentage of User Space Time. After evaluating this chart, there was a strong motivation to replace miss rate related metrics with those related with Percentage of User Space Time.

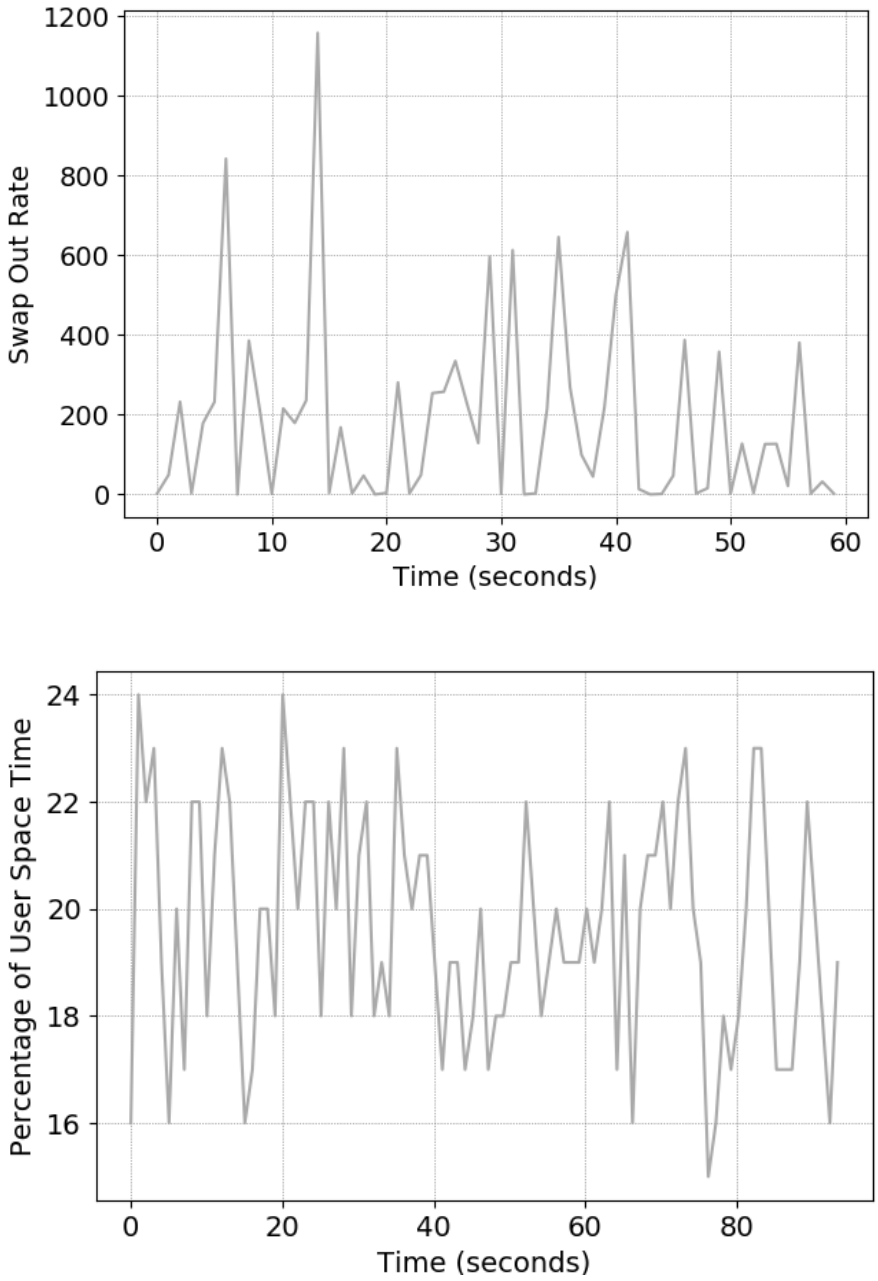


Figure 4.1: Visualizing the noise for the swap out rate measured from a VM with 1GB of RAM running an application that needs 1.2GB of memory (above), and the Percentage of User Space Time for the same VM and the same application (below).

Action space, A

The action $a_t^{VM_j}$ chosen for VM_j in timestep t is referred to as the VM's **memory bid**, which ranges from -1.0 to 1.0. This definition is identical to the one used for CAVMem. A positive bid is a request for more memory and a negative bid is to release memory. The

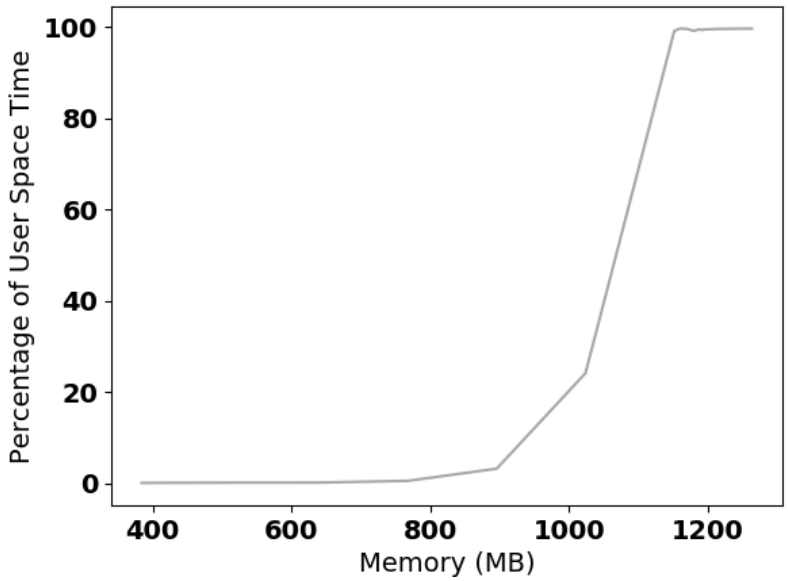


Figure 4.2: Plotting the relationship between memory allocations in MB and the Percentage of time spent by the processor a VM executing user space processes.

learning agent of a VM requests a total memory allocation equal to $(1 + a_t^{VM_j}) \times M_t^{VM_j}$. In many cases, it may not be possible to fully satisfy the request, for instance when there is insufficient memory for all requests or the memory allocation for the VM would become too small.

Action Space Exploration

To explore the continuous action domain in CARLEMM, the same approach than CAVMem was used: implement a Ornstein–Uhlenbeck process [11, 58] to create a noise signal that in turn generates a Brownian motion around the deterministic action generated by the DDPG agent. At every timestep, a noise signal is added to the action with a probability of ϵ. Similarly to CAVMem, initially there's a phase of aggressive exploration in which $\epsilon = 1$. This probability is annealed [102] until it reaches $\epsilon = 0.001$ over 18000 steps, whereas in CAVMem it was reduced over 38000 steps.

Reward function, R

To ensure a balance between the memory usage among the VMs and the memory bid, the reward function (shown in Equation 4.1) has two parts: (1) the absolute difference between the Percentage of User Space Time of the VM ($usrtime^{VM_j}$) and the target percentage ($usrtimeTgt$) and (2) a penalty for making bids of large magnitudes. The first part ensures that the bids generated by the agent make the Percentage of User Space Time $usrtime^{VM_j}$ converge to a consistent percentage value ($usrtimeTgt$) across all VMs, and the latter part ensures that the agent does not generate large bids in order to improve bidding behavior

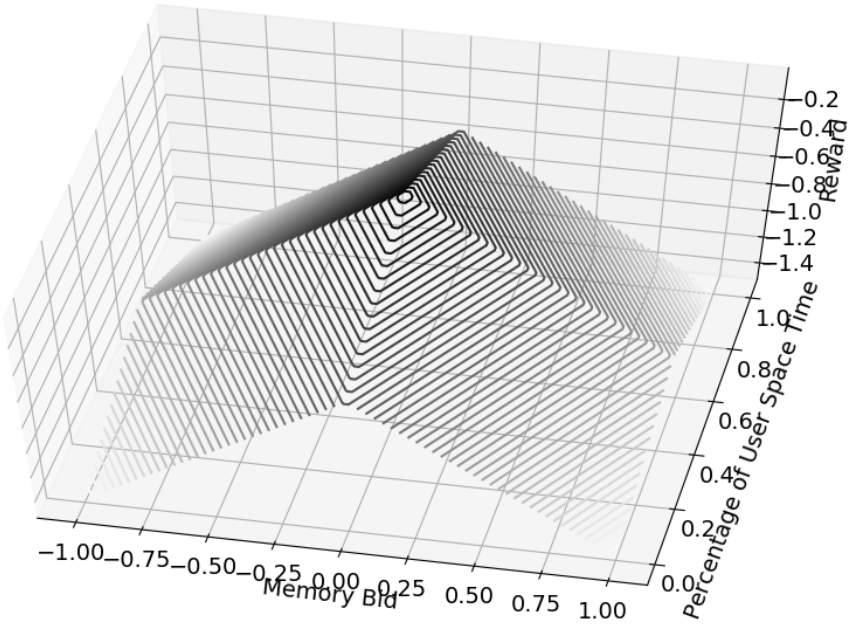


Figure 4.3: Plot of Equation 4.1 showing the the reward as a function of the user space time percentage and memory bid (both values normalized) of VM_j for $\beta = 1.0, k = 1.0$.

stability.

$$r_{t+1}^{VM_j} = -|usrtime_{t+1}^{VM_j} - usrtimeTgt_{t+1}| \times k - \beta \times |a_t| \tag{4.1}$$

Both parameters k and β where made equal to 1.0. The target percentage is defined as:

$$usrtimeTgt_{t+1} = \frac{avgUsrtime_t}{avgUsrtime_t + (1.0 - avgUsrtime_t) * totalMemUse_t} \tag{4.2}$$

In Equation 4.2, $totalMemUse$ determines the total amount of memory used by all VMs in the node. In turn, the average user space time percentage across all active VMs is defined as $avgUsrtime = \frac{1}{N} \sum_{j=1}^{N} usrtime^{VM_j}$.

Figure 4.3 shows the plot for Equation 4.1 for a target percentage value of $usrtimeTgt = 0.5$. Notice that the maximum value of Equation 4.1 is zero, and it occurs when both $usrtime^{VM_j} = 0.5$ and $a^{VM_j} = 0.0$.

4.4 Environment Constraints for Decentralized Control

Figure 4.4 shows a high-level view of the learning agents for the memory management problem, including the computing node, a module called Bid Analyzer (BA) and the VMs. The high-level view of the learning agents still remains valid as the one corresponding to CAVMem in Section 3.3.2.

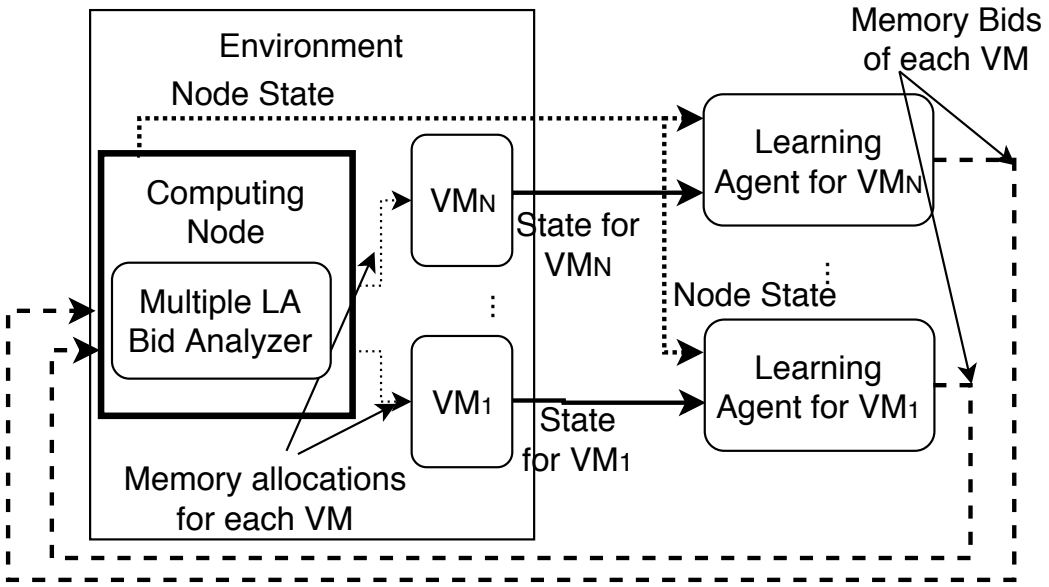


Figure 4.4: Diagram of the DDPG learning agents in the Memory Management Problem, with one learning agent per VM. The real implementation still looks identical to th

4.4.1 Pre-processing Bids for Memory

For the case of CARLEMM, the BA is looked at more closely. The BA checks the bids coming from the actor neural network of every learning agent before sending them to the balloon driver. The BA performs a series of high-level control tasks. First, it prevents VMs for bidding less than the minimum amount they need to function, as well as preventing the VMs to bid more than the actual memory available in the node. The BA also enforces bid prioritization for VMs that issue negative bids as an additional optimization.

The BA also performs a bid arbitration process to prevent aggregated overallocation of memory. Since the learning agents bid for memory independently, this may lead to cases where one or more agents bid for more memory than the amount available. To prevent overallocation, we first check if the requested bids exceed the total memory available in the node using Equation 4.3 (values are not normalized in this equation), where the subscript t is used to represent a respective variable in a specific timestep.

$$\sum_{j=1}^{j=N} [(a_t^{VM_j} + 1) \times M_t^{VM_j}] = totalMemUse <= Total\_Mem \tag{4.3}$$

In case equation 4.3 is not met when the bids are analyzed by the BA, then an arbitration process is called to adjust the bids for memory. This process is triggered whenever the amount of memory available for reallocation i.e. the amount of spare memory in the node plus the aggregated memory of the current of VMs that will release memory (negative bids), is smaller than the amount of aggregated memory needed by a subset of VMs that are asking for memory, then the bids of all VMs that are asking for memory will be readjusted.

Intuitively, the arbitration process is only required for the VMs bidding positively, as the remaining VMs either don't change their memory allocation (zero bid case) or are bidding negatively. Concretely, equation 6.2 summarizes the arbitration process and the proportional adjustment process, at time t, for the positive bids.

$$\bar{a}_t^{VM_j} = a_t^{VM_j} \times \frac{SpareMemory_k}{\sum_{j=1}^{j=P-1}(a_t^{VM_j} \times M_{t-1}^{VM_j})} \tag{4.4}$$

where P is the number of VMs that are bidding positively in timestep t, and $SpareMemory_k$ represents the memory available in the node after processing the negative bids, and is defined as follow:

$$SpareMemory = Total\_Mem - \sum_{i=1}^{i=N} M_t^{VM_j} + \sum_{j=1}^{j=L}(a_t^{VM_j}) \times M_t^{VM_j}) \tag{4.5}$$

in which L is the number of VMs that are bidding negatively and N is the total number of VMs..

All the tasks of the BA so far are illustrated in Algorithm 5. In Lines 12–13, the positive bids and negative bids are separated. The functions responsible of doing so generate two lists that are keyed with the IDs of each VM. In Lines 15–25, Algorithm 5 calculates first the de-allocations resulting from the negative bids first in order to obtain the amount of memory that will be freed. It also checks that the a VMs do not bid less than the minimum allowed, modifying the potentially offending bids (originally obtained from the Actor neural network of the DDPG agent) in order to meet this constraint

In Lines 27–33, Algorithm 5 calculates how much additional memory the positive bids are requesting for. In other words, how much memory is necessary to satisfy all the bids for memory. Lines 34 and 35 calculate the amount of spare memory in the system (memory that remains unallocated) and the amount of memory that is possible to reallocate, calculated as the sum of the spare memory and the memory that has been freed by negative bids.

In Lines 36–38, Algorithm 5 checks if the amount of requested memory exceeds the amount of re-allocatable memory. In case it does, the Arbitration process is called that readjusts the positive bids proportionally to the amount of memory in excess, in order to distribute more fairly the available memory for reallocation. In this way, a degree of fairness is provided for the memory distribution among the VMs.

Lines 39 and 40 then issue the bids to the Memory Manager (MM, as in previous chapters), so that it can pass it on the balloon driver and the HV. Notice that the negative bids are issued first, in order for the memory to be released before attempting to request more memory through the positive bids. It is not necessary to add synchronicity in this step, but it is an optimization that allows the balloon driver to respond faster.

4.4.2 Heuristic for Assisted Learning

Even though it was shown there is a good correlation between the memory allocations and the Percentage of User Space Time for cases in which the allocation remains static, the Percentage of User Space Time still presented responsiveness issues when the memory allocations vary.

Algorithm 5 Bid Analyzer (BA)

Let $state$ be defined as $state[VM_{ID}] = State$, where $State$ is defined as in Section 4.3.1
Let $actions$ be a list of the form $actions[VM_{ID}] = bid$, where $bid \in [-1.0, 1.0]$
Let pos_bids be a list of the form $pos\_bids[VM_{ID}] = bid$, where $bid \in [-1.0, 1.0]$
Let neg_bids be a list of the form $neg\_bids[VM_{ID}] = bid$, where $bid \in [-1.0, 1.0]$
procedure ARBITRATION($actions$, $spare\_mem$, $requested\_mem$)
    for each $vm\_id \in actions$ **do**
        $actions[vm\_id] = actions[vm\_id] \times \frac{spare\_mem}{requested\_mem}$
    end for
    return $actions$
end procedure
function BID_ANALYZER($actions, state$)
    $pos\_bids \leftarrow get\_positive\_bids(actions)$
    $neg\_bids \leftarrow get\_negative\_bids(actions)$
    $mem\_to\_release = 0$
    for $vm\_id \in neg\_bids$ **do**
        $bid \leftarrow neg\_bids[vm\_id]$
        $M^{vm\_id} \leftarrow state[vm\_id].M$
        $new\_alloc \leftarrow calc\_new\_alloc(bid, state[vm\_id])$
        if $new\_alloc < min\_alloc$ **then**
            $bid \leftarrow adjust\_bid(bid, state[vm\_id])$
            $neg\_bids[vm\_id] \leftarrow bid$
            $new\_alloc \leftarrow min\_alloc$
        end if
        $mem\_to\_release \leftarrow M^{vm\_id} - new\_alloc$
    end for
    $sum\_allocs \leftarrow 0$
    for $vm\_id \in pos\_bids$ **do**
        $bid \leftarrow pos\_bids[vm\_id]$
        $M^{vm\_id} \leftarrow state[vm\_id].M$
        $new\_alloc \leftarrow calc\_new\_alloc(bid, state[vm\_id])$
        $sum\_allocs \leftarrow sum\_allocs + new\_alloc$
        $requested\_mem \leftarrow new\_alloc - M^{vm\_id}$
    end for
    $spare\_mem \leftarrow Total\_Mem - sum\_allocs$
    $realloc\_mem \leftarrow spare\_mem + mem\_to\_release$
    if $requested\_mem > realloc\_mem$ **then**
        $pos\_bids \leftarrow Arbitration(pos\_bids, spare\_mem, requested\_mem)$
    end if
    issue_bid(neg_bids)
    issue_bid(pos_bids)
end function

This highlighted other problems when attempting to correlate the memory allocations to the

Percentage of User Space Time. This is analyzed in more detail in Section 4.8.1.

In order to deal with the responsiveness behavior of the Percentage of User Space Time, a heuristic was introduced to assist the learning agent. This heuristic, which we call Heuristic for Assisted Learning (HAL), was included as part of the action policy of the agent. Algorithm 6 shows the details of the heuristic.

Algorithm 6 Heuristic for Assisted Learning (HAL)

```
1: reward_array ← []
2: window_size ← 10
3: function HEUR_LEARNING(reward_t^{VM_j}, state^{VM_j})
4:     usrtime^{VM_j} ← state^{VM_j}.usrtime
5:     usrtimeTgt ← state^{VM_j}.usrtimeTgt
6:     reward ← reward_t^{VM_j}
7:     action ← 0.0
8:     if |reward_array|== window_size then
9:         reward_array.pop(0)
10:    end if
11:    reward_array.append(reward)
12:    avgReward ← sum(reward_array)/|reward_array|
13:    if avgReward < −usrtimeTgt/2.0 then
14:        if  thenusrtime^{VM_j} < usrtimeTgt
15:            action ← 4.0%
16:        else if usrtime^{VM_j} > usrtimeTgt then
17:            action ← −4.0%
18:        end if
19:    else
20:        action ← actor_inference(state^{VM_j})
21:    end if
22:    return action
23: end function
```

HAL starts by calculating the average reward obtained in the last ten timesteps of the execution (lines 7–11), which it accumulates on an array. Once the average reward is calculated, HAL checks if the *avgReward* is smaller (that is, more negative) than the negative of the half of the current User Space Time Target (line 12). This condition means that the reward is too far away from its maximum, which implies a large deviation from the performance target. If such condition occurs, then it proceeds to check if the $usrtime^{VM_j}$ of that VM is smaller (line 13) or larger (line 15) than the current target, and then generates an action of 4.0% or -4.0%, respectively (lines 12–18). However, if the *avgReward* is not in fact smaller, then the action is generated from the actor neural network of the learning agent using the state of the VM.

4.5 The Control Flow for CARLEMM

As explained in Section 4.3.1, CARLEMM instantiates a learning agent for each VM, allowing each learning agent to bid for memory such that the Percentage of User Space Time of its VM is equal to the target percentage value. Figure 4.5 shows the control flow of CARLEMM considering all of its building blocks: MM, balloon driver, the bid analyzer, the HV and the

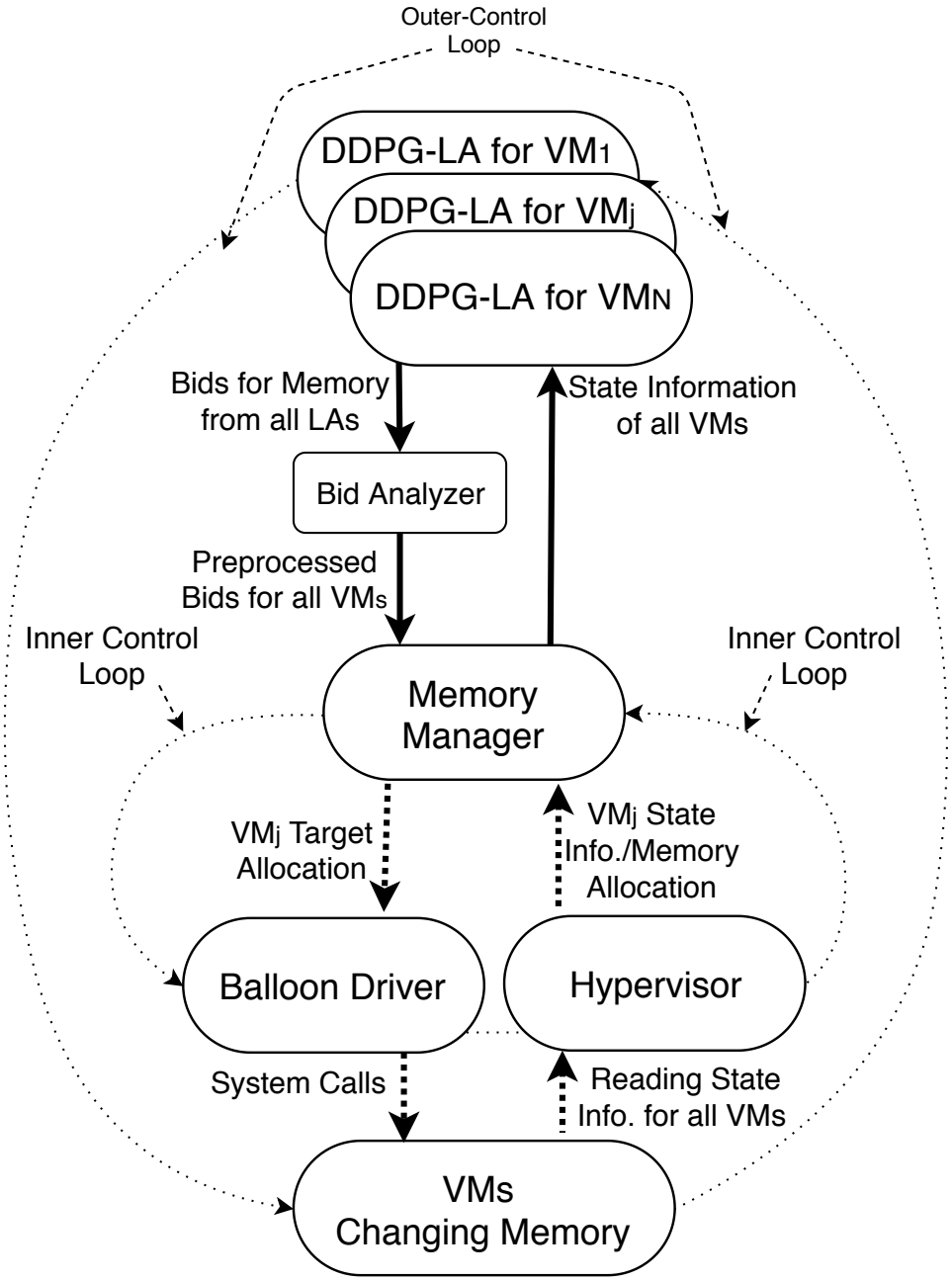


Figure 4.5: Diagram of the DDPG learning agents in a virtualized node deployed to manage memory, where each VM_j is bound to one learning agent. LA stands for "learning agent" in this figure.

learning agents. The learning agent of each VM is implemented as a Deep Deterministic Policy Gradient (DDPG) algorithm [60].

The state variables for each agent are obtained from the VM it is bound to and the node in which it is executed. All the values of these variables are obtained through a specialized

HV interface. The HV used in this implementation of CARLEMM is Xen, which provides a specialized interface called Xenstore [2], that allows the exchange of information among all the VMs in the node.

In CARLEMM, every VM runs a process that monitors the state of the VM and outputs the state information through a location in Xenstore. The MM reads the information from Xenstore through an API provided by Xen, does some pre-processing on this information and then sends it to the DDPG learning agents.

Each bid generated by the learning agent is sent to the balloon driver to allocate memory to the VM. The bid frequency for each agent is constrained by the reaction time of the balloon driver. This is because the balloon driver is relatively slow to allocate or deallocate, as explained in Section 1.3 of Chapter 1. To ensure that the requested bid is satisfied, the MM periodically checks the memory allocated to each VM every time it reads the memory statistics from Xenstore. In every cycle, the MM makes sure that the last memory allocations that were issued are met before sending the state information back to the learning agents. If these target allocations are not met, the MM will count the amount of cycles in which the target allocations and the real allocations are not met, and after a certain amount of cycles have gone through, it will transmit the state information back to the learning agents.

This is done so to allow the balloon driver to meet the target allocations before receiving new targets from the learning agents and issuing them to the balloon driver. It counts the cycles in order to prevent deadlock, and change the targets in case the balloon driver is unable to meet them after a certain amount of time. Experimentally, we found that when the target allocations for the balloon driver are modified too frequently, faster than the time necessary for the balloon driver to converge to an allocation target, the balloon driver turns unstable and presents an erratic behavior.

Additionally, the MM also checks the updates done in Xenstore by the monitoring process inside the VM. This process also generates a field for timestamps, that provides information on the current and last times that the status information was updated. When the MM reads the information from Xenstore, it checks whether if the timestamps have increased with the respect to the previous read before sending the information to the learning agents. This timestamp verification mechanism is used to guarantee that a state transition has occurred with respect to the previous timestep, preventing the sampling of the same state multiple times.

The MM thus implements two control flow mechanisms in the inner loop of the system: awaiting convergence of the balloon driver and ensuring state transitions by the VMs. Both of these features are indeed necessary to ensure a correct learning process, but in turn increase the running time of our experiments significantly.

4.6 Software Stack for CARLEMM

CARLEMM acquires the necessary memory statistics from the Xenstore API [2] provided by the Xen HV. The software stack for CARLEMM, shown in Figure 4.6, consists of the HV, the MM and the learning agents which are inside a separate Python process, all within the privileged domain of Xen. The DDPG learning agents are all implemented using Python Tensorflow. The communication between the MM and Python process is built using POSIX sockets.

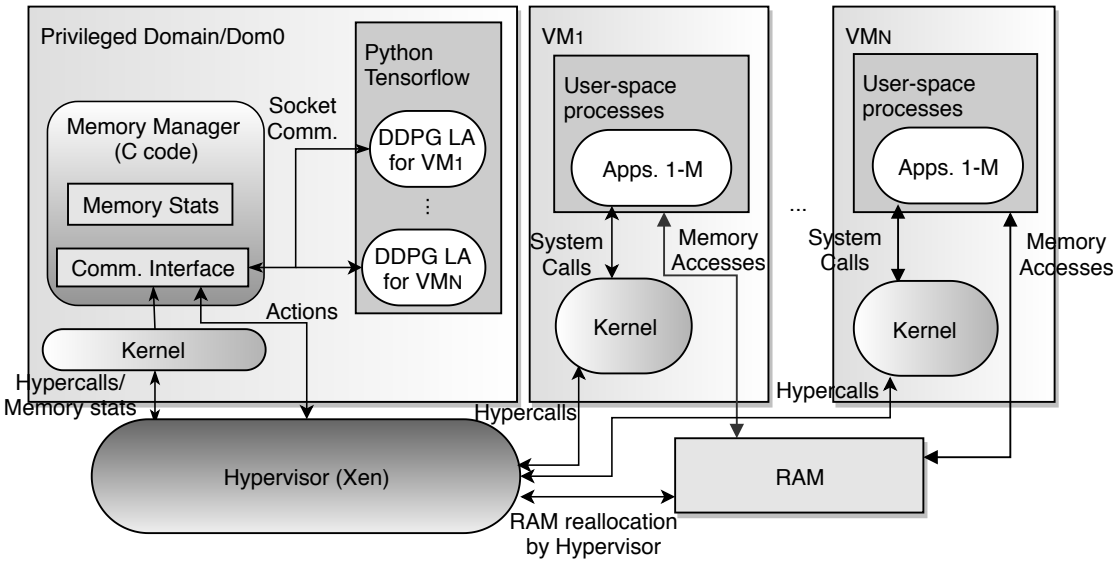


Figure 4.6: Diagram of the software stack of CARLEMM in a virtualized computing node.

The interface with the balloon driver is done through the MM, which is written in C. The MM uses Libxl tools to change VMs' memory allocation through balloon driver interface. The Libxl tools are used to send target allocations to the HV layer. To gather the memory utilization statistics for every VM and other metrics from the HV, the MM periodically reads data from the Xenstore interface in the HV, for which it used the appropriated API provided by Xen.

4.7 Experimental Framework for CARLEMM

We evaluated CARLEMM in a platform consisting of one computing node with Xen 4.5 running Ubuntu 14.04 with Linux kernel 3.19 as the OS in every VM. The node has an Intel i7 QuadCore Processor, 16 GB of RAM memory and 500 GB of storage.

Up until now, we have evaluated the effectiveness of CARLEMM running a series of experiments with a synthetic benchmark we called "Constant Memory Stress" (CMS) inside a VM. This benchmark is designed to stress the memory subsystem by allocating a fixed amount of memory and periodically performing writes and reads to the allocated blocks.

CMS performs the reads and writes following a Normal distribution across the address space of the allocated blocks. The parameters of the Normal distribution are specified when launching CMS. When specifying the mean and the standard deviation of the Normal distribution, it translates into the CMS process generating memory accesses around an address that corresponds to the mean with accesses around that region consistent with the standard deviation. If the memory accesses generated try to access an address beyond the largest or the smallest addresses, the access is clipped to the corresponding largest or smaller value, respectively.

In the experiments that we have analyzed so far, we launch one VM running CMS and setting the $usrtimeTgt$ to a constant value. At this stage, what we are more interested

to see is if the learning agent can learn to meet the performance target as intended, how long does it take to converge to the target and how well does it achieve this objective. The experiments were executed for 2000 episodes, where each episode has a 100 steps. The real time it took for such an experiment to run was of 60 to 72 hours.

The parameters used for the learning agents are shown in Table 4.2. The values used were the ones who yielded the best results after a series of experiments.

Parameter	Value
Actor Learning Rate	0.0001
Critic Learning Rate	0.001
Size of Experience Replay Buffer	10000
Annealing Episodes	180
Hidden Layers	2
Size of First Hidden Layer	64
Size of Second Hidden Layer	64
Activation function of Hidden Layers	Relu
Activation function of output layer	tanh
Total Episodes	2000
Timesteps per Episode	100

Table 4.2: Parameters for the neural networks of the actor and critic in every learning agent.

For the parameters k and α of Equation 3.10, we have used a value of 1.0 for both of them, which yielded the best results experimentally. The parameter β is used to drive the agent into avoiding actions that allocate or deallocate large amounts of memory in a single timestep. The parameter k acts as a gain factor to the difference between the VMs swap rate and the target swap rate. This last parameter is used to drive the agent into minimizing this difference.

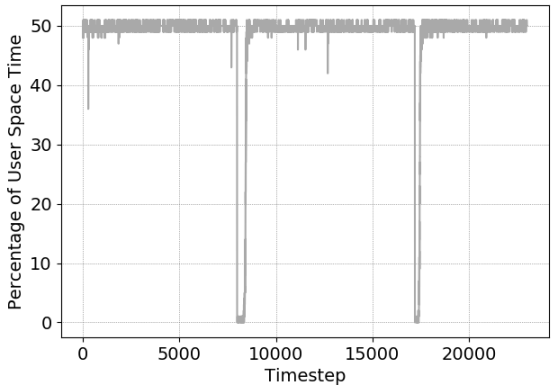
4.8 Experimental Results for CARLEMM

4.8.1 Experiments without Using HAL

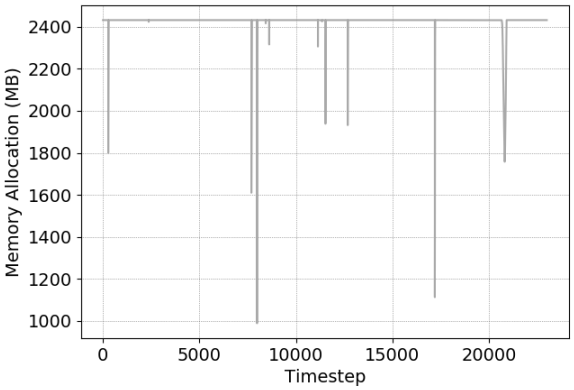
As explained in Section 4.7, most of the experiments we have run so far using CARLEMM consisted on running CMS in one single VM with a fixed $usrtimeTgt$ that is equal to 32.5.

The first experiments were run without HAL as described in Section 4.4.2. These first results showed that the learning agent by itself was not able to learn a good policy, and would instead get stuck on the minimum and/or maximum values (in this case, maximum), generating a large percentage of error and failing to learn how to generate useful actions. Figure 4.7 shows the Percentage of User Space Time and the Memory Allocation for the last 250 episodes of the experiment.

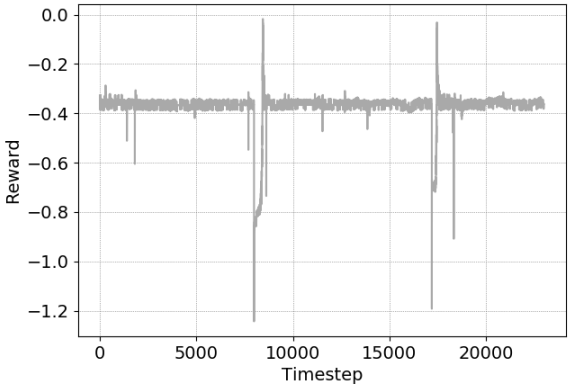
Figure 4.7 shows that the memory remains at its maximum, with a Percentage of User Space equal to 50.0 (its maximum as well), and it does not adjust whatsoever. The plot for the reward is also shown, which shows that the agent is unable to maximize the reward (bring it closer to 0.0) in the timesteps presented.



(a) Percentage of User Space Time at every Timestep



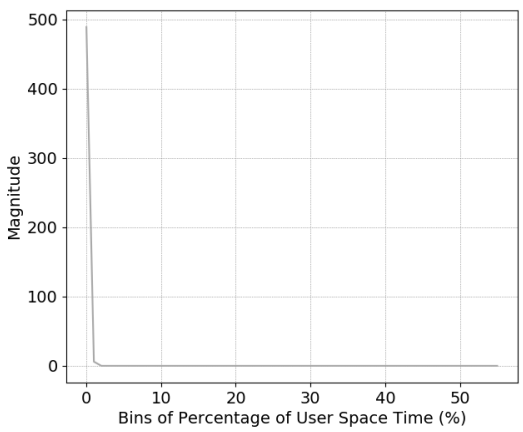
(b) Memory Allocation at every Timestep



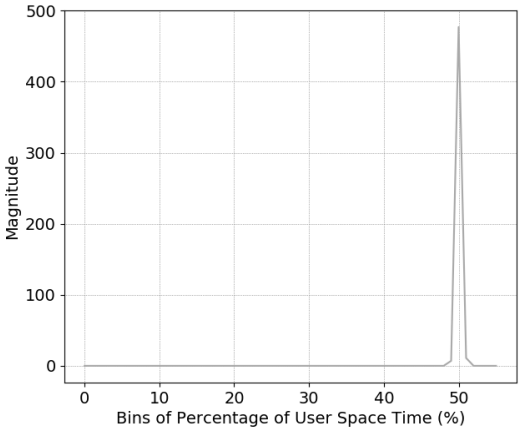
(c) Reward at every Timestep

Figure 4.7: Visualizing the behavior of the Percentage of User Space Time, memory allocation and the Reward at each Timestep according to Equation 4.1, for a VM running CMS over the last 250 epidodes.

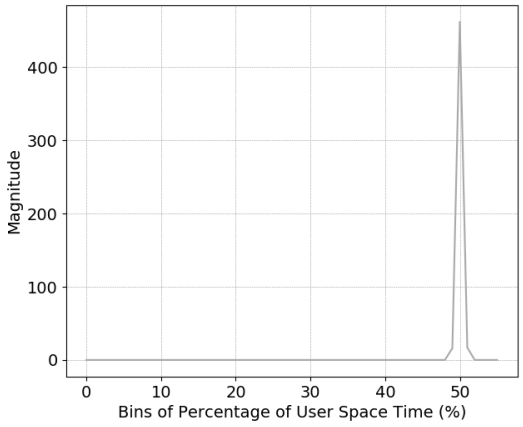
In order to provide more insight on the behavior of the learning agents of CARLEMM, Figure 4.8 shows four histograms in which the bins are the values of the Percentage of User Space Time rounded to the nearest integer. The bins are generated by sampling data gathered from CARLEMM in five consecutive episodes at different stages during the execution. The first histogram is taken around the 500th episode, the second is taken around the 1000th episode, the third one is taken around the 1500th and the fourth one is taken a bit before the 2000th episode, which is near the end of the experiment.



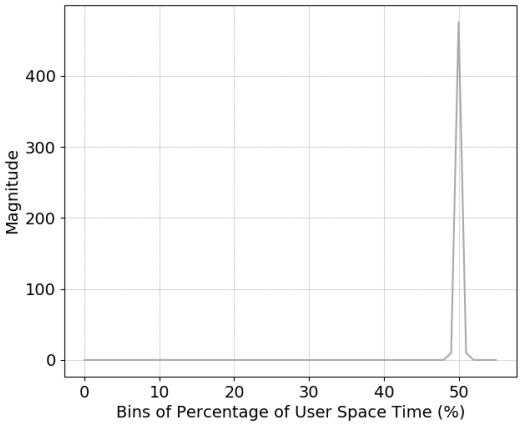
(a) Histogram sampled from the initial stage of learning.



(b) Histogram sampled from the second stage of learning.



(c) Histogram sampled from the third stage of learning.

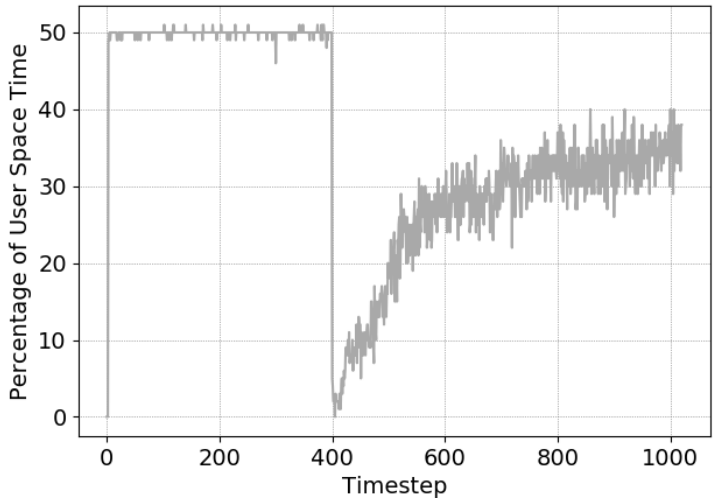


(d) Histogram sampled from the last stage of learning.

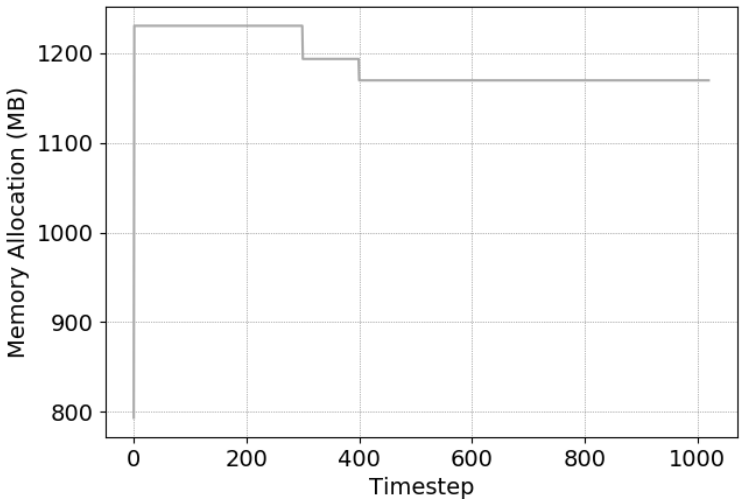
Figure 4.8: Histograms for the Percentage of User Space Time without HAL sampled at 500 episode intervals from the beginning until the end of the execution.

Notice that in Figure 4.8(a), the Percentage of User Space Time is mostly at zero, and in the subsequent histograms it moves toward the maximum value, which corresponds to the results observed in Figure 4.7. Different targets were tried and different parameters for CMS

inside the VM, but the results were fairly similar and consistent.



(a) Percentage of User Space Time corresponding to specific memory allocations



(b) Memory Allocations Decreasing at very controlled intervals

Figure 4.9: Analyzing the temporal response of the Percentage of User Space Time to variations in Memory Allocations.

The reason why the agent failed to learn in these circumstances is related to the way the Percentage of User Space Time responds to the variations on the memory allocation. Figure 4.9 illustrates this, by plotting in Figure 4.9(a) the time response of the Percentage of User Space Time to the corresponding variation of the Memory Allocation, shown in Figure 4.9(b), for the same timestep.

Notice that the memory allocated is reduced twice very slightly. During the first reduc-

tion, around 300th timestep, the Percentage of User Space Time remains at its maximum, meaning that the VM is executing its applications without any interruptions. But in the second reduction, at the 400th timestep, which is not a very significant drop, the Percentage of User Space Time drops to zero and then it increases slowly until it becomes stable around a value of 35%.

The second reduction of memory created a situation in which data previously allocated by a process inside the VM had to evicted from memory, because the new allocation was not enough to fit all the data in. This situation triggered an I/O operation to the swap device, which interrupted the execution of the application, thus reducing the Percentage of User Space Time abruptly. This abrupt reduction occurs even if the amount of data evicted was not very significant or even if the data evicted did not belong to CMS (a possibility, if there are other processes inside the vM).

However, the Percentage of User Space Time regains slowly in value as the data it generated starts to be read from disk replacing data that is accessed with less frequency, allowing the Percentage of User Space Time to adjust to a value that is better correlated to the memory allocation. The main problem with this behavior, is that it takes more than 600 timesteps to reach the stable value, which makes it difficult for the learning agent to capture the correlation between the Percentage of User Space Time, the memory allocation and the rest of the state variables.

Therefore, we concluded that we needed to add some *heuristic* to compensate for the behavior of the Percentage of User Space Time, and assist the learning process to prevent the agents from getting stuck on extreme values and approximate better the relationship between the Percentage of User Space Time and the rest of the state variables.

4.8.2 Experiments Using HAL

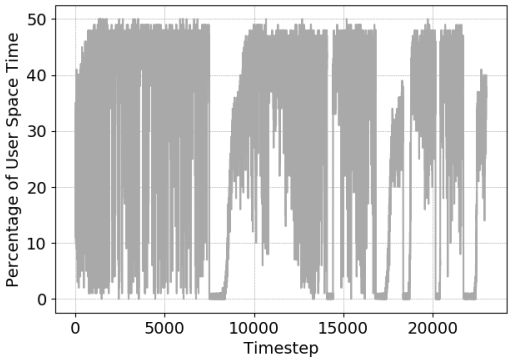
Figure 4.10 shows the same experiment, where there is one VM executing CMS with $usrtimeTgt = 32.5$, but this time enabling HAL, which is described in Section 4.4.2.

Notice that in the case of Figure 4.10(c), the values of the reward are able to reach their maximum towards the end of the run, but with some instability in the way the allocations are issued, as shown in Figure 4.10(b). Notice the Percentage of User Space Time presents the behavior explained in the previous section, but with the help of HAL, the agent is able to approximate its value to the desired target.

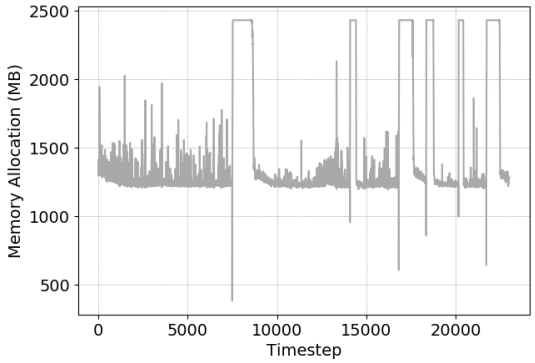
To reinforce this claim, Figure 4.11 shows four histograms in which the bins are the values of the Percentage of User Space Time rounded the nearest integer, and notice how the values approximate the desired target. This is clear proof that CARLEMM is able to learn to approximate the desired performance target, but does not do it perfectly.

Notice that in the last histogram shown in Figure 4.11(d), the agent is able to approximate the desired performance target very closely towards the end of the experiment, but with still some inaccuracies and instability.

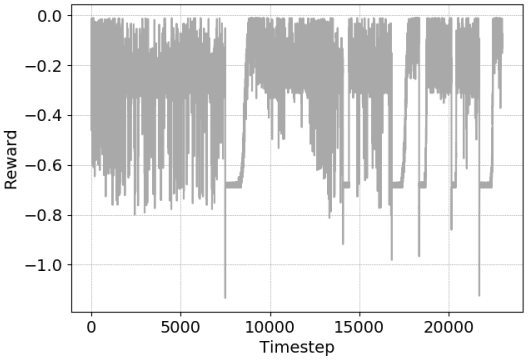
The results with HAL work better because HAL prevents the learning agent to continuously bid around values that keep Percentage of User Space Time away from its extreme values. Due to the lack of immediate correlation between the Percentage of User Space Time and the memory allocation, it is necessary to ensure that the memory allocations remain around certain values that allow for Percentage of User Space Time to become more responsive, thus improving the correlation relationship with respect to the memory allocated,



(a) Percentage of User Space Time at every Timestep



(b) Memory Allocation at every Timestep

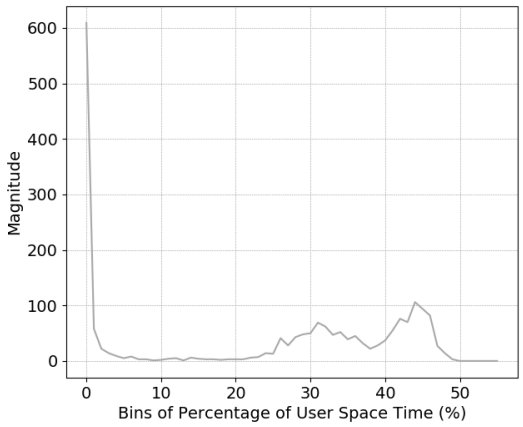


(c) Reward at every Timestep

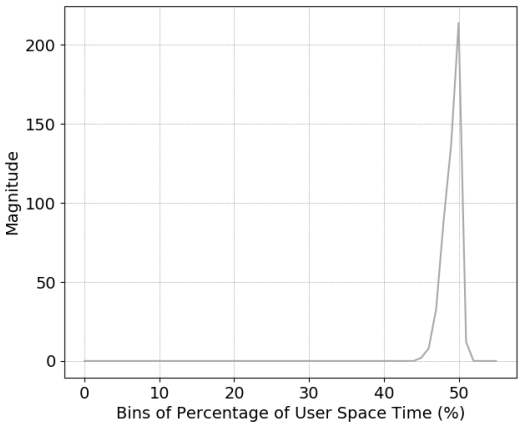
Figure 4.10: Visualizing the behavior of the Percentage of User Space Time, memory allocation and the Reward at each Timestep according to Equation 4.1, for a VM running CMS over the last 250 episodes with HAL enabled.

therefore improving the learning possibilities of the agent.

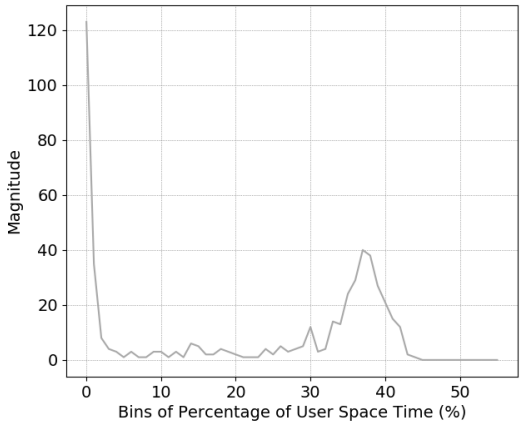
Even with the introduction of HAL and the improvements of the learning agent, it still



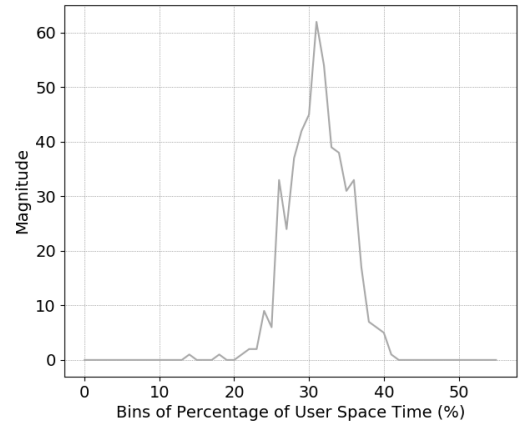
(a) Histogram sampled from the initial stage of learning.



(b) Histogram sampled from the second stage of learning.



(c) Histogram sampled from the third stage of learning.



(d) Histogram sampled from the last stage of learning.

Figure 4.11: Histograms for the Percentage of User Space Time with HAL enabled, sampled at 50 episodes interval from the beginning until the end of the execution.

presents some instability and it takes around 72 hours just to learn to approximate the scenario with a constant target of $usrtimeTgt = 32.5$. Arguably, by letting the learning process run for longer, it is possible that it could have been refined and approximate the value better. However, more experiments are still pending, specially those with variable target.

4.9 Related Work

Many RL-based solutions have been proposed to solve the resource allocation problem in cloud infrastructures [13, 87] but so far, there have not been too many efforts that use

RL to exclusive solve the memory management problem, and much less continuous action RL solutions. Many of the related work for CARLEMM overlaps with the related work CAVMem [37].

As explained in the beginning of this chapter, the design or CARLEMM is based on the conclusions obtained from CAVMem [37]. However, when deploying a continuous action RL agent in a real virtualized node, different constraints have to be considered due to the dynamics of the computing node. For example, some of the constraints that can be mentioned are the responsiveness of the balloon driver, the times of convergence and state transitions for the VMs, the definition of the state of a VM (that can be defined with different sets of variables) and other issues related to control flow of the computer system.

4.10 Conclusions for this Chapter and Future Work

This chapter proposed a method for dynamic memory allocation for virtual machines (VMs) called CARLEMM that applies continuous action RL, based on the proof-of-concept provided by CAVMem [37]. As it was the case with CAVMem, CARLEMM addresses two key drawbacks of prior work. Firstly, it uses continuous action RL, which avoids the unnecessary limitations introduced by discretization. Second, it formulates the problem in a distributed way, i.e. with one agent per VM, which supports an arbitrary number of VMs as well as the starting and stopping of VMs.

The most important contribution of CARLEMM presented in this chapter is that it is the first known attempt to deploy a continuous action RL agent to solve the memory allocation problem. The constraints and the limitations of the real system are way more complicated than in the simulated environment used in Chapter 3, but our experiments showed an important progress in CARLEMM's to approximate the performance targets somewhat reasonably. However, for future work, more experimentation and revision is necessary to make CARLEMM work for more general cases and more reliably.

Chapter 5

Aggregating and Managing Memory Across Computing Nodes in Cloud Environments

In the chapters so far, this thesis has focused on the memory management problem within a single virtualized node. Some mechanisms were developed for RAM and Tmem, but still the mechanisms for cross-node memory aggregation have not been described, which was initially the main purpose of this thesis work.

In this chapter, we introduce the first of two mechanisms that were developed to achieve this. The first of these mechanisms is called GV-Tmem (acronym for Globally Visible Transcendent Memory, as previously stated), which was the first effort to enable remote memory capacity sharing across a cloud infrastructure. GV-Tmem provides a more simplified version of a more thoroughly designed mechanism called vMCA, introduced in Chapter 6

When a cloud infrastructure is able to support sharing of memory capacity across nodes, memory becomes a global resource, enabling the accessibility of each node to a large pool of memory that could be instrumental to avoid costly swaps or VM migrations. GV-Tmem achieves this by exploiting Tmem and uses a user-space process to manage the memory available to a node. This user-space process is also in charge of distributing the aggregated memory across the computing infrastructure.

5.1 Introduction to GV-Tmem

Current data centres use large numbers of servers provisioned with their own computing resources. These servers are designed to share none of their resources, and they communicate over an Ethernet (or similar) network. Newer system architectures, such as Euroserver [26] try to improve over this resource isolation approach, allowing servers to share their resources through the memory address space. These systems provide a shared global physical address space, accessing memory at low latency using very fast interconnects. In these systems, the computing nodes of the infrastructures are seen as *coherence islands*, and each of these have their own cache coherency enforced locally, with no global hardware coherence.

Even in infrastructures with these features, each node has its own HV which distributes the node's memory among one or more VMs. The demand for memory resources generated by the VMs varies over time due to the different workloads they execute. To improve utilization of the memory capacity of the node, physical memory is often overcommitted, which causes a VM to have less memory than the amount it was configured with at boot time. As in mentioned in Chapter 1.1, the physical memory given to a VM is usually adjusted using memory ballooning, memory hotplug or Tmem [69].

This chapter presents a mechanism, called GV-Tmem, that illustrates the first extensions to a state-of-the-art hypervisor to enable the sharing of memory capacity across nodes in a cloud computing infrastructure. GV-Tmem introduces changes to the hypervisor, keeping it small and self-contained. Most of the complexity is in a user-space memory manager process running in the privileged domain that supports memory management policies and inter-node communication. Our main contributions in this chapter are:

1. A software architecture to aggregate memory across nodes using Tmem.

2. A two-tier mechanism for allocation and management of aggregated memory.

The rest of the sections in this chapter are organized as follows. Section 5.2 gives the necessary background to give a sound context to GV-Tmem. Section 5.3 explains GV-Tmem. Section 5.4 describes the experimental methodology and Section 5.5 shows our results. Section 5.6 compares with related work and Section 5.7 concludes the paper and outlines future work.

5.2 Background for Remote Memory Aggregation

This section recaps on virtualization and memory management in cloud data centers, highlighting the most relevant aspects of each to build GV-Tmem. It also recaps on Xen's Tmem and outlines the hardware characteristics that GV-Tmem exploits. This section is important to understand both the contributions of this chapter and the next.

5.2.1 Virtualization In Cloud Computing Infrastructures

There are many cloud service models available in the industry, such as SaaS (Software-as-a-Service), PaaS (Platform-as-a-Service) and IaaS (Infrastucture-as-a-Service). Considered one of the most important, IaaS allows users to dynamically access a configurable (seemingly

unlimited) pool of computing resources. In IaaS, the customers share the underlying hardware computing resources of the cloud infrastructure in a transparent way. These sharing of resources is possible by deploying commercially available HVs such as Xen [10], KVM [41] or VMWare [74].

As explained in previous chapters, the HV virtualizes the computing resources of the node, creates VMs and allocates memory for them. The HV is able to overcommit memory, and there are cases in which a VM may be over-provisioned or under-provisioned with memory, both conditions which are undesireable. To prevent these cases, HVs have mechanisms, like ballooning, hotplug and Tmem, that enable it to reallocate memory using high-level policies [35, 15, 37].

However, when the memory demand keeps increasing, at some point the memory available on the node won't be enough to ensure an acceptable performance for the applications running inside the VMs. When this occurs in a cloud infrastructure, some of the VMs that are creating the memory load in a node will either face a large performance drop (because they are blocked from execution or they are generating a large amount of swap operations) or will be migrated to a different node that can handle the memory load better. Even though migration will be preferred in most cases, the reality is that both of these situations (swapping or migration) compromise the performance of the applications inside the VMs and consume a significant amount of energy.

Another alternative to migration or costly swap operations, would be to allow for the nodes to pool their memory resources (at least partially) through their address space, disaggregating the memory from a node and allowing it to be aggregated to any node in the infrastructure in a more flexible way.

As we have mentioned, the HVs provide mechanisms for dynamic memory management, like memory ballooning and memory hotplug, but the interfaces provided by these mechanisms are not very well-suited for aggregation of memory capacity across nodes, because the memory that is made available to the VMs as RAM (as it would be through ballooning or hotplug) imposes important constraints on the coherency and consistency of that memory. On the other hand, Tmem provides a more flexible interface to aggregate memory across nodes and even for its use in heterogeneous memory hierarchies. However, for a mechanism to aggregate memory across nodes to become viable and more attractive than just letting a VM do swap operations or migrate, the hardware needs to have certain features.

5.2.2 Hardware Support in Virtualized Nodes for Remote Memory Aggregation

In systems such as Venice [24] and EUROSERVER [26] (based on ARM processors), the processors in each node are connected in clusters via a local cache-coherent interconnect to local resources. In particular, the Euroserver architecture implements the UNIMEM (*Unified Memory*) model [26]. Figure 5.1 illustrates how UNIMEM works.

Systems that are based on UNIMEM usually consist of N processor cores (6 to 8 in current implementations), grouped in clusters connected via a local cache-coherent interconnect to local DRAM and I/O devices. Remote memory is visible through the global physical address space, and the *inter-node interface* together with the *global interconnect* supports communication among the nodes, routing the remote memory accesses to the appropriate node.

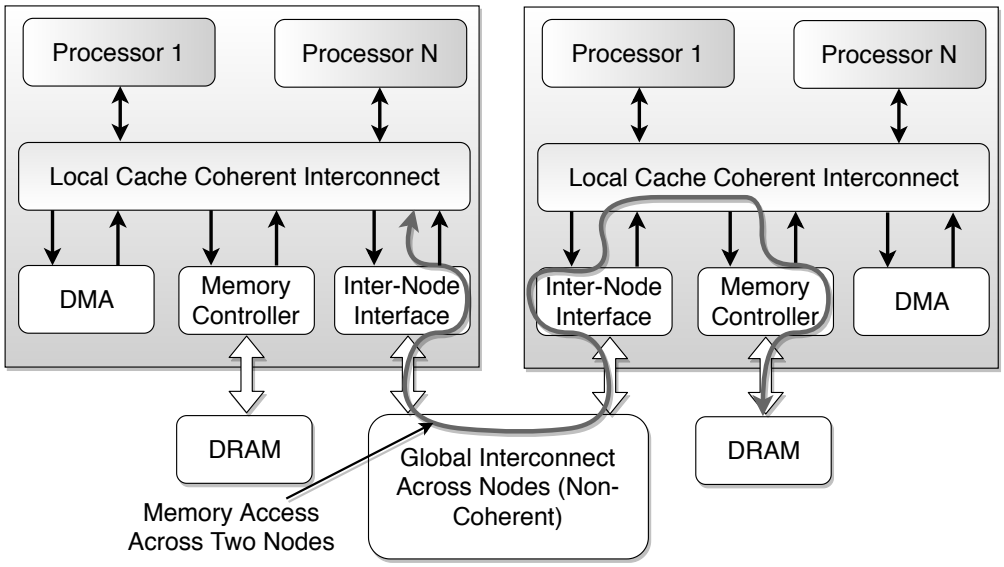


Figure 5.1: UNIMEM architecture with two nodes i.e. two coherence islands

There are other architectures that are also based on similar principles to UNIMEM but using different memory models, such as dRedBox [49] and Beehive [106]. The essential characteristics of all of these architectures are:

- Each node executes its own server stack, including hypervisor and OSs.
- Global physical memory address space with low-latency access.
- Routing is based on the global physical address (e.g. high-order bits).
- Fast communication is provided across the nodes of the system, bypassing traditional network protocols (e.g. TCP/IP).

5.3 GV-Tmem Design

The GV-Tmem stack consists of three software components:

- Extended Xen HV for GV-Tmem (Section 5.3.1)
- Tmem Kernel Module in the kernel of all domains to support GV-Tmem (Section 5.3.2)
- Memory Manager (MM) in user space in Dom0 for GV-Tmem (Section 5.3.3)

5.3.1 Xen Hypervisor with Extensions for GV-Tmem

As was the case for SmarTmem in Chapter 2, it was necessary to extend a state-of-the-art VH (Xen in this case as well) in order to support GV-Tmem. The HV extensions for GV-Tmem are minimum and localized in the Tmem subsystem. First, the HV enforces the memory allocation constraints determined by the MM. Second, it allocates and deallocates

physical pages and passes ownership of blocks of pages in and out of the HV. Third, it collects information of Tmem utilization that it sends to the MM, similar to SmarTmem presented in Chapter 1.1.

Enforcing Local per-VM Memory Constraints

The HV constrains the local and remote Tmem consumption of its VMs, based on the allocation determined by the MM. The MM specifies the maximum number of pages a VM can use, but it does not differentiate between the amount of local or remote Tmem pages.

Page Allocation and Transfer of Ownership

GV-Tmem ensures that each physical page is *owned* by at most one HV. Tmem pages owned by a HV are allocated using a zoned Buddy allocator, with a zone for each node from which it has ownership of at least one page.

A Tmem put operation allocates the closest free page from the allocators. Thus, it will preferably allocate a local page to a remote page. A Tmem flush operation causes a page to be returned to the corresponding Buddy allocator, regardless if the page is remote or local. In order words, a *flush* of any kind does not change the ownership of the page.

A Grant hypercall is used when the MM receives ownership of a list of blocks, where each block is appropriately-aligned to power-of-two number of pages. These are added as free blocks to the appropriate Buddy allocator. In contrast, a Request hypercall is used to release ownership of pages on behalf of another node.

Memory Statistics Gathered in GV-Tmem

The HV collects information about the Tmem utilization of the VMs. More specifically, the HV monitors:

- Number of active VMs
- Amount of total Tmem capacity available to the HV
- Amount of Tmem capacity in use by each VM
- Number of put, get and flush operations of each VM
- Number of failed put operations of each VM

The information gathered needs to be minimized in order to avoid excessive communication overheads between different contexts of execution, which result when multiple page-copy operations take place from the HV to the user-space MM. The HV sends this information to the MM approximately every second by issuing a custom virtual interrupt request (VIRQ) to the privileged domain. The TKM (Section 5.3.2) captures the interrupt and forwards the information to the MM

5.3.2 Tmem Kernel Module (TKM) for GV-Tmem

The TKM in this case performs a similar function as its counterpart for SmarTmem in Chapter 2.2.3, and it is also an extension of the baseline state-of-the-art TKM present in current Linux kernels which only provide basic support Tmem. In most of the VMs, the TKM acts as an interface between the VM and the underlying Tmem support in the HV. But in the privileged domain (Dom0), the TKM acts as an interface between the HV and the node's MM using netlink sockets.

The TKM needs to be extended with respect to the case of SmarTmem to support the different commands configured and implemented in the MM and the different hypercalls in the HV.

5.3.3 Dom0 User-space Memory Manager for GV-Tmem

The MM was created from scratch, as it was the case for SmarTmem in Chapter 2, and bears most of the functionalities to support GV-Tmem. Each node has a user-space MM in Dom0. The MMs perform most of the work of GV-Tmem by cooperating to:

1. Distribute memory owned by each node among its guests
2. Distribute global memory capacity among nodes
3. Implement the flow of page ownership among nodes
4. Enable nodes to join and leave, and handle failures

Most of the complexity of GV-Tmem has been kept in the MM. This has been designed in this way for the same reasons explained in Section 2.2.4 for the case of SmarTmem. Moreover, there is an additional restriction for the case of GV-Tmem (and for vMCA as well): GV-Tmem has to support communication with different MMs across remote nodes, which is implemented as TCP/IP sockets (not through the partial global physical address space). Adding socket communication support in the HV entails a series of complications, which may yield an over-grown and insecure HV.

Joining the GV-Tmem System

There is one MM Master (MM–M) that controls the system and distributes the global memory capacity. The messages passed among the MMs are listed in Table 5.1. Every node requires a configuration file, which provides the network addresses of all nodes, their mappings to a node ID and credentials to establish secure connections. When a node R wishes to join the GV-Tmem system, it sends Register to the MM–M (see Table 5.1), which enables the node to partake in the memory resource pool created across the infrastructure.

Distributing Memory Owned by a Node Among Guests

The local MM determines the maximum number of pages for each VM. This is done using a policy that determines this maximum based on the statistics received from the HV. This is the first tier of the memory management strategy. Pages are distributed subject to a memory consumption limit, set by the MM–M using the Mem-Limit hypercall.

Command	Direction	Description	Slave state
Distribution of global memory capacity			
Statistics(S)	SL→MT	Send node statistics S to Master	A
Grant-Any(n, x)	SL→MT	Request grant of n pages to slave x	A
Grant-Fwd(n, x)	MT→SL	Forward request of n pages to y from x	A
Force-Return(x)	MT→SL	Return pages located at x and disable it	A
Mem-Limit(n)	MT→SL	Limit allocated pages to store local data	A
Flow of page ownership			
Grant($b, \cdots$)	SL/MT→SL	Transfer ownership of blocks of pages	A
Node state changes			
Register	SL→MT	Register a new node	I→A
Leave-Req	SL→MT	Node requests to leave or shutdown	A→L
Leave-Notify	MT→SL	MM–M notifies that the recipient has left	L→I
Enable-Node(x, e)	MT→SL	Accept ($e = 1$) or reject ($e = 0$) pages at x	A

Table 5.1: MM message types. *SL*: *slave*, *MT*: *master*, *I*: *Inactive*, *A*: *Active*, *R*: *Recovery*, *L*: *Leaving*

Distributing Global Memory Capacity Among Nodes

Nodes in the Active state, regularly send Statistics messages to the MM–M. These messages consist of the statistics gathered by the HV of each node and information regarding the utilization and requests for remote memory of the node. In this way, the MM–M has a general view of the status of each node from a perspective of memory utilization.

Based on these statistics and the global memory policy, the MM–M redistributes the memory among nodes when it receives a new Grant-Any message, which is a request to transfer ownership of a number of free physical pages to a node that requested memory. The MM–M can forward the request to another node by sending a Grant-Fwd message or give some of its pages to the requesting node. This is the second tier of the memory management strategy.

Implementing Flow of Page Ownership

The MM–M rebalances Tmem capacity, without knowing the physical addresses of the pages. Ownership of physical pages is transferred in a peer-to-peer way using Grant messages, which passes a list of blocks of addresses, each an appropriately-aligned power-of-two number of addresses of physical pages.

Once a node is granted ownership of remote pages, it has exclusive access to them, and it is free to allocate these pages to store data on behalf of its VMs. The other nodes in the system, including the donor node, will not perform any reads or writes to those pages, since a page can be owned by only one node at a time. Thus, there is no need to order writes among multiple nodes.

With this way of distributing memory, the OS inside VMs is oblivious to the amount and location of the Tmem pages, only the HV has a clear view of these details. The VMs are able to store any data that its VMs attempt to swap to disk, without any constraints regarding the nature of the data.

Leaving the GV-Tmem System

To cleanly shutdown a node R that is in GV-Tmem, the following procedure must be followed.

1. Node R sends a Leave-Req message to the MM–M.
2. Upon receiving the Leave-Req, the MM–M sets the node to Leaving state and sends Force-Return(R) to all nodes. The nodes return the pages at R that they own and will reject any pages received in future Grant messages.
3. Node R frees all pages used by Tmem and returns ownership of all remote pages to their home nodes.
4. Periodically, each node sends Grant messages to node R to return ownership of the pages that it had borrowed.
5. Once the MM–M has received Statistics messages from all nodes indicating that R is disabled and that it owns no pages at R, the MM–M moves R to Inactive and sends Leave-Notify to R.
6. At this point, the node R may shutdown.

5.3.4 Hardware Support for Memory Aggregation

As it was explained in Section 5.2.2, GV-Tmem is suitable to exploit the functionalities of UNIMEM-based architectures such as Euroserver [26], that possess certain hardware characteristics. Coupling these hardware features, with a virtualization layer and the software stack provided by GV-Tmem, we can enumerate in a clearer way the requirements that the underlying virtualized node needs to provide in order fo fully support GV-Tmem:

1. A fast interconnect, providing a synchronous interface across the system.
2. Direct memory access from the HV to all the memory available. Memory accesses from the HV could be either through load/store instructions or RDMA, bypassing TCP/IP or similar protocols that require OS functionality.
3. Remote access to a node's pages is disabled on hardware boot. Access is enabled only when the node joins GV-Tmem by sending the Register message.
4. Given a physical address, it must be possible to extract the Node ID i.e. identify each node and route packets toward it.

5.3.5 Putting Everything Together

So far we have described the elements of the software stack of GV-Tmem and the hardware features it exploits for its correct functionality. We have described the message passing between the nodes and the procedures necessary to enable the pooling of memory resources.

Once every node inside the infrastructure implements the GV-Tmem software stack, they will be able to take part in the pooling of memory capacity. Figure 5.2 shows the architecture of a GV-Tmem system fully deployed.

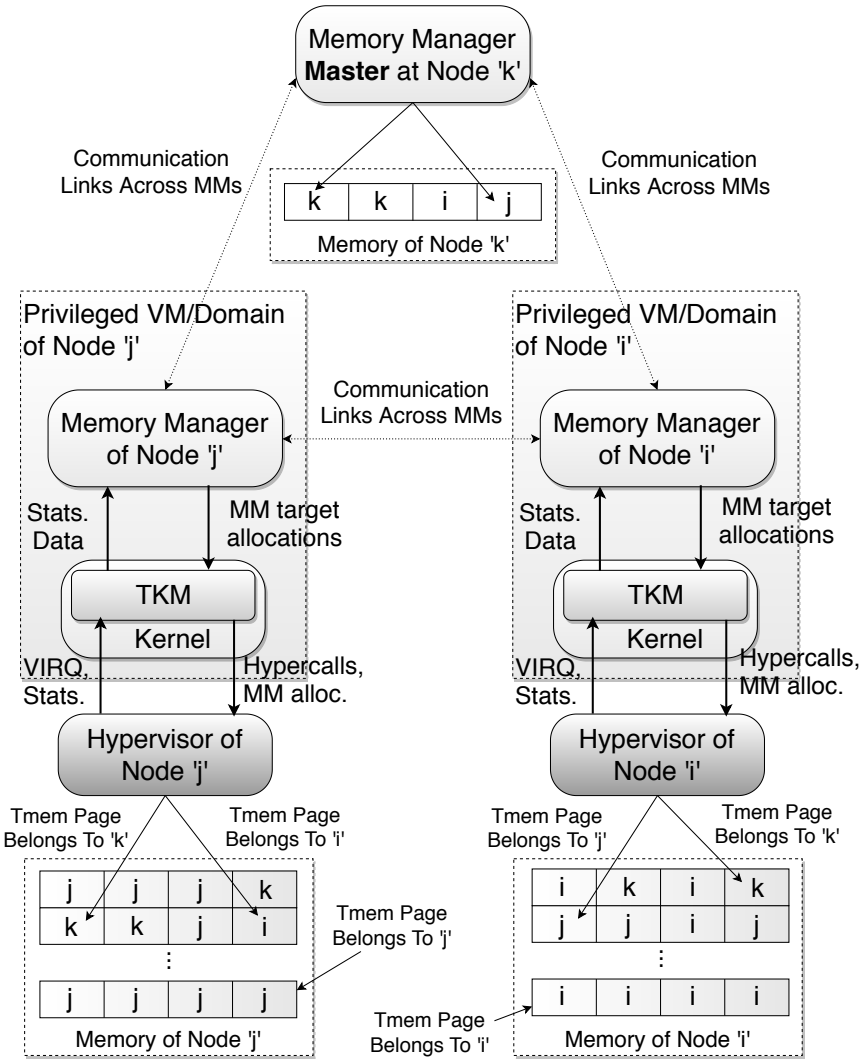


Figure 5.2: A diagram of the architecture of a GV-Tmem system fully deployed.

Figure 5.2 presents nodes "i", "j", "k", where node "k" has the MM–M, with the respective communication links across the privileged VMs of every node, in which the corresponding MM is running. Every node implements the software stack and the functionalities for every component. The software stack for node "k" was omitted only on the figure for simplicity.

As it can be seen, this is very complex architecture with a lot of moving parts. It also has similarities with the architecture presented in Chapter 2 for SmarTmem. Even though the building blocks (HV support, a specialized kernel module and a user-space process) are seemingly common components between SmarTmem and GV-Tmem, the design of the functionalities and interfaces within each component vary dramatically, enabling the architectures to solve different issues, making them into totally different contributions altogether.

Node	CPU	Frequency	Memory
Node 1	AMD FX Quad-Core	1.4 GHz	6 GB
Node 2	Intel Core i7	2.10 GHz	8 GB
Node 3	Intel Xeon	2.262 GHz	64 GB

Table 5.2: Hardware characteristics of the nodes used to evaluate GV-Tmem

Scns.	VM Parameters	Description
Scn. 1	VM1, VM2: 768 MB RAM, 1 CPU; VM3: 1 GB RAM, 1 CPU	All VMs execute in-memory-analytics, sleep 5 seconds and then execute it again.The data set used is from [43].
Scn. 2	VM1, VM2, VM3: 512 MB RAM, 1 CPU	VM1 and VM2 execute *usemem*, and VM3 starts when VM1 allocates 640 MB.
Scn. 3	VM1, VM2: 512 MB RAM, 1 CPU	Every VM executes graph-analytics once. They use the dataset provided by [93], [92], [91].

Table 5.3: List of scenarios used to evaluate GV-Tmem

5.4 Experimental Methodology

We tested GV-Tmem in a platform consisting of three nodes, which is consistent with UNIMEM-based architectures that currently have 4 to 8 nodes, with 6 to 8 processor cores per node [26].

Every VM runs Ubuntu 14.04 with Linux kernel 3.19.0+ as the OS, and Xen 4.5. The MMs in the nodes communicate using Ethernet TCP/IP sockets. Node 2 acts as the Master node and executes no VMs. Table 5.2 summarizes the hardware properties of the nodes.

The shared global address space was emulated using the node's local memory. Remote access emulation was the best choice in order to offer the possibilty to analyze the impact of the latency of the interconnect on the memory aggregation and management mechanisms. However, the analysis of the impact of latency and non-uniform latencies is out of the scope of GV-Tmem for now. We modified Xen to start up using a portion of the physical memory capacity, equalling the emulated memory capacity of the node. The rest of the node's memory capacity was reserved to emulate remote data storage. Whenever the HV performs an emulated remote access after remote memory becomes available, we add a delay in the HV lasting 50 µs to model hardware latency.

We evaluate GV-Tmem using CloudSuite 3.0 [31] and a microbenchmark we designed called *usemem* that was also used for the evaluation of SmarTmem, as mentioned in Section 2.3. We execute at most three DomUs simultaneously, and refer to each set of DomUs as a *scenario* (or *Scn*). Table 5.3 shows the scenarios used. For Scenarios 1 and 3, all nodes have 1 GB of Tmem capacity. For Scn. 2, Node 2 has 1 GB of Tmem, while Nodes 1 and 3 have 384 MB.

As it was with the case of SmarTmem in Chapter 2, the scenarios created here, the configurations of the VMs and the datasets were chosen in order to generate a considerable amount of memory pressure within the VMs, generating memory footprints two or three times of their initial memory capacity [100].

This paper uses three memory management policies:

- **greedy-local**: Default policy used in Tmem with only local memory, which gives memory away on demand. No maximum values are set to limit the amount of memory taken by a VM.
- **greedy-remote**: An extended version of greedy-local using remote memory.
- **TTM**: A two-tier memory management strategy that allocates memory locally for each VM (first-tier) depending on the node's statistics, and issues requests for remote memory (second-tier) depending on the perceived memory pressure. The pages allocated and deallocated to a VM are increased by a percentage $\%P$ of the pages owned by the node (local or remote).

5.5 Results

5.5.1 Results for Scenario 1

Figure 5.3 shows the average running times of each VM for Scn. 1. The running time improves by an average of 19.4% and 23.5% in nodes 1 and 3, respectively, when going from greedy-local to greedy-remote. When implementing TTM with $P = 2.0\%$, there is further improvement of 6.0% and 4.0% over greedy-remote, demonstrating the need to implement memory management policies when there is significant memory pressure.

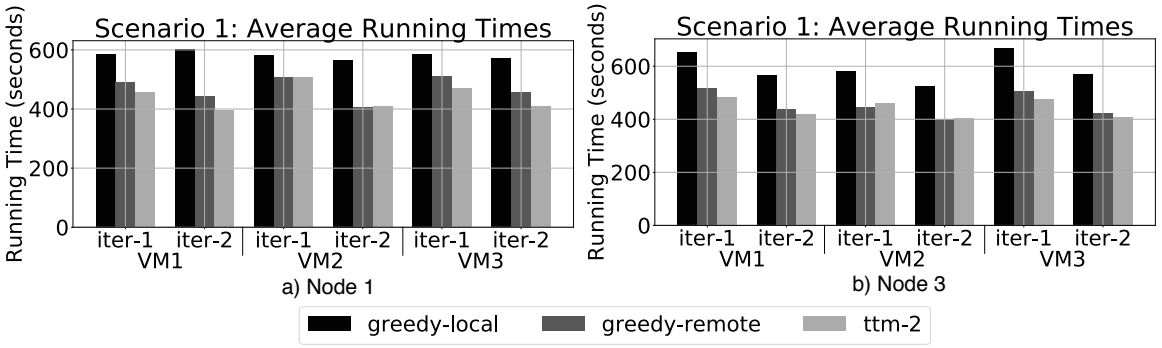


Figure 5.3: Running time for Scn. 1 in nodes 1 and 3. Time is in seconds (less is better).

Figure 5.4 shows the amount of Tmem capacity that each VM is able to use for the three policies mentioned in node 3. With greedy-local (Figure 5.4(a)), VM3 in both iterations cannot obtain a fair share of the available Tmem capacity. With greedy-remote (Figure 5.4(b)), the VMs are able to get more total Tmem, but because of the lack of memory management policies, some VMs are unable to obtain a fair share of Tmem. With TTM (Figure 5.4(c)), every VM is ensured a fair amount of the available Tmem, demonstrating that TTM is able to ensure fairness regarding the VMs' allocation of Tmem, improving the running times.

Similar as it was defined in Section 2.4, fairness is define qualitatively as being equivalent to all the VMs having an equal proportion of the available Tmem because of the amount of memory pressure they impose on the node.

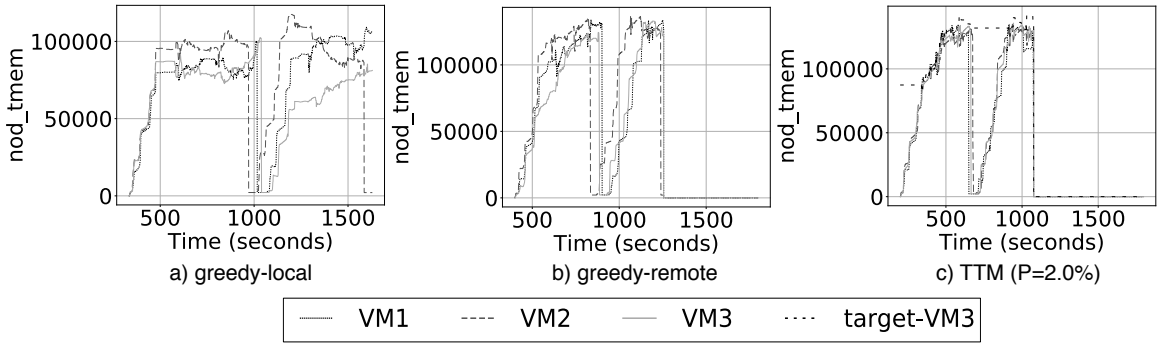


Figure 5.4: Tmem capacity (nod-tmem) obtained by every VM in node 3 for Scn. 1. The label *target-VM3* refers to the target allocation of VM3.

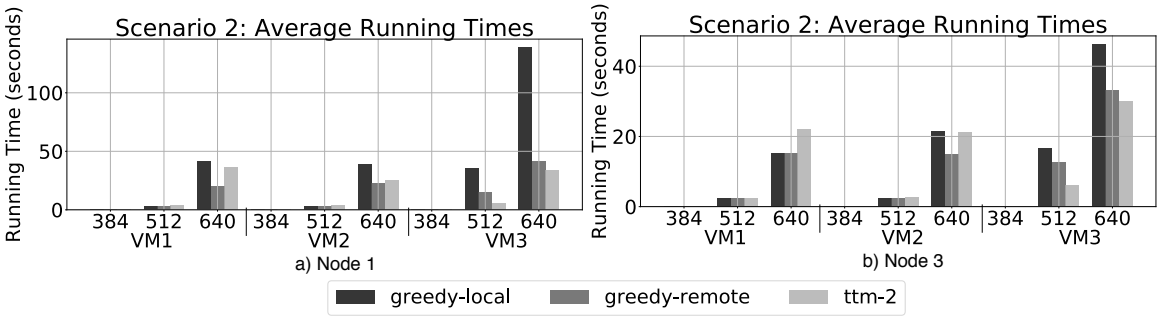


Figure 5.5: Running time for Scn. 2 for nodes 1 and 3.

5.5.2 Results for Scenario 2: the Usemem Scenario

The average running times for Scn. 2 are shown in Figure 5.5. When enabling greedy-remote, VM3, VM2 and VM1 reach an average performance improvement of 63%, 20% and 13% respectively. When TTM is enabled, VM3 shows a maximum and minimum improvement of 27% and 5.5% in node 1, respectively, and a maximum and minimum improvement of 51% and 9.3%, respectively, in node 3. However, VM1 and VM2 both experience a performance loss.

Figure 5.6 shows the remote memory capacity that each VM is using for the three policies. Figure 5.6(a) shows that VM3 struggles to obtain memory pages using *greedy-remote*. This is similar to the case in Figure 5.4(a), in which the VM3 was unable to reach its fair share of Tmem. With TTM, VM3 obtains a larger amount of Tmem, improving its performance. Here, VM3's improvement comes at the expense of VM1 and VM2, balancing the Tmem pages every VM can have.

5.5.3 Results for Scenario 3

The average running times for Scn. 3 are shown in Figure 5.7. In this case, node 1 improves by a maximum and a minimum of 92.3% and 92.1%, respectively, when comparing greedy-local to greedy-remote. When enabling TTM, it improves by a maximum and a minimum of 6.0% and 0.9%.

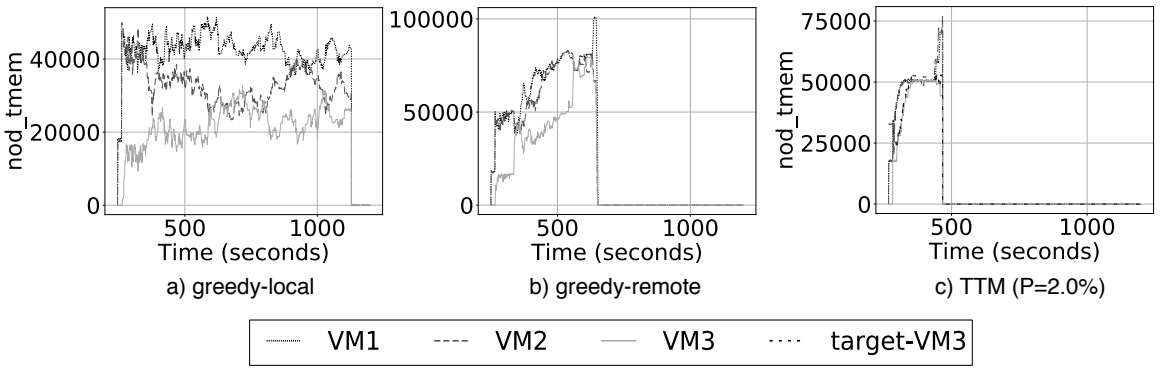


Figure 5.6: GV-Tmem: Tmem capacity (nod-tmem) obtained by every VM in node 3 for Scenario 2.

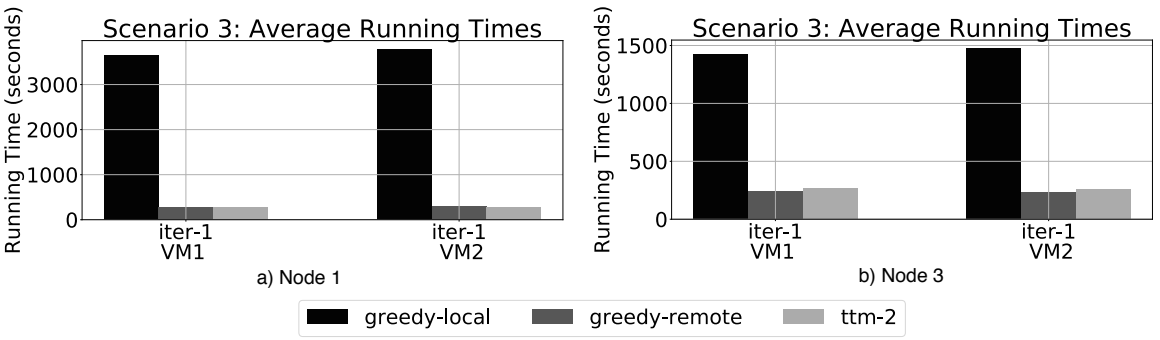


Figure 5.7: Running time for Scn. 3 for nodes 1 and 3.

In node 3, there's a maximum and mininum improvement of 84.4% and 83.1% when comparing greedy-local to greedy-remote. However, performance degrades by 10% with TTM compared to greedy-remote. When enabling TTM, the VMs require more memory but TTM enforces limits, although flexible, on the memory they can take, similar to what occurs in Scn. 2 for VM1 and VM2. When disabling TTM, the VMs take memory unrestrained thus performing slightly better than TTM. This highlights the need for more adaptive memory management policies.

5.6 Related Work

Zcache [68] is a backend that provides a compressed cache for swap and clean filesystem pages. *RAMster* [68] is an extension of zcache that uses kernel sockets to store pages in the RAM of remote nodes. In contrast, GV-Tmem grants and releases blocks of pages at greater granularity, reducing the amount of software communication between nodes, since we exploit the shared global address space. RAMster is implemented in the kernel whereas our approach uses a user-space process in a privileged VM, providing greater flexibility for memory management.

RAMCloud [82] is a POSIX-like filesystem in which all data is stored in DRAM across

nodes, placing the data in one or more nodes, introducing issues of global coherency and exclusivity of access. GV-Tmem aggregates memory exploiting a global shared address space, without requirements for global coherency. Hecatonchire [104] achieves resource aggregation by decoupling virtual resource management from physical resources. It uses a mediation layer that arbitrates how applications access resources. We differ from [104] by making memory available to the hypervisor through a user-space process.

In [12], Bielski et al. propose a mechanism to extend the amount of memory of the VMs without relying on the ACPI standard support. They use a custom driver to allocate memory buffers for the VMs that abstracts the actual memory locations, allowing for memory to be provided by a local or remote node, allowing the VMs to access disaggregated memory transparently. They exploit memory ballooning capabilities and to increase the memory limits beyond the upper-bounds, they employ memory hotplug. Our work differs from [12] mainly in the fact that our mechanism for memory addressing memory remote (or even locally) relies on the shared address space and the additional level of indirection provided by the HV, which means that we don't require a special allocator to address remote memory.

In [40], Gu et al. proposed a mechanism called INFINISWAP that works as a remote memory paging mechanism by introducing a custom block device that is used as swap space. It devides the swap space of each machine into many slabs and distributing them across many machines' remote memory, and it relies on RDMA network for low-latency networking. This approach is somewhat similar to GV-Tmem in which our work accesses remote memory through the block device interface of the kernel, but it leverages Tmem (hypervisor-level caching) instead of implementing a customized block device exposed to the HV.

5.7 Conclusions for this Chapter and Future Work

This chapter introduced GV-Tmem, a method that exploits Tmem to share memory capacity across multiple nodes. We evaluated GV-Tmem using CloudSuite, obtaining up to 51% performance improvement using simple memory management policies. The results demonstrate the effectiveness of GV-Tmem.

Future work will investigate how to integrate GV-Tmem with other resource management mechanisms of other cloud software. It is also necessary to develop more sophisticated two-tier global memory management policies, in order to improve adaptability and responsiveness to changes in memory demand. Aspects of resiliency fault tolerance also need to be addressed and aspects regarding the efficiency and optimal use of the available Tmem capacity. Both of these issues will be addressed on Chapter 6

Chapter 6

vMCA: Memory Capacity Aggregation and Management in Cloud Environments

The problem of aggregating memory across nodes has a lot of implications. Besides exploiting the interconnection infrastructure and the capabilities of Tmem, it is also necessary to address issues related to the status of each node, the potential of memory leaks, the efficient use of Tmem capacity, among other issues.

This chapter presents a memory capacity aggregation mechanism for cloud environments called vMCA (Virtualized Memory Capacity Aggregation) based on Xen's Tmem. This solution is an an extension of GV-Tmem presented in Chapter 5.

vMCA distributes the system's total memory globally across multiple nodes with special considerations to the distrubtion of Tmem within a node, similar to SmarTmem presented in Chapter 2, but extended as well to consider the presence of remote memory.

6.1 Introduction

As explained in Section 5.1, cloud data center environments are built using "share nothing" servers, each provisioned with their own computing resources, communicating over a TCP/IP (or similar) network. New system architectures [70, 24, 49], however, have been proposed that present a shared global physical address space and use a fast interconnect to share physical resources through the memory hierarchy. An example is the UNIMEM (*Unified Memory*) memory model, which is an important feature of the "EuroEXA" family of projects [70, 89]. In such systems, remote memory is addressable at low latency using Remote DMA and/or load/store instructions. Cache coherency is only enforced inside a node (a *coherence island*), which avoids global coherence traffic and enables scalability to large numbers of nodes.

GV-Tmem and vMCA are developed considering this same framework related to the hardware architectures of the node. Both of these contributions are designed using Tmem in order to exploit two key capabilities of UNIMEM-like architectures: the shared global (across the cloud infrastructure) physical address space and the fast interconnects. In the contributions presented in this and Chapter 5, we have added the virtualization layer (the HV, basically) to leverage these hardware capabilities in a more flexible way, without requiring modifications to the commercially available OSs and runtime systems. In this way, the new hardware architectures can be used and different OSs can be deployed in a transparent way to the application and kernel developers. However, there are some important differences between GV-Tmem and vMCA.

This chapter describes vMCA (virtualized Memory Capacity Aggregation), an extension of Xen's Tmem that leverages virtualization software and a global address space to allow the whole system's memory capacity to be shared among nodes. vMCA extends and improves upon an earlier proposal known as GV–Tmem [36], in order to handle resiliency, and with a deeper discussion and improvements of the memory aggregation and allocation policies. Like GV–Tmem, vMCA introduces minimal changes to the hypervisor to enforce constraints on local and remote page allocation. The complexity is situated in a user-space Memory Manager (MM) process that runs in the privileged domain of the nodes, which implements a resilient and reliable mechanism for distributing memory capacity across nodes according to a high-level memory management policy. The main contributions of this chapter are:

1. An extended software stack to aggregate memory capacity across multiple nodes in a resilient and efficient way.

2. An extension of the multi-level user-space mechanism for allocation/management of aggregated memory capacity developed for GV-Tmem, with special considerations for memory allocation within a node and memory distribution across nodes.

3. A more complete analysis of high-level policies for memory aggregation and allocation.

This chapter is organized as follows. Section 6.2 presents vMCA and its components, explaining its resiliency to failures and the necessary hardware support. Section 6.3 describes the experimental methodology and Section 6.4 shows the evaluation results. Section 6.5 compares with related work. Finally, Section 6.6 outlines future work and provides some conclusions.

6.2 Design of vMCA

The vMCA stack consists of three components:

- Hypervisor support for vMCA (Sect. 6.2.1)
- Tmem Kernel Module (TKM) in Dom0 for vMCA (Sect. 6.2.2)
- Memory Manager (MM) in Dom0 for vMCA (Sect. 6.2.3)

6.2.1 Hypervisor Support for vMCA

The Xen HV has been extended to support vMCA, and all extensions are localized in the Tmem subsystem, which originally supports the necessary features to allocate and deallocate local Tmem pages on behalf of the VMs.

Page ownership

Each HV *owns* a subset of the physical pages in the system, which constitutes the pool of memory that it can use and allocate to its VMs. When a HV boots and first joins the vMCA system, it owns all of its *local* physical memory; i.e. all pages whose *home* node is that node. If it needs additional memory, it may request ownership of additional *remote* memory pages. Conversely, it may release ownership of some of its local or remote pages, so that they can be granted to another node that needs them. As described in Sect. 6.2.3, the MMs collectively ensure that each physical page in the system is owned by at most one HV. The only times when a page is not owned by any HV are when the page is in transit between HVs, and, rarely, following a memory leak caused by a failed node (Sect. 6.2.3).

Page allocation

The pages owned by a HV are allocated using a zoned Buddy allocator, with a zone for each node, including itself, from which it has ownership of at least one page, similar to the case of GV-Tmem.

One improvement done in vMCA done with respect to GV-Tmem is in the way the allocators are configured and ordered in the HV in order to optimize the allocation of pages. In vMCA, the allocators are designed and arranged in a tree data structure corresponding to the hierarchy of the system, in which the nodes are at the leaves, then, e.g. boards, chasses and racks. This design is thought to somewhat optimize the accesses by ensuring that remote pages are taken from the closest node possible when a Tmem put occurs.

In this scheme, a Tmem put will traverse the tree to find the closest non-empty allocator, and take a page from it. In this arrangement, the preferred nodes (the closest nodes) is one that has a common parent with the node that issues the put. At this point, it is assumed that the local HV has ownership of the page. This implies that every node in the system implements a similar tree structure. When a tmem flush operation frees a page, returning it to the corresponding Buddy allocator.

Transfer of page ownership

In order to minimize communication overheads, page ownership is transferred with hypercalls in *blocks* of addresses, each an appropriately-aligned power-of-two number of pages. The HV supports four hypercalls in order to transfer page ownership. The Grant hypercall is used to receive ownership of a list of blocks, which are added to the appropriate Buddy allocator. In contrast, a Request hypercall requests that the HV releases ownership of a number of pages on behalf of another node. In response, the HV takes the necessary blocks from its allocators using a simple heuristic (see **Page allocation**) to prefer large blocks located physically close to the target node. The Return hypercall is issued when another node wishes to shut down or leave vMCA. It asks the HV to release ownership of all pages that are physically located on that node. It returns free pages, and, executing asynchronously, searches for pages in use by the Tmem clients and migrates their contents to free pages. Finally, the Invalidate-Xen hypercall is used if a remote node fails: it discards all free or allocated pages located in that node, and terminates all VMs that were using that data.

Enforcing local per-VM Tmem constraints

The HV enforces the Tmem allocations determined by the MM. Algorithm 7 illustrates this mechanism. The DO_TMEM_PUT function is called when a VM attempts to store a page in Tmem through the corresponding hypercal. The HV checks the amount of Tmem the VM has (line 4), and returns $-ENOMEM$, denying the request, if the VM is already at the limits of its allocation. If not (lines 9–11), the Tmem page will be given to the VM and the necessary data structures are updated.

The mechanism to enforce the Tmem allocations of each VM as defined by the high-level policies in concept is very similar to the one used in Algorithm 1 developed for SmarTmem, but the one developed for vMCA has important design enhancements. Mainly, the allocation mechanism is more complex for the case of vMCA (details omitted in Algorithm 7 for simplicity), as well as an additional mechanism to keep track of the location of the remote memory in use by each node. Even though this information is not visible to the MM, it is highly critical to shutdown a node in a control way and for the resiliency mechanism that allow the node and memory to be restored in case it fails. The details of this will be explain in Sections 6.2.3 and 6.2.3.

Tmem Statistics

Table 6.1 presents the most relevant Tmem statistics gathered by the HV and kept by the MM. The HVs send statistics to their respective MMs every second, and their size is minimized to prevent large communication overheads.

It is possible to see that Table 6.1 presents some similarities with Table 2.1 for the SmarTmem design. However, there are a more fields that needed to be added to implement vMCA, and we enumerate some of the most important modifications:

1. In the case of vMCA, we have used put rates instead of number of puts
2. We have added additional fields in the information gathered by the HV that allows to identify the node

Algorithm 7 Enforce memory allocation in the HV

```
1: function DO_TMEM_PUT(vm_data_hyp, id)
2:     tmem_used ← vm_data_hyp[id].tmem_used
3:     mm_target ← vm_data_hyp[id].mm_target
4:     if tmem_used ≥ mm_target then
5:         return_value ← −ENOMEM
6:     else if node_info.free_tmem == 0 then
7:         return_value ← −ENOMEM
8:     else
9:         vm_data_hyp[id].tmem_used += 1
10:        vm_data_hyp[id].puts_succ += 1
11:        return_value ← 1
12:    end if
13:    vm_data_hyp[id].puts_total += 1
14:    return return_value
15: end function
```

3. A variable to keep track of the variable amount of total Tmem owned by the node was also added. The update process of this variable is more complex than the one designed for SmarTmem.

6.2.2 Dom0 Tmem Kernel Module (TKM)

The Tmem client interface requires a kernel module in each guest domain. vMCA needs a kernel module in the privileged domain Dom0 that acts as an interface between the HV (using hypercalls) and the node's MM.

6.2.3 Dom0 User-space Memory Manager

Each node has a user-space MM in its privileged domain, Dom0. The MMs cooperate to:

- Distribute memory owned by each node among its VMs
- Distribute global memory capacity among nodes
- Implement the flow of page ownership among nodes
- Enable nodes to join/leave vMCA, and handle failures

In the current design, one of the MMs is designated to be the Memory Manager Master (MM–M), which is responsible for system control and global memory capacity distribution. The MMs communicate using a secure and reliable packet transport such as SSL/TLS. The commands passed among the MMs are listed in Table 6.2.

In general, the inter-node messages used to support the functionalities of vMCA are fairly similar to the ones used to support GV-Tmem presented in Table 5.1. However, additional messages have been added to the repertoire of vMCA in order to support the resiliency features offered in this case.

Memory Statistics	Description
$ENOMEM$	Code used in the HV to signify that a *put* failed due to a lack of Tmem capacity
$node\_info$	Data structure that holds general status information of the computer host.
$node\_info.total\_tmem$	Total number of pages available to Tmem (free or allocated). It varies depending on how the specific node gains ownership or relinquish ownership of Tmem pages.
$node\_info.id$	Identifier of the node inside vMCA-supported system
vm_data_{hyp}	Data structure that holds the parameters of all of the VMs within the HV
$vm\_data_{\text{hyp}}[id].vm\_id$	Identifier of the VM within Xen
$vm\_data_{\text{hyp}}[id].tmem\_used$	Number of Tmem pages currently used by the VM
$vm\_data_{\text{hyp}}[id].mm\_target$	Target number of pages for the VM, held by the HV and previously sent by the MM
$vm\_data_{\text{hyp}}[id].puts\_total\_rate$	Total number of *puts* issued by the VM in the most recent period
$vm\_data_{\text{hyp}}[id].puts\_succ\_rate$	Total number of successful *puts* issued by the VM in the most recent period
$memstats$	Data structure that holds the last sampled statistics that the HV sent to the MM.
$memstats.vm\_count$	Amount of active VMs as seen by the MM.
$memstats.vm$	Array where each entry holds statistics about an active VM
$memstats.vm[i].vm\_id$	Identifier of the VM within the MM, as received from the HV
$memstats.vm[i].puts\_total\_rate$	Total number of *puts* that a VM has issued to the HV in the recent period
$memstats.vm[i].puts\_succ\_rate$	Total number of successful *puts* that a VM has issued to the HV in the recent period
$memstats.total\_alloc\_pages$	Total number of pages available in every zoned buddy allocator of the HV
mm_out	Pointer to a data structure that holds the output parameters of the MM policy
$mm\_out[i].vm\_id$	VM identifier that maps a VM to its target allocation as calculated by the MM
$mm\_out[i].mm\_target$	Memory allocation target as calculated by the policy in the MM

Table 6.1: Summary of memory statistics used in the MM for vMCA

Joining the vMCA system

In order to join vMCA, a node requires a configuration file, which provides the network address of all the nodes, security credentials for secure connections, and the mapping for all nodes from node ID to network address. For managing locality, it also needs to know the location of each node in the NUMA hierarchy, so that this information can also be passed to the allocators and the mechanisms to keep track of the locations of used Tmem in the HV.

When a node R wishes to join vMCA, it first sends a message with a Register command to the MM–M (Table 6.2). The MM–M sets its state to Active (Figure 6.1) and sends an Enable-Node(R,1) command to all the registered nodes. Each node maintains a bitmap of the enabled nodes, in order to keep track of which nodes are active. When a node receives such a message, it checks whether it already has ownership of any pages with from node R, which would be a fatal error. This mechanism is necessary to prevent data reads to nodes that just entered the system, which would read garbage data that could compromise the functionality of the node performing the reads.

Distributing ownership of memory

The flow of memory pages across vMCA considers many aspects.

Command	Direction	Description	Slave state
Distribution of global memory capacity			
Statistics(S)	S→M	Send node statistics S to Master	Active
Grant-Any(n, x)	S→M	Request n pages to slave x	Active
Grant-Fwd(n, x, y)	M→S	Forward request of n pages coming from y to slave x	Active
Force-Return(n, x)	M→S	Disable node and return all pages located at slave x (for leaving)	Active
Flow of page ownership			
Grant($b, \cdots$)	S/M→S	Transfer ownership of blocks of pages	Active
Node state changes			
Register	S→M	Register a new node	Inactive→Active
Leave-Req	S→M	Node requests to leave the system (e.g. for shutdown)	Active→Leaving
Leave-Notify	M→S	MM–M notifies that the recipient has left the system	Leaving→Inactive
Enable-Node(x, e)	M→S	Inform slave to accept ($e = 1$) or reject ($e = 0$) pages at node x	Active
Resiliency support			
Invalidate	S→M	Invalidate all pages located at sending node	Active/Leaving→Recovery
Invalidate(x)	M→S	Request recipient to invalidate pages at node x	Active
Invalidate-Notify(x)	M→S	MM–M notifies that the recipient has been invalidated	Recovery→Inactive

Table 6.2: MM message types. In the table, "S" stands for Slave and "M" for Master.

Distributing memory owned by a node among its local VMs: Each node's MM determines the maximum Tmem allocation for each VM, using a policy (Sect. 6.2.4) that uses the Tmem statistics gathered by the HV. Pages are dynamically allocated to every VM subject to total (local plus remote) consumption limits, which are sent to the HV using a Mem-Limit hypercall.

For every policy, the following condition must be met:

$$\sum_{i=1}^{i=m} vm\_data_{\mathrm{MM}}[i].mm\_target = total\_tmem(t_i) \tag{6.1}$$

Eq. 6.1 means that at time t_i, the amount of total Tmem pages available to the node ($total\_tmem = local\_tmem + remote\_tmem$) is equal to the sum of the allocations of all m active VMs. This implies that every Tmem page must be used, and that Tmem pages are not overallocated. However, when calculating the allocation limits for every VM, it is possible that the sum of the allocations exceeds $total\_tmem(t_i)$. Representing $total\_tmem$ as a function of time t_i implies that the amount of total Tmem of a node changes as it requests or gives away ownership of Tmem pages. This is a fundamental difference when compared to SmarTmem, that only considers constant local Tmem, a difference that has important implications on the design of the software stack (hypercalls to support, data that needs to be stored, allocation mechanisms, etc).

Overallocation of pages causes VMs to compete for the available Tmem capacity, defeating the purpose of the management policies. When over-allocation occurs, the MM recalculates the target of the VMs as follows:

$$tgt_{vm_i} = \frac{total\_tmem \times vm\_data_{\mathrm{MM}}[i].mm\_target}{\sum_{i=1}^{i=m} vm\_data_{\mathrm{MM}}[i].mm\_target} \tag{6.2}$$

Eq. 6.2, where tgt_{vm_i} is an abbreviated form of $vm\_data_{\mathrm{MM}}[i].mm\_target$ shown in Table 6.1, ensures that the proportional allocation of each VM follows the policy's desired proportion while also satisfying Eq. 6.1.

Distributing global memory capacity among nodes: All nodes in the Active state send statistics to the MM–M using the Statistics command. A node that needs memory sends a Grant-Any command to the MM–M, which then decides which of the nodes in the vMCA realm is best suited to satisfy the request for memory of the node. The MM–M makes this decision based on the amount of available memory the possible candidate node has.

In this scheme, the MM–M redistributes the memory based on the statistics and the global memory policy. The MM–M then sends Grant-Fwd commands, forwarding requests to a donor node to transfer ownership of a number of free physical pages to a requesting node.

Flow of page ownership: Conceptually, the flow of pages ownership looks identical both in GV-Tmem and vMCA. The MM–M rebalances memory capacity without specific knowledge of the physical addresses, but the HV in vMCA needs to keep track of which nodes are borrowing pages from the local node and from which remote nodes the local node is borrowing pages from.

Ownership of global physical addresses is transferred peer-to-peer using Grant commands, after the MM–M has allowed for the remote node to give page ownership to the node that has requested the memory. A Grant command passes a list of blocks, each an appropriately-aligned power-of-two number of physical pages. The MM–M can also satisfy the request for memory of another node, for which it will send a Grant command itself to the requesting node.

To avoid race conditions during node shutdown/failure, the recipient checks the home node of each received block against the bitmap of enabled nodes (Sect. 6.2.3). If the home node of a block is disabled, then ownership is returned back to its home using a new point-to-point Grant.

Leaving the vMCA system

Shutting down a node of vMCA requires the following procedure, which has a similar procedure to GV-Tmem, but with one additional step that checks that the page ownership of the pages of the node that wants to shut down have been returned to it.. Note that if a node fails to retrieve the ownership of its pages or it shuts down in an unexpected way, then the procedure in Sect. 6.2.3 is followed. We reproduce the process for node shutdown in vMCA similar to the one for GV-Tmem, with the additional step(s):

1. The node R that wishes to shutdown sends Leave-Req to the MM–M.

2. The MM–M in response moves the node to the Leaving state (see Figure 6.1). It sends a Force-Return(R) command to all nodes in the system. Each recipient returns all of

the pages at R that it owns and will reject any such pages received in future Grant commands.

3. Node R frees all pages used by Tmem and returns ownership of all remote pages to their home nodes using peer-to-peer Grant messages.

4. Each node sends a Grant command to node R to return ownership of the pages that it borrowed.

5. The MM–M is disabled once it has received Statistics from every node indicating that R and that it owns no pages at R, then the MM–M moves R to Inactive (Figure 6.1) and sends Leave-Notify to R.

6. Node R at this point check that it has ownership of all its local non-leaked pages, officially leaves the vMCA system, and may proceed to shutdown.

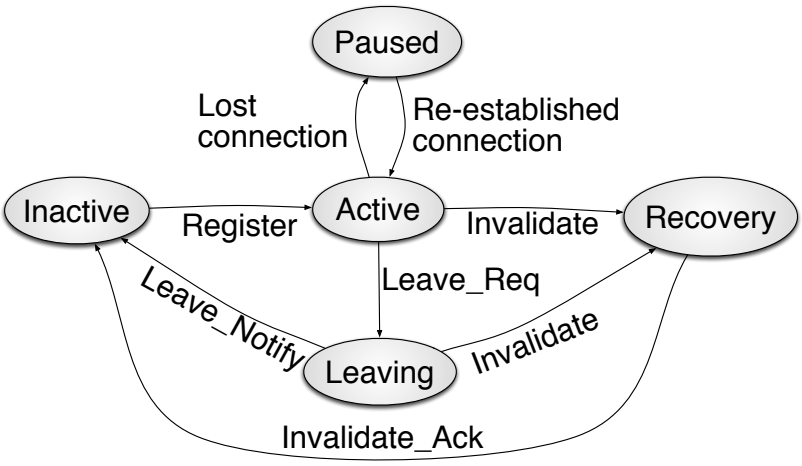


Figure 6.1: Node state transition diagram (at MM–M)

Resilient memory capacity aggregation

There are two aspects to ensuring resiliency in the face of node failures: 1) protecting the integrity of the data stored in Tmem, and 2) restoring lost system state.

Ensuring data integrity on slave failure: The Tmem interface guarantees that a get returns the data previously written by the corresponding put. In addition, if the pool is *persistent*, the get operation must succeed.

Assume that node A has ownership of memory located at node R. If R fails, then this data is lost. If R reboots, it will restart and may even attempt to re-join vMCA. Since any get performed at A must never access the (incorrect) new contents of the physical page at node R, the following procedure must be followed after a failure:

1. When a node R boots, remote memory accesses to it must raise a hardware exception (Sect. 6.2.5), also raised when a node accesses a page from a failed node. If a remote

access causes such an exception, the VM must be shutdown if the page is in a *persistent* pool. If the pool is *non-persistent*, the get fails. On any failed access, read or write, the page is returned to the corresponding Buddy allocator, and it is disabled.

2. When node R is initialized without a previous clean shutdown, it sends Invalidate to the MM–M.

3. The MM–M moves the node to the Recovery state (Figure 6.1) and sends Invalidate(R) to all nodes. Each node then disables node R, and if it has ownership of pages with home R, it makes an Invalidate-Xen hypercall to invalidate the pages.

4. Once the MM–M has received Statistics from every node, i.e. it has disabled R and owns no pages at R, then the MM–M moves the node to the Inactive state and sends an Invalidate-Notify command to R.

5. Node R begins its normal initialization procedure upon receiving the Invalidate-Notify command.

Restoring lost pages on slave failure: After a node R fails, vMCA is unable to recover the pages that it owned. It is possible to approximate the number of lost pages from a home node using the statistics sent by the nodes to the MM–M, by adding the number of pages owned and comparing against the number of physical pages at the node. However, since this is done using potentially outdated information from Statistics commands, the value will not be exact.

Restoring system state on MM–M failure: If the MM–M fails, the system will continue to work except that a) no memory capacity will be redistributed, and b) nodes can't enter/leave the system.

When the MM–M attempts to restart, it will listen for connections from the other nodes. When a node connects and starts sending Statistics, it is added to the system in the Active state. A node that is in the process of leaving or invalidation will send a new Leave-Req or Invalidate command on re-connecting to the MM–M, which will restart the leaving or invalidation process. At any time, all nodes in the Active state are part of the global memory allocation.

6.2.4 Memory Management Policies

We tested three memory management policies: 1) greedy-local [36] (single-node Tmem implementation), 2) greedy-remote [36] and a 3) two-level memory management policy (TLP), divided in a first and a second level of memory management, implemented for this work. greedy-remote, like greedy-local, allows for the node's HV to give Tmem pages away to its VMs on demand without any constraints. In contrast to greedy-local, greedy-remote issues a Grant-Any to the MM–M requesting 1000 of pages when the MM detects that the node needs memory.

Algorithm 8 shows the first level of *TLP*. It uses the statistics provided by the HV (*memstats*, in Table 6.1), checks the number of VMs running (line 5) and if the VMs have failed *puts* since receiving the previous statistics set (lines 6–8). Initially, all VMs have an equal portion of the node's available Tmem. If $put\_fail\_rate > 0$, the MM increases the

Algorithm 8 First-Level of TLP

```
1: function FLM_POLICY(memstats, node_info, P, threshold)
2:     ttmem ← memstats.total_alloc_pages
3:     node_id ← node_info.id
4:     sumtgt ← 0
5:     for i ← 1, memstats.vm_count do
6:         ptr ← memstats.vm[i].puts_total_rate
7:         psr ← memstats.vm[i].puts_succ_rate
8:         puts_fail_rate ← ptr − psr
9:         if puts_fail_rate > 0 then
10:            ctgt ← memstats.vm[i].mm_target
11:            mm_target ← ctgt + (P ∗ ttmem)/100
12:        else
13:            ctgt ← memstats.vm[i].mm_target
14:            curr_use ← memstats.vm[i].tmem_used
15:            difference ← ctgt − curr_use
16:            if difference > threshold then
17:                mm_target ← ((100 − P) ∗ ctgt) /100
18:            else
19:                mm_target ← ctgt
20:            end if
21:        end if
22:        mm_out[i].vm_id ← memstats.vm[i].vm_id
23:        mm_out[i].mm_target ← mm_target
24:        sumtgt ← sumtgt + mm_target
25:    end for
26:    if sumtgt > ttmem then
27:        send_msg(Grant-Any(node_id, (sumtgt − ttmem)))
28:        for i ← 1, memstats.vm_count do
29:            nt ← (ttmem/sumtgt) ∗ mm_out[i].mm_target
30:            mm_out[i].mm_target ← nt
31:        end for
32:    end if
33:    send_hypercall(Mem-Limit(mm_out))
34:    send_msg(Statistics(memstats) )
35: end function
```

allocation of the VM by P% of the total amount of Tmem pages available to the node (lines 9–11).

If $put\_fail\_rate = 0$, the VM allocation is reduced by P% assuming that it currently has more pages allocated than its actual use by a $threshold$ ($difference > threshold$). In case not, then it keeps the current allocation (lines 12–19).

The MM then sums and assigns the target allocations to the corresponding data structures (lines 20–22). If the sum exceeds the amount of Tmem available, the MM does the following: 1) send a Grant-Any command to the MM–M to request memory pages (line 24), and 2) readjust the VMs' allocations according to Eq. 6.2 (lines 25–27). The MM sends the Grant-Any command because the node is under memory pressure, so it needs remote pages. There's no guarantee it will get them, thus it's necessary to readjust the allocations until more pages are available. After this, the MM issues the Mem-Limit hypercall (line 28) and

the Statistics message to the MM–M if the node is a slave (line 29).

Upon receiving the Grant-Any request, the MM–M decides how many pages to actually request and from which node(s) to get them from. This is the second-level of TLP shown in Algorithm 9. The MM–M has an extended version of *memstats*, where it stores its own statistics and from other slave nodes, that it gets through Statistics messages.

Algorithm 9 Second-level of TLP

```
1: function SLM(memstats, GrAny, R, np ← {})
2:     for i ← 1, memstats.node_count do
3:         np[i].id ← memstats[i].node_id
4:         np[i].tmem ← memstats[i].total_tmem
5:     end for
6:     sort_nodes(np)
7:     for i ← 1, memstats.node_count ∧ GrAny.n > 0 do
8:         diff ← np[i].tmem − GrAny.n
9:         if diff > R then
10:            nr ← GrAny.n
11:            send_msg(Grant-Fwd(nr, np[i].id, GrAny.x))
12:            GrAny.n ← 0
13:        else if diff < R ∧ np[i].tmem ≥ R then
14:            nr ← np[i].tmem − R
15:            send_msg(Grant-Fwd(nr, np[i].id, GrAny.x))
16:            GrAny.n ← GrAny.n − nr
17:        end if
18:    end for
19: end function
```

Algorithm 9 uses a data structure (*np*) that stores the amount of Tmem pages available in each node and their IDs (lines 3–4). Then, it sorts the nodes (using merge sort) in descending order in *np* (line 5). This is to decide which node to forward the request to (lines 6–15), usually the one with more memory available. The MM–M first checks that the potential donor node will have enough memory for itself (above a threshold R) . If the node is able to satisfy the request while meeting the threshold, then the request is forwarded to it using a Grant-Fwd message (lines 7–11).

If the node is unable to satisfy the request but has memory available, the MM–M can still forward the request (line 14) but requesting less pages (line 13) to meet the threshold of the node. Then, a new node from *np* is selected to request the rest of the remaining pages. From this process, it's clear that there's no guarantee that requests for pages will be satisfied.

6.2.5 Hardware Support for Memory Aggregation

vMCA requires the computing node to provide support for specific hardware features. vMCA and GV-Tmem require similar hardware support and they have a similar high-level view when both systems are deployed, similar to what is shown in Figure 5.2 in Chapter 5. But vMCA requires additional features to support the new resiliency constraints. We enumerate all of the necessary hardware features necessary for vMCA in their entirety for convenience:

1. Fast interconnect providing a synchronous interface to the NUMA distributed memory.

Node	CPU	Frequency	Memory
Node 1	AMD FX Quad-Core	1.4 GHz	8 GB
Node 2	Intel Core i7	2.10 GHz	16 GB
Node 3	Intel Xeon	2.262 GHz	64 GB
Node 4	AMD FX-4300 Quad-Core	3.8 GHz	8 GB

Table 6.3: Hardware characteristics of the nodes used to evaluate vMCA.

2. Direct memory access from the HV to all the memory available: the HV has to be able to access transparently its local and remote memory through the same mechanisms, either through load/store instructions and/or through RDMA.

3. Remote access to a node's pages is disabled on hardware boot. Access is enabled when the node joins vMCA, when it sends the Register command. A series of hardware control mechanisms, like interrupts and event controls, need to be implemented as well to support the resiliency constraints within vMCA

4. Extraction of the node ID given a physical address, for instance depending on some higher-order bits.

6.3 Experimental Methodology

We evaluated vMCA in a platform consisting of four computing nodes. UNIMEM-based architectures usually have 4 to 8 nodes sharing the global physical address space. Every node and VM runs Ubuntu 14.04 with Linux kernel 3.19.0, and Xen 4.5. The MM of the nodes communicate using TCP/IP sockets over Ethernet. Node 2 executes the MM–M. The hardware properties of the nodes are given in Table 6.3.

Accesses to remote sections of the shared global address space are emulated using the node's local memory, similar as it was done with GV-Tmem. Whenever the hypervisor performs an emulated remote access, we add a delay in the hypervisor lasting 50 µs to model a reasonable worst-case hardware latency. We evaluate vMCA using the CloudSuite 3.0 Benchmarks [31] as well, with every VM having 1 vCPU and every node having 1 GB of Tmem initially available. We run at most three VMs simultaneously with memory intensive applications that generate memory pressure on the node, and also refer to each set of VMs as a *scenario*, shown in Table 6.4. We run each scenario at least four times. As it was with the case of SmarTmem and GV-Tmem in Chapters 2 and 5, the scenarios created here, the configurations of the VMs and the datasets were chosen in order to generate a considerable amount of memory pressure within the VMs, generating memory footprints two or three times of their initial memory capacity [100].

Scenario	VM RAM (MB)	Description
Scenario 1	VM1: 768 , VM2: 1024 , VM3: 1024	Every VM executes simultaneously in-memory analytics once, sleeps 5 seconds and then executes it again. The data set was taken from [43]
Scenario 2	VM1: 768 , VM2: 768 , VM3: 768	Every VM executes simultaneously graph analytics once. The data set was taken from [16, 92, 91], chosen to generate enoug memory pressure
Scenario 3	VM1: 768 , VM2: 768 , VM3: 1024	VM1 and VM2 execute graph analytics while VM3 executes in-memory analytics, all at once. Every VM executes its benchmarks twice.
Scenario 4	VM1: 768 , VM2: 512 , VM3: 512	VM1 executes graphs analytics, while VM2 and VM3 execute the client and server (memcached [32]) for data-caching, respectively.

Table 6.4: List of scenarios used to evaluate vMCA.

6.4 Results

6.4.1 Results for Scenario 1

Figures 6.2(a, b) show the average running times of each VM for Scenario 1 for all policies, showing average improvements of 11.94% and 12.9% in nodes 3 and 4, respectively, when going from greedy-local to greedy-remote. TLP shows an improvement of 7.3% ($P = 2.0\%$) and 7.0% ($P = 6.0\%$) in nodes 3 and 4 over greedy-remote.

Figures 6.2(c, d, e) present the Tmem capacity (remote and local) that each VM takes for all policies in node 3. For greedy-local and greedy-remote (Figures 6.2(c, d)), VM3 and VM4 take a smaller proportion of Tmem when compared to VM2, but this disparity reduces with TLP, as shown in Figure 6.2(e). TLP tries to be *fair* on the distribution of the Tmem capacity among the VMs. In this case, fairness is defined in the same way as it was in Sections 2.4 and 5.5.

For some values of P, TLP degrades the performance of in-memory analytics. Consider the first iteration of VM1 in node 3 for $P = 1\%$ (tlp-1, Figure 6.2(a)). With TLP, limits are enforced on the Tmem capacity of a VM. When a VM tries to go beyond its initial allocation, the hypervisor prevents it from taking pages until the MM updates its allocation target. But with $P = 1\%$, the targets increase slower than needed, forcing the VM to access the disk device.

With higher P, the targets increase faster. But when P becomes too high, a VM might have excess memory allocated, making it unlikely for other VMs to take a fair share, thus degrading performance. In Figures 6.2(a), the performance improves when increasing P up to 2.0% but degrades again as P keeps increasing. This highlights a trade-off between the adaptability of TLP to adjust to changes in memory demand and to achieve a fair distribution of Tmem, similar as it was seen for the case of SmarTmem.

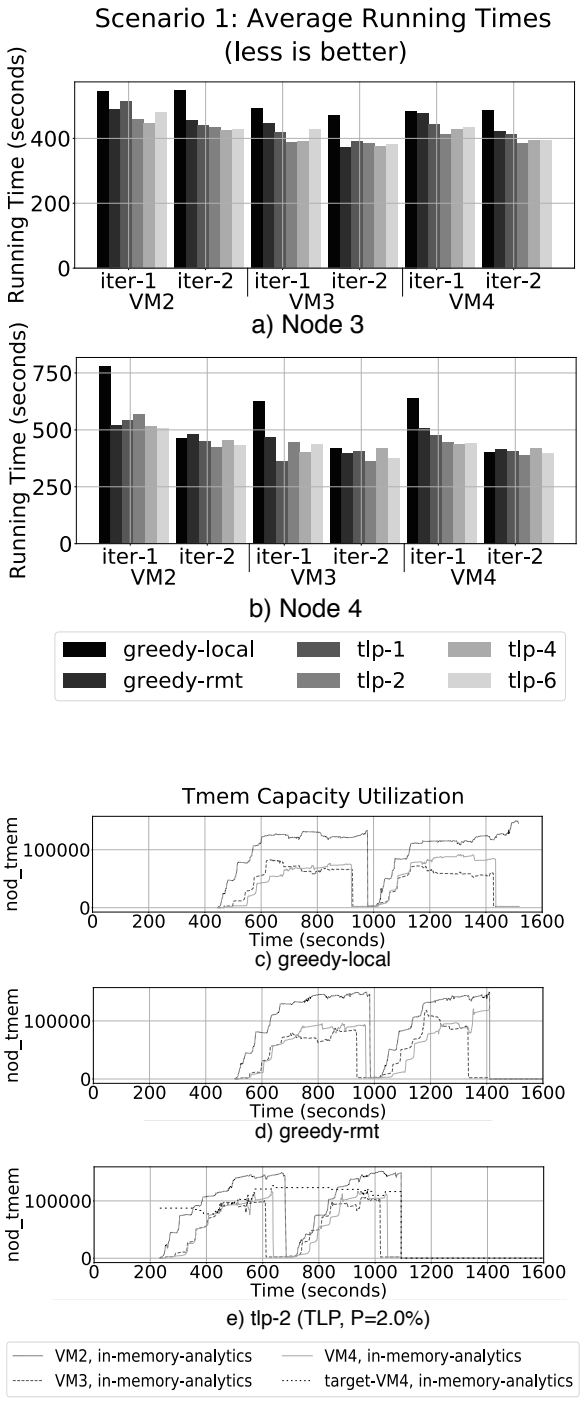


Figure 6.2: Running time for Scenario 1 in nodes 3 (a) and 4 (b). Tmem capacity (nod-tmem) used by every VM in node 3 for Scenario 1 in all three policies (c, d ,e). *target-VM4* refers to the target allocation of VM4.

6.4.2 Results for Scenario 2

Figures 6.3(a, b) present the average running times of each VM for Scenario 2 for the three policies. They show average improvements of 50.84% and 45.96% in nodes 3 and 4, respectively, when going from greedy-local to greedy-remote. With TLP, there is a further improvement of 32.1% ($P = 2.0\%$) and 31.8% ($P = 2.0\%$) in nodes 3 and 4, respectively, over greedy-remote.

Figures 6.3(c, d, e) show the amount of Tmem taken by each VM on node 3, for each policy. When $P = 2.0\%$, the MM enforces *fairness* in the Tmem allocation of the VMs when comparing greedy-local to greedy-remote. The difference in Tmem ownership between VM4 and other VMs is significant for greedy-local but reduces for greedy-remote, because of the remote memory availability. In both cases, the VMs are competing for the Tmem capacity. Figures 6.3(a, b) highlight the *adaptability-vs-fairness* trade-off. As P increases, the performance improvements hit a minimum and then increase. For this scenario, this trend is maintained for large P, making $P = 2.0\%$ an optimal value.

6.4.3 Results for Scenario 3

Figures 6.4(a, b) show the average running times of each VM for Scenario 3 for the three policies. They show average improvements of 50.6% and 55.4% in nodes 3 and 4, respectively, when going from greedy-local to greedy-remote. When implementing TLP, there is a further improvement of 10.1% ($P = 6.0\%$) and 15.6% ($P = 4.0\%$) in nodes 3 and 4 over greedy-remote.

Figures 6.4(c, d, e) show the Tmem capacity taken by each VM for all policies in node 3. As remote memory becomes available to the node, the running time dramatically drops while VM4, running in-memory analytics, takes the longest. The adaptability-vs-fairness tradeoff is observed in Figures 6.4(a, b). Performance improves as P increases, but it can change suddenly with P, as in the second iteration of VM1 and VM2 for node 3 (Figure 6.4(a)). In both nodes, the second iteration of VM4 (in-memory analytics) performs the same for all policies because it runs on its own, while the others have completed (Figure 6.4(c, d, e)). In this Scenario, it is more difficult to obtain an optimal value of P.

After VM2 and VM3 execute graph analytics, they keep a large part of the Tmem pages they acquired. The MM is unable to allocate Tmem pages that are *in use*, until the VMs flushes them. But these VMs never actually flush. If the node runs more VMs with benchmarks that keep pages in this way, the Tmem pages may be inefficiently depleted.

6.4.4 Results for Scenario 4

Figures 6.5(a, b) show the average running times of each VM of Scenario 4 for all policies, showing average improvements of 63.5% and 76.8% in nodes 3 and 4, respectively, when going from greedy-local to greedy-remote. TLP shows improvements of 23.7% ($P = 2.0\%$) and 37.5% ($P = 2.0\%$) in nodes 3 and 4 over greedy-remote.

Figures 6.5(c, d, e) show the Tmem capacity of each VM for all policies in node 3. In all cases, VM4 (memcached server) uses the same amount of Tmem, while VM2 (graph analytics) and VM3 (data-caching client) compete for the Tmem capacity. With TLP (Figure 6.5(e)), fairness is achieved quickly, showing the adaptability-vs-fairness tradeoff in Fig-

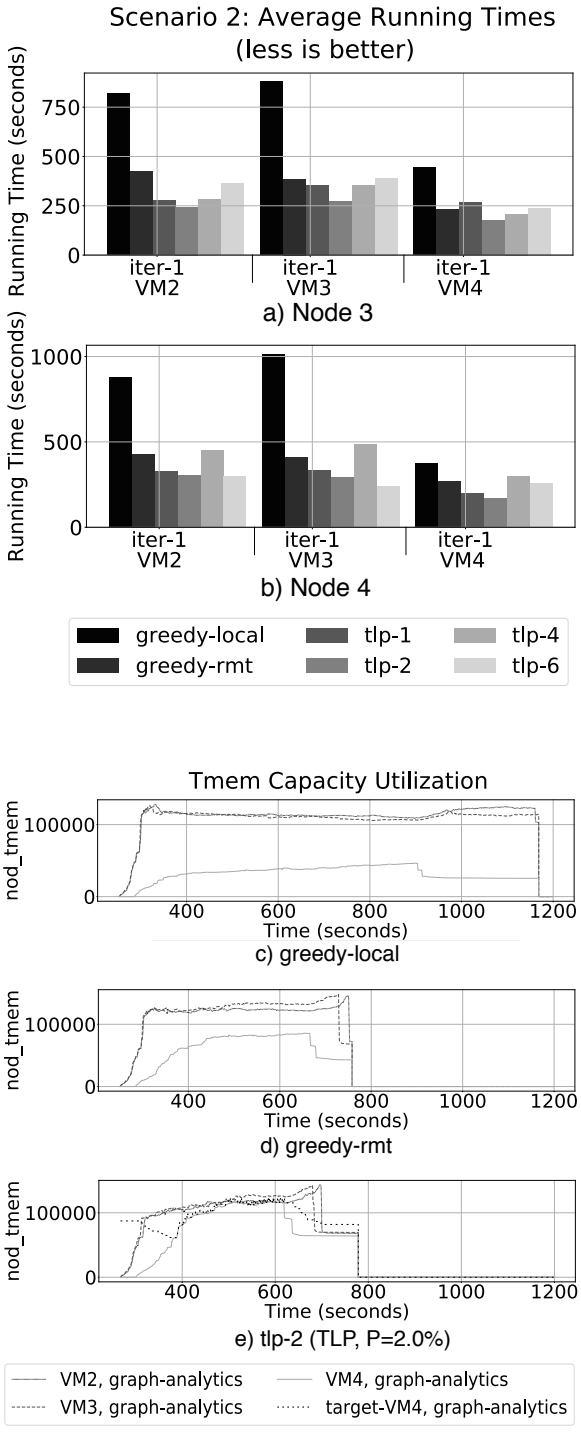


Figure 6.3: Running time for Scenario 2 in nodes 3 (a) and 4 (b). Tmem capacity (nod-tmem) used by every VM in node 3 for Scenario 2 in all three policies (c, d, e). *target-VM4* refers to the target allocation of VM4.

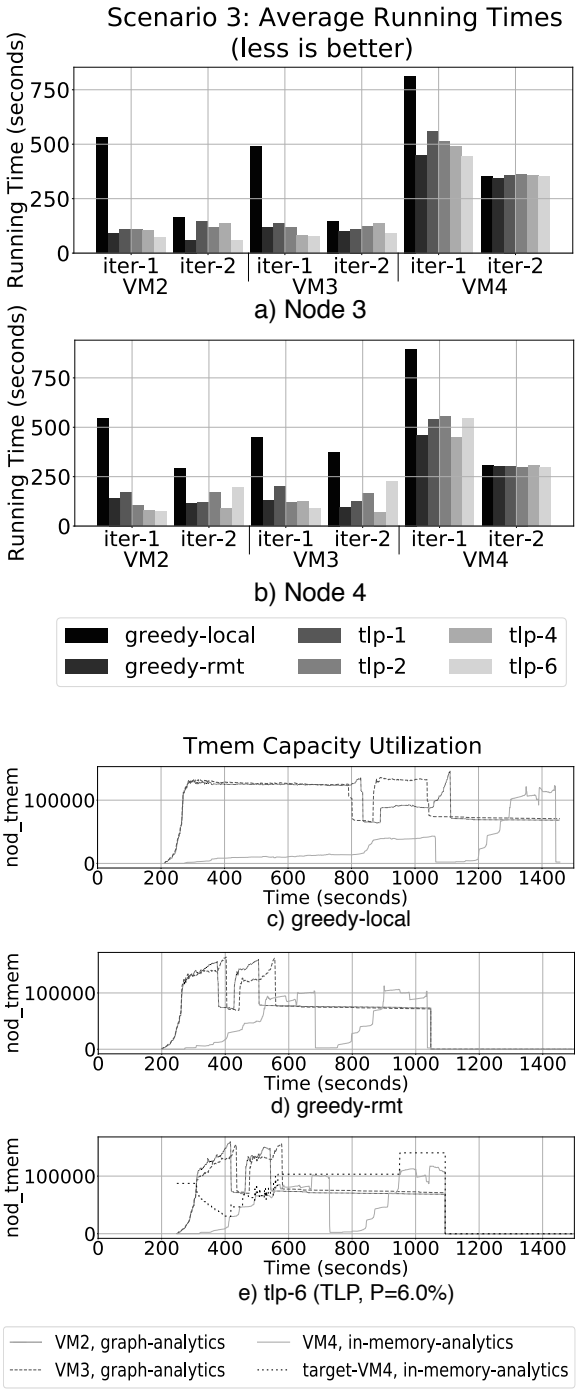


Figure 6.4: Running time for Scenario 3 in nodes 3 (a) and 4 (b). Tmem capacity (nod-tmem) used by every VM in node 3 for Scenario 3 in all three policies (c, d ,e). *target-VM4* refers to the target allocation of VM4.

ures 6.5(a, b). In this case, $P = 2.0\%$ is optimal for most VMs, except for VM3 in node 4, in which $P = 1.0\%$ is better.

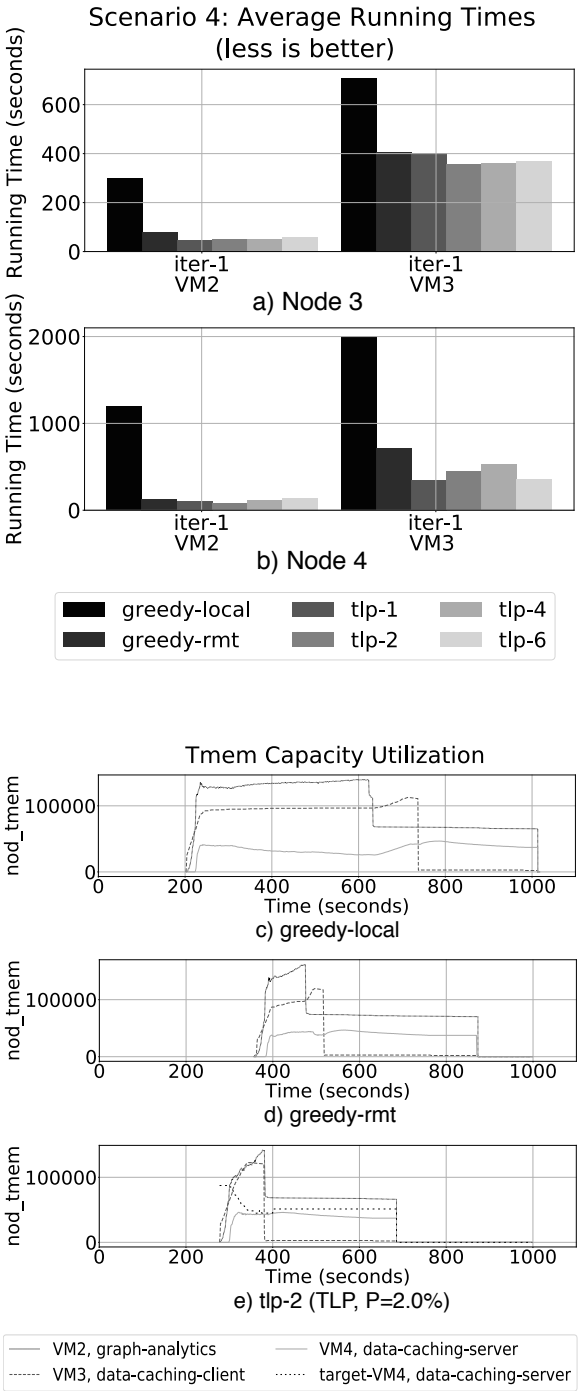


Figure 6.5: vMCA: Running time for Scenario 4 in nodes 3 (a) and 4 (b). Tmem capacity (nod-tmem) used by every VM in node 3 for Scenario 4 in all three policies (c, d, e). *target-VM4* refers to the target allocation of VM4.

6.5 Related Work

Venkatesan et al. [110] use Tmem and non-volatile memory (NVM) to reduce accesses to disk by VMs in a system with DRAM and NVM, showing that Tmem is useful in systems with non-uniform memory hierarchies. The non-uniformity in our case is a consequence of the different memory hierarchies across the nodes, the amount of memory available to a node and the physical location of this memory.

Lorrillere et al. [66] proposed a mechanism, called PUMA, to improve memory utilization based on remote caching to pool VMs memory across a data center for the benefit of VMs having I/O intensive workloads. PUMA uses an interface similar to Tmem to access the cache in a remote server. vMCA differs from [66] because the latter targets clean file-backed pages for I/O intensive applications, while vMCA targets pages generated by the processes of the applications during their computations (not file-backed).

Zcache [68] is a backend for frontswap and cleancache that provides a compressed cache for swap and clean filesystem pages. *RAMster* [68] is an extension of zcache that uses kernel sockets to store pages in the RAM of remote nodes. Similarly, RAMCloud [82] is a POSIX-like filesystem in which all data is stored in RAM across the nodes. vMCA differs from all these approaches in: 1) vMCA grants and releases memory capacity at a higher granularity larger than a single page (reducing communication between nodes), 2) vMCA is not entirely within the kernel, leveraging user-space flexibility, and 3) vMCA exploits a global shared address space with low-latency communication.

Hwang et al. [46] proposed Mortar: a mechanism that pools spare memory on a server and exposes it as a volatile data cache managed by the hypervisor using a similar interface to Tmem, and it can evict objects from the cache in order to reclaim memory for other VMs. They test two use cases, the first using memcached protocol to aggregate free memory across data center, and the second works at the OS-level to cache and prefetch disk blocks. vMCA puts all the memory aggregation and management details on a user-space process, keeping the hypervisor small and secure.

In [5], Aguilera et al. pointed out that one of the challenges for systems that enable remote memory access was the situation in which a remote host crashes or fails, effectively causing a VM to lose access to the memory it was using at the remote location. In this work, we provide a mechanism to recover the memory capacity for the VM that was accessing a remote region in order to prevent memory leakage problems. However, the problem of recovering the actual data that was lost upon remote host failure is out of the scope of this work. As proposed in [5], this could be done by enabling data replication at some level, but this is left for future work.

In [55], Lagar-Cavilla et al. proposed a mechanism to use remote memory that relies on zswap, which is related to a Tmem frontend Zcache in frontswap mode. Their approach is totally software-based, and does not require special hardware support for inter-node communication i.e. their approach is network latency agnostic. In this work, they use the remote memory regions as caches between DRAM and permanent storage, and they encrypt and decrypt the pages that are stored in remote memory, effectively protecting the data in remote memory regions for the cases in which remote hosts get compromised [5]. vMCA is similar to [55] in that it uses remote memory as a cache between DRAM and permanent storage, but differs in some fundamental ways, and we explain some of them.

First, vMCA leverages hardware support in two respects: partially shared global physical

address space and fast communication links among the nodes, which is an important enabler for remote memory capacity support to maintain application performance within specific bounds [33]. Second, vMCA relies on the functionality provided by Tmem to make memory transparent to the VM without any encryption to prevent taking CPU time and reducing application performance. Although, we still believe that securing the data in remote memory is necessary.

6.6 Conclusions and Future Work

This chapter introduced vMCA, a resilient mechanism that exploits Tmem to aggregate memory capacity across multiple nodes, based on thework developed for GV-Tmem [36]. We evaluated vMCA using CloudSuite, obtaining up to 37.5% performance improvement when enabling memory aggregation and management policies. The results demonstrate the effectiveness of vMCA for improving the performance of the evaluated CloudSuite applications.

Future work will investigate how to integrate in vMCA other resource management mechanisms and to create a more complete solution for data centers. We need to consider reclamation of in-use Tmem pages, VM migration and more adaptive and predictive memory management policies.

As it was the case with SmarTmem, future work will also need to analyze its effectiveness with a larger set of workloads for cloud data centers. In this context, it will be necessary to implement ways for vMCA to optimize other performance metrics such as energy consumption, cost of ownership, tail latency and define a more general way to determine fairness quantitatively.

6. VMCA: MEMORY CAPACITY AGGREGATION AND MANAGEMENT IN CLOUD ENVIRONMENTS

Chapter 7

Conclusions

In Chapter 2, it was demonstrated that there the demand-based approach to hand out Tmem pages to VMs in a virtualized node was sub-optimal, and that current state-of-the-art Tmem implementations needed high-level strategies to optimize the available Tmem of a system. SmarTmem provides the first known solution for this, and was able to obtain up to 35% over the state-of-the-art implementations of Tmem, opening the door for the introduction of more Tmem management policies

Chapter 3 introduced a formulation of the dynamic memory re-allocation problem as a de-centralized and continuous-action MDP, and CAVMem proved that a continuous-action RL agent with this formulation can in fact learn to track performance objectives, and that it may be a cost-effective solution when compared to discrete-action RL algorithms (DQL) and to tabular RL approaches (QL).

Chapter 4 builds on the work presented in Chapter 3 to generate CARLEMM. The system dynamics add an important level of complexity when deploying RL based algorithms into the virtualized node, but CARLEMM demonstrated that the continuous-action RL approach with our novel MDP formulation learns to track performance objectives for a limited set of workloads, with limitations regarding its stability.

Chapter 5 presented GV-Tmem, which is the first mechanism to aggregate memory capacity across multiple nodes in a cloud infrastructure. GV-Tmem was designed to exploit the hardware features provided by state-of-the-art UNIMEM-like hardware architectures, which provide support for a partial global physical address space, fast interconnects and local enforcement of memory coherency.

Chapter 6 builds on the work of Chapter 5 to generate vMCA. Similar as it was the case with GV-Tmem, vMCA also builds on the specialized features of UNIMEM-like hardware architectures, and also includes resiliency support and efficiency constraints for the use of remote memory.

To close this conclusion section, the main contributions of this thesis are presented once again:

- A mechanism called SmarTmem for the optimal allocation of Tmem in a single virtualized node.
  - A demonstration that the default way of allocating Tmem at the single node level, used in state-of-the-art hypervisors, is not optimal.

  - Development of a high-level memory management policies to manage Tmem through SmarTmem in a single node.

  Luis Garrido, Rajiv Nishtala, and Paul Carpenter. SmarTmem: Intelligent management of transcendent memory in a virtualized server. In *RADR 2019: 1st Workshop on Resource Arbitration for Dynamic Runtimes*, 2019 [35].

- A mechanism called CAVMem (Continuous Action Agent for Virtualized Memory Management) developed as a proof-of-concept for the use of Continuous-Action Reinforcement Learning for RAM allocation in a *simulation* of a virtualized node

  - Formulation of the RAM management problem as a Markov Decision Process (MDP) in a simulated environment that models memory management constraints.
  - Development of a continuous-action off-policy model-free reinforcement learning algorithm for memory allocation in a simulated environment.
  - Comparison of the CARL algorithm in CAVMem with non-continuous action reinforcement learning algorithms like Q-Learning and Deep Q-Learning in the context of RAM management within the simulated environment

  Luis Garrido, Rajiv Nishtala, and Paul Carpenter. Continuous-Action Reinforcement Learning for Memory Allocation in Virtualized Servers, In *14th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'19), International Conference on High Performance Computing*, 2019 [37].

- A mechanism called CARLEMM (Continuous Action Reinforcement Learning for Memory Management), based on the proof-of-concept provided by CAVMem, but enhanced to solve the memory allocation problem in a *real* computing node.

  - Formulating the RAM management problem as an MDP for a real virtualized computing node.
  - A continuous-action reinforcement algorithm for RAM allocation in a real virtualized node
  - A flexible software stack for the implementation of CARLEMM and performance evaluation.

- A mechanism called GV-Tmem (Globally Visible Tmem) for remote memory capacity sharing across multiple computing nodes in a cloud infrastructure

  - A software architecture to share memory capacity across nodes using Tmem.
  - A two-tier mechanism for allocation and management of dis-aggregated memory.

  Luis A. Garrido and Paul Carpenter. Aggregating and managing memory across computing nodes in cloud environments. In *12th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'17), International Conference on High Performance Computing*, pages 642–652. Springer International Publishing [36].

- A mechanism called vMCA (virtualized Memory Capacity Aggregation) for the sharing of memory capacity across nodes, but enhanced for resiliency and efficiency in comparison to GV-Tmem.

– A software stack to aggregate memory capacity across multiple nodes, focused on resiliency and efficiency.
– A multi-level user-space mechanism for allocation management of aggregated memory capacity, with special considerations for memory allocation within a node and memory distribution across nodes.
– A more complete analysis of high-level policies for memory aggregation and allocation.

Luis A. Garrido and Paul Carpenter. vMCA: Memory capacity aggregation and management in cloud environments. In *IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, December 2017 [34].

Glossary

CARL	Continuous Action Reinforcement Learning
CARLEMM	Continuous Action Reinforcement Learning for Memory Management
CAVMem	Continuous Action Agent for Virtualized Memory Management
CPU	Central Processing Unit
DDPG	Deep Deterministic Policy Gradient
DRAM	Dynamic Random Access Memory
DQL	Deep Q-Learning
GV-Tmem	Globally Visible Tmem
HV	Hypervisor
I/O	Input/Output
MDP	Markov Decision Process
MM	Memory Manager
MM–M	Memory Manager Master
NUMA	Non-Uniform Memory Access
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RL	Reinforcement Learning
SLA	Service Level Agreement
TKM	Tmem Kernel Module
TLP	Two-Level memory management Policy
Tmem	Transcendent Memory
TTM	Two-Tier Memory Management
VM	Virtual Machine
vMCA	virtualized Memory Capacity Aggregation

Bibliography

[1] Size of the public cloud computing services market from 2009 to 2022. https://www.statista.com/statistics/273818/global-revenue-generated-with-cloud-computing-since-2009. ix, 2

[2] Xenstored. https://wiki.xen.org/wiki/XenStore. Accessed: 2016-06-06. 82

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association. 51

[4] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 41–52, New York, NY, USA, 2014. ACM. 13

[5] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 121–127, New York, NY, USA, 2017. ACM. 14, 15, 126

[6] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 120–126, New York, NY, USA, 2019. ACM. 14, 15

[7] B. W. Arden, B. A. Galler, T. C. O'Brien, and F. H. Westervelt. Program and addressing structure in a time-sharing environment. *J. ACM*, 13(1):1–16, January 1966. 5

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. 22

[9] Yonathan Bard. An analytic model of cp-67 - vm/370. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 170–176, New York, NY, USA, 1973. ACM. 5

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, December 2003. 6, 95

[11] Enrico Bibbona, Gianna Panfilo, and Patrizia Tavella. The ornstein–uhlenbeck process as a model of a low pass filtered white noise. *Metrologia*, 45(6):117, 2008. 51, 75

[12] Maciej Bielski, Alvise Rigo, and Renaud Pacalet. Dynamic guest memory resizing - paravirtualized approach. *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 181–186, 2019. 12, 106

[13] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Trans. Parallel Distrib. Syst.*, 24(4):681–690, April 2013. 12, 13, 66, 67, 90

[14] Yue Cheng, Zheng Chai, and Ali Anwar. Characterizing co-located datacenter workloads: An alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, pages 12:1–12:3, New York, NY, USA, 2018. ACM. 14

[15] Jui-Hao Chiang, Han-Lin Li, and Tzi cker Chiueh. Working set-based physical memory ballooning. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 95–99, San Jose, CA, 2013. USENIX. 12, 36, 66, 70, 95

[16] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1082–1090, New York, NY, USA, 2011. ACM. 120

[17] Tom Clark. *Storage Virtualization: Technologies for Simplifying Data Storage and Management*. Addison-Wesley Professional, 2005. 5

[18] Cloudstack. https://cloudstack.apache.org/. 22

[19] Maxime C. Cohen, Philipp Keller, Vahab Mirrokni, and Morteza Zadimoghadddam. Overcommitment in cloud services bin packing with chance constraints. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '17 Abstracts, pages 7–7, New York, NY, USA, 2017. ACM. 13

[20] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 185–196, New York, NY, USA, 1965. ACM. 5

[21] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 153–167, New York, NY, USA, 2017. ACM. 13, 14

[22] Antonio Cuomo, Giuseppe Di Modica, Salvatore Distefano, Antonio Puliafito, Massimiliano Rak, Orazio Tomarchio, Salvatore Venticinque, and Villano Umberto. An sla-based broker for cloud infrastructures. *JOURNAL OF GRID COMPUTING*, 11:1–25, 03 2012. ii

[23] Todd Deshane, Zachary Shepherd, Jeanna Matthews, Muli Ben-Yehuda, and Balaji Rao. Quantitative comparison of xen and kvm. 01 2008. 6

[24] Jianbo Dong, Rui Hou, Michael Huang, Tao Jiang, Boyan Zhao, Sally A McKee, Haibin Wang, Xiaosong Cui, and Lixin Zhang. Venice: Exploring server architectures for effective resource sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 507–518. IEEE, 2016. 95, 108

[25] Gabriel Dulac-Arnold, Richard Evans, Peter Sunehag, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *CoRR*, abs/1512.07679, 2015. 46

[26] Yves Durand, Paul M Carpenter, Stefano Adami, Angelos Bilas, Denis Dutoit, Alexis Farcy, Georgi Gaydadjiev, John Goodacre, Manolis Katevenis, Manolis Marazakis, Emil Matus, Iakovos Mavroidis, and John Thomson. Euroserver: Energy Efficient Node For European Micro-servers. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 206–213. IEEE, 2014. 4, 14, 15, 17, 94, 95, 100, 102

[27] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: towards a fully automated workflow. In *7th International Conference on Autonomic and Autonomous Systems (ICAS'2011)*, pages 67–74, Venice, Italy, May 2011. 12, 66

[28] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 42:1–42:13, New York, NY, USA, 2018. ACM. 15

[29] Rick F. van der Lans. Data virtualization for business intelligence systems. *Data Virtualization for Business Intelligence Systems*, 12 2012. 5

[30] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 17–17, Berkeley, CA, USA, 2015. USENIX Association. 14

[31] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 37–48, New York, NY, USA, 2012. ACM. 22, 30, 52, 102, 119

[32] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004. 120

[33] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 249–264, Berkeley, CA, USA, 2016. USENIX Association. 14, 15, 127

[34] L.A. Garrido and P. Carpenter. vMCA: Memory capacity aggregation and management in cloud environments. In *IEEE 23rd Intl. Conference on Parallel and Distributed Systems (ICPADS)*, December 2017. 19, 29, 37, 38, 71, 131

[35] Luis Garrido, Rajiv Nishtala, and Paul Carpenter. SmarTmem: Intelligent management of transcendent memory in a virtualized server. In *RADR 2019: 1st Workshop on Resource Arbitration for Dynamic Runtimes, 2019*, 2019. 17, 95, 130

[36] Luis A. Garrido and Paul Carpenter. Aggregating and managing memory across computing nodes in cloud environments. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *12th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'17), International Conference on High Performance Computing*, pages 642–652, Cham, 2017. Springer International Publishing. 18, 37, 71, 108, 116, 127, 130

[37] Luis A. Garrido and Paul Carpenter. Continuous-action reinforcement learning for memory allocation in virtualized servers. In *14th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'19), International Conference on High Performance Computing*, pages 642–652, Cham, 2019. Springer International Publishing. 18, 91, 95, 130

[38] Charles T. Gibson. Time-sharing in the ibm system/360: Model 67. In *Proceedings of the April 26-28, 1966, Spring Joint Computer Conference*, AFIPS '66 (Spring), pages 61–78, New York, NY, USA, 1966. ACM. 5

[39] Chunye Gong, Jie Liu, Qiang Zhang, Haitao Chen, and Zhenghu Gong. The characteristics of cloud computing. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 275–279, Washington, DC, USA, 2010. IEEE Computer Society. 22

[40] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 649–667, Berkeley, CA, USA, 2017. USENIX Association. 14, 15, 106

[41] Irfan Habib. Virtualization with kvm. *Linux Journal.*, 2008(166), February 2008. 6, 95

[42] Jia Hao, Binbin Zhang, Kun Yue, Hao Wu, and Jixian Zhang. Measuring performance degradation of virtual machines based on the bayesian network with hidden variables. *Int. J. Communication Systems*, 31, 2018. 52

[43] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015. 31, 102, 120

[44] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. 52

[45] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Proceedings of the 11th IFIP/IEEE International Conference on Symposium on Integrated Network Management*, IM'09, pages 630–637, Piscataway, NJ, USA, 2009. IEEE Press. 12, 66

[46] Jinho Hwang, Ahsen Uppal, Timothy Wood, and Howie Huang. Mortar: Filling the gaps in data center memory. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 53–64, New York, NY, USA, 2014. ACM. 13, 126

[47] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 14–24, New York, NY, USA, 2006. ACM. 12, 66

[48] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM. 14

[49] Kostas Katrinis, Dimitris Syrivelis, D Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, K Hasharoni, D Raho, C Pinto, F Espina, et al. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 690–695. IEEE, 2016. 14, 15, 17, 96, 108

[50] K. T. Kearney, F. Torelli, and C. Kotsokalis. Sla* an abstract syntax for service level agreements. In *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 217–224, Oct 2010. ii

[51] Jinchun Kim, Viacheslav Fedorov, Paul V. Gratz, and A. L. Narasimha Reddy. Dynamic memory pressure aware ballooning. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 103–112, New York, NY, USA, 2015. ACM. 12, 36

[52] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Group-based memory oversubscription for virtualized clouds. *J. Parallel Distrib. Comput.*, 74(4):2241–2256, April 2014. 13, 14

[53] Hajime Kimura and Shigenobu Kobayashi. An analysis of actor/critic algorithms using eligibility traces: reinforcement learning with imperfect value functions. In *Proc. 15th International Conf. on Machine Learning*, pages 278–286, 1998. 47

[54] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1008–1014. MIT Press, 2000. 47

[55] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 317–330, New York, NY, USA, 2019. ACM. 14, 17, 126

[56] Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Reinforcement learning in continuous action spaces through sequential monte carlo methods. In *Advances in Neural Information Processing Systems*, 2007. 47

[57] Min Lee, Vishal Gupta, and Karsten Schwan. Software-controlled transparent management of heterogeneous memory resources in virtualized systems. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM. 16

[58] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. Model-free control for distributed stream data processing using deep reinforcement learning. *Proc. VLDB Endow.*, 11(6):705–718, February 2018. 51, 75

[59] Xi Li, Pengfei Zhang, Rui Chu, and Huaimin Wang. Optimizing guest swapping using elastic and transparent memory provisioning on virtualization platform. *Front. Comput. Sci.*, 10(5):908–924, October 2016. 22

[60] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. 44, 46, 47, 48, 70, 81

[61] Kevin T P Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009. 14, 15

[62] Tim Lindholm and Frank Yellin. The java virtual machine specification. 1997. 5

[63] Haikun Liu, Hai Jin, Xiaofei Liao, Wei Deng, Bingsheng He, and Cheng-zhong Xu. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1350–1363, 2015. 7, 8, 12, 22, 36

[64] Huan Liu. A measurement study of server utilization in public clouds. In *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, DASC '11, pages 435–442, Washington, DC, USA, 2011. IEEE Computer Society. 14

[65] Jiuxing Liu and Bulent Abali. Virtualization polling engine (vpe): Using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 225–234, New York, NY, USA, 2009. ACM. 6

[66] Maxime Lorrillere, Julien Sopena, Sébastien Monnet, and Pierre Sens. Puma: Pooling unused memory in virtual machines for i/o intensive applications. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 1–11, New York, NY, USA, 2015. ACM. 16, 126

[67] D. Magenheimer. Memory overcommit... without the commitment. In *Extended Abstract for Xen Summit*, June 2008. 7, 70, 71

[68] D. Magenheimer. Zcache and ramster (oh, and frontswap too) overview and some benchmarking, 2012. 17, 105, 126

[69] David Magenheimer, C. Mason, D. McCracken, and K. Hackel. Transcendent memory and linux. In *Ottawa Linux Symposium*, pages 191–200, July 2009. 9, 11, 22, 71, 94

[70] Manolis Marazakis, John Goodacre, Didier Fuin, Paul Carpenter, John Thomson, Emil Matus, Antimo Bruno, Per Stenstrom, Jerome Martin, Yves Durand, et al. Euroserver: Share-anything scale-out micro-server design. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 678–683. EDA Consortium, 2016. 108

[71] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970. 12

[72] Daniel A. Menasce and Mohamed N. Bennani. Autonomic virtualized environments. In *Proceedings of the International Conference on Autonomic and Autonomous Systems*, ICAS '06, pages 28–, Washington, DC, USA, 2006. IEEE Computer Society. 12, 66

[73] F.P. Miller, A.F. Vandome, and M.B. John. *IBM Cp-40*. VDM Publishing, 2010. 5

[74] Dave Mishchenko. *VMware ESXi: Planning, Implementation, and Security*. Course Technology Press, Boston, MA, United States, 1st edition, 2010. 6, 95

[75] Debadatta Mishra and Purushottam Kulkarni. Comparative analysis of page cache provisioning in virtualized environments. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, pages 213–222, Washington, DC, USA, 2014. IEEE Computer Society. 13

[76] Debadatta Mishra and Purushottam Kulkarni. Comparative analysis of page cache provisioning in virtualized environments. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, pages 213–222, Washington, DC, USA, 2014. IEEE Computer Society. 66

[77] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1537–1550, May 2016. 15

[78] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop 2013*, abs/1312.5602, 2013. 46

[79] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. 13, 44, 46, 48

[80] Germán Moltó, Miguel Caballer, Eloy Romero, and Carlos de Alfonso. Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements. In *International Conference on Computational Science*, volume 18, 2013. 12, 36

[81] Openstack. http://www.openstack.org/. 22

[82] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010. 17, 105, 126

[83] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM. 12, 66

[84] P.Lu and K.Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 3:1–3:15, Berkeley, CA, USA, 2007. USENIX Association. 12, 66

[85] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. 3, 5, 22

[86] Matthew Portnoy. *Virtualization Essentials*. SYBEX Inc., Alameda, CA, USA, 1st edition, 2012. 5

[87] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. Vconf: A reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 137–146, New York, NY, USA, 2009. ACM. 13, 44, 47, 67, 90

[88] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM. 14

[89] Alvise Rigo, Christian Pinto, Kevin Pouget, Daniel Raho, Denis Dutoit, Pierre-Yves Martinez, Chris Doran, Luca Benini, Iakovos Mavroidis, Manolis Marazakis, et al. Paving the way towards a highly energy-efficient and highly integrated compute node for the exascale revolution: the exanode approach. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 486–493. IEEE, 2017. 4, 108

[90] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the ACM*, 17:365–375, 1974. 5

[91] Rossi, R. A. and Ahmed, N. K. An interactive data repository with visual analytics. *SIGKDD Explor.*, 17(2):37–41, 2016. 31, 102, 120

[92] Rossi, R. A. and Ahmed, Nesreen K. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. 31, 102, 120

[93] Rossi, Ryan A. and Ahmed, Nesreen K. soc-twitter-follows - social networks. http://networkrepository.com/soc-twitter-follows.php, 2013. 31, 102

[94] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Proceedings of the 2010 Second International Conference on Computer and Network Technology*, ICCNT '10, pages 222–226, Washington, DC, USA, 2010. IEEE Computer Society. 22

[95] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 337–350, New York, NY, USA, 2013. ACM. 12, 36

[96] Vasily A. Sartakov and Rüdiger Kapitza. Nv-hypervisor: Hypervisor-based persistence for virtual machines. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 654–659, Washington, DC, USA, 2014. IEEE Computer Society. 15

[97] Joel H. Schopp, K.Fraser, and Martine J. Silbermann. Resizing memory with balloons and hotplug. In *In Proceedings of the Linux Symposium*, pages 313–319, 2006. 7, 8, 22, 71

[98] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14. JMLR.org, 2014. 47, 48

[99] José Simão, Jeremy Singer, and Luís Veiga. A comparative look at adaptive memory management in virtual machines. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01*, volume 1 of *CLOUDCOM '13*, pages 452–457, Washington, DC, USA, 2013. IEEE Computer Society. 22

[100] Rebecca Smith and Scott Rixner. A policy-based system for dynamic scaling of virtual machine memory reservations. In *SoCC*, 2017. 30, 36, 37, 102, 119

[101] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 953–966, New York, NY, USA, 2008. ACM. 12, 66

[102] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. 51, 75

[103] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press. 47, 48

[104] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Hecatonchire: Towards multi-host virtual machines by server disaggregation. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, pages 519–529, 2014. 106

[105] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ICAC '06, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society. 12, 66

[106] Chuck Thacker and MSR Silicon Valley. Beehive: A many-core computer for FPGAs (v5). *MSR Silicon Valley*, 2010. 96

[107] Aimilios Tsalapatis, Stefanos Gerangelos, Stratos Psomadakis, Konstantinos Papazafeiropoulos, and Nectarios Koziris. utmem: Towards memory elasticity in cloud workloads. In *ISC Workshops*, 2018. 13

[108] Dmitrii Ustiugov, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, and Dionisios Pnevmatikatos. Design guidelines for high-performance scm hierarchies. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, pages 3–16, New York, NY, USA, 2018. ACM. 14, 15

[109] Hado Van Hasselt. Reinforcement learning in continuous state and action spaces. *Reinforcement Learning*, 12:207–251, August 2012. 46

[110] Vimalraj Venkatesan, Wei Qingsong, and Y. C. Tay. Ex-tmem: Extending transcendent memory with non-volatile memory for virtual machines. In *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, HPCC '14, pages 966–973, Washington, DC, USA, 2014. IEEE Computer Society. 13, 15, 37, 38, 43, 126

[111] Vimalraj Venkatesan, Y. C. Tay, and Qingsong Wei. Sizing cleancache allocation for virtual machines' transcendent memory. *IEEE Transactions on Computers*, 65:1949–1963, 2016. 13

[112] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002. 7, 8, 22

[113] Lizhe Wang, Gregor von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. Cloud computing: a perspective study. *New Generation Comput.*, 28:137–146, 04 2010. 3

[114] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. 44, 46

[115] Steven D. Whitehead and Long-Ji Lin. Reinforcement learning of non-markov decision processes. *Artificial Intelligence*, 73(1–2):271–306, February 1995. 45

[116] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 205–216, New York, NY, USA, 2011. ACM. 7

[117] Georgios S. Zervas, Hui Min Yuan, A. Saljoghei, Qianqiao Chen, and Vaibhawa Mishra. Optically disaggregated data centers with minimal remote memory latency: Technologies, architectures, and resource allocation [invited]. *IEEE/OSA Journal of Optical Communications and Networking*, 10:A270–A285, 2018. 14

[118] Chongjie Zhang, Victor Lesser, and Prashant Shenoy. A multi-agent learning approach to online distributed resource allocation. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 361–366, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc. 47

[119] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. 22

[120] Qi Zhang, Ling Liu, Calton Pu, Wenqi Cao, and Semih Sahin. Efficient shared memory orchestration towards demand driven memory slicing. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1212–1223, 2018. 12, 36, 37, 52

[121] W. Zhang, H. Xie, and C. Hsu. Automatic memory control of multiple virtual machines on a consolidated server. In *IEEE Transactions on Cloud Computing, vol. 5, no. 1*, pages 2–14, 2017. 66

[122] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM. 15

[123] Zhizhong Zhang, Chuan Wu, and David W.L. Cheung. A survey on cloud interoperability: Taxonomies, standards, and practice. *SIGMETRICS Perform. Eval. Rev.*, 40(4):13–22, April 2013. 22

[124] Ming Zhao, Lixi Wang, Yun Lv, and Jing Xu. Cross-layer optimization for virtual machine resource management. *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 90–98, 2018. 13

[125] Weiming Zhao, Zhenlin Wang, and Yingwei Luo. Dynamic memory balancing for virtual machines. In *SIGOPS Oper. Syst. Rev*, volume 43, pages 37–47. ACM, July 2009. 12, 36, 66

[126] Guoliang Zhu, Kai Lu, Xiaoping Wang, Yiming Zhang, Pengfei Zhang, and Sparsh Mittal. Swapx: An nvm-based hierarchical swapping framework. *IEEE Access*, 5:16383–16392, 2017. 15

[127] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A Mckee, Paul M Carpenter, Petar Radojković, and Eduard Ayguadé. Main memory in hpc: do we need more or could we live with less? *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):3, 2017. 15

Index