Memory management in a virtual environment

Aram Kocharyan

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier

Cotutelle internationale : Institute for Informatics and Automation Problems (IIAP)

Présentée et soutenue par

Aram KOCHARYAN

Le 2 juillet 2019

Gestion de la mémoire dans un environnement virtuel

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Daniel HAGIMONT et Hrachya ASTSATRYAN

Jury

M. Noel DEPALMA, Rapporteur
M. Eric GRESSIER-SOUDAN, Rapporteur
M. Daniel HAGIMONT, Directeur de thèse
M. Hrachya ASTSATRYAN, Co-directeur de thèse
Mme Mariam HAROUTUNIAN, Président

Acknowledgements

I would like to express my special appreciation and thanks to my advisors Professor Daniel Hagimont and Hrachya Astsatryan, you have been tremendous mentors for me. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my career have been priceless. I would like to express special thanks to professor Alain Tchana for his motivation, continuous support and advice.

Besides my advisors, I would also like to thank the rest of my thesis committee, Prof. Noel Depalma and Prof. Eric Gressier-Soudan, for serving as my committee members even at hardship. I also want to thank you for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions, thanks to you.

I would like to thank my labmates: Boris Teabe, Vlad Nitu, Mathieu Bacou, Gregoire Todeschi, Kevin Jiokengfor, Wahi Narsisian and Hayk Grigoryan for their feedback, cooperation and of course friendship. I could not complete this work without their help and fruitful discussions and advice.

I would like to express my gratuity to Erasmus plus program for financial support to this research.

My sincere thanks also goes to Stphane Kojayan and Amicale des Armniens de Toulouse et de Midi-Pyrnes, who provided me a hospitality and care. Without their precious support, it would not be possible to conduct this research.

A special thanks to my family. Words cannot express how grateful I am to my family, for all of the sacrifices that youve made on my behalf. My father who motivated me to start my scientific trip and my mother and sister who were always a source of motivation for me. I would also like to thank all of my friends who supported me in writing, and incented me to strive towards my goal. At the end, I would like to express appreciation to my beloved wife Sirarpi and my son Seyran, who spent sleepless nights with me and was always my support in the moments when there was no one to answer my queries.

Rsum

L'augmentation des besoins en calcul pour les applications modernes (par exemple dans le big data) a conduit au dveloppement d'infrastructures de calcul de moyenne trs grande taille. Dans ce contexte, le cloud est devenu la solution permettant la mutualisation des ressources. De plus en plus d'entreprises ou d'institutions scientifiques mettent en place leur propre structure de cloud prive. Une technologie cl sous-jacente au dveloppement de ces infrastructures est la virtualisation. Les infrastructures virtualises apportent de nombreux avantages pour la gestion des ressources, mais l'optimisation de la gestion des ressources reste un dfi, avec l'objectif d'assurer un taux d'utilisation lev des ressources matrielles et un faible gaspillage.

La consolidation des serveurs a t introduite dans le but d'optimiser ces infrastructures. Le principe est de rassembler les machines virtuelles (VMs) sur un nombre minimal de serveurs, permettant alors de suspendre les serveurs inutiliss. Cependant, les stratgies de consolidation sont complexes mettre en oeuvre, car elles doivent prendre en compte diffrents types de ressources (CPU, mmoire, I/O). De plus, l'utilisation de ces ressources peut varier fortement au cours du temps et la consolidation repose sur la migration de VM qui est une opration trs lourde. En consquence, les consolidations sont effectues une frquence relativement faible.

Dans cette thse, nous proposons la conception d'un systme de gestion mmoire permettant la mutualisation de la mmoire entre les VMs. Ce systme est orthogonal et complmentaire la consolidation. La premire contribution est un systme de surveillance qui permet de mesurer le working-set (WS) de chaque VM l'excution avec une faible intrusivit. L'tape suivante est de reprendre la mmoire inutilise par les VMs ayant un petit WS et de l'allouer aux VMs ayant un gros WS pour les aider surmonter des surcharges mmoire temporaires. Ainsi, nous proposons un systme de mutualisation mmoire la fois local et global, permettant la surveillance de la taille du WS de chaque VM et la mutualisation de la mmoire inutilise, soit localement (avec des VMs sur le mme serveur) ou globalement (avec des VMs sur des serveurs voisins).

La solution a t value avec des benchmarks du HPC et du traitement de donnes massives, et galement des applications scientifiques et du big data (Apache Spark). Les rsultats dmontrent la pertinences des choix effectus.

Abstract

The increasing computation needs of modern applications led to the development of medium to large scale computer infrastructures. Cloud computing became a key solution for resource mutualization. More and more entreprises and scientific institutions set-up their own private cloud facilities. The key technology behind the development of these infrastructures is virtualization. Virtualized infrastructures bring many advantages for resource management, but resource optimization is still a challenge, for ensuring high hardware utilization and low waste.

Server consolidation was introduced for optimizing such infrastructures. Its principle is to gather VMs on as less servers as possible, thus allowing unused servers to be suspended. However, consolidation strategies have to take into account many types of resource (CPU, memory, I/O) thus inducing a high complexity. Additionally, these resources may be fluctuating at runtime and consolidation relies on VM migration which is a heavy operation. Therefore consolidations are performed infrequently.

In this thesis, we propose the design of a memory management system which allows mutualizing memory between VMs. This system is orthogonal and complementary to consolidation. The first issue is to design a monitoring system that should track the working set of the VMs at runtime with low intrusiveness. The next important step is to reclaim unused memory from under-loaded VMs and finally grant it to over-loaded VMs to help them to overcome temporary memory difficulties. As a result, it is proposed a both local and global memory mutualization system which allows to monitor the working set size of each VM and to mutualize unused memory, either locally (with the VMs located on the same node) or globally (with the VMs located on neighbor nodes).

The solution has been evaluated with modern HPC, data intensive benchmarks as well as scientific and Big Data (Apache Spark) applications. The results demonstrate the effectiveness of our design choices.

Contents

Chapter 1

Introduction

Modern applications have increasing computational needs. They include traditional applications such as web applications or social networks, but also scientific pillars such as physics, biology or life sciences which require more and more resources. Moreover, the important development of data processing fields such as Big Data, Artificial Intelligence or Machine Learning significantly accelerated this tendency. Due to these tremendous resource needs, we observed the development of datacenters which include the computational infrastructures (clusters of servers) for hosting these applications.

Due to the cost of creation and maintenance of these infrastrutures, it was proposed to mutualize them, following the cloud computing principle. With cloud computing, a provider is maintaining an infrastructure which can be used on demand by its clients. The main benefit is to cut maintenance costs as they are shared between clients who don't have to invest in the management of their own infrastructure. Another advantage is the high scalability, as clients have potentially access to an infrastructure that they could not afford individually. Moreover, the pay-as-you-go pricing made cloud computing even more popular, allowing to pay only for the amount of resources that were effectively allocated according to the needs.

The evolution of cloud computing prompted many companies or institutions to set-up their own cloud computing infrastructures. In this regard, there are three main types of cloud computing infrastructures:

1. Public cloud infrastructures are owned and maintained by third parties (providers) which deliver their services (computational resources) to clients via internet. In this case, the infrastructure and the software are owned by the provider which is in charge of maintenance and providing high availability.

2. Private cloud infrastructures are owned by a company or scientific organization for its internal use. A company is owning its hardware and

software infrastructure to satisfy the needs of its own workloads (within the company). The main motivation for managing a private cloud (compared to relying on a public cloud) is security.

3. Hybrid clouds are the combination of the two previous solutions. This is applied in the case when an organization manages its own private cloud but extends it with resources from a public cloud in case of resource limitation.

In such infrastructures, different types of services may be managed:

1. Infrastructure as a Service (IaaS). This is the lowest level of service. The cloud provides clients with hardware resources (computing, storage, etc.). These resources may be real hardware resources or virtualized resources.

2. Platform as a Service (PaaS). This is a higher level service where the cloud provides its clients with a platform for the development, deployment and execution of a class of applications.

3. Software as a Service (SaaS). Here, the cloud directly provides the applications needed by its clients.

In this thesis, we are interested in resource management in computing platforms independently from any application domain, so we mainly consider the IaaS model.

1.1 Research Domain

With the cloud computing model, the clients benefit from the fact that providers inherit from the infrastructure management responsibility. The main objective of the providers is to cut the costs by saving energy and hardware consumption.

Energy consumption is a primary concern for datacenter (DC) management. Its cost represents a significant part of the total cost of ownership (about 80% [14]) and it is estimated that in 2020, US DCs will spend about $13 billion on energy bills [28]. The electricity consumption of computing infrastructures is already going upwards to 810% of global consumption [39] and the total global footprint is 2% of global CO_2 emissions [33]. This means that beside money wasting, the environment is directly influenced by the energy spend to support such infrastructures.

One of the most efficient ways to minimize the energy consumption of computational infrastructures is resource optimization. Resource management is crucial for every datacenter provider. The aim of resource management is to provide the same service with less resources and with the same quality (without

Service Level Agreement (SLA) violation). The main resources of the datacenters are: CPU, Memory, Disk and Network.

1.2 Problem Statement

The previous section described the importance of resource management and the impact of energy spendings on the world environmental system. Many researchers in the world try to tackle the issue of resource optimization nowadays. However, it is very complex to target all types of resources at once. Thus, there are many works in the field on CPU resource optimization, since it was considered to be the most expensive and main resource in datacenters. However, the situation has changes during years. Over last years, we have seen the emergence of new applications with growing memory demands, while hardware platforms' evolution continued to offer more CPU capacity growth than memory, referred to as the memory capacity wall [57]. In this PhD, we focus on optimization of memory resource management and optimization in datacenters.

1.3 Background

1.3.1 Virtualized infrastructures

A majority of datacenters implements the Infrastructure as a Service (IaaS) model where *customers* buy (from *providers*) Virtual Machines (VMs) with a set of reserved resources. The VMs host general purpose applications (e.g. web services), as well as High Performance Computing applications. In such IaaS DCs, virtualization is a fundamental technology which allows optimizing the infrastructure by colocating several VMs on the same physical server.

Generally, in computer science, the term virtualization is associated with the creation of a virtual instance of a physical machine called virtual machine. This is achieved by virtualizing the main resources such as CPU, Memory, Disk, Network etc. Virtualization adds an abstract layer over the hardware, which allows to run several VMs on a single physical host. The concept of VM was introduced by IBM [26, 42] a long time ago, way before this concept has been widely re-used in cloud computing. Virtualization changed the way resources are allocated in computing infrastructures. Before, providers were allocating a physical machine to run each given application. Virtualization allows to host a set of VMs on a single physical server and therefore to host several applications on the same server. VMs run isolated on their portion of hardware with their own operating system (OS). This means that they are fully protected one from another and totally independent. An abstracted software

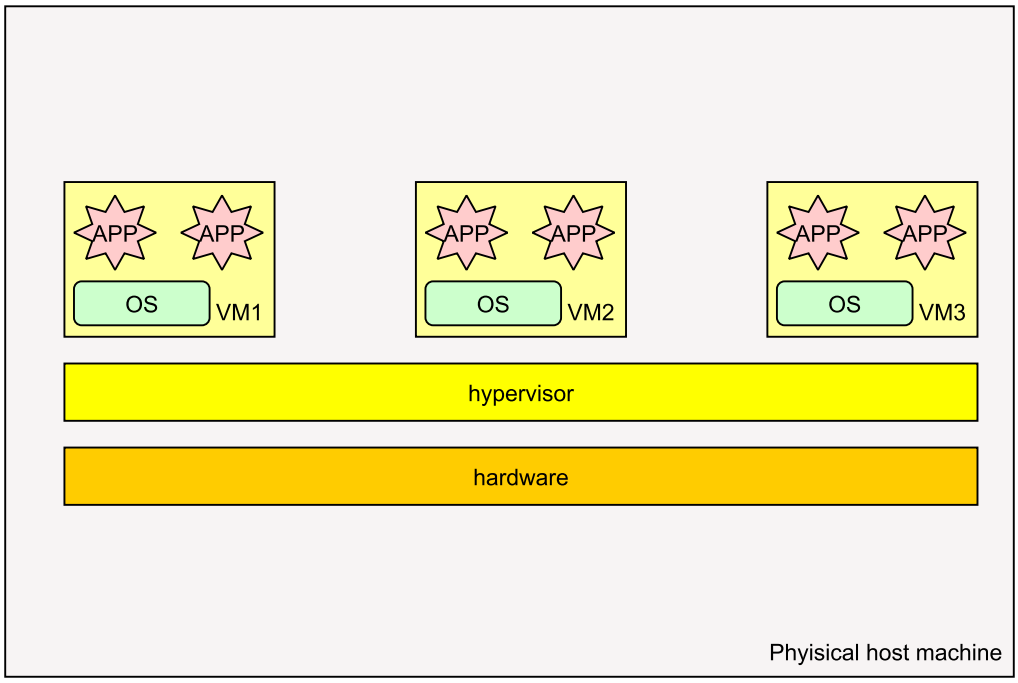


Figure 1.1: Type 1 hypervisor architecture

layer called hypervisor (or virtual machine monitor (VMM)) is in charge of distributing hardware resources between VMs and managing these VMs. There are two main types of hypervisors:

1. Type 1 or bare metal hypervisor. This type of hypervisor is installed directly on top of the hardware and distributes physical resources to VMs. The VMs (with their own OSs) run on-top of the hypervisor. Thus, the hypervisor acts as an OS and VMs run one level above it. Figure 1.1 illustrates the architecture of type 1 hypervisors. This architecture allows to minimize the overhead caused by the virtualization layer, allowing to perform close to the speed of a native OS. However, as the hypervisor should be installed in the place of the traditional OS, deployment and maintenance might be harder.

2. Type 2 or hosted hypervisor. Such an hypervisor runs on-top of an existing OS. This adds an extra level of virtualization which obviously results in higher overhead than type 1 hypervisors. However, in this type of hypervisor, deployment and management of the hypervisor are much easier and flexible. Figure 1.2 illustrates the architecture of hosted hypervisors and points the extra level added by this type of hypervisor.

There are three main virtualization techniques: paravirtualization, full vir-

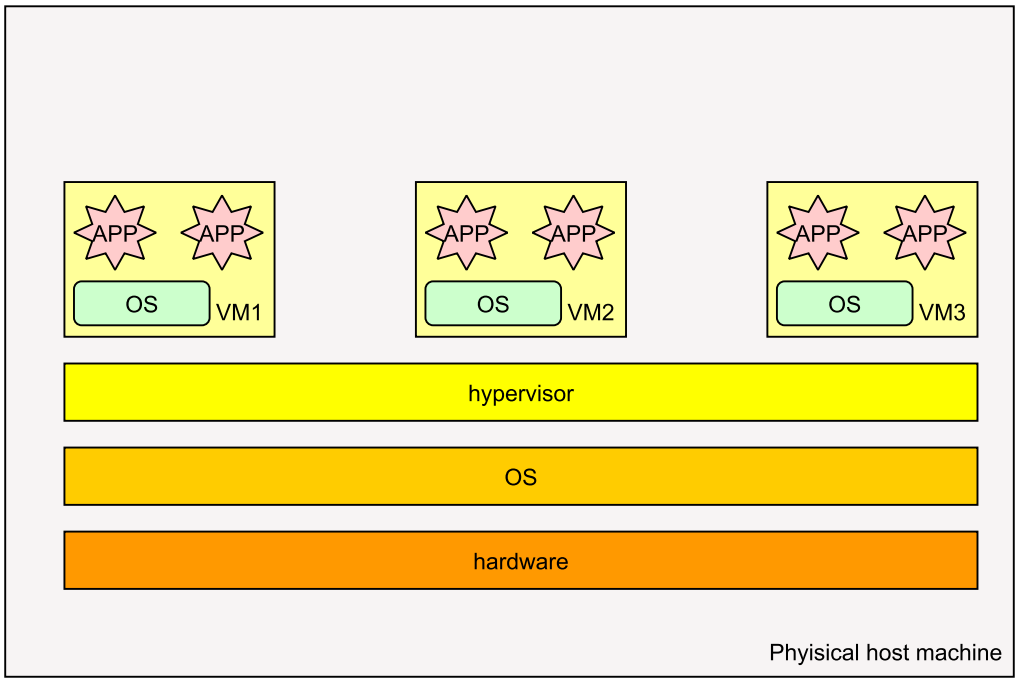


Figure 1.2: Type 2 hypervisor architecture

tualization and Hardware assisted virtualization.

1. In paravirtualization, the OS (called the guest OS) is modified in a way that it is aware of the virtualization system (the hypervisor) on top of which it runs. The guest OS is capable to directly initiate direct calls to the hypervisor called hyper-calls. This is a means to reduce the overhead caused by virtualization.

2. In full virtualization, the hypervisor emulates the hardware for the guest OS. So the guest OS does not have to be aware that it runs on a virtualized hardware. This allows to run an unmodified guest OS in a virtualized environment. However, emulation has a significant cost.

3. Hardware-assisted virtualization uses specific virtualization support provided by modern hardware, and it does not require any kind of custom kernel or patches in the guest OS. Thus, it allows to run unchanged guest OSes with close to native OS speeds.

Nowadays, almost every modern physical machine supports virtualization. Virtualization made a step forward and it was even introduced for embedded devices [40], allowing to run VMs on your smartphones or tablets. The motivations of using virtualization in most of the infrastructure changed. Initially

the main motivation was the full utilization of a hardware resources. Than, many started to use it due to its fault tolerance and ease of maintenance. However, during last years more and more providers adapted it to optimize resource consumption and minimize the energy consumption of the infrastructures.

1.3.2 Resource Management in Virtual Infrastructures

Virtualization is a key to resource management techniques. It allows to manage isolated execution environments (VMs) to which resources are allocated. Generally there are two approaches of resource allocation:

1. Static: where a VM is created with a given amount of resources and the resource allocation never changed during the lifetime of the VM.

2. Dynamic: where resources are allocated at creation time but might be dynamically adapted at runtime according to VM's needs.

In case of static allocation, the user estimates the resource consumption and asks for a VM according to the peak workload of the application in order to avoid VM saturation. Thus the reserved resources may stay underutilized most of the time. This leads to resource wasting.

Dynamic resource allocation

Dynamic resource allocation [53] allows to change the resource (CPU, memory, I/O) allocation during the runtime of VMs, according to resource utilization and therefore to avoid wasting. For CPU and I/O resources, a time-sharing approach is generally used, where VMs are granted access to the resource for a given percentage of time. Therefore, dynamically changing the resource allocation is relatively easy. However, regarding memory, things are more tricky. The reason is the complexity of working set estimation (the amount of memory effectively used by a VM) and fair memory sharing.

Regardless of this functionality, most of the e-infrastructures use a static allocation approach.

VM live migration and consolidation

Another powerful resource management technique is VM live migration [24] which allows to migrate VMs from one physical server to another without disturbing the applications running inside a VM. This means that applications do not have to be aware of migrations and users connected to these VMs do not notice any changes while a VM is migrated to another machine. Dynamic resource allocation associated with live migration may allow to gather VMs on

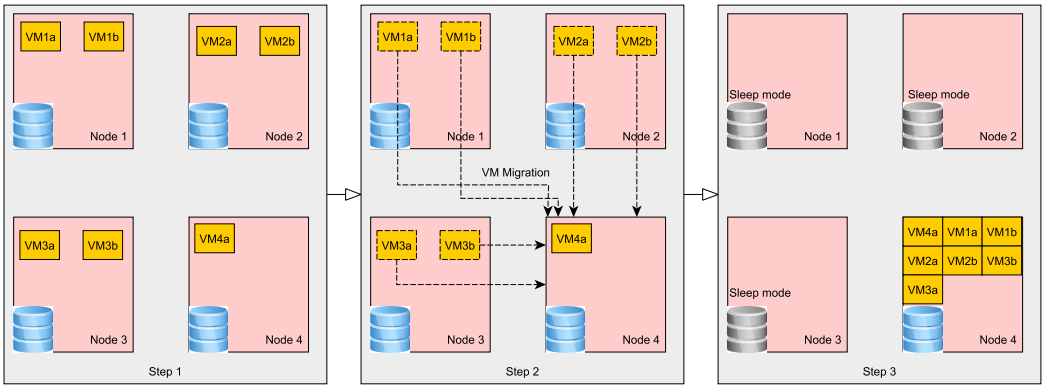


Figure 1.3: How consolidation is achieved via VM migration

a minimal number of physical servers and to power off the idle servers. This can dramatically reduce the energy consumption of a datacenter. Symetrically, overloaded VMs may be migrated to a relatively underloaded physical machines in order to re-allocate additional resources to these VMs and to boost their performance or avoid Service Level Agreement (SLA) violation.

The density of VMs collocated on a single physical machine is called consolidation ratio. VM migration is used to raise the consolidation [25, 77] of physical machines by hosting as much VMs as possible on each single node. Consolidation is one of the main techniques used to improve resource utilization in datacenters. Specific tools called "consolidators" are managing clusters by implementing migration and placement policies in datacenters. A consolidator decides where each VM should be placed or migrated in order to achieve the highest consolidation ratio with the minimal number of active (powered on) physical machines. Figure 1.3 illustrates how consolidation is achieved via VM migration.

1.3.3 Memory Management in Virtual Environments

Ideally, consolidation should lead to highly loaded servers. Although consolidation may increase server utilization by about 5-10%, it is difficult to actually observe server loads greater than 50% for even the most adapted workloads [15, 29, 66]. The main reason is that VM collocation is memory bound, as memory saturates much faster than the CPU[72]. This situation was accentuated over the last several years, as we have seen emerging new applications with growing memory demands, while physical platforms had an opposite tendency; the augmentation of the CPU capacity was faster than that of physical memory. Therefore, memory is key to resource management.

As it is extremely complex to tackle the management of all the resources at

the same time, in this thesis, we focus on memory management.

Optimizing memory management consists in avoiding memory waste. The general principle is to identify VMs with weakly used memory, then reclaiming that memory, and finally distributing this memory to VMs which lack memory. Thefore, such a memory management consists of the following three steps:

1. Monitoring: the most important component of this memory management system is a monitoring system. The monitoring system should allows to track the memory consumption of VMs. This will allow to reclaim the unused pages.

2. Reclaiming: This functionality should allow the memory management system to reclaim unused memory from under-loaded VMs (regarding memory).

3. Re-distributing: This technique should allow to redistribute the reclaimed memory locally (to the VMs on the same node) or globally (to the VMs on the neighboring nodes).

Monitoring

In a general purpose OS, a process allocates some number of pages in memory to load necessary data. However, during the run of the process, some of the allocated pages are accessed more often than others. The pages that are accessed frequently are called warm pages while less frequently accessed pages are called cold pages. In case of memory shortage, the system will have to send some of the allocated pages to swap. Hence, the optimal way is to swap out cold pages, because swap is much slower than memory and frequent accesses to pages located in the swap would degrade performance. The working set of the system is the set of warm pages. Therefore, the system tries to predict its memory behavior in order to be more efficient. The prediction is based on the hypothesis that that pages in the working set are more likely to be accessed in nearest future rather than cold pages that are out of working set.

Monitoring the working set of VMs is a challenge for datacenter providers as it allows to measure the memory need by VMs and the memory which can be reclaimed. The reclaimed memory can then be used to satisfy memory needs of other VMs in order to raise the consolidation ratio.

Reclaiming

Memory ballooning [13, 85] is a memory management technique which allows memory to be dynamically reclaimed from a VM by the hypervisor. Most of the modern hypervisors implement this technique in order to reclaim unused

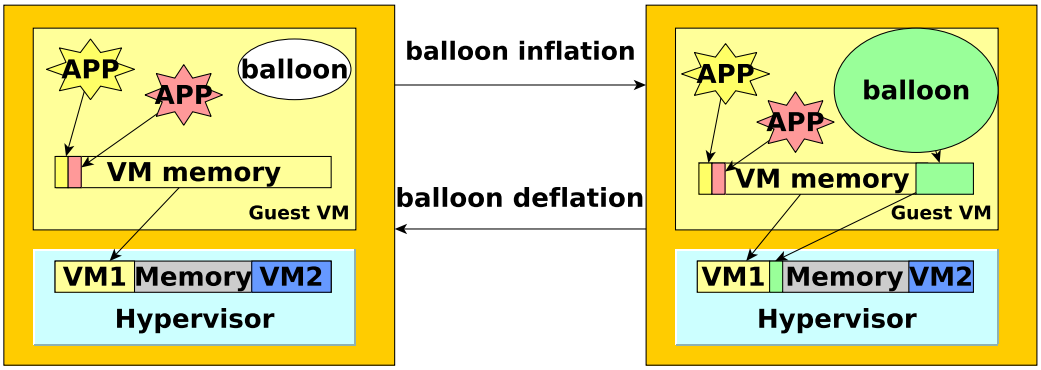


Figure 1.4: Memory ballooning principles.

memory from VMs, thus avoiding resource waste. In such systems, every VM is equipped with a balloon driver which can be inflated or deflated (by the hypervisor/dom0). Fig. 1.4 presents the general functioning of the balloon driver. Balloon inflation raises memory pressure on the VM, as follows. As soon as the balloon driver receives a higher balloon target size, it allocates a portion of memory and pins it, thus ensuring that memory pages cannot be swapped-out by the VM's OS. Then, the balloon driver reports the addresses of the pinned pages to the hypervisor so that they can be used for other purposes (e.g. assigned to a VM which is lacking memory). In the case of a balloon deflation, the balloon driver receives the addresses of pages that are freed by the hypervisor, and deallocates them. Thereby, the pages reenter under the control of the VM's OS.

Therefore, ballooning provides the ability to reclaim memory from some VMs and allocate it to other VMs, the selection of these VMs being based on the previous working set monitoring tool.

Re-Distributing

Memory reclaimed by the hypervisor on one server can be granted to VM which lack memory on the same server. However, this reclaimed memory cannot simply be allocated to remote VMs.

Generally, swapping (on local disk) is considered as an unwanted operation due to the extremely slow speed. Swapping to local disk is a bottleneck for every computer system. However, in modern computing infrastructures, the networking technologies are providing high speed connections to remote machines, including the connection to the memory of remote machines. More precisely, networking technologies such as RDMA (Remote Direct Memory Access) have been introduced, which allow to access the memory of a remote machine without interaction with the CPU of the remote machine. This allows

to consider the use of the memory of remote machines as swap devices. Thus remote swapping can be used global memory mutualization, i.e. re-distributing reclaimed memory to remote machines.

1.4 Contributions

1.4.1 Working Set Size Estimation Techniques in Virtualized Environments: Badis

Numerous datacenters are relying on virtualization, as it provides flexible resource management means such as virtual machine (VM) checkpoint/restart, migration and consolidation. However, one of the main hindrances to server consolidation is physical memory. In nowadays cloud, memory is generally statically allocated to VMs and wasted if not used. Techniques (such as ballooning) were introduced for dynamically reclaiming memory from VMs, such that only the needed memory is provisioned to each VM. However, the challenge is to precisely monitor the needed memory, i.e., the working set of each VM. In this context, we thoroughly reviewed the main techniques that were proposed for monitoring the working set of VMs. We implemented the main techniques in the Xen hypervisor and we defined different metrics in order to evaluate their efficiency. Based on the evaluation results, we proposed Badis, a system which combines several of the existing solutions, using the right solution at the right time. We also proposed a consolidation extension which leverages Badis in order to pack the VMs based on the working set size and not the booked memory. The implementation of all techniques, our proposed system, and the benchmarks we have used are publicly available in order to support further research in this domain.

1.4.2 Local Memory Mutualization Based on Badis

Virtualization allows to run several Virtual Machines (VMs) in parallel and isolated on a single physical host. In most virtualized environments, memory is statically allocated to VMs which means it is given to the VM at creation time and for the VM's lifetime. Such a memory management policy has one main drawback: some VMs may be lacking memory while others have unused memory. Relying on a VM working-set estimation facility that we previously implemented, we designed a memory management policy which allows reclaiming unused memory from unsaturated VMs and to lend it to saturated VMs. We implemented this memory management service in the Xen virtualization environment and evaluated its accuracy with several benchmarking applica-

tions.

1.4.3 Global Memory Mutualization system for Virtualized Computing Infrastructures

Resource management is a critical issue in today's virtualized computing infrastructures. Consolidation is the main technique used to optimize such infrastructures. It allows gathering overloaded and underloaded VMs on the same server so that resources can be mutualized. However, consolidation is much complex as it has to manage many different resources at the same time (CPU, memory, IO, etc.). Moreover, it has to take into account infrastructure constraints which limit VM migrations making it difficult to optimize resource management.

In this context, besides consolidation, we propose a service which allows a global memory mutualization between VMs. It relies on remote memory sharing for mutualizing memory. We implemented a system which monitors the working set of virtual machines, reclaims unused memory and makes it available (as a remote swap device) for virtual machines which need memory.

Our evaluations with HPC and Big Data benchmarks demonstrate the effectiveness of this approach. We show that remote memory can improve the performance of a standard Spark benchmark by up the 17% with an average performance degradation of 1.5% (for the providing application).

1.5 Thesis Statement

The problem During the last years, the number of memory (data) intensive applications has significantly increased. Many such applications are related with Artificial Intelligence (AI), Machine Learning (ML) and Big Data (BD). It is obvious that such applications process huge amount of data and require large amounts of memory resource for execution. Such applications make more pressure on the memory resource. The traditional memory management systems in DCs (which are based on static memory allocation) are not able to optimize memory management and especially to mutualize memory between VMs.

Solution We propose to implement a system which allows both local and global mutualization. Local memory mutualization system should deal with sharp memory increase of applications' working set in VMs. Local memory mutualization should amortize such peaks with unused memory from neighboring VMs located on the same node. If the local memory resources are not

enough to amortize these peaks, global memory mutualization should try to satisfy memory needs of the overloaded VMs with unused memory from neighboring nodes. Global memory mutualization is achieved via remote swapping technique using RDMA technologies which reduces the overhead on neighbor node CPUs.

Methods While the solution seems to be basic and intuitive, however it includes several touchy aspects of memory managements such as:

1. Working set estimation which should not be application or VM intrusive, and should not induce a significant overhead.

2. Local memory mutualization which gathers working set estimations from VMs and makes decisions based following a policy (from which VM to reclaim, to which VM to grant extra memory or how much to grant).

3. Global memory mutualization which should be coordinated with local memory mutualization and should allow the use of remote memory.

In this regards, we propose solutions to all the above issues. Our prototype was evaluated with modern memory and CPU intensive benchmarks as well as popular scientific applications.

Novelty Regarding working set estimation, we have evaluated the following techniques: Self-ballooning [63], Zballoond [22], the VMware technique [85], Geiger [47], Exclusive Cache [60] and Dynamic Memory Pressure Aware (MPA) Ballooning [51]. Based on the experimental results, it is possible to conclude that the method described in chapter 2 (Badis) is performing the best among the list of working set estimations techniques. Our evaluations were based on several metrics (such as accuracy, overhead, code base intrusiveness etc.).

This thesis introduces a Local memory mutualization system which is based on Badis as working set estimation technique and maintains a free memory reservoir which identifies available resources and distributes them in the most optimal way based on its local memory distribution policy.

The work is completed with a global memory mutualization system, which includes above described two contributions and is able to distribute memory globally, thus providing a complete solution for datacenters to tackle memory optimization problems. There have been several related works that had the following drawbacks compared to our solution: Ginkgo [45] requires application profiling, SpongeFiles [32] is targeting only few applications with no virtualization support, while Nswap2L [71] does not support RDMA which is a key technology in remote swapping.

1.6 Publications

Publications that constitute this thesis:

1. Hrachya Astsatryan, Wahi Narsisian, Aram Kocharyan, Georges Da Costa, Albert Hankel, Ariel Oleksiak: Energy optimization methodology for ein-frastructure providers - Concurrency and Computation: Practice and Experience, Volume 29, Issue 10, 2017

2. Vlad Nitu, Aram Kocharyan, Hannas Yaya, Alain Tchana, Daniel Hagimont, Hrachya V. Astsatryan: Working Set Size Estimation Techniques in Virtualized Environments: One Size Does not Fit All. Proceedings of the ACM on Measurement and Analysis of Computing Systems - SIGMETRICS 2018: Volume 2 Issue 1, March 2018 Article No. 19, 2018.

3. Aram Kocharyan, Boris Teabe, Vlad Nitu, Alain Tchana, Daniel Hagimont, Hrachya Astsatryan, Hayk Kocharyan: Intra-Node Cooperative Memory Management System for Virtualized Environments - 2018 Ivannikov Memorial Workshop (IVMEM), IEEE Conference, Pages 56-60, 2018

4. Aram Kocharyana, Alain Tchana, Hrachya Astsatryanb, Daniel Hagimonta: A remote memory sharing system for virtualized computing infrastructures ,Future Generation Computer Systems, 2019 (pending approval)

Other publications:

1. Hrachya Astsatryan, Hayk Grogoryan, Eliza Gyulgyulyan, Anush Hakobyan, Aram Kocharyan, Wahi Narsisian, Vladimir Sahakyan, Yuri Shoukourian, Rita Abrahamyan, Zarmandukht Petrosyan, Julien Aligon: Weather Data Visualization and Analytical Platform - Scalable Computing: Practice and Experience,Volume 19, Issue 2, Pages 79-86, 2018

1.7 Roadmap

In Chapter 2, we survey the state-of-the-art for working set size estimation techniques and propose Badis, a system that is able to estimate a VM's working set size with high accuracy and no VM codebase intrusiveness. Chapter 3 describes the coordinated memory management system for local memory mutualization which is based on Badis. Chapter 4 presents the global memory mutualization system based on remote swapping.

Chapter 2

Studies on Working Set Size Estimation Techniques in Virtualized Environments: Badis

2.1 Introduction

The existing consolidation systems [6, 36] take the CPU as a pivot, i.e. the central element of the consolidation. The memory is considered constant (i.e. the initially booked value) all over the VM's lifetime. Nevertheless, we consider that the memory should be the consolidation pivot since it is the limiting resource. In order to reduce the memory pressure, the consolidation should consider the memory actually consumed (i.e. the VM's working set size) and not the booked memory (see Fig. 2.1). Thereby, we need mechanisms to (1) evaluate the working set size (WSS) of VMs, (2) to anticipate their memory evolution and (3) to dynamically adjust the VMs' allocated memory. Numerous research papers propose algorithms to estimate the WSS of VMs. However, most of them are able to follow either up-trends (the increase) or down-trends (the decrease) of WSS. The few of them which are able to follow both trends are highly intrusive. Moreover, to the best of our knowledge, no previous work has shown the implications of dynamically adjusting the VM's allocated memory according to the WSS estimation. Finally, as far as we know, no previous consolidation algorithm considers the WSS as a pivot. In this work we address all the above limitations.

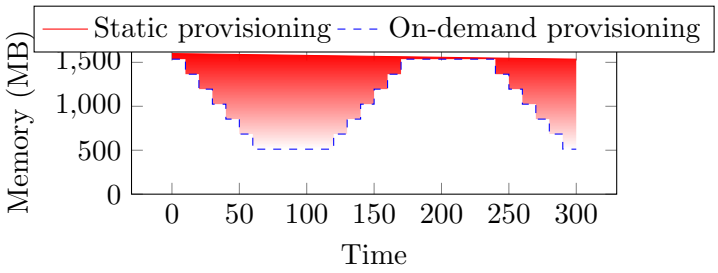


Figure 2.1: Static provisioning vs on-demand provisioning.

In summary, the contributions of this work are the following:

- We define evaluation metrics that allow to characterize WSS estimation solutions.
- We evaluate existing WSS techniques on several types of benchmarks. Each solution was implemented in the Xen virtualization system.
- We propose Badis, a WSS monitoring and estimation system which leverages several of the existing solutions in order to provide high estimation accuracy with no codebase intrusiveness. Badis is also able to dynamically adjust the VM's allocated memory based on the WSS estimations.
- We propose a consolidation system extension which leverages Badis for a better consolidation ratio. Both the source and the data sets used for our evaluation are publicly available [7], so that our experiments can be reproduced.

The rest of this work is structured as follows: Section 2.2 covers a quick background overview. Section 2.3 presents the general functioning of a WSS estimation solution. Section 2.4 presents the existing WSS estimation techniques that we analyze and evaluate in this article. Section 2.5 reports the evaluation results for the main studied techniques. Section 2.6.1 exposes the details of Badis while Section 2.6.2 presents the way we integrated Badis in an OpenStack cloud. Section 2.6.3 evaluates our solution. After a review of related works in Section 2.7, we present our conclusions in Section 2.8.

2.2 Background on virtualization: illustration with Xen

2.2.1 Generalities

The main goal of virtualization is to multiplex hardware resources between several guest operating systems also called Virtual Machines (VMs). Xen [13]

is a well-known virtualization system employed by Amazon [1] to virtualize its DCs. Xen relies on a hypervisor which runs on the bare hardware, and a particular VM (the *dom0*) which includes all OS services. The latter are not included in the hypervisor in order to keep it as lightweight as possible. The other (general purpose) VMs are called *domUs*. In the next subsections, we provide details about memory management and I/O management in Xen, necessary for understanding the WSS techniques we study in this chapter.

2.2.2 Memory and I/O virtualization

In a fully virtualized system, the VM believes it controls the RAM. However, the latter is actually under the control of the hypervisor which ensures its multiplexing between multiple VMs. In this respect, one of the commonly used techniques is the following. The page frame addresses presented to the VM and used in its page tables are fictitious addresses (called pseudo-physical). They do not designate a page frame's actual location in the physical RAM. The real addresses (i.e. host-physical) are known only by the hypervisor which maintains for each *guest page table* in the VMs (mapping guest-virtual $\rightarrow$ pseudo-physical), an equivalent called *shadow page table* (mapping guest-virtual $\rightarrow$ host-physical). Each shadow page table is synchronized with its equivalent guest page table. The shadow page tables are the ones used by the MMU[1]. The guest page tables play no role in the address translation process. However, how the hypervisor ensures this synchronization knowing that the VM is a "black box"? In this respect, the hypervisor runs each guest kernel at Ring 3 and sets as read-only the address ranges corresponding to guest page tables. Thereby, any attempt (from the guest kernel) to update a guest page table or the guest %cr3 traps to the hypervisor. Based on the trap error, the hypervisor updates the corresponding shadow page table (in the case of a guest page table write attempt) or switches the execution context (in the case of a guest %cr3 write attempt).

By leveraging this mechanism, a WSS estimation technique can monitor a VM's memory activity in a transparent way, in the hypervisor (see Section 2.3).

2.3 On-demand memory allocation

2.3.1 General functioning

As argued in the introduction, the memory is the limiting resource when performing VM collocation. To alleviate this issue, the commonly used approach

[1]The shadow page table's address is loaded into %cr3 at context switch. The CR3 register enables the processor to translate virtual addresses into physical addresses.

consists of managing the memory in the same way as the processor, by doing on-demand allocation. Indeed, considering a VM whose booked memory capacity is m_b (representing the SLA that the provider should meet) but which actively uses m_u ($m_u \leq m_b$), the on-demand approach would assign only m_u memory capacity to the VM (instead of m_b as in a static strategy); m_u is called the WSS of the VM. This approach requires the implementation of a feedback loop which operates as follows. The memory activity of each VM is periodically collected and services as the input of a WSS estimation algorithm. Once the latter has estimated the WSS (noted wss_{est}), the VM's memory capacity is adjusted to wss_{est}. In short, the implementation of the on-demand memory allocation strategy raises thee main questions:

- (Q_1) How to obtain the VM's memory activity knowing that the VM is a "black-box" for the cloud provider?
- (Q_2) How to estimate the VM's WSS from the collected data?
- (Q_3) How to update the VM's memory capacity during its execution?

Regarding Q_3, the solution is self-evident. Indeed, it leverages the balloon driver inside the VM (see the previous section). Furthermore, the hypervisor provides an API to control the balloon driver's size. Thus, by inflating or deflating the balloon, the actual memory capacity of the VM can be updated at runtime. The rest of the section focuses on Q_1 and Q_2, which are more complex.

Answering Q_1 raises two challenges. The first one relates to the implementation of the method used for retrieving the memory activity data. The method is either active or passive. An active method modifies the execution of the VM (e.g. deliberately inject page faults) while a passive method does not interfere in the VM's execution process. The active method could impact the VM's performance. For instance, a naive way for capturing all memory accesses may be to invalidate all memory pages in the VM's shadow page table. All subsequent accesses would result in page faults which are trapped by the hypervisor. This solution would be catastrophic for the VM's performance because of the page faults' overhead. The second challenge is related to the level where the method is implemented. Three locations are possible: exclusively inside the hypervisor/dom0, exclusively inside the VM, or spread across both. In the last two locations, the method is said to be *intrusive* because the "black-box" nature of the VM is altered. In this situation, the implementation of the method requires the end-user's agreement. Otherwise, one could exploit only the memory activity data available at the hypervisor/dom0 level. Concerning Q_2, two main challenges should be tackled: the accuracy of the estimation technique (a

wrong estimation will either impact the VM's performance or lead to resource waste) and the overhead. In the rest of the document, the expression "WSS estimation technique" is used to represent a solution to both Q_1 and Q_2.

2.3.2 Metrics

With respect to the above presentation, the metrics we propose for characterizing a WSS estimation technique are the following: the *intrusiveness* (requires the modification of the VM), the *activeness* (alters the VM's execution flow), the *accuracy*, the overhead on the VM (noted *vm_over*), and the overhead on the hypervisor/dom0 (noted *hyper_over*). Both the *intrusiveness* and the *activeness* are qualitative metrics while the others are quantitative. Among the qualitative metrics, we consider the *intrusiveness* as the most important. We note that the balloon driver alone is not considered an intrusiveness since it is de facto accepted and integrated in most of the OSs. Concerning the quantitative metrics, the ranking is done as follows. Metrics which are related to the VM performance (thus the SLA) occupy higher positions since guaranteeing the SLA is one of the most important provider's objectives. In this respect, we propose the following ranking:

1. *vm_over*: it directly impacts the VM performance. It could be affected by both the *intrusiveness* and the *activeness*.

2. *accuracy*: a wrong estimation leads to either performance degradation (under-estimation) or resource waste (over-estimation).

3. *hyper_over*: a high overhead could saturate the hypervisor/dom0, which are shared components. This could lead, in turn, to the degradation of VMs' performance (e.g. the I/O intensive VMs). In this work we mainly focus on the CPU load induced by the technique.

The metrics presented above characterize the WSS estimation techniques. Apart from these, we also define a metric which characterizes the WSS itself, namely the *volatility*. The latter represents the degree/speed of WSS variation and is very important for the VM consolidation (see Section 2.6.2).

2.4 Studied techniques

This section presents the main WSS estimation techniques proposed by researchers up to the writing time of this document. We have thoroughly studied them both qualitatively and quantitatively. This section focuses on the former aspect while Section 2.5 is dedicated to the latter aspect. The presentation

of each technique is organized as follows. First, we present the technique description, while highlighting how Q_1 and Q_2 are answered. Second, we explain (whenever necessary) the way in which we implement the technique in Xen (our illustrative virtualization system). Last but not least, we present both the strengths and the weaknesses of the technique, knowing that they are validated in Section 2.5.

2.4.1 Self-ballooning

Description. Self-ballooning [63] entirely relies on the VM, especially the native features of its OS. It considers that the WSS of the VM is given by the `Committed_AS` [2] kernel statistic (`cat /proc/meminfo`), computed as follows. The OS monitors all memory allocation calls (e.g. `malloc`) - Q_1 - and sums up the virtual memory committed to all processes. The OS decrements the *Committed_AS* each time the allocated pages are freed. For illustration, let us consider a process which runs the C program presented in Fig. 2.2. After the execution of line 2, the value of *Committed_AS* is incremented by 2GB, even if only one octet is actively used. In summary, the *Committed_AS* statistic corresponds to the total number of anonymous memory pages allocated by all processes, but not necessary backed by physical pages.

Implementation. No effort has been required to put in place this technique since it is the default technique already implemented in Xen. The balloon driver (which runs inside the VM) periodically adjusts the allocation size according to the value of the *Committed_AS*.

Comments. As mentioned above, this technique completely depends on the VM. In addition, the implementation of the feedback loop is shift from the hypervisor/dom0 to the VM, making this technique too intrusive. The heuristic used for estimating the WSS is not accurate for two reasons. First, *Committed_AS* does not take into account the page cache, and thus may cause substantial performance degradation for disk I/O intensive applications [22]. Second, this technique could lead to resource waste since the committed memory is most of the time greater than the actively used memory. These two statements are also validated by the evaluation results. The only advantage of the *Committed_AS* technique is its simplicity.

2.4.2 Zballoond

Description. Zballoond [22] relies on the following observation: when a VM's memory size is larger than or equal to its WSS, the number of swap-in and refault (occurs when a previously evicted page is later accessed) events is close to zero. The basic idea behind *Zballoond* consists in gradually decreasing the

```
void main(void){
        char* tab=(char*)malloc(2*1024*1024*1024);
        do{
                tab[1]=getchar();
        }while(tab[1]!='a');
        free(tab);
}
```

Figure 2.2: The `Committed_AS` value increases with the amount of malloc-ed memory even if it is not backed by physical memory.

VM's memory size until these counters start to become non-zero (the answer of Q_1). Concerning Q_2, the VM's WSS is the lowest memory size which leads the VM to zero swap-in and refault events.

Implementation. *Zballoond* is implemented inside the VM as a kernel module which loops on the following steps. (1) The VM's memory size is initialised to its `Committed_AS` value. (2) Every epoch (e.g. 1 second), the memory is decreased by a percentage of the `Committed_AS` (e.g. 5%). (3) Whenever the `Committed_AS` changes, *Zballoond* considers that the VM's WSS has changed significantly. In this case, the algorithm goes to step (1). Our implementation of *Zballoond* is about 360 LOCs.

Comments. Like the previous technique, *Zballoond* is entirely implemented in the VM's OS. Furthermore, *Zballoond* is very active in the sense that it performs memory pressure on the VM. The overhead introduced by this technique comes from the fact that it actively forces the VM's OS to invoke its page reclamation mechanism (every epoch). Therefore, the overhead depends on both the epoch length and the pressure put on the VM (how much memory is reclaimed).

2.4.3 The VMware technique

Description. The VMware technique [85] is an improvement of the naive method presented in Section 2.3. Instead of invalidating all memory pages, it relies on a sampling approach which works as follows. Let us note m_{cur} the current VM's memory size. To answer Q_1, the hypervisor periodically and randomly selects n pages from the VM's memory (e.g. $n = 100$) and invalidates them. By so doing, the next access to these pages trap in the hypervisor. The latter counts the number of pages (noted f) among the selected ones which were subject to a non present fault during the previous time interval. The WSS of the VM is $\frac{f}{n} \times m_{cur}$, thus answering Q_2.

Implementation. Two implementations of this technique are possible de-

pending on the way the memory pages are invalidated. A memory page can be invalidated by clearing either the present bit or the accessed bit. In the first implementation the hypervisor counts the number of page faults generated by the selected pages while in the second, it counts the number of pages being accessed (the accessed bit is set) during the previous time frame. Notice that the access bit is automatically set by the hardware each time a page is accessed; no trap is triggered in the hypervisor. The implementation of the two methods requires around 160 LOCs.

Comments. This technique is completely non intrusive. The feedback loop is entirely implemented in the hypervisor/dom0. However, the technique has two main drawbacks. First, the method used for answering Q_1 modifies the execution flow of the VM, which could lead to different performance degradation levels depending on the adopted implementation. The first implementation leads to higher performance degradation comparing to the second implementation. This is explained by the cost of resolving a non-present page fault which is higher than the cost of setting the accessed bit (performed in the hardware). However, the accuracy of the second implementation (the number of accessed pages) could be biased if the hypervisor/dom0 runs another service which clears the accessed bit. Such a situation could occur in a KVM environment because the hypervisor (i.e. Linux) runs services like kswapd (the swap daemon) which monitors and clears the accessed bit. As a second drawback, this techniques is unable to estimate WSSs greater than the current allocated memory. In the best case, the technique will detect that all monitored pages are accessed, thus estimating the WSS as the current size of the VM.

2.4.4 Geiger

Description. *Geiger* [47] monitors the evictions and subsequent reloads from the guest OS buffer cache to the swap device (the answer of Q_1). To deal with Q_2, Geiger relies on a technique called the ghost buffer [74]. The latter represents an imaginary memory buffer which extends the VM's physical memory (noted m_{cur}). The size of this buffer (noted m_{ghost}) represents the amount of extra memory which would prevent the VM from swapping-out. Knowing the ghost buffer size, one can compute the VM's WSS using the following formula: $WSS = m_{cur} + m_{ghost}$ if $m_{ghost} > 0$.

Implementation. The first challenge was to isolate the swap traffic from the rest of the disk IO requests. In this respect, we forced the VM to use a different disk backend driver for the swap device (e.g. *xen-blkback*). This driver is patched to implement the *Geiger* monitoring technique as follows. When a page is evicted from the VM's memory, a reference to that page is added to a tail queue in the disk backend driver, located inside the dom0. Later, when a

page is read from the swap device, *Geiger* removes its reference from the tail queue and computes the distance D to the head of the queue. D represents the number of extra memory pages needed by the guest OS to prevent the swapping out of that page (i.e. the ghost buffer size at that timestamp). However, to update the VM's memory size after each reloaded page from swap would be too frequent. Thereby, we leverage D values to compute the miss ratio curve [74]. This curve is an array indexed by D which represents how many times we saw the D distance in the last interval. For example, if the computed $D = 50$, we increment `array[50]` by one. When the timer expires, we iterate through the array and we sum up its values until we got X% of its total size. In our implementation, we found out that $X = 95$ yields good results. The index corresponding to the position where the iterator stops represents the number of extra memory pages needed by the VM to preserve 95% of swapped out pages.
Comments. Like the *VMware* technique, *Geiger* is also completely transparent from the VM's point of view. Thereby it does not require the VM user's permission. As stated before, the VM has to be started with a different disk backend driver for the swap device. However, this is not an issue since the VMs are created by the cloud provider who is also the one deciding the disk backend drivers to be used. Additionally, Geiger has an important drawback which derives from its non-intrusiveness. It is able to estimate the WSS only when the size of the ghost buffer is greater than zero (the VM is in a swapping state). Geiger is inefficient if the VM's WSS is smaller than the current memory allocation.

2.4.5 Hypervisor Exclusive Cache

Description. The *Exclusive Cache* technique [60] is fairly similar with Geiger in the way that both of them rely on the ghost buffer to estimate the WSS. In the *Exclusive Cache*, each VM has a small amount of memory called direct memory, and the rest of the memory is managed by the hypervisor as an exclusive cache. Once the direct memory is full, the VM will send pages to the hypervisor memory (instead of sending to the swap). Thereby, in the Exclusive Cache technique, the ghost buffer is materialized by a memory buffer managed in the hypervisor.
Implementation. In the same way as Geiger, the Exclusive cache technique is also implemented as an extension to the XEN disk backend driver. In the vanilla driver, the backend receives the pages to be swapped through a shared memory between the VM and dom0. Subsequently, the backend creates a block IO request that is passed further to the block layer. In our implementation, instead of creating the block IO request, we store the VM's page content in

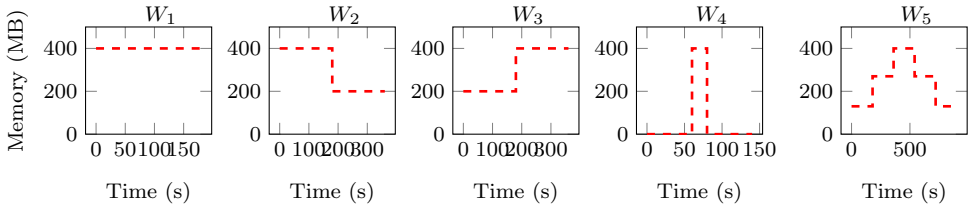


Figure 2.3: The set of synthetic workloads.

a dom0 memory buffer. The latter represents the materialization of the ghost buffer.

Comments. In comparison with Geiger, this technique is more active since it may force the VM in eviction state. However, the performance impact of the *Exclusive cache* technique is lower since the block layer is bypassed and the evicted pages are stored in memory.

2.4.6 Dynamic MPA Ballooning

Description. The Dynamic Memory Pressure Aware (MPA) Ballooning [51] studies the memory management from the perspective of the entire host server. It introduces an additional set of hypercalls through which all VMs report the number of their anonymous pages, file pages and inactive pages to the hypervisor (Q_1). Based on this information, the technique defines three possible memory pressure states: *low* (the sum of anonymous and file pages for all VMs is less than the host's total memory pages), *mild* (the sum of anonymous and file pages is greater than the host's total memory pages) and *heavy* (the sum of anonymous pages is greater than the host's total memory pages); this answers Q_2. Depending on the current memory pressure state, the host server adopts a different memory policy. In the case of low memory pressure, this technique divides the hypervisor's free memory to $nb_{VMs} + 2$ slices. Each slice (called *cushion*) is assigned to a VM as a memory reserve. The two remaining cushions stay in the control of the hypervisor for a sudden memory demand. The cushion may be seen as the exclusive cache in the Hypervisor Exclusive Cache technique. In the mild memory pressure state, the hypervisor reclaims the inactive pages from all VMs and rebalance them in $nb_{VMs} + 1$ cushions. In heavy memory pressure, most of the page cache pages are evicted so the technique rebalance exclusively the anonymous pages.

Comments. This technique has high intrusiveness since it requires additional hypercalls in the guest OS. Thereby, it may be effective in the case of a private data center where the cloud manager has a high degree of control over the guest OS. Additionally, the new hypercalls export precise and important information about the VM's memory layout; this may increase the risk of attacks on VMs.

2.5 Evaluation of the studied techniques

This section presents the evaluation results for most of the techniques described above. We do not evaluate the Dynamic MPA Ballooning since is not a WSS estimation technique. The memory utilization values are directly communicated by the VM to the hypervisor.

2.5.1 Experimental environment

The experiments were carried out on a 2-socket DELL server. Each socket is composed of 12 Intel Xeon E5-2420 processing units (2.20 GHz), linked to a 8GB NUMA memory node (the machine has a total of 16GB RAM). The virtualization system on the server is Xen 4.2. Both the dom0 and the VMs run Ubuntu server 12.04. One socket of the server is dedicated to dom0 in order to avoid interference with other VMs. Unless otherwise specified each VM is configured with two vCPUs (pined to two processing units) and 2GB memory (the maximum memory it can use).

Concerning the applications which run inside VMs, we rely on both micro and macro benchmarks. The former is an application which performs read and write operations on the entries of an array whose size could be dynamically adjusted in order to mimic a variable workload. Each array entry points to a data structure whose size is equivalent to a memory page. The micro-benchmark allows to compare experimental values with the exact theoretical values, necessary for evaluating the *accuracy* metric. To this end, we build five synthetic workloads which cover the common memory behaviors of a VM during its lifetime. Fig. 2.3 presents these workloads, noted W_i, $1 \leq i \leq 5$. Each workload is implemented in two ways. In the first implementation (noted $W_{i,s}$), the array size is malloced once, at VM start time, to its maximum possible value. In the second implementation (noted $W_{i,d}$), the array's allocated memory size is adjusted to each step value.

In addition, we also rely on three macro-benchmarks, namely DaCapo [19], CloudSuite [37], and LinkBench [12]. DaCapo is a well known open source java benchmark suite that is widely used by memory management and computer architecture communities [93]. We present the results for 5 DaCapo applications which are the most memory intensive:

- Avrora is a parallel discrete event simulator that performs cycle accurate simulation of a sensor network.
- Batik produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.

- Eclipse executes some of the (non-gui) jdt performance tests for the Eclipse IDE.
- H2 executes a JDBC-like in-memory benchmark, executing a number of transactions against a model of a banking application.
- Jython inteprets the PyBench python benchmark

CloudSuite is a benchmark suite which covers a broad range of application categories commonly found in today's datacenters. In our experiments, we rely on Data Analytics, a map-reduce application using Mahout (a set of machine learning libraries). LinkBench is a database benchmark developed to evaluate database performance for workloads similar to those at Facebook. The performance metric of all these applications is the complete execution time. By choosing these benchmarks, we wanted to cover the most important and popular applications executed in the cloud nowadays.

2.5.2 Evaluation with synthetic workloads

As stated above, these evaluations focus on the *accuracy* metric. Fig. 2.4 and Fig. 2.5 present the results for each workload and each WSS estimation technique. To facilitate the interpretation of the results, each curve shows both the original workload (noted W_i^o) and the actual estimated WSSs (noted W_{ij}^e), $1 \leq i \leq 5$ (represents the workload type) and j=s,d (represents the implementation type - static or dynamic).

Xen *self-ballooning*. Fig. 2.4 line 1-2. The accuracy of this technique is very low for all $W_{i,s}$ (see line 1) while it is almost perfect for all $W_{i,d}$ (see line 2). This is because the technique relies on the value of `Committed_AS` as the WSS. Thus, it is able to follow all `Committed_AS` changes. The accuracy of this technique depends on the implementation (i.e. the memory allocation approach) of applications which run inside the VM.

***Zballoond*.** Fig. 2.4 line 3-4. This technique behaves like *self-ballooning* on all $W_{i,d}$ (see line 4) because it tracks all `Committed_AS` changes. Unlike *self-ballooning*, *Zballoond* is also quite efficient on all $W_{i,s}$ (see line 3). This is because *Zballoond* continuously adjusts the VM's memory size so that swap-in or refault events occur, thus avoiding resource waste. However, if the WSS reduction is faster than the memory reclaim percentage (i.e. 5%), the estimation diverges from the real WSS (see line 3, columns 2 and 4). Even if a higher memory reclaim percentage may solve the problem, this means more memory pressure on the VM and thereby, it would increase the *vm_over*.

		Self-ballooning		*Zballoond*		$VMware_{present}$	
Benchmark and app.		**vm_over**	**hyp_over**	**vm_over**	**hyp_over**	**vm_over**	**hyp_over**
Dacapo	avrora	1	1	1.19	1	2.77	1.06
	batik	1	1	1.09	1	15.44	2.0
	eclipse	1	1	3.67	1	18.79	1.01
	h2	1	1	2	1	24.12	2.05
	jython	1	1	1.58	1	21.42	1.16
Cloud suite	Data Anal.	1	1	1.4	1	45.05	2.06
LinkBench	MySQL	1	1	2.92	1	20.17	1
		$VMware_{access}$		*Geiger*		*Exclusive Cache*	
Benchmark and app.		**vm_over**	**hyp_over**	**vm_over**	**hyp_over**	**vm_over**	**hyp_over**
Dacapo	avrora	2.14	1.1	1.22	1.2	1	5.06
	fop	13.06	2.2	1.41	1.32	1.5	5.6
	h2	15.63	1	1	1.02	1	5.0
	jython	20.51	2	1.12	1.5	1.7	4.9
	luindex	18.2	1.5	1.04	1.45	1.08	5.52
Cloud suite	Data Anal.	40.22	1.06	1.15	1.22	2.03	6.04
LinkBench	MySQL	19.22	2	1.76	1.09	1.80	5.2

Table 2.1: Evaluation results of each technique with macro-benchmarks.

From now on (Fig. 2.5), we only discuss $W_{i,s}$ results because we observed no difference with $W_{i,d}$ regardless the WSS technique. In fact, only `Committed_AS`-based techniques are sensitive to the way by which the workload is implemented.

VMware. Fig. 2.5 line 1. Without access to the implementation details of this technique, we considered two versions according to the way the sampled pages are invalidated: the present bit based version (noted $VMware_{present}$) and the access bit based version (noted $VMware_{access}$). The evaluation results of these versions show that they have almost the same accuracy. They are only different from the perspective of other metrics (see the next section). From Fig. 2.5 line 1, we can see that the *VMware* technique has a main limitation. Although it is able to detect WSS when the VM is wasting memory, it is not able to detect shortage situations. This happens because the percentage of memory pages (among the sampled ones) which is used for estimating the WSS is upper bounded by 100%.

Geiger. Fig. 2.5 line 2. *Geiger* is the opposite of the *VMware* technique; it is only able to detect shortage situations. This is because it monitors the swap-in and refault events, which only occur when the VM is lacking memory. Another advantage of this technique is its reactivity; it quickly detects WSS changes.

Hypervisor exclusive cache. Fig. 2.5 line 3. This technique behaves like *Geiger* in the perspective of the *accuracy* metric. They are different in terms of the *vm_over* metric presented in the next section.

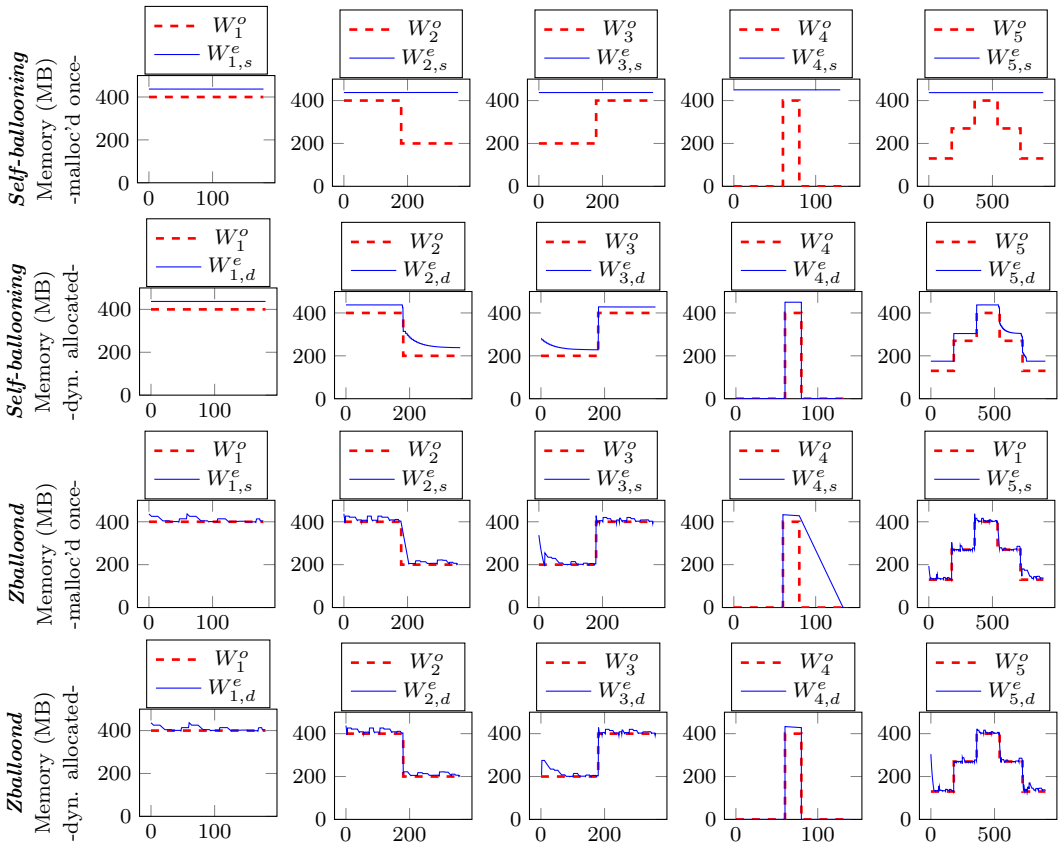


Figure 2.4: Evaluation results of *self-ballooning* and *Zballoond* with synthetic workloads. The original workload is noted W^o_i while the actual estimated WSSs are noted W^e_{ij}. "j" is s (the static implementation) or d (the dynamic implementation).

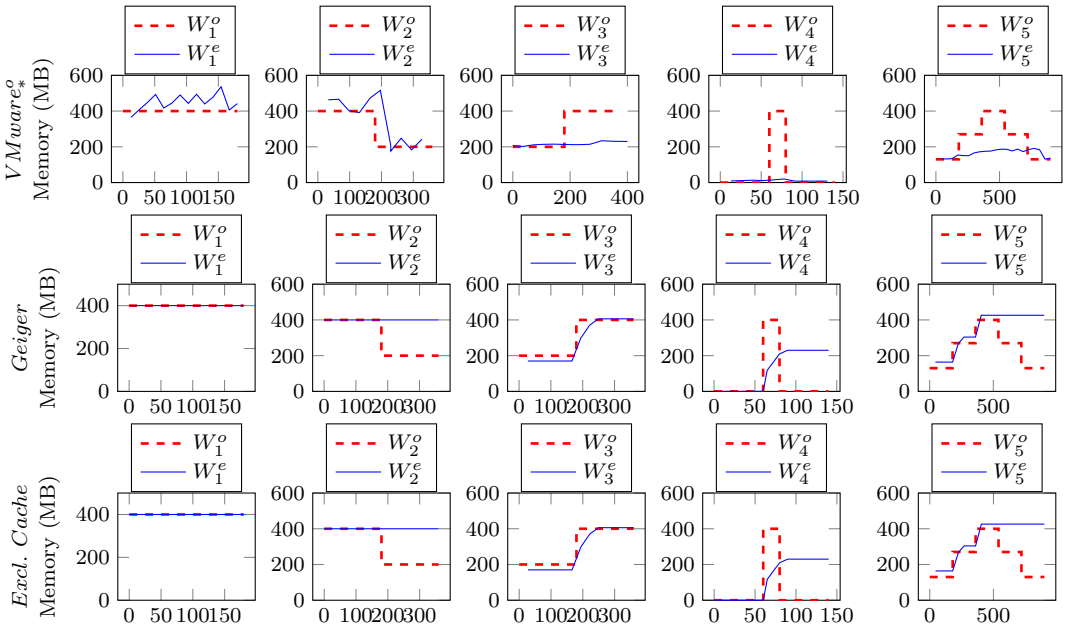


Figure 2.5: Evaluation results of *VMware*[2], *Geiger*, and *Exclusive cache* with synthetic workloads.

2.5.3 Evaluation with macro-benchmarks

Table 2.1 presents the evaluation results of each technique with macro-benchmarks. We only focus on the *vm_over* and the *hyper_over* metrics. The *vm_over* value represents the normalized runtime performance of each benchmark while the *hyper_over* represents the normalized CPU utilization by the hypervisor. For example, $vm\_over = 2$ means that the benchmark execution time is twice longer. The interpretation of Table 2.1 is as follows.

Self-ballooning. It incurs no overhead neither on the hypervisor/dom0 nor on the benchmark.

Zballoond. Like *self-ballooning*, it incurs no overhead on the hypervisor/-dom0. However, the VMs' performance is impacted (between 1.09x and 3.67x).

VMware. We can see that the two versions we implemented ($VMware_{present}$ and $VMware_{access}$) incur a relatively low overhead on the hypervisor/dom0. However, the two versions severely impact the benchmark performance (up to 45x degradation in the case of the Data Analytics applications). As presented in the previous section, this is due to the fact that the *VMware* technique is

[2]The accuracy of the *VMware* method is orthogonal to the implementation approach thereby, it is represented only once.

not able to detect memory lacking situations. $VMware_{present}$ leads to more impact on VMs than $VMware_{access}$ (about 3x).

Geiger. Its overhead on either the hypervisor/dom0 or the VM is negligible (less than 2x). Even if the technique does not entirely address the issue of WSS estimation, the VM performance is not strongly impacted since Geiger never leads the VM to a lacking situation like the *VMware* technique.

Exclusive cache. Its overhead on the hypervisor/dom0 is not negligible (about 5x). However, its impact on the VM performance is almost nil (swapped-out pages are store in the main memory).

2.5.4 Synthesis

Table 2.2 summarizes the characteristics of each technique according to both qualitative and quantitative criteria presented in Section 2.3.2. Besides these criteria, the evaluation results reveal that not all solutions address the issue of WSS estimation in its entirety. Indeed, a WSS estimation technique must be able to work in the following two situations:

- (S_{more}) the VM is wasting memory,
- (S_{less}) the VM is lacking memory.

The *VMware* technique [85] is only appropriate in (S_{more}) while *Geiger* and *Hypervisor exclusive cache* are effective in (S_{less}). Only *Zballoond* and *self-ballooning* cover both (S_{more}) and (S_{less}). Our study also shows that each solution comes with its strengths and weaknesses. The next section presents our solution.

2.6 Badis

2.6.1 Presentation

The previous section shows that the WSS estimation problem is addressed by a wide range of solutions. However, to the best of our knowledge, none of them are consistently adopted in the mainstream cloud. We assert that one reason which leads the cloud customers to the denial of such solutions is their intrusiveness (both from the codebase and from the performance perspective). This is confirmed by our cloud partner, Eolas [3]. We claim that a solution easily adopted in the mainstream cloud should provide (1) no codebase intrusiveness and (2) low performance impact. In order to reduce the performance impact

	Self-b.	*Zballoond*	*VMware*	*Geiger*	*Excl. Cache*
intrusive	yes	yes	no	no	no
active	no	yes	yes	no	yes
addressed situations	all	all	S_{more}	S_{less}	S_{less}

	Self-b.	*Zballoond*	*VMware*	*Geiger*	*Excl. Cache*
accuracy	depends on the app.	high	high in S_{more} zero in S_{less}	high in S_{less} zero in S_{more}	high in S_{less} zero in S_{more}
vm_over	nil	almost nil	nil in S_{more} high in S_{less}	almost nil	almost nil
hyper_over	nil	nil	almost nil	almost nil	not negligible

Table 2.2: Study synthesis of all WSS estimation techniques according to both qualitative (left) and quantitative (right) metrics.

the solution should provide high accuracy and thereby, address both (S_{more}) and (S_{less}).

This section presents *Badis*, a system which smartly combines existing techniques in such a way that both (S_{more}) and (S_{less}) are covered with no codebase intrusiveness. Indeed, we found that even if the *VMware* and *Geiger* solutions have a fairly high performance impact they have no intrusiveness in the VM's codebase. The second observation is that these solutions are complementary (*VMware* addresses S_{more} while *Geiger* addresses S_{less}). The *Hypervisor exclusive cache* is also a solution that only addresses (S_{less}) but it has higher *hyper_over*. Thereby, a system which is able to combine *VMware* and *Geiger* satisfies all our requirements.

Fig. 2.6 top presents the architecture of our system. The *VMware* technique is implemented at the hypervisor level while *Geiger* as well as the feedback loop decision module are located inside the dom0. Concerning the *VMware* technique, we rely on the accessed bit instead of the present bit for memory page invalidation. The former introduces less overhead on the VM than the latter. The decision module is implemented as a kernel module inside the dom0, thus keeping the hypervisor as lightweight as possible. The communication between *Geiger* and the decision module is straightforward since they both run inside the dom0. Concerning the *VMware* technique, it communicates with the decision module via a shared memory established between the dom0 and the hypervisor. To this end, we extend the native Xen `share_info` data structure, which implements the shared memory used by the hypervisor to provide the VM with hardware information necessary at VM boot time (e.g. the memory size). Having described the mechanisms which allow the global functioning of our system, let us now present how the two WSS estimation techniques are leveraged.

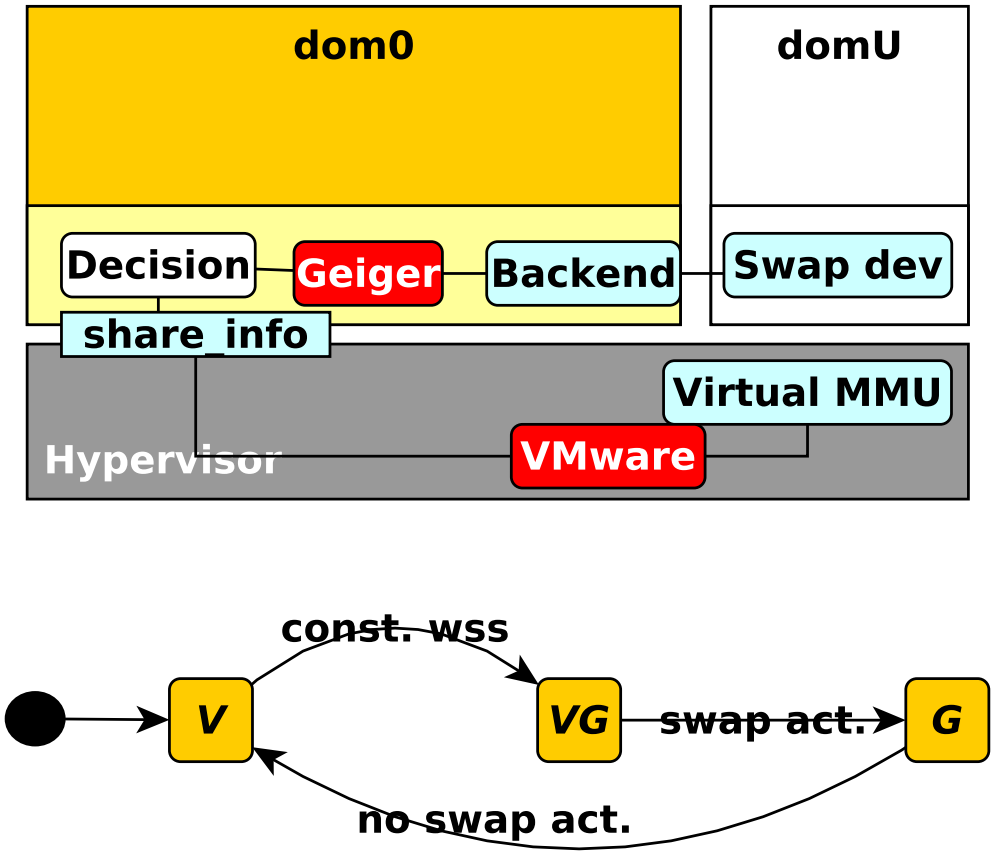


Figure 2.6: (top) The architecture of Badis. (bottom) The finite-state machine used to track a VM's WSS in Badis.

For each VM, the system implements a 3-state finite state machine (FSM), as shown in Fig. 2.6 bottom. Once setup, the VM enters the V state in which the WSS is estimated using the *VMware* technique (*Geiger* is disabled). In fact, it is more likely that the memory allocated to the VM at boot time (booked by its owner) is larger than its WSS. While in the V state, if the estimated WSS moves closer to the VM's allocated memory, the FSM transitions to the VG state in which *Geiger* is enabled. While in the VG state, the WSS of the VM is given by the *VMware* technique if *Geiger* does not measure any swap activity. Otherwise, the WSS is given by *Geiger*. The FSM transitions from VG to the G state (in which the *VMware* technique is disabled) when *Geiger* reports swap activities during two consecutive rounds. Finally, the transition from G to V is triggered if *Geiger* does not observe any swap activity during two consecutive rounds. One may doubt the need of VG state. However, we consider it necessary because of a more subtle VMware limitation. As presented before, VMware chooses a set of sample pages and based on the number of pages accessed during an observation interval, it computes the WSS as a percentage of the total memory. For example, if VMware chooses 100 sample pages and 60 of them are accessed, it concludes that the WSS size is 60% of the total VM's memory. However, in most of the cases this is wrong and not only because of the estimation error. The VMware technique considers all pages equal and swappable. Nevertheless, some of the pages are pinned down by the OS. If they are not accessed during a VMware observation interval, they are considered out of the working set. When the memory is adjusted to the WSS the OS cannot swap out this pinned pages and thereby, it has to chose from the active pages. This issue is an important source of performance degradation.

Further we will present how Badis cope with this problem. When in VG, the VM is in a swapping state which means that all of its allocated memory is necessary. In this state we still continue to read estimations from the *VMware* technique which theoretically should be 100% (i.e. all pages are accessed during a time frame). However, the estimations are generally less than 100% (e.g. 80%) because of the pinned pages which are inactive. The difference to 100% (e.g. 20%) should also be included in the working set because, even if these pages are inactive, they cannot be swapped-out. This correctional value is stored and leveraged later, in the V state, for a conclusive estimation. The next section presents the way our estimation system is leveraged in a virtualized cloud.

2.6.2 Badis in a virtualized cloud

In the last section we presented the advantages of Badis over the state-of-the-art. However, one may ask which are the benefits of WSS estimation in the

cloud? Clearly, there is no benefit in shrinking a VM's memory unless there is some other VM ready to make use of that. Thereby, the WSS estimation should be integrated in a higher level system that has a wide image on the datacenter's compute resources. Such a system is the cloud manager (e.g. OpenStack [5]) which is the one controlling the VM lifecycle and taking consolidation decisions.

Generally, the factor that limits the server consolidation is memory, for two main reasons. The first one is the *the memory capacity wall* presented in [57]. Second, in most of the virtualization systems, the booked memory (m_b) is entirely allocated when the VM is booted. This quantity should meet the highest possible memory demands the VM will have during its lifetime. However, most of the time, the memory demands are lower than m_b which implies some degree of memory waste (see Fig. 2.7). **The WSS estimation could help improving the memory efficiency and thereby, increase the consolidation ratio.** However, in some circumstances, the server consolidation based on the VMs' current WSS estimation may do more harm than good. If a recently consolidated VM requests more memory than available on the hosting server, it should be migrated back on a server which can provide enough memory. This excessive VM dynamics may increase the datacenter's energy consumption [59] and impact the hosted applications' performance [84]. Thereby, the research question is: how to leverage the WSS estimation techniques not only for a better but also for a stable consolidation? Further we will present our solution to this problem.

Our solution is implemented as an extension to a popular consolidation system, namely OpenStack Neat [6]. The latter takes consolidation decisions when a server is (1) underloaded or (2) overloaded. In the first case it relocates all VMs in order to free up the server and switch it to a lower energy state. In the latter case it migrates one VM, generally the one with the smallest allocated memory, to reduce the migration time. We mention that Neat places VMs based on the booked memory and not the WSS estimation. In order to decide when a server is underloaded or overloaded, Neat has a data collection module that fetches the CPU utilization of all VMs and stores the data in both, the local datastores on each physical server and a global datastore for the entire datacenter. However, since Neat does not overcommit memory, it does not collect any memory utilization data. The underload and overload detection algorithms only take into account the CPU. Further we will present how Badis adjusts a VM's allocated memory based on its WSS.

First, Badis continuously computes the moving average of the last n WSS estimation samples (e.g. $n = 5$). We monitor the moving average of each WSS using time slices of size s (e.g. $s = 1$ *hour*). The allocated memory of VM *id_vm* is adjusted to *the maximum value of the moving average in the*

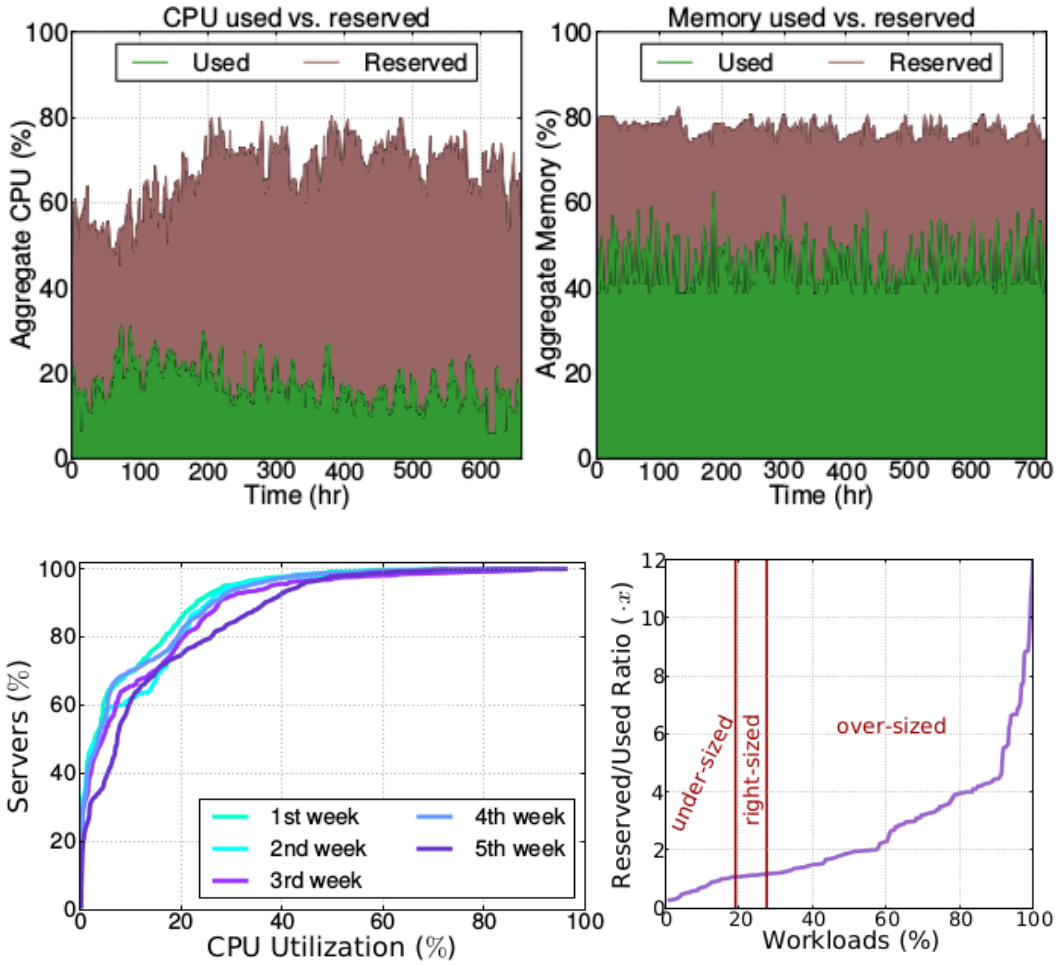


Figure 2.7: "Resource utilization over 30 days for a large production cluster at Twitter managed with Mesos. (a) and (b): utilization vs reservation for the aggregate CPU and memory capacity of the cluster; (c) CDF of CPU utilization for individual servers for each week in the 30 day period; (d) ratio of reserved vs used CPU resources for each of the thousands of workloads that ran on the cluster during this period." [29]

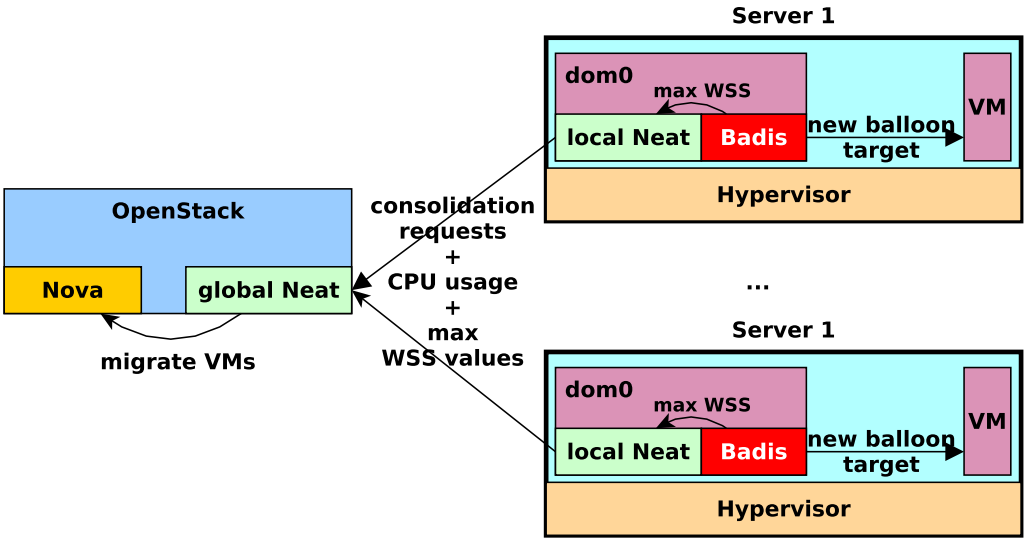


Figure 2.8: The integration of Badis in OpenStack. Badis estimates the WSS and sets the *id_vm*'s allocated memory to $WSS_{id\_vm}^{max\_avg}$. It also transmits $WSS_{id\_vm}^{max\_avg}$ values to the local Neat. The latter collects these values along with the CPU loads and sends them in batches to the global Neat. The local Neat may also send consolidation requests to the global Neat in the case of CPU/RAM overload/underload. These consolidation requests are decomposed into individual VM migrations which are executed by OpenStack Nova.

last time slice, noted $WSS_{id\_vm}^{max\_avg}$. The latter value is also transmitted to the data collection module (see Fig. 2.8). We have modified the Neat's underload and overload detection algorithms to also take into account the memory load and pack the VMs based on $WSS_{id\_vm}^{max\_avg}$. Since $WSS_{id\_vm}^{max\_avg} \leq m_b$, the VM packing is tighter. If the allocated resources of all VMs on a server overpasses the underload or the overload threshold, Neat will trigger a new consolidation round. However, the volatility of the memory load is generally lower than the CPU. In our experiments only 3% of the consolidation rounds were triggered because of the memory load (see Section 2.6.3).

2.6.3 Evaluations

The experimental environment is the same as presented in Section 2.5. We evaluated our solution with both micro and macro benchmarks.

Micro-benchmark based evaluations. We first validated the effectiveness of our solution using a synthetic workload, see the dashed blue curve in Fig. 2.9. This workload includes situations a WSS estimation technique should

		Self-ballooning	*Zballoond*	Badis	
Benchmark and app.		**vm_over**	**vm_over**	**vm_over**	**hyper_over**
Dacapo	avrora	1	1.19	1.26	1.8
	batik	1	1.09	1.57	1.05
	eclipse	1	3.67	1	1.68
	h2	1	2	1.16	1.3
	jython	1	1.58	1.05	1.15
Cloud suite	Data Analytics	1.29	1.4	1.16	1.2
LinkBench	MySQL	1.11	2.92	1.09	1

Table 2.3: Evaluation of our solution with macro-benchmarks, and comparison with two existing solutions.

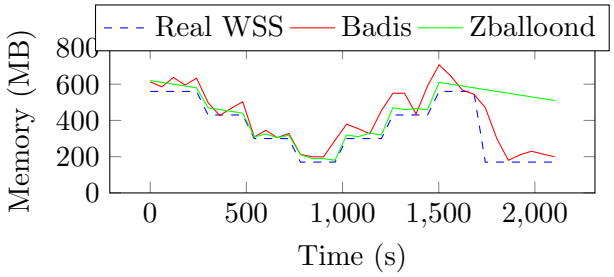


Figure 2.9: Badis and Zballoond evaluated with a synthetic workload.

cope with. One can observe that the accuracy of our solution is comparable with Zballoond but without any VM codebase intrusiveness. In the last part of Fig. 2.9 we can observe a case where our solution even outperforms Zballoond: the WSS drops quickly and the inactive pages are still allocated. In this case Badis is able to quickly track the new WSS while Zballoond slowly decreases the WSS leading to a lot of resource waste.

Macro-benchmark based evaluations. We also evaluated our solution with macro-benchmarks, see Table 2.3. The latter focuses on the *hyper_over* and the *vm_over* metrics since the *accuracy* metric has been evaluated above. We compare our solution with the only solutions which address the issue of WSS estimation in its entirety, namely *self-ballooning* and *Zballoond*. We can see that our solution leads to a negligible overhead on both the VM and the hypervisor/dom0 (less than 2x).

Simulations on traces from a Google datacenter. In the last sections we have demonstrated the capability of our solution to follow the WS variation with high precision. This section will show the effect of WSS estimation on the VM consolidation. In this respect, we leverage traces from a Google datacenter [4]. They represent the execution of thousands of jobs on a cluster of about 12,5k servers, monitored for about 29 days. Each job can be composed of several tasks and each task runs inside a container. For each container, the traces provide data such as the creation time, the destruction time, the amount of CPU/memory requested at creation time. Moreover the traces provide the

amount of CPU/memory actually assigned to the container[3]. By relying on GloudSim [30] (a cloud simulator with VMs based on Google traces) we have simulated both, a consolidation based on the booked memory and a consolidation based on the actually assigned memory. In the first case the datacenter has an average of 9562 active servers while in the second case the average number of active servers is 4676. These figures prove that the memory is indeed the resource which limits the VM consolidation. In the second consolidation type, the packing ratio is more than 2x higher. Regarding the VM dynamics, there were executed around 2.5M migrations in total. Only 75k migrations (i.e. 3.17%) were caused by memory overload/underload. These results prove that the memory volatility is net inferior to the CPU volatility. However, the paradox is that most of the popular consolidation systems overcommit CPU but not RAM memory. Our evaluation results are totally reproducible using the code provided at [7].

2.7 Related work

The reader should refer to Section 2.4 for the presentation of the main WSS estimation techniques in virtualized environments. In this section we focus on other studies related to the concept of WSS, memory management and VM consolidation in a virtualized datacenter.

Working set size estimation. WSS estimation [67] could require large data collection and complex processing. Weiming Zhao et al. [92] have introduced a working set size estimation system which computes a VM's WSS based on its miss-ratio curve (MRC). The latter shows the fraction of the cache misses that would turn into cache hits if the VM's allocated memory increases. Moreover, Weiming Zhao et al. have evaluated the overhead of their solution by providing the relationship between performance and allocated memory size. Pin Zhou et al. [94] have proposed two similar methods which dynamically track the MRC of applications at run time. These techniques represent the hardware and the software implementations of the Mattsons stack algorithm. The latter relies on a "stack" which stores the references to accessed pages (the most recently used page is on the top of the stack). Similarly to the ghost buffer, this algorithm computes the miss ratio curve based on the distance to the top of the stack. Carl Waldspurger et al [86] have proposed an approximation algorithm that reduces the space and time complexity of reuse-distance analysis. This algorithm is appropriate for online MRC generation due to its modest resource requirements.

Memory optimization techniques. Memory deduplication is one of the

[3]The sampling time interval for this data is around 5 minutes.

most popular memory optimization techniques. It consists in merging identical memory pages by keeping only one copy of it. This is mostly useful in case of read-only pages that stay unchanged during the VM run time. Depending on the algorithm used to identify similar pages, there are several implementations of page sharing [20, 69, 86, 44]. These techniques are often combined with memory compression tools to achieve better optimization rates [81, 75, 91]. Another memory optimization tool is the transcendent memory [64] which gathers the VMs' idle memory and the VMM non-allocated memory to a common pool.

Memory balancing is a memory optimization technique, that tries to adjust the VM's allocated memory depending on its necessities. Memory ballooning is the main concept behind this approach. The balancing techniques typically rely on working set size estimation techniques to optimize the memory usage [93]. In a latter work, Zhao et al. [87] leverages inexpensive working set tracking systems to correctly estimate the working set size for the Memory Balancer (MEB) [93]. Xiaoqiao Meng et al. [68] leverage the concept of statistical resource multiplexing between multiple VMs. Specifically, this paper proposes to form pairs of VMs that have complementary temporal behavior (i.e. the peaks of one VM coincide with the valleys of the other). Thereby, if consolidated together, the unused resources from the VM with low demands could be lent to the VM with high demands. These pairs of VMs are found out by computing the correlation between all combinations of two VMs in the datacenter. As one can notice, this approach requires high amount of computation even for small datacenters.

Improving Memory balancing drawbacks. Memory balancing techniques have several drawbacks. First, in the case where several VMs reach their respective memory limit simultaneously, they will all generate a high amount of I/O requests which may saturate the secondary storage. On the other hand, memory balancing is not aware of the hosted applications. Thus, memory intensive applications (e.g. database engines) face serious issues because of memory balancing techniques. To overcome these issues, [78] extends the VM memory ballooning to user level, for applications that manage their own memory.

VM consolidation. The VM consolidation is an NP hard problem [50]. Thereby, numerous papers came up with heuristics for this problem [52, 16, 8, 48]. However, few of these projects provide real implementations to the proposed algorithms [6, 36]. Among the implemented systems, to the best of our knowledge, no system consistently performs memory overcommitment. Even if memory is the main consolidation impediment, most of the existing systems consolidate the VMs based on their booked memory and not on the actually used memory. In this work, we propose a system that monitors the

WSS of VMs and takes consolidation decisions based on the observed memory utilization.

2.8 Conclusion

In this work, we presented a systematic review of the main WSS estimation techniques, namely *Self-ballooning*, *Zballoond*, *VMware*, *Geiger* and *Hypervisor exclusive cache*. From far of our knowledge, this is the first work which deeply compares existing WSS techniques. To this end, we propose a set of qualitative and quantitative metrics allowing the classification of these techniques and we evaluate each technique using both micro and macro benchmarks. The evaluation results reveal the strengths and the weaknesses of each technique. More important, they show that not all solutions address the issue in its entirety. Unfortunately, those which entirely address the issue are intrusive, thus requiring the permission of the VM's owner. This is unacceptable from the datacenter operator's point of view. We also propose Badis, a system which combines several of the existing solutions, using the right solution at the right time. In addition, we have implemented a consolidation extension which leverages Badis for an improved consolidation ratio. The evaluation results reveal a 2x better consolidation ratio with only 3% additional VM migrations.

Chapter 3

Local Memory Mutualization Based on Badis

3.1 Introduction

This chapter aims to present a cooperative memory management system (CMMS) that tackles the challenge of dynamic memory allocation. The basic idea is to reclaim memory from over-provisioned VMS in order to provide it to under-provisioned VMs. Three scientific applications, with different memory behaviours, have been studied to evaluate the effect of CMMS on the performance of applications.

3.2 Background

Working set estimation is the critical issue for providing dynamic memory allocation. The working set measures statistically the amount of memory pages that are currently actively in use. We have used the working set estimation technique described in chapter 2. We assume here that the amount of memory equal to the working set size (WSE) should be practically enough for a VM to run without performance degradation. Dynamic memory adjustment is reached via the memory ballooning technique. A balloon driver is installed in every VM and it is allowed to allocate memory pages, so that these pages are reclaimed from the VM and given to the Virtual Machine Manager (VMM). Then the VMM can decide how to distribute freed memory pages.

3.3 Motivation

A majority of datacenters relies on static allocation rather than dynamic allocation, which results in a huge amount of memory wasting. Figure 3.1 shows

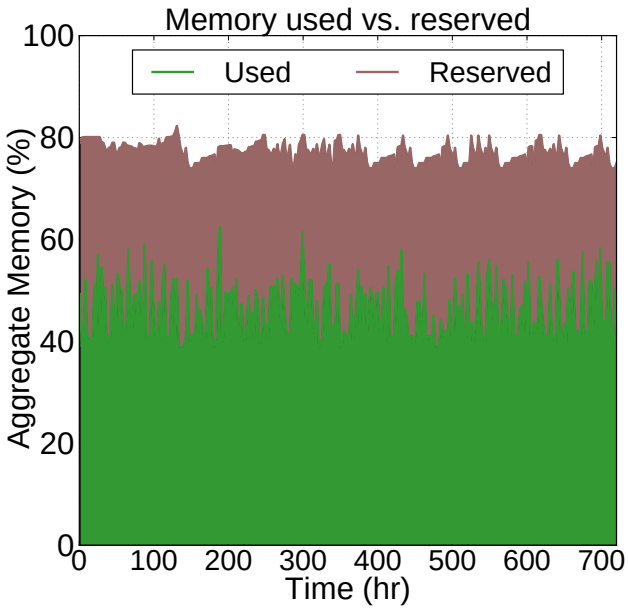


Figure 3.1: : Memory utilization over 30 days for Twitter datacenter managed with Mesos

the memory utilization in a Twitter datacenter managed with Mesos. It illustrates that the average memory consumption is at the level of 50%, which means that almost half of the memory is wasted most of the time. VMs are over-provisioned because it is anticipated that at some times, applications will have short memory peaks and memory is allocated to prevent this issue. Studies [49, 82] shows that such a usage of memory is common to most datacenters and most of the servers are underutilized. Thus an efficient memory management system is required to tackle this issue.

3.4 Contribution

Figure 3.2 illustrates our cooperative memory management system consisting of three main parts: Working set estimation (WSE) technique which periodically calculates the working set size of each VM and updates the values. The memory manager adjusts the memory size of VMs according to the new working set values. If a VM is over-provisioned, then unused memory (according to the working set) is reclaimed and sent to the free memory pool. In case of memory shortage of a VM, the memory needs can be satisfied from the free memory pool. The system guarantees that at least the VM's initially allocated memory size is allocated in case of memory shortage and some extra memory can be allocated if the free memory pool is not empty.

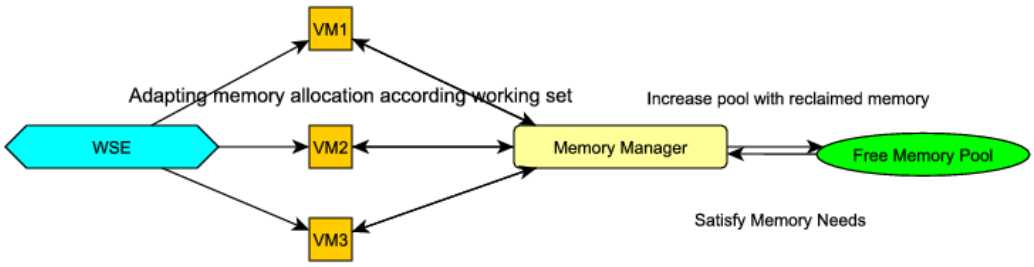


Figure 3.2: Memory Management system schematic illustration

The system maintains 2 variables for each VM: initial allocation (Mem_{init}) and current allocation (Mem_{act}). The VMs are distinguished into two groups: Servers and Clients. Servers are the VMs that gave memory to the pool ($Mem_{init} > Mem_{act}$) and Clients are the VMs that owe memory to the pool ($Mem_{init} < Mem_{act}$).

The system is designed to optimize memory consumption of VMs and amortize temporary memory shortage. However, it is not always possible to have enough memory in the free memory pool and it is anticipated that at some moments, applications will face memory peaks simultaneously and the system should somehow choose the best strategy to overcome such situations. Algorithm 1 describes the approach which is used in our solution.

Data: Working set sizes (WSS) are updated periodically by the estimator

foreach *VM where* Mem_{act}¿WSS **do**

Reclaim extra memory and send it to free memory pool (M_{pool});

$Mem_{act} = WSS$;

if Mem_{act}¡Mem_{init} **then**

| VM = Server;

end

;

let $GROW_{server}$ be the set of VMs from $GROW$ which are Server;

foreach *VM in* $GROW_{server}$ **do** grant memory up to Mem_{init} to from pool if possible;

reclaim memory up to Mem_{init} from Clients (following policy);

if *(*WSS¿Mem_{init}*)* **then**

| $Mem_{act} = Mem_{init}$;

| VM = Client;

else

| $Mem_{act} = = WSS$

end

;

let $GROW$ be the set of VMs such that Mem_{act}¡WSS;

forall *VM in* $GROW$ **do**

| let $CLIENT_{MEM}$ be the memory from pool and borrowed by Client VMs;

| distribute memory from $CLIENT_{MEM}$ up to WSS if possible (following policy);

| $Mem_{act} = Mem_{allocated}$;

end

Algorithm 1: Memory distribution algorithm

The system may be used with two different memory distribution policies: equal and proportional. According to the equal distribution policy, the memory will be granted or reclaimed equally between VMs. With the proportional distribution policy, memory reclamation or granting is distributed proportionally according to VM's memory usages.

3.5 Related Work

The memory size of a VM is generally assigned at creation time. However, optimal memory management requires dynamic memory allocation during VM's runtime. Waldspurger introduced a technique called ballooning [85] which has been widely adopted for VM memory resizing. The balloon is installed into the guest OS as a kernel space driver; it provides a means to dynamically adjust the memory size of the VM. From there, many dynamic memory management policies may be implemented and this is a hot topic. However, there are only few works addressing this issue citeVMCTune describes a resource balancing tool based on dynamic resource allocation (for different resource types). It tracks the live resources consumption (CPU, memory and network) of VMs and Physical Machines, then uses instant resource reallocation for VMs running on same PM to achieve local VMs load balancing. Another paper[76] proposes a reinforcement learning algorithm that facilitates self-adaptive VM resource provisioning. Ginseng[73] is a market-driven cloud system that collects client's bids of a true value for the needed memory, then re-allocates physical memory to the clients thanks to an efficient memory allocation according to these bids. XHive[54] and VSWAPPER[9] are cooperative caching systems for virtual machines that try to reduce the overhead caused by the swap activity within VMs. The main advantages of our solution is that it uses precise working set estimation technique (badis presented in chaper 2) to trigger memory balancing before the swap activity occurs and it maintain a free memory pool which allows to utilize unused memory for other purposes such as hosting new VMs to raise consolidation.

3.6 Results and discussions

This section presents the evaluation results of our CMMS system. The evaluation includes two memory distribution algorithms, equal and proportional.

The experiments were carried out on a Dell Precision server (Intel CPU E5-1603 2.80GHz and 8 Gb of RAM). The virtualization system on the server is Xen 4.2. Both the host and guest OS are Ubuntu Server 12.04. At creation time, each VM is allocated 1Gb of memory.

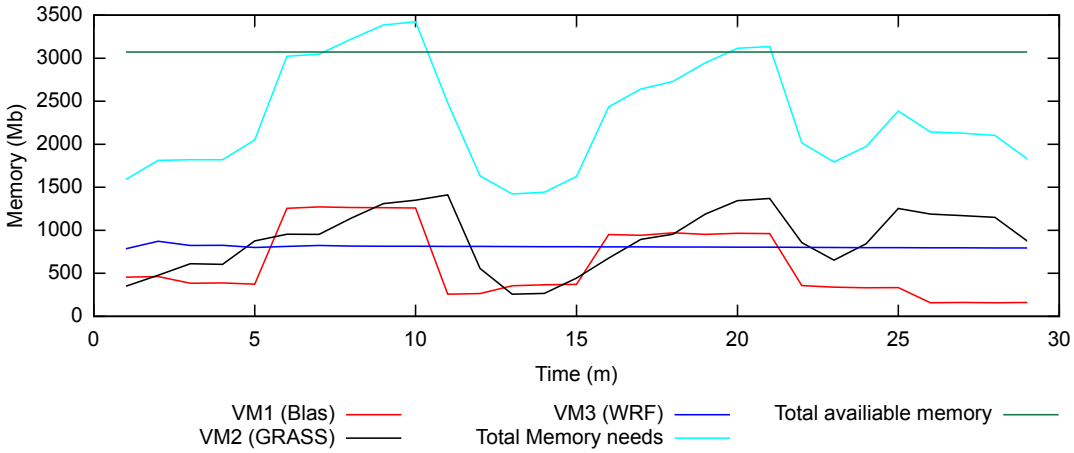


Figure 3.3: Memory behaviour of VMs

To evaluate the benefits of CMMS, the following scientific applications were ran in VMs during experiments.

- The Weather Research and Forecasting (WRF) [89] Model is a next-generation mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting applications.
- GRASS GIS [70] commonly referred to as GRASS (Geographic Resources Analysis Support System) is a free and open source Geographic Information System (GIS) software suite used for geospatial data management and analysis, image processing, graphics and maps production, spatial modeling, and visualization.
- BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations.

The memory behaviour of WRF does not include significant fluctuations (it is mainly flat). The memory profile of BLAS and GRASS GIS changes significantly during the runtime of the application.

Figure 3.3 reports the memory behaviour of the VMs running the enlisted applications. Red, Blue and Black lines illustrate the working set of each VM. The Light blue line indicates the total memory needs of all VMs. The green line represents the maximum available memory dedicated to all VMs. The running time of each application is around 30 minutes.

The experiments were carried out with three different settings: (1) with static allocation (i.e. the memory size of the VM does not change at runtime), then with CMMS (which adapts VM's memory allocation according to the working set), free memory being distributed based on the equal (2) or the proportional (3) algorithms.

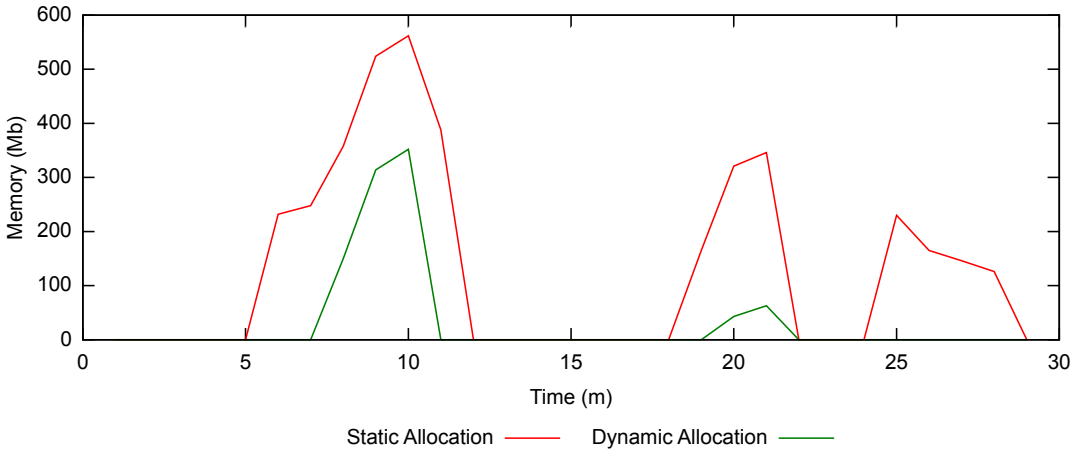


Figure 3.4: Difference between static and dynamic allocation in terms of swapped out memory

Table 3.1: Memory distribution in case of equal policy

Min	From	To	Memory Size
8	VM3	VM1, VM2	104, 104
9	VM3	VM1, VM2	105, 105
10	VM3	VM1, VM2	105, 105
-	-	-	-
21	VM1, VM3	VM2	58, 220
22	VM1, VM3	VM2	62, 221

The observations on Figure 3.3 show that VM1(Blass) and VM2(Grass) are crossing the line of 1GB several times during the running time of the applications, which is the initially allocated size of the VMs. Thus in case of static allocation, these VMs are swapping during these periods. However, these peaks are supposed to be amortised with dynamic allocation. Furthermore, we can notice that at some points, the total of the memory needs is higher than the size of the available memory on the physical machine. This means that at these points, the amount of memory in the pool is 0 and the memory management system faces a challenge of fair memory distribution when free resources are not enough. This challenge is tackled by the implementation of equal and proportional memory distribution policies. These peaks occur during minutes 8-11 and 21-22. Table 3.1 and table 3.2 show how the free memory was distributed when the resources in the pool are not sufficient to satisfy the memory needs of all VMs.

Figure 3.4 demonstrates that dynamic allocation may significantly reduce the amount of swapped out memory. The experiments show that by applying dynamic allocation, we can reduce the amount of swapped out memory by 4.2

Table 3.2: Memory distribution in case of proportional policy

Time	From	To	Memory Size
8	VM3	VM1, VM2	140, 68
9	VM3	VM1, VM2	96, 115
10	VM3	VM1, VM2	88, 122
-	-	-	-
21	VM1, VM3	VM2	58, 220
22	VM1, VM3	VM2	62, 221

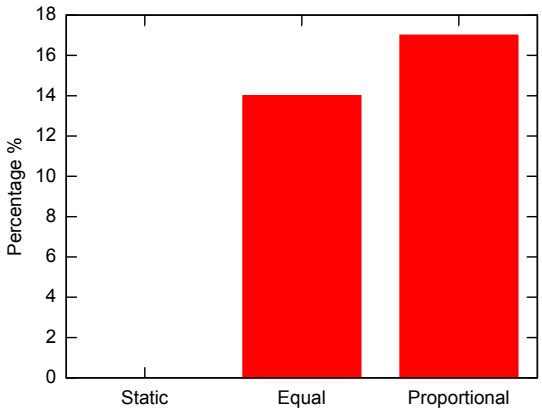


Figure 3.5: Performance evaluation

times. This leads to a significant speed-up for applications. This is possible as it is highly probable that all VMs will not reach peaks at the same time. Thus the over-provisioned memory is transferred to the pool and used to satisfy memory needs of the VMs that reach peaks. Figure 3.6 show the evolution of the number of client and server VMs over time. Most of the time, the number of servers is greater than the number of clients, so there is free memory to provide to VMs facing memory shortage.

The decrease of swapped memory delivers performance boost in case of dynamic allocation. Figure 3.5 shows that the proportional distribution policy provides better speedups than the equal distribution policy.

3.7 Conclusion

Memory management is a complex task, especially in virtualized environments. We showed that a static allocation policy leads to resource wasting, and that a dynamic policy allows to amortize load peaks. We implemented a cooperative memory management system which allows to reclaim unused memory from VMs, and to provision this reclaimed memory to VMs whih need it. Experi-

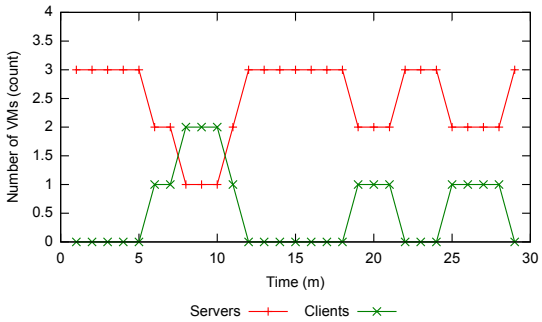


Figure 3.6: VMs status change over time

mental results show that our solution is able to decrease the overhead caused by swap activity for VMs which lack memory, while maintaining a low overhead for other VMs. On our benchmarks, the amount of swapped-out memory was reduced by 4.2 times and performance boosts of about 14% and 17% were detected when testing with two different memory distribution policies.

However, there are several directions to improve our system. First, we are considering an evaluation in a full scale cluster which includes a consolidation system. Then there might be a need to study more memory distribution algorithms.

Chapter 4

Memory Mutualization system for Virtualized Computing Infrastructures

4.1 Introduction

An increasing number of applications require huge amounts of computational and data resources provided by large scale hardware infrastructures, i.e. clusters of servers. As such infrastructures can be mutualized, this contributes to the development of data centers following the cloud computing model. Such data centers may target the public market (public clouds) or be operated by companies for their internal use (private clouds).

A majority of these clusters rely on virtualization for resource management simplification [46]. Virtualization allows hosting several virtual machines (VMs) / operating systems on a single physical machine. Each VM is allocated a given amount of resources (CPU, memory, networking, storage) and it represents the unit of allocation in the infrastructure. Thanks to VM migration, a consolidation policy can potentially be implemented [80], which consists of packing VMs on as less physical servers as possible.

In this context, two main types of resource intensive applications are generally exploited: High Performance Computing (HPC) and Big Data (BD). HPC applications are mainly CPU bound [34] while BD applications are mainly IO and memory bound [21].

This contribution aims to improve the memory management in such environments. The generally adopted approach is to monitor the working set of each VM and to reclaim weakly used memory (cold pages) without degrading the VM performance. Then, the reclaimed memory can be given to VMs with high memory requirements [83]. However, this can only be done on a per server basis as reclaimed memory on one server can only be given to VMs running

on that server. Therefore, we have to trust the placement and consolidation systems for gathering on the same server memory providing VMs and memory consuming VMs.

However, this approach is difficult to implement for two main reasons:

1. Consolidation limitations. Consolidation is known to be a NP hard problem [50], especially since it has to simultaneously take into account multiple resource types whose availability is continuously varying. Therefore, it is a challenge to colocate VMs so that memory can be mutualized.

2. Infrastructure concerns. VMs' placement may be constrained by rules linked with the hardware type or with administration policies (e.g. different sub-clusters for HPC or BD applications), thus limiting the use of VM migration and dynamic consolidation.

Therefore, requiring VM colocation for memory mutualization appears to be a substantial limitation. The principle followed by the suggested contribution is to make the reclaimed memory accessible remotely.

In this chapter, we present the implementation of a system which allows to dynamically monitor the working set of each VM, to aggregate this memory into a distributed memory reservoir, and to make it available to requiring VMs. This memory can be used directly by the VM if available locally. It can be used as a fast remote swap device when available remotely. Our evaluations with HPC and BD benchmarks demonstrate the effectiveness of this approach. We show that a remote memory reservoir provided by a HPC cluster through an Infiniband network can improve the performance of a standard Spark BD application by up to 17% with an average performance degradation of 1.5% (for the providing application).

The rest of the chapter is structured as follows. Section 4.2 presents the motivations and main design choices. Section 4.3 details the designed and implemented system. Section 4.5 presents the performance evaluation which demonstrates the effectiveness of the approach. After a review of related works in Section 4.4, we conclude the article in Section 4.6.

4.2 Motivation and Design Choice

4.2.1 Motivation

An increasing number of applications require huge amounts of resources, especially HPC (CPU bound) and BD (IO and memory bound) applications. Consequently, companies are setting up private cloud infrastructures (datacenters) where the resources required by such applications can be mutualized. For

ease of administration, most of these infrastructures are virtualized. Therefore, VMs are the unit of resource allocation and placement in the datacenter.

Resource allocation to VMs can be static or dynamic:

1. static allocation: resources are allocated to each VM at creation time and never reclaimed during the lifetime of the VM. This approach is widely used due to its ease of use. However, it leads to a waste of resources, since users estimate and allocate the maximum amount of resources needed for their applications to prevent performance degradations during the peak workloads. Thus, when VMs workloads are lower, the allocated resources stay unutilized.

2. dynamic allocation: even if a VM is configured with a given amount of resources (a maximum), resources are effectively allocated on demand. This allows to mutualize the resources within one node and increase resource utilization, since resources which are not used by a VM can be reclaimed and reused for another VM.

Finally, thanks to VM migration, the placement of VMs in the datacenter can be modified dynamically, following a consolidation strategy. Consolidation consists in packing VMs on as less physical servers as possible, and aims at optimizing resource management. Such consolidation can be implemented either with static or dynamic resource allocation. Most of the works regarding consolidation were either based on static allocation or dynamic allocation, but considering for most of them the CPU resource only. Addressing the consolidation problem with dynamic allocation and multiple fluctuating resources is a much tricky issue.

In this work, we are interested in improving memory management in such environments. We assume a dynamic allocation approach where we monitor the working set of each VM and reclaim weakly used memory (cold pages) without degrading the VM performance. Then, the reclaimed memory can be given to VMs with high memory requirements. However, this can only be done on a per server basis as reclaimed memory on one server can only be given to VMs running on that server. Therefore, we have to trust the placement and consolidation systems for gathering on the same server memory providing VMs and memory consuming VMs.

However, this approach is difficult to implement for three main reasons:

1. Consolidation limitations. As we have seen previously, consolidation with dynamic allocation and multiple resources is tricky and it is known to be a NP hard problem [50], especially since it has to simultaneously take into account multiple resource types whose availability is continuously

varying. Therefore, many providers renounce using consolidation and rely on simple predictable policies, thus making it difficult to colocate VMs so that memory can be mutualized.

2. Infrastructure concerns. VMs' placement may be constrained by hardware type concerns, e.g. the availability of a specific device like a GPU. Or sometimes, the hardware may just not enable VM migration, e.g. SR-IOV network devices [58] significantly reduce the overhead of virtualization, but forbid migration in major hypervisors.

3. Administration concerns. Different parts of the infrastructures may be dedicated to different types of application, e.g. different sub-clusters for HPC or BD applications. Indeed, the infrastructures built to run HPC and BD applications are quite different from one another. In HPC clusters, there is generally a centralized file system which is shared among all the (diskless compute) nodes via NFS (or a similar technology). In Big Data clusters, every node has its own local storage device because every node accesses its local storage intensively. Moreover, the schedulers used for distributing jobs in these clusters are different due to the difference in the type of job they distribute. Consequently, migrations between these sub-clusters can hardly be operated.

Therefore, relying on consolidation to enforce VM colocation for memory mutualization appears to be an hazardous strategy. The principle followed by our contribution is to make the reclaimed memory accessible remotely.

4.2.2 Design Choice

In this work, we present a memory mutualization system which relies on dynamic memory allocation. The working set of each VM is monitored and weakly used memory can be reclaimed. This reclaimed memory can be used to provision VMs which lack memory on the same server. It can also be used to provision a memory reservoir which behaves as a remotely accessible fast storage. Modern networking technologies (such as Infiniband) provide low latency and high bandwidth communication allowing to use this memory as a fast swap device. Therefore, the free memory reservoir is used as an extension of the local memory of VMs.

We observed that VMs in HPC clusters are mainly CPU bound and their memory consumption is quite stable, allowing memory to be reclaimed to provision the memory reservoir. Most applications in BD clusters are memory and IO bound and can significantly benefit from extra memory from the memory reservoir.

The next sections present the design of our system and its evaluation.

4.3 Contribution

In the presented distributed memory sharing system, memory may be mutualized locally (to a node) or globally (between remote machines). In the case of local mutualization, unused memory from local VMs can be used to help overloaded local VMs. In the case of global mutualization, unused memory from VMs on one node can be used (as a fast swap) to mitigate memory shortage of remote VMs.

Physical machines act either as a Client (memory consumer) or a Server (memory provider). A client machine can benefit from remote memory from server machines. A machine which does not use all of its memory becomes a server. A machine which requires more memory (than its capacity) becomes a client. However, every VM is guaranteed to have at least its initially allocated memory in case of memory shortage. Thus VMs and client machines can get their memory back in such cases.

The suggested system is composed of two parts: dynamic memory allocation within one node (local memory mutualization) and remote memory allocation from server machines (global memory mutualization).

4.3.1 Design

Overall System architecture

The design of our system relies on two main entities:

- A Local Memory Controller (*LMC*) is in charge of memory management within a single node. Every node (client or server) is running an *LMC*. The *LMC* manages a *Free Memory Reservoir*. This memory may be used for local or global mutualization.

- A Global Memory Controller (*GMC*) manages the coordination between machines (clients and servers). It is connected with all the *LMCs*. It implements a *Global Memory Reservoir* by federating the distributed *free memory reservoirs*. It is responsible for remote memory distribution among clients.

Figure 4.1 illustrates the architecture of the distributed memory sharing system. Each single machine executes an instance of *LMC*, which is responsible for local memory management within the node. The virtualization system

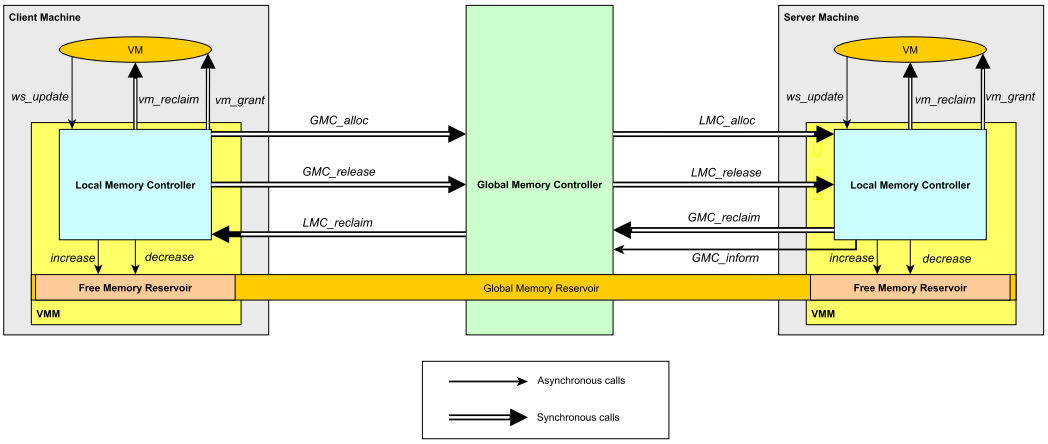


Figure 4.1: Architecture of the distributed memory sharing system

includes a memory monitoring system which periodically evaluates the working set size of each VM. The estimations are transmitted to the *LMC* which is allowed to reclaim or grant memory from/to VMs depending on their memory needs and current allocations. The memory reclaimed by the *LMC* provisions the *Free Memory Reservoir* (local to the machine). This memory can be allocated to local VMs. The information on the size of the *Free Memory Reservoir* is sent (when modified) to *GMC* which implements the *Global Memory Reservoir*. This memory can be allocated to client machines (their *LMC*) and then be used as a remote swap device by overloaded VMs.

LMCs in all machines keep communication with the *GMC* for requesting and releasing remote memory. This communication can rely on different types of networks. We experimented both with Gb Ethernet and Infiniband. Infiniband brings the advantage of enabling Remote Direct Memory Access (RDMA). With Infiniband, the communication framework between *LMC* and *GMC* implements the concepts of RPC over RDMA [38, 79]. Communications between VMs and remote swap devices rely on low-level RDMA primitives which directly access remote memory.

Monitoring Service

A proper working set estimation facility for the memory management system should have a very low performance impact on VMs. It should also be accurate regarding the estimation of the working set size (WSS). For implementing such a facility, the main issues is to enforce accuracy and low overhead in two states, when a VM does not use all its memory (S_{less}) and when it lacks memory (S_{more}).

In our system, we rely on badis 2, a WSS monitoring system implemented

in our research group. Badis combines a statistical working set estimation method based on *page invalidation* [85] and a *buffer cache* monitoring based method [47].

Badis accurately estimates WSS in both the S_{less} case (with the *page invalidation* method) and the S_{more} case (with the *buffer cache* method). Thus, these solutions complete one another.

Memory Management Controllers

The memory management protocol defines the interactions between controllers (*LMC* and *GMC*) and between *LMCs* and the virtualization system. The protocol is composed of two parts:

1. the Local Memory Management Protocol defines the exchanged messages between the virtualization system and the *LMC*. It allows the *LMC* to obtain the WSS from the virtualization system and to grant or reclaim memory to/from a VM.

2. the Global Memory Management Protocol defines the exchanged messages between *GMC* and *LMC*. It allows a client *LMC* to request (to the *GMC*) the allocation or the release of extra memory, or it allows a server *LMC* to reclaim its memory back. A *LMC* can also inform the *GMC* about the size of the local reservoir. Symetrically, the *GMC* can request (to a server *LMC*) a memory allocation or release, or it can reclaim some memory back.

The defined protocol is described in Figure 4.1, with one arrow per message type.

Local Memory Controller behavior The *LMC* periodically gathers the WSS from all the VMs running on the machine. The *ws_update* message is used to receive the WSS information of all VMs. Based on this information, the *LMC* decides wether to *grant* extra memory to VMs or *reclaim* memory from VMs. The implementation of *vm_grant* and *vm_reclaim* are based on memory ballooning [85, 13] (using a balloon driver installed in all the VMs). These operations naturally force a change of the VM sizes and a change of the size of the *Free Memory Reservoir*: a *vm_reclaim* (respectively *vm_grant*) on a VM increases (respectively decreases) the size of the *Free Memory Reservoir*. Later, the memory gathered in the *Free Memory Reservoir* may be provided as a remote swap device.

Global Memory Controller behavior The *GMC* periodically receives updates on the current size of all the *Free Memory Reservoir*. The *GMC_inform* message is used to notify every modification of the size of a *Free Memory Reservoir*. When a machine needs extra memory (a remote swap device) for one of its VMs, the *LMC* of the client machine sends a *GMC_alloc* message to the *GMC* asking to check if it may provide the needed amount of memory. *GMC* is aware of the current state of the *Free Memory Reservoirs* in all machines, thus it can decide where it can allocate a remote swap device for a client machine. Then, it sends an *LMC_alloc* message to the chosen server machine (which can provide memory to the client machine). The *LMC* (in the server machine) allocates the memory and prepares it as a swap device and returns all the necessary information to the *GMC*. After a successful allocation on the server side, the *GMC* returns to the *LMC* on the client machine all necessary information allowing to mount and use the remote swap device which is physically located on a server machine. If an *LMC* of a client machine does not need its extra memory anymore, it can release it with a *GMC_release* message, which propagates to the *LMC* on the server machine with a *LMC_release* message. In the case where a server machine needs its memory back, its *LMC* sends a *GMC_reclaim* message to the *GMC*. Then, the *GMC* notifies the client machine that the remote swap device has to be unmounted (*LMC_reclaim* message) and then returns to the server machine that the memory is free. To avoid high latencies that we would have if you were moving the data from the removed swap device, we asynchronously store the data of a remote swap device on a local disk (on the client machine). Thus, the local disk can replace a remote swap device (rapidly and temporarily as the data will potentially be cached in another remote swap device) in such cases.

Remote Swapping Service

Swapping is a key functionalities provided by Linux OS. It uses swap space to increase the amount of virtual memory available to a machine. When the OS faces memory shortage, some pages have to be swapped out to local disk. The OS invokes the *kswapd* kernel module which is responsible for paging. It sends pages to the swap device having the highest priority in the list of devices, which performs I/O operations that are specific for the given type of device. The remote swapping service implements a new type of device which operates with the same logic and gets the highest priority among swap devices. Applications are not aware of this process. Remote swapping is used to provide the unused memory collected in the *memory reservoir* to remote machines.

4.3.2 Implementation

Implementation Environment

Xen [13] is a popular open-source virtualization system which is widely espoused by several cloud providers such as Amazon EC2. Its implementation follows the para-virtualization [88] model. In the latter, VMs' OS are modified to be aware of the fact that they are virtualized, which reduces virtualization overhead. In this model, a small kernel called the *hypervisor* runs directly on top of the hardware, so taking the traditional place of the OS. Thus, it has all privileges and rights to access the entire hardware and provides the way to run several OS called Virtual Machines (VMs) concurrently. The host OS (seen as a special VM) is called the *Virtual Machine Monitor* (VMM). It has much more privileges than other VMs since it is responsible for running Xen management toolstack.

Monitoring Service

As mentioned before for working set monitoring, we rely on Badis, the details of implementation has been presented in Chapter 2.

Memory reservoir Service

This service is composed of two parts: the *LMC* and the *GMC*. The *LMC* is implemented as a loadable kernel module installed in the VMM in order to allow interactions with VMs for monitoring and management. The *GMC* can be run in any process on any machine in the cluster.

The *LMC* implements dynamic memory allocation based on the memory ballooning technique which is a simple driver located in the VM (guest OS). The driver communicates with the VMM and may receive two types of command: *inflate* and *deflate*. In case of *inflate* command, the driver allocates memory in the VM and gives its control to the VMM (so that the *LMC* can use it). In case of *deflate* command, the VMM releases its control over pages, and the driver can deallocate these pages (therefore making these pages available in the VM). This technique is used to implement dynamic memory allocation (to VMs) in our system.

The *GMC* is managing its communication via standard network interfaces such as Ethernet and Infiniband. The communication via Ethernet is implemented by using low level netpoll APIs, which allow to send UDP packets from the Linux OS Kernel. In case of communication over RDMA (Infiniband), the standard RDMA *SEND/RECV* functions from the IBverbs library are used.

Remote Swapping Service

In the guest OS of a VM, *kswapd* can use swap devices. Such a swap device is implemented in the VMM on the local host. Remote swapping is implemented with two modules, on the client side (memory consumer) and on the server side (memory provider):

1. The client side module is located in the VMM of the client host and provides a means for *kswapd* to swap pages to the swap device.

2. The server side module is located in the VMM of the server host and is responsible for allowing reads and writes to the memory reservoir from remote locations.

Figure 4.2 describes the architecture of the remote swapping service. These two modules can be interconnected via the Infiniband network which makes possible remote swapping without interaction with the remote CPU. For its implementation on Infiniband, remote swapping is based on Infiniswap [43] where *RDMA READ* and *RDMA WRITE* requests allow direct addressing of remote pages.

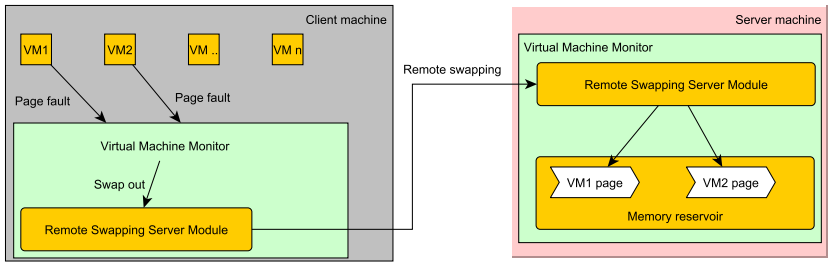


Figure 4.2: The architecture of remote swapping

4.3.3 Memory Management Policy

The *LMC* adjusts the memory size of VMs according to the working set values. If a VM is over-provisioned, then unused memory (according to the working set) is reclaimed and sent to the *Free Memory Reservoir*. In case of memory shortage of a VM, the memory needs can possibly be satisfied from the *Glocal Memory Reservoir*, locally or remotely. The system guarantees that at least the VMs initially allocated memory size is provisioned if its working set grows up to that size, and some extra memory can be allocated if the *Global Memory Reservoir* is not empty.

There are many configurable parameters in our system, which give the opportunity to tune the system according to the needs of an administrator. Some

administrators would give priority to local VMs (satisfy local needs first), others would give higher priority to a given type of VM (potentially remote).

In our case, we tried to satisfy local needs first and a *Free Memory Reservoir* is always used as a remote swap device when all the VMs on that host are satisfied. In our policy, we also added a configurable limit (per VM) on the amount of memory that can be provided as a remote swap device.

4.3.4 Memory Allocation for Application

Our distributed memory sharing system allows VMs to obtain extra memory. This will change the size of the VM dynamically, i.e. the amount of memory granted to the VM.

We can consider two situations regarding the use of this extra memory by applications:

- Applications with dynamic memory allocation. These are applications which dynamically allocate and realease memory. Such application will naturally benefit from the extra memory granted to the VM.
- Applications with static memory allocation. These are applications which allocate as much memory as possible at startup and implement their own memory management internally. The Java virtual machine is an example of such application. Therefore, such applications will not benefit from extra memory granted to the VM.

Regarding the second class of application, we use our system in the following way. First, we monitor the behavior of VM images to determine the type of VM which needs extra memory (e.g. Big Data VMs). These VMs are configured with an increased memory size, so that they will use extra memory. We show in the evaluation that a Big Data VM improves its performance with up to 40% of extra memory. For such VMs, the extra memory must be available if the VM uses it, else the VM would swap to disk and it would degrade performance. Second, the quantity of extra memory which can be allocated is based on statistics. We monitor the use of memory in the datacenter and therefore have an estimation of the extra memory which can be allocation without damage.

The question of which VM should receive extra memory (static application VMs, dynamic application VMs, which VM) is a question of policy.

4.4 Related work

The design of our system was inspired by Global Memory System [35] which was one of the first attempts to implement such an infrastructure-wide memory

management service. As a memory management system which aims at memory mutualization, our contribution can be compared to related works within two categories:

1. Local Memory Mutualization: where the memory optimization and redistribution is done within a single physical node.

2. Global Memory Mutualization: where the memory distribution is implemented datacenter-wide.

Local Memory Mutualization.

Ginkgo [45] is a mutualization system which allows to dynamically adjust the memory size of a VM. It determines the minimum acceptable amount of memory that a VM may run with, based on application performance, memory usage, and submitted load. Then, it relies on ballooning for reclaiming unused memory. The main drawback of the method is that the user or the provider should profile the application in advance and inform the system with that profile.

W. Zhao and Z.Wang [93] have introduced dynamic MEmory Balancer (MEB) for memory balancing between virtual machines. It estimates the memory consumption of a VM and periodically re-allocates memory of VMs based on their needs. In a more recent paper, they upgraded it with a non-intrusive working set estimation system [87]. This approach is close to our regarding local mutualization. However, they did not evaluate it with memory intensive applications such as Big Data applications.

Statistical resource multiplexing strategies have been introduced by Xiaoqiao Meng et al [68]. They analyze VMs' memory usage for deducing predictions and they create couples of VMs in such a way that when one reaches its peak memory consumption the second one is in low point of memory consumption. This forcasting of memory usage is complementary to our and could be inegrated in our system.

Global Memory Mutualization.

Remote swapping mechanisms [10, 65, 31, 90] were introduced years ago and deeply evaluated. However, as today's traditional Ethernet networks are not fast enough and remote paging causes a significant overhead on remote CPUs, successors have been implemented [23, 56] relying on low latency networking infrastructures (mainly Infiniband) enabling the use of RDMA technologies. Infiniswap [43] enables to create a swap device from the memory of one machine and to use that swap device from another machine through Infiniband. Infiniswap implements the fundamental mechanisms that we used to build a global memory mutualization system.

SpongeFiles [32] is a remote memory sharing systems for Hadoop. For large data sets generated by a job, it allows to avoid sending these data to disk but rather to route them to another node where sufficient memory is available. SpongeFiles addresses the issue of memory mutualization for Big Data application but limited to Hadoop applications. Our system addresses this issue in a virtualized infrastructures with a wider scope of applications.

Nswap2L [71] is a swap device extension for Linux which allows to add an abstraction level on swapping process. It appears to the OS as a single swap device partition whose data can be stored to or migrated between various heterogeneous storages (RAM, SSD, HDD etc.) including the memory of a remote host. We share many objectives with Nswap2L, but our experiments target different environments. First, our system address memory mutualization in virtualized infrastructures. Second, while Nswap2L targets mutualization for HPC applications, we rather address mutualization between HPC clusters (with average memory consumption) and Big Data clusters (with high memory consumption). Finally, Nswap2L does not exploit RDMA based networking technologies.

4.5 Evaluation

This section presents the details and results of our evaluation. The provided numbers are the average of ten executions. We do not provide standard deviation as it was not significant.

4.5.1 Methodology

This subsection presents the benchmarks we used and evaluation methodology we relied on. In our evaluations, we consider the client side where remote memory may be used, and the server side from where remote memory is provided. On the client side, we evaluate the performance benefit with memory intensive applications when additional memory is provided. On the server side, we evaluate the performance degradation since that memory is used remotely. On the server side, applications are not memory intensive applications, but rather CPU intensive applications which don't use all their allocated memory.

On the client side, we chose the following types of application for the evaluation of our system:

1. **Microbenchmark** - a simple application that we implemented, whose memory behavior is known in advance, thus allowing to precisely evaluate the contribution.

2. **Memory Intensive benchmarks** - standard benchmarks which are known to stress memory. In this regards, we chose the following benchmarks: Data Caching from CloudSuite [37], Elasticsearch nightly benchmarks [17] and BigBench [41].

3. **Big Data benchmarks** - modern memory intensive benchmarks which are used in the domain of Big Data computing. The following benchmarks have been chosen: Spark SQL [11], TestDFSIO [18] and SparkPi from Spark bench [55].

On the server side, we evaluate the performance degradation using the High Performance Linpack [62] benchmark which is used to evaluate performance on Top 500 supercomputers.

4.5.2 Experimental environment

Hardware. We evaluated our contribution on Dell PowerEdge R610 servers with the following configuration: Intel(R) Xeon E5-2630LV4 CPU, 64 GB of memory, 4 x 512GB of SSD storage. The servers were used with the following scenario: one machine hosts the *GMC*, one machine acts as a Server and one machine acts as a Client.

Virtualization. The virtualization environment we used is Xen 4.2. All the VMs are running an Ubuntu Server 12.04 with Linux kernel 3.6. In all our experiments, the VM configuration that we used is a *medium* size VM, as defined by Amazon web Services: 2 VCPU and 4 GB of memory.

Network. Our default implementation relies on the Infiniband network. Our evaluations were performed with Mellanox ConnectX-3 cards. However, to evaluate the impact of Infiniband, we also performed an evaluation with Ethernet networks.

Software. For benchmarks which rely on Java, the Java Virtual Machine (JVM) configuration (maximum memory allocation) has been changed to make the JVM aware of extra memory available in the remote memory reservoir. Some of the testing applications are based on Apache Spark. In this regard, we evaluated our contribution both with standard apache spark and with an improved version (presented in Xiaoyi et al. [61]) of Spark with RDMA (Infiniband) implemented as a plug-in in Spark which overrides the shuffle methods (this optimized version was implemented at Ohio University, therefore we call it *Ohio Spark*). Comparing with this RDMA optimized version of Spark is a means to know where the benefit comes from.

4.5.3 Evaluation Results

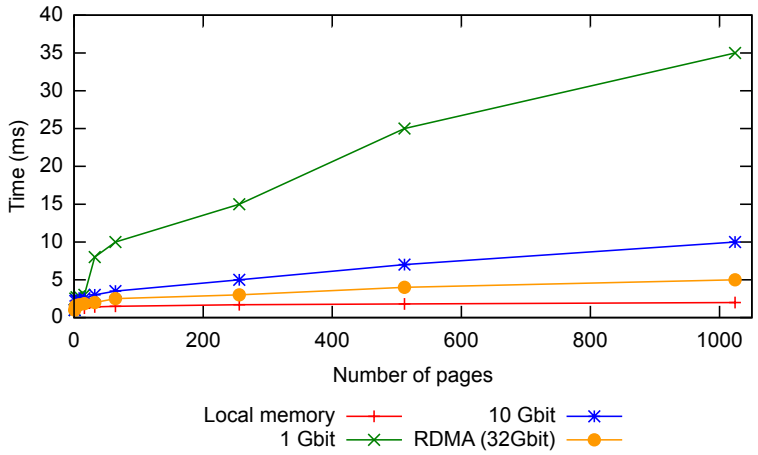


Figure 4.3: Latency while accessing pages in different locations

Micro-benchmarks

Micro-benchmark loops and accesses (reads or writes) on every cell of an array that has a fixed size assigned at creation time. Every cell in the array has the size of a page (4Kb). The performance metric of this benchmark is the latency.

Fig. 4.3 shows the latency of Micro-benchmark while accessing pages locally or remotely using several network technologies (1Gbit, 10Gbit, Infiniband with RDMA 32Gbit). The overhead caused by remote memory location with a small amount of data is not significant. Logically, when the data size increases, the latency increases and the overhead caused by network communications becomes significant. At the level of 1024 pages, using the 1Gb and 10Gb networks makes the application respectively 17 and 5 times slower compared to local memory. We can also observe that the Infiniband network with the RDMA technology can minimize the network overhead down to an acceptable level. During our experiments, we observed that RDMA is only 50% slower than local memory.

Memory Intensive Benchmarks

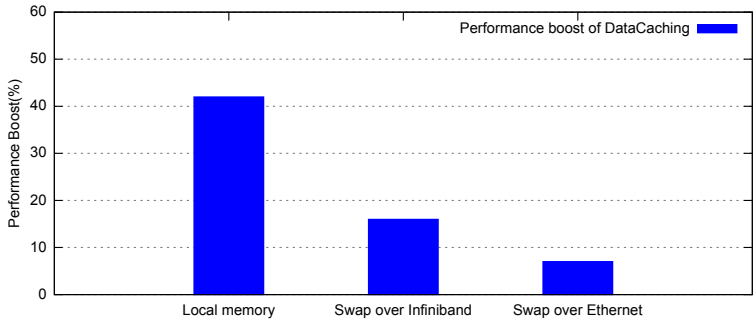


Figure 4.4: Data Caching benchmark performance boost with memory extension (local, Infiniband, Ethernet)

We performed experiments with memory intensive benchmarks which are known to stress memory. The following benchmarks were used:

Data Caching [37] uses the Memcached data caching server to simulate the behavior of a Twitter caching server using a Twitter dataset.

Elasticsearch nightly benchmarks [17] is a benchmark suite. We only performed evaluations for the NYC taxi benchmark. The NYC taxi data set contains the rides that have been performed in yellow taxis in New York in 2015. This benchmark evaluates the performance of Elasticsearch for structured data.

BigBench [41] includes more than 30 queries. We chose query 23 because it has the longest execution time.

For each benchmark, we designed 4 types of workloads to investigate the correlation between performance improvement and swap activity. Each workload determines the amount of memory used by the benchmark.

1. *zero* is a workload where the memory allocated to the VM is enough to execute the application.

2. *light* is a workload where the used memory exceeds the memory allocated to the VM so that up to 10% of its memory goes to swap.

3. *medium* is a workload where up to 20% of its memory goes to swap.

4. *heavy* is a workload where up to 30% of its memory goes to swap.

Fig. 4.4 shows the performance improvement obtained with the Data Caching benchmarks with a heavy workload. The baseline is the execution without memory extension, so that the required swap is managed on disk. We measured the improvement when the VM is given its (required) memory extension (1) with local memory (2) with remote memory (swap) through Infiniband and

(3) with remote memory (swap) through Ethernet (10Gb). A local memory extension would bring an improvement of 40%, which is the ideal case and corresponds to local memory mutualization. We observed that with remote memory, we still have a significant improvement with Infiniband (17%) and with Ethernet (8%). It demonstrates the interest to mutualize memory even remotely.

Fig. 4.5 shows the performance improvement for our 3 selected memory intensive benchmarks with the different workload sizes, when being provided memory extension over Infiniband. Naturally, the improvement is proportional to the memory extension required by the workload. We observe that the improvements are significant for all benchmarks.

One important aspect is that the memory reservoir is provisioned by memory reclaimed on the server side. The reclamation and the remote use of such memory may have a negative impact on applications running on the server side. Remind that such applications are not memory intensive applications but rather CPU intensive applications which do not fully use their memory. We have used the HPL [62] benchmark to evaluate the performance degradation on the server side. On Fig. 4.6, we can see that the performance degradation is in the 1-3% range which is fairly low.

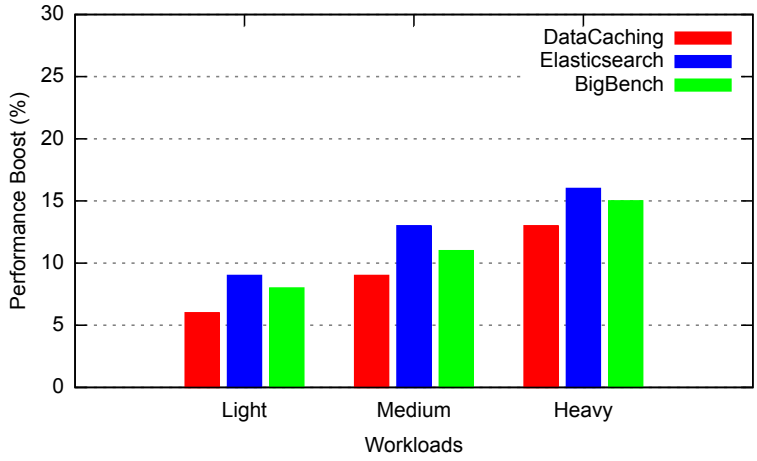


Figure 4.5: Performance boost of Memory intensive benchmarks with memory extension on Infiniband.

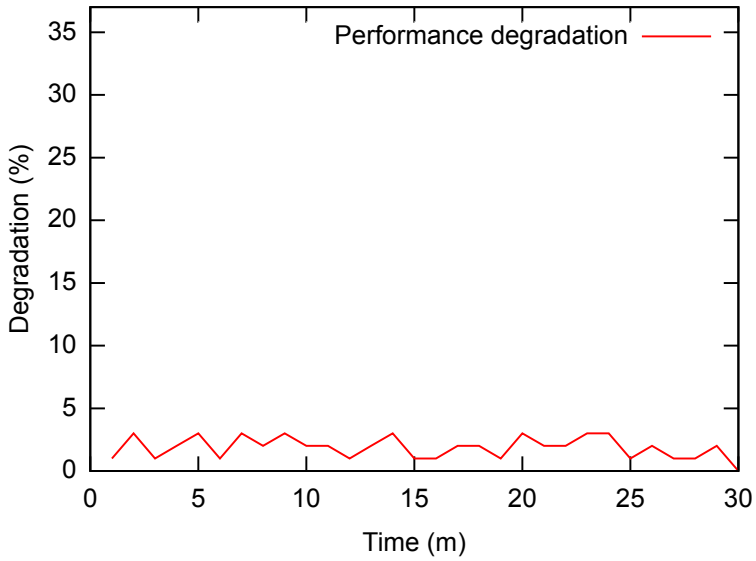


Figure 4.6: Performance degradation for server side VMs executing the HPL benchmark.

Spark Benchmarks

We also experimented with modern memory intensive benchmarks which are used in the domain of Big Data computing. All the benchmarks run on Spark. We relied on the following benchmarks:

Spark SQL [11] is a Spark extension (as module) that enables to support relational processing to benefit from the advantages of relational processing and Spark analytical libraries. As a workload, we have selected JoinPerformance [27] which compares the performance of joining different table sizes and shapes with different join types.

TestDFSIO [18] is a benchmark that attempts to measure the capacity of HDFS for reading and writing bulk data.

SparkBench [55] is a benchmark suit for benchmarking and simulating Spark jobs. In this contribution, the evaluation has been made with PageRank benchmark due to its memory intensive behavior. It is based on the algorithm which was used by Google as an initial algorithm for ranking web pages according their significance and their back-links.

In the case of Spark applications, Spark adapts its memory consumption according to the memory available in the VM. Therefore, for each of these benchmarks, we evaluated the impact of providing a memory extension to the VM. For all experiments, memory was extended by 30% of the original memory size of the VM, assuming that this memory is statistically available in the cluster.

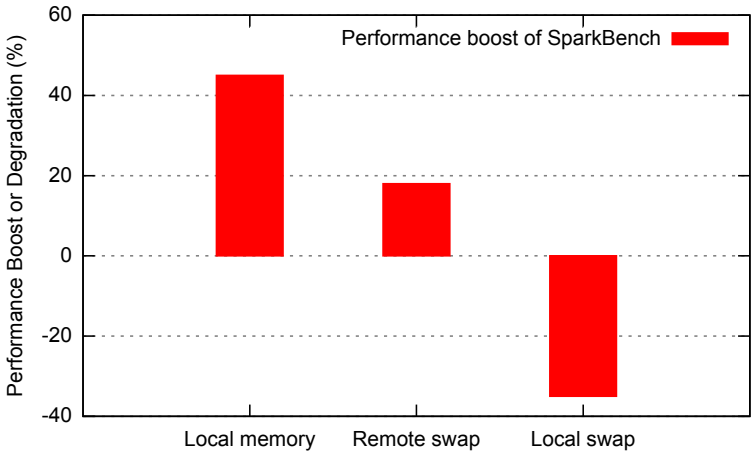
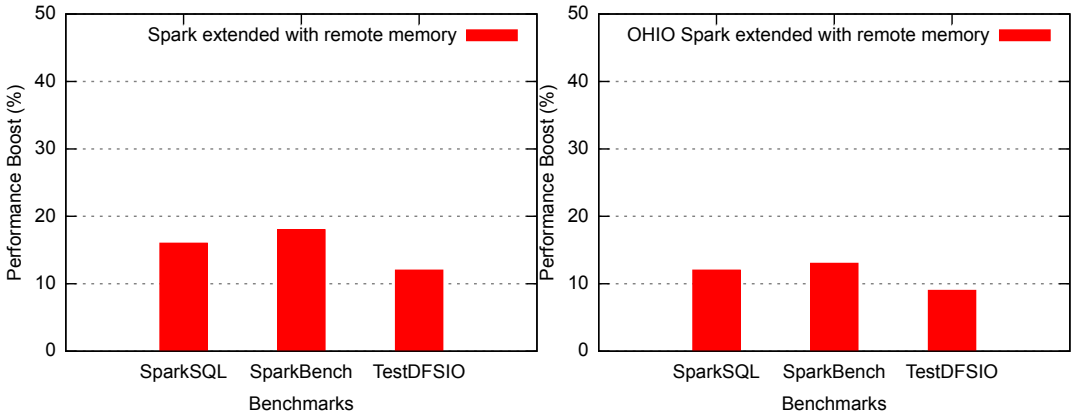


Figure 4.7: Performance evaluation of SparkBench when extending memory with different types of memory.

Fig. 4.7 presents the performance impact on SparkBench when extending the memory allocated to the VM with a different type of memory: (1) with local memory (2) with remote memory (swap) through Infiniband (default Spark) and (3) with a local disk (swap). Again here, local memory is the ideal case, but we can see that remote memory on Infiniband also brings a significant benefit (about 19%). It also confirms that local disk swap is a bad idea, but it should not happen if the memory extension is allocated based on statistics about used memory in the cluster (Section 4.3.4).



(a) Default implementation of Spark.

(b) Ohio Spark implementation.

Figure 4.8: Performance evaluation with two different implementations of Spark extended with remote memory

Fig. 4.8 shows the performance improvement for our 3 selected big data benchmarks when being provided memory extension over Infiniband. The evaluation was performed both with the default implementation of Spark (Fig. 4.8

left) and with the OHIO Spark version (Fig. 4.8 right). We observe that remote memory extension allows a performance improvement from 10% to 17% with default Spark, and from 7% to 10% with OHIO Spark. It is worth to mention that even with an optimized version of Spark which fully takes advantage of RDMA, remote memory extension still brings a significant benefit.

It is well known that Spark is considered to be much efficient since it does its computations in memory. In this regard, it is obvious that the most valuable resource for Spark applications is memory. Thus, we tested several configurations of JVM allowing it to exploit remote memory. Usually, Spark takes all the memory allocated to the JVM and fills it with data. Thus extra memory, which is slightly slower than local memory, will boost its performance until a level. To evaluate that level, we varied the amount of remote memory (up to 70%) in the JVM.

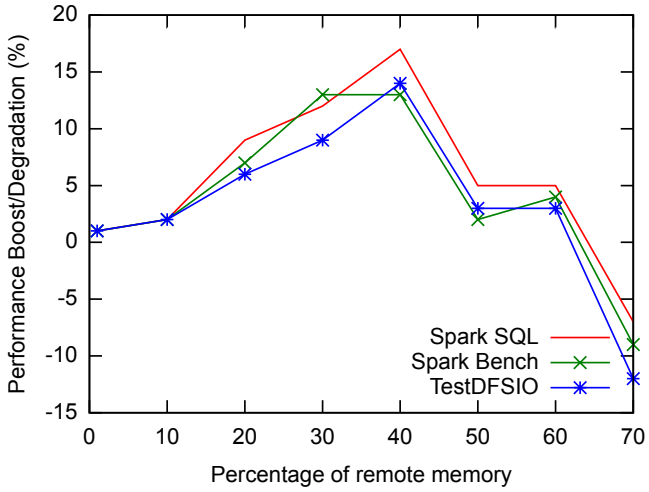


Figure 4.9: Performance evaluation while increasing the allocated remote memory.

Figure 4.9 shows the performance improvement while the amount of remote memory allocated to the VM is increasing. The graph shows that for all benchmarks, the performance is boosted until 40% of memory extension.

4.6 Conclusion

The implemented platform enables to improve the memory management of HPC and BD infrastructures via dynamically monitoring the working set of each VM, aggregating this memory into a distributed memory reservoir, and making it available to requiring VMs. Microbenchmarks, memory intensive benchmarks and Big Data benchmarks were used to evaluate our contribution.

The results show that remote memory mutualization can improve the performance of a standard Spark benchmark by up 17% with an average performance degradation of 1.5%.

Chapter 5

Conclusion

5.1 Conclusion

Recent developments of computing infrastructures encouraged users and companies to externalize their computing resources. This externalization followed the cloud computing principle which promotes resource mutualization. It is not possible to imagine modern IT ecosystems without cloud solutions. The providers started to compete with each other. The main aspects of the competition are the price for their services and the provided quality of service. Thus, they all try to optimize resource consumption, cut the costs and provides always better services. Moreover, raising economical and ecological issues pushed the development of resource optimization techniques even harder.

This thesis suggests to concentrate on memory resource, which is one of the most important types of resource used in large scale infrastructures. The challenge we address is to optimize memory management in a virtualized infrastructure. It raises three main issues: monitoring the woking set of VMs, mutualizing memory between VMs on the same server and mutualizing memory globally between distributed VMs.

First of all, one of the main issues of every memory management system is working set monitoring. In this regard, Badis is proposed to evaluate the amount of effectively used memory of VMs. Badis was favourably compared to the main working set estimation techniques that are used nowadays. Badis is based on the combination of two previously proposed techniques and is capable to use the right method at the right place. In addition to this, it was proposed a consolidation system based on Badis which allows the consolidator to operate based on the actual working set size, not the size of allocated memory.

The second addressed issue is to enable memory mutualization within a node. This type of systems allows to reclaim memory from unused VMs and provide it to the overloaded VMs to help them amortizing memory shortages. The local memory mutualization system is based on Badis and a memory man-

ager which makes decisions on memory re-balancing. In this regard, it maintains a free memory pool which allows to react rapidly to changes in the memory state of the system. The experimental results show that the system is capable to reduce the amount of swapped memory by 4.5 times.

The last addressed issue is to enable global memory mutualization between distributed VMs. This contribution includes the two previous (Badis and local memory mutualization), but extends them to enable global mutualization. The portion of locally unused memory can be used to satisfy global needs (memory needs from VMs located in remote hosts). The system is very productive in mutualizing memory between HPC clusters (which don't fully use their memory) and Big Data clusters (which fully exploit memory), due to specific memory needs of such applications. The experimental results demonstrate the effectiveness of the system showing up to 17% performance boost of overloaded VMs, while only 1.5% degradation on remote hosts.

All the above described solutions have been evaluated with specifically created microbenchmarks and modern HPC and Big Data application benchmarks.

5.2 Perspectives

In this section, we present the possible future works that were planned but did not fit into the 3 year PhD thesis.

The thesis describes a prototype of memory mutualization system. The prototype is implemented on a specific version of the Xen hypervisor. A first perspective would be to adapt the solution to make it available on various hypervisors (such as KVM, VMware etc.). Moreover, the configuration of the system is hard-coded at the moment and it would be convenient to create a configuration dashboard where the administrator can define his policies or configure other parameters. It would be a significant milestone for the project to introduce it as an opensource project to the community which would be able to evaluate it more widely and contribute to the future developments of the system.

Another perspective is to work on properties such as fault tolerance, heterogeneity or security. There have been some works done in this area (e.g. remotely stored data are asynchronously backed up locally for fault tolerance), but it was not the main concern and these are still issues to be addressed. These properties have not been evaluated during our experiments and have not been tested in real life environments. Moreover, in datacenters the solution will have to deal with the heterogeneity of the networking facilities with different speeds and typologies. The solution is capable to operate with different types

of networks, but not with heterogeneous infrastructures.

Finally, it would also be a great achievement to evaluate the solution in different situations, i.e. when the memory reservoir (of mutualized memory) is exploited for different purposes. For instance, the memory reservoir can be used by a consolidator in order to host new VMs, or it can be used to grant extra memory (a bonus) to client VMs in order to attract clients. The memory reservoir could also be made accessible at the application level, as there are many applications that are managing their own memory cache (e.g. the Java virtual machine or a database server).

Bibliography

[1] Elastic Compute Cloud (EC2) Cloud Server and Hosting - AWS. https://aws.amazon.com/ec2/. Accessed on 30/05/2018.

[2] */proc/meminfo* virtual file. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/s2-proc-meminfo. Accessed on 30/05/2018.

[3] Eolas. https://www.eolas.fr/. Accessed on 30/05/2018.

[4] Google datacenter traces. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md. Accessed on 30/05/2018.

[5] OpenStack. https://www.openstack.org/. Accessed on 30/05/2018.

[6] OpenStack Neat. http://openstack-neat.org/. Accessed on 30/05/2018.

[7] Working Set GIT. https://github.com/papers02/working_set.git. Accessed on 30/05/2018.

[8] Abdelsalam, H. S., Maly, K., Mukkamala, R., Zubair, M., and Kaminsky, D. Analysis of energy efficiency in clouds. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:* (2009), IEEE, pp. 416–421.

[9] Amit, N., Tsafrir, D., and Schuster, A. Vswapper: A memory swapper for virtualized environments. *ACM SIGPLAN Notices 49*, 4 (2014), 349–366.

[10] Anderson, T. E., Culler, D. E., Patterson, D. A., , and the NOW team. A case for now (networks of workstations). *IEEE Micro 15*, 1 (Feb. 1995), 54–64.

[11] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al.

Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1383–1394.

[12] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 1185–1196.

[13] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 164–177.

[14] Barroso, L. A., Clidaras, J., and Hölzle, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 8*, 3 (2013), 1–154.

[15] Barroso, L. A., and Hölzle, U. The case for energy-proportional computing. *Computer 40*, 12 (2007).

[16] Beloglazov, A., and Buyya, R. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing* (2010), IEEE Computer Society, pp. 826–831.

[17] benchmarks, E. Elasticsearch nightly benchmarks. https://elasticsearch-benchmarks.elastic.co/ Visited on 2019-03-01.

[18] Benchmarks, S. Testdfsio spark benchmark. https://github.com/BBVA/spark-benchmarks Visited on 2019-03-01.

[19] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices* (2006), vol. 41, ACM, pp. 169–190.

[20] Bugnion, E., Devine, S., Govil, K., and Rosenblum, M. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS) 15*, 4 (1997), 412–447.

[21] Chen, C. P., and Zhang", C.-Y. "data-intensive applications, challenges, techniques and technologies: A survey on big data". *"Information Sciences" "275"* ("2014"), "314 – 347".

[22] Chiang, J.-H., Li, H.-L., and Chiueh, T.-c. Working set-based physical memory ballooning. In *ICAC* (2013), pp. 95–99.

[23] Choi, H.-H., Kim, K., and Kang, D.-J. Performance evaluation of a remote block device with high-speed cluster interconnects. In *Proceedings of the 8th International Conference on Computer Modeling and Simulation* (New York, NY, USA, 2017), ICCMS '17, ACM, pp. 84–88.

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

[25] Corradi, A., Fanelli, M., and Foschini, L. Vm consolidation: A real case based on openstack cloud. *Future Generation Computer Systems 32* (2014), 118 – 127. Special Section: The Management of Cloud Systems, Special Section: Cyber-Physical Society and Special Section: Special Issue on Exploiting Semantic Technologies with Particularization on Linked Data over Grid and Cloud Architectures.

[26] Creasy, R. J. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev. 25*, 5 (Sept. 1981), 483–490.

[27] Databricks. Joinperformance workload for spark sql. https://github.com/databricks/spark-sql-perf Visited on 2019-03-01.

[28] Delforge, P. Americas data centers are wasting huge amounts of energy. *National Resources Defense Council, vol. Issue Brief* (2014), 14–08.

[29] Delimitrou, C., and Kozyrakis, C. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices 49*, 4 (2014), 127–144.

[30] Di, S., and Cappello, F. Gloudsim: Google trace based cloud simulator with virtual machines. *Software: Practice and Experience 45*, 11 (2015), 1571–1590.

[31] Dwarkadas, S., Hardavellas, N., Kontothanassis, L., Nikhil, R., and Stets, R. Cashmere-vlm: Remote memory paging for software distributed shared memory. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999* (April 1999), pp. 153–159.

[32] Elmeleegy, K., Olston, C., and Reed, B. Spongefiles: Mitigating data skew in mapreduce using distributed memory. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 551–562.

[33] eSustainability Initiative 2015, G. Gesi marter 2030: Ict solutions for 21st century challenges. http://smarter2030.gesi.org. Accessed on 01/04/2019.

[34] Expósito, R. R., Taboada, G. L., Ramos, S., Touriño, J., and Doallo, R. Performance analysis of hpc applications in the cloud. *Future Generation Computer Systems 29*, 1 (Jan. 2013), 218–229.

[35] Feeley, M. J., Morgan, W. E., Pighin, E. P., Karlin, A. R., Levy, H. M., and Thekkath, C. A. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 201–212.

[36] Feller, E., Rilling, L., and Morin, C. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (2012), IEEE Computer Society, pp. 482–489.

[37] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 37–48.

[38] Gao, P. X., Narayan, A., Karandikar, S., Carreira, J., Han, S., Agarwal, R., Ratnasamy, S., and Shenker, S. Network requirements for resource disaggregation. In *OSDI* (2016), vol. 16, pp. 249–264.

[39] Gelenbe, E., and Caseau, Y. The impact of information technology on energy consumption and carbon emissions. *Ubiquity 2015*, June (June 2015), 1:1–1:15.

[40] Gernot Heiser University of New South Wales, Sydney, A. The role of virtualization in embedded systems. *Proceedings of the 1st workshop on Isolation and integration in embedded systems* (2008), 11–16.

[41] Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., and Jacobsen, H.-A. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data* (2013), ACM, pp. 1197–1208.

[42] Goldberg, R. P. Survey of virtual machine research. *Computer 7*, 9 (Sept. 1974), 34–45.

[43] Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., and Shin, K. G. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 649–667.

[44] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM 53*, 10 (2010), 85–93.

[45] Hines, M. R., Gordon, A., Silva, M., Da Silva, D., Ryu, K., and Ben-Yehuda, M. Applications know best: Performance-driven memory overcommit with ginkgo. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science* (Nov 2011), pp. 130–137.

[46] Infrastructure, V. Resource management with vmware drs. *VMware Whitepaper 13* (2006).

[47] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. *ACM SIGARCH Computer Architecture News 34*, 5 (2006), 14–24.

[48] Jung, G., Hiltunen, M. A., Joshi, K. R., Schlichting, R. D., and Pu, C. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *2010 International Conference on Distributed Computing Systems* (2010), IEEE, pp. 62–73.

[49] Kaplan, J. M., Forrest, W., and Kindler, N. Revolutionizing data center energy efficiency. Tech. rep., Technical report, McKinsey & Company, 2008.

[50] Karve, A., Kimbrel, T., Pacifici, G., Spreitzer, M., Steinder, M., Sviridenko, M., and Tantawi, A. Dynamic placement for clustered web applications. In *Proceedings of the 15th international conference on World Wide Web* (2006), ACM, pp. 595–604.

[51] Kim, J., Fedorov, V., Gratz, P. V., and Reddy, A. Dynamic memory pressure aware ballooning. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 103–112.

[52] Kim, K. H., Beloglazov, A., and Buyya, R. Power-aware provisioning of cloud resources for real-time services. In *Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science* (2009), ACM, p. 1.

[53] Kochut, A., and Beaty, K. On strategies for dynamic resource management in virtualized server environments. In *2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (Oct 2007), pp. 193–200.

[54] Lee, H. K. . H. J. . J. Xhive : Efficient cooperative caching for virtual machines. *IEEE Transactions on Computers 60*, 1 (2011), 106–119.

[55] Li, M., Tan, J., Wang, Y., Zhang, L., and Salapura, V. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers* (New York, NY, USA, 2015), CF '15, ACM, pp. 53:1–53:8.

[56] Liang, S., Noronha, R., and Panda, D. K. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing* (Sep. 2005), pp. 1–10.

[57] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 267–278.

[58] linux, R. Sr-iov. https://goo.gl/qUtsQz Visited on 2019-03-01.

[59] Liu, H., Xu, C.-Z., Jin, H., Gong, J., and Liao, X. Performance and energy modeling for live migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing* (2011), ACM, pp. 171–182.

[60] Lu, P., and Shen, K. Virtual machine memory access tracing with hypervisor exclusive cache. In *Usenix Annual Technical Conference* (2007), pp. 29–43.

[61] Lu, X., Rahman, M. W. U., Islam, N., Shankar, D., and Panda, D. K. Accelerating spark with rdma for big data processing: Early experiences. In *Proceedings of the 2014 IEEE 22Nd Annual Symposium on High-Performance Interconnects* (Washington, DC, USA, 2014), HOTI '14, IEEE Computer Society, pp. 9–16.

[62] Luszczek, P., Meek, E., Moore, S., Terpstra, D., Weaver, V. M., and Dongarra, J. Evaluation of the hpc challenge benchmarks in virtualized environments. In *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2* (Berlin, Heidelberg, 2012), Euro-Par'11, Springer-Verlag, pp. 436–445.

[63] Magenheimer, D. Add self-ballooning to balloon driver. *Discussion on Xen Development mailing list and personal communication* (2008).

[64] Magenheimer, D., Mason, C., McCracken, D., and Hackel, K. Transcendent memory and linux. In *Proceedings of the Linux Symposium* (2009), Citeseer, pp. 191–200.

[65] Markatos, E. P., and Dramitinos, G. Implementation of a reliable remote memory pager. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), ATEC '96, USENIX Association, pp. 15–15.

[66] Meisner, D., Gold, B. T., and Wenisch, T. F. Powernap: eliminating server idle power. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 205–216.

[67] Melekhova, A., and Markeeva, L. Estimating working set size by guest os performance counters means. *CLOUD COMPUTING 48* (2015).

[68] Meng, X., Isci, C., Kephart, J., Zhang, L., Bouillet, E., and Pendarakis, D. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing* (2010), ACM, pp. 11–20.

[69] Miłós, G., Murray, D. G., Hand, S., and Fetterman, M. A. Satori: Enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (2009), pp. 1–1.

[70] Neteler, M., Bowman, M. H., Landa, M., and Metz, M. Grass gis: A multi-purpose open source gis.

[71] Newhall, T., Lehman-Borer, E. R., and Marks, B. Nswap2l: Transparently managing heterogeneous cluster storage resources for fast

swapping. In *Proceedings of the Second International Symposium on Memory Systems* (New York, NY, USA, 2016), MEMSYS '16, ACM, pp. 50–61.

[72] Nitu, V., Teabe, B., Tchana, A., Isci, C., and Hagimont, D. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 16:1–16:12.

[73] Orna Agmon Ben-Yehuda Eyal Posener, Muli Ben-Yehuda, A. S. A. M. Ginseng: Market-driven memory allocation. *10th ACM SIGPLAN/SIGOPS, International Conference on Virtual Execution Environments* (2014), 41–52.

[74] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. *Informed prefetching and caching*, vol. 29. ACM, 1995.

[75] Pekhimenko, G., Mowry, T. C., and Mutlu, O. Linearly compressed pages: A main memory compression framework with low complexity and low latency. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (2012), ACM, pp. 489–490.

[76] Rao, J., Bu, X., Wang, K., and Xu, C.-Z. Self-adaptive provisioning of virtualized resources in cloud computing. *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (2011), 129–130.

[77] Rina Panigrahy, Vijayan Prabhakaran, K. T. U. W., and Ramasubramanian, R. Validating heuristics for virtual machines consolidation. Tech. rep., January 2011.

[78] Salomie, T.-I., Alonso, G., Roscoe, T., and Elphinstone, K. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 337–350.

[79] Su, M., Zhang, M., Chen, K., Guo, Z., and Wu, Y. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 1–15.

[80] Subramanian, C., Vasan, A., and Sivasubramaniam, A. Reducing data center power with server consolidation: Approximation and evaluation. In *High Performance Computing (HiPC), 2010 International Conference on* (2010), IEEE, pp. 1–10.

[81] Tuduce, I. C., and Gross, T. R. Adaptive main memory compression. In *USENIX Annual Technical Conference, General Track* (2005), pp. 237–250.

[82] Vasan, A., Sivasubramaniam, A., Shimpi, V., Sivabalan, T., and Subbiah, R. Worth their watts?-an empirical study of datacenter servers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (2010), IEEE, pp. 1–10.

[83] VMware. Understanding memory resource management in vmware esx server. *VMware Whitepaper* (2009).

[84] Voorsluys, W., Broberg, J., Venugopal, S., and Buyya, R. Cost of virtual machine live migration in clouds: A performance evaluation. In *IEEE International Conference on Cloud Computing* (2009), Springer, pp. 254–265.

[85] Waldspurger, C. A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[86] Waldspurger, C. A., Park, N., Garthwaite, A. T., and Ahmad, I. Efficient mrc construction with shards. In *FAST* (2015), pp. 95–110.

[87] Wang, Z., Wang, X., Hou, F., Luo, Y., and Wang, Z. Dynamic memory balancing for virtualization. *ACM Transactions on Architecture and Code Optimization (TACO) 13*, 1 (2016), 2.

[88] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and performance in the denali isolation kernel. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002* (2002).

[89] William C. Skamarock, Joseph B. Klemp, J. D.-D. O. G. D. B. M. G. D. X.-y. H. W. W. J. G. P. A description of the advanced research wrf version 3. *NCAR Technical Notes* (2008).

[90] Williams, D., Jamjoom, H., Liu, Y.-H., and Weatherspoon, H. Overdriver: handling memory overload in an oversubscribed cloud. In *Proceedings of the 7th International Conference on Virtual Execution Environments, VEE 2011, Newport Beach, CA, USA, March 9-11, 2011 (co-located with ASPLOS 2011)* (2011), pp. 205–216.

[91] Yang, L., Lekatsas, H., and Dick, R. P. High-performance operating system controlled memory compression. In *Proceedings of the 43rd annual Design Automation Conference* (2006), ACM, pp. 701–704.

[92] Zhao, W., Jin, X., Wang, Z., Wang, X., Luo, Y., and Li, X. Low cost working set size tracking. In *USENIX Annual Technical Conference* (2011).

[93] Zhao, W., Wang, Z., and Luo, Y. Dynamic memory balancing for virtual machines. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 37–47.

[94] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review* (2004), vol. 38, ACM, pp. 177–188.