THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue par :

M. VLAD-TIBERIU NITU

le vendredi 28 septembre 2018

Titre :

Improving energy efficiency of virtualized datacenters

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. DANIEL HAGIMONT

M. ALAIN TCHANA

Rapporteurs :

Mme SONIA BEN MOKHTAR, CNRS

M. TIMOTHY ROSCOE, ECOLE POLYTECHNIQUE FEDERALE DE ZURICH

Membre(s) du jury :

M. ANDRE LUC BEYLOT, INP TOULOUSE, Président

M. ALAIN TCHANA, INP TOULOUSE, Membre

M. COSTIN RAICIU, UNIVERSITE POLITEHNICA BUCAREST, Membre

M. DANIEL HAGIMONT, INP TOULOUSE, Membre

M. RENAUD LACHAIZE, UNIVERSITE GRENOBLE ALPES, Membre

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Daniel Hagimont for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I would also like to thank to my co-advisor Prof. Alain Tchana whose office was always open whenever I ran into a trouble spot or had a question about my research. His guidance went beyond the professional context and conferences were surely less joyful without him. This is why I could not have imagined having better advisors and mentors for my Ph.D study.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Timothy Roscoe, Dr. Sonia Ben Mokhtar, Dr. Renaud Lachaize, Dr. Costin Raiciu and Prof. Andr-Luc Beylot for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

My sincere thanks also goes to Prof. Timothy Roscoe, Dr. Gabriel Parmer, and Dr. Timothy Wood, who provided me an opportunity to join their team as intern, and who gave access to the laboratory and research facilities. Without they precious support it would not be possible to conduct this research.

I also thank my fellow labmates for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years. In particular, I am thankful to Boris Teabe who helped me since my arrival when the research project seemed too complex for my limited skills and to my office-mate Mathieu Bacou who helped me being a more proficient French speaker.

My sincere thanks also goes to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. Last but not least, I express my very profound gratitude to my girlfriend Roxana who knows better than me how many working hours are behind a PhD project (because she counted them). Without her love and support I could not accomplish this work. Thank you!

Abstract

Nowadays, many organizations choose to increasingly implement the cloud computing approach. More specifically, as customers, these organizations are outsourcing the management of their physical infrastructure to data centers (or cloud computing platforms). Energy consumption is a primary concern for datacenter (DC) management. Its cost represents about 80% of the total cost of ownership and it is estimated that in 2020, the US DCs alone will spend about $13 billion on energy bills. Generally, the datacenter servers are manufactured in such a way that they achieve high energy efficiency at high utilizations. Thereby for a low cost per computation all datacenter servers should push the utilization as high as possible. In order to fight the historically low utilization, cloud computing adopted server virtualization. This technology enables a cloud provider to pack (consolidate) the entire set of virtual machines (VMs) on a small set of physical servers and thereby, reduce the number of active servers. Even so, the datacenter servers rarely reach utilizations higher than 50% which means that they operate with a set of long-term unused resources (called 'holes'). My first contribution is a cloud management system that dynamically splits/fusions VMs such that they can better fill the holes. However the datacenter resource fragmentation has a more fundamental problem. Over time, cloud applications demand more and more memory but the physical servers provide more an more CPU. In nowadays datacenters, the two resources are strongly coupled since they are bounded to a physical sever. My second contribution is a practical way to decouple the CPU-memory tuple that can simply be applied to a commodity server. The underutilization observed on physical servers is also true for virtual machines. It has been shown that VMs consume only a small fraction of the allocated resources because the cloud customers are not able to correctly estimate the resource amount necessary for their applications. My third contribution is a system that estimates the memory consumption (i.e. the working set size) of a VM, with low overhead and high accuracy. Thereby, we can now consolidate the VMs on based on their working set size (not the booked memory). However, the drawback of this approach is the risk of memory starvation. If one or multiple VMs have an sharp increase in memory demand, the physical server may run out of memory. This event is undesirable because the cloud platform is unable to provide the client with the booked memory. My fourth contribution is a system that allows a VM to use remote memory provided by a different rack server. Thereby, in the case of a peak memory demand, my system allows the VM to allocate memory on a remote physical server.

Contents

Chapter 1

Introduction

We live in an economic era of recursive outsourcing. Companies are becoming more and more specialized and those tasks falling outside of their narrow expertise area are outsourced to other companies which, in turn, follow the same pattern. IT services are not exempted from this trend. They are centralized to datacenters which increase the energy efficiency by sharing resources among multiple tenants. Initially, datacenters were populated with specialized machines but they were soon replaced by commodity servers for cost efficiency matters. *Commodity servers are built in a way that they are more energy efficient at high utilization* [89]. When the utilization is low, servers waste an important fraction of energy in stalled cycles and empty memory banks [46].

This problem is attacked from two different angles. First, a research axis focuses on improving the energy proportionality of servers by reducing their energy consumption at low utilization rates. The second research axis takes the opposite way and focuses on increasing as much as possible the utilization of turned-on servers. An energy efficient datacenter should host either fully used servers or turned-off servers. In this second research category, a notable evolutionary step was achieved when cloud datacenters adopted virtualization. This technology enables the execution of multiple virtual machines (VMs) on top of a single physical server. Since the VM is completely decoupled from the hardware, it can easily be relocated (*migrated*) between physical servers with minimal service downtime. This is a powerful feature since it enables *VM consolidation* whose objective is to pack VMs on as few physical servers as possible. By mutualizing server resources among multiple tenants, virtualization along with VM consolidation brought task density to a level unseen before. However, even if VM consolidation increased server utilization by 5-10%, we rarely observe datacenter servers with an utilization above 50% for even the most adapted workloads [46, 63, 113]. Considering that datacenters may have tens or hundreds of thousands of servers, huge amounts of energy and resources end up being wasted.

1.1 Contribution overview

StopGap: reduce datacenter resource fragmentation. In the previous section we have seen that VM consolidation is a powerful technique but it is often not as effective as we want. There are multiple reasons that limit the consolidation rate. First, cloud providers propose a wide range of general purpose or specialized virtual machines. For example, Amazon EC2 proposes 164 VM types tuned for general purpose use or specialized for CPU, memory, GPU, or storage intensive workloads. In addition, this wide range of choices is not on a single dimension (i.e. a single resource type) but at least three (CPU, memory and network bandwidth). In this context, VMs almost always fail to completely fill up the capacity of the hosting servers and this limits consolidation rates. Datacenter servers will execute with a set of long-term unused resources, hereinafter called *holes*. However, we observed that smaller VMs lead to higher consolidation rates because it is more probable that they will fit available holes. Second, we observed that many internet applications (e.g. internet services, MapReduce, etc.) are *elastic* which means that they can be reconfigured on top of an arbitrary number of VMs. Based on this two observations, in Chapter 2 we introduce *StopGap*, an extension which comes in support to any VM consolidation system. StopGap dynamically replaces (when needed) "big" VMs with "smaller" ones and automatically reconfigures the user application on top of the newly created VMs. This process is called *VM split*. However, a larger number of VMs will also introduce a higher overhead because each VM executes additional OS services along with the user application or stateful servers (such as databases) generate additional coherency traffic. Thereby, in cases where the higher number of VMs does not lead to a higher consolidation rate, StopGap may chose to *fuse* multiple VMs colocated on the same physical server which execute the same application replica.

Decouple memory and CPU with a new ACPI state. Even if a datacenter enhanced with StopGap is able to increase the consolidation rate, we observed that the CPU utilization of servers is still very low. Thereby, we presumed that a more fundamental problem is the source of this low CPU utilization. After analyzing the amount of memory and CPU for the Amazon EC2 VMs in the last decade, we found out that even if both resources have grown substantially, *the memory per CPU ratio is two times higher today than one decade ago*. Applications are gradually migrating datasets from HDD to faster storage such as NVMe or RAM and, on top of this, datasets are also becoming larger and larger over time. On the other hand, by analyzing the SPECpower_ssj2008 [28] reports, we've also computed the memory per CPU

ratio for the physical servers in the last decade. In this case, we have observed an inverse trend; the memory per CPU ratio is today more than two times lower than about a decade ago. In conclusion, applications demand more and more memory but physical servers provide more and more CPU. However, in the current datacenters the CPU:memory tuple is strongly coupled inside a physical server which is the smallest granularity of the power domain. To get more memory in the datacenter, one needs to completely turn on a new server, including the CPU. However, CPU could not be fully used because memory saturates faster and limits the consolidation rates. Since the two resource demands are not correlated anymore, cloud computing research is looking for ways to decouple them such that each resource can be allocated independently. To solve this problem, the concept that gained significant momentum in recent years is *disaggregated computing* which aims to change the server-centric view of the datacenter to a resource-centric view. A disaggregated datacenter can be seen as a single huge and modular physical machine whose amount of resources can be independently and dynamically allocated. Since resource disaggregation completely revolutionizes the datacenter computing paradigm, it is still a research topic with plenty of unanswered questions and thereby, not yet implemented in the mainstream cloud. In Chapter 3, we propose a short term solution that can have the benefits of memory disaggregation and can be introduced with only small changes to the hardware of commodity servers. We propose a new ACPI state (called *zombie*) which is very similar to Suspend-to-RAM in power efficiency and transition latency. Our new zombie state (noted S_z) keeps the memory banks completely active and ready to be used by other servers in the rack, even when all the CPUs are turned off. Thereby, we provide a simple and practical way to decouple memory from the computing resources. In this way, zombie servers have the potential to considerably increase the energy efficiency of cloud datacenters.

ZombieStack and working set size estimation. Cloud computing does not provide any software stack ready to take advantage of our new Zombie ACPI state. Thereby, in Chapter 4 we present the architecture and the implementation details of *ZombieStack*, a prototype cloud operating system based on OpenStack [22] and a modified version of the KVM [80] hypervisor. In a nutshell, a cloud operating system manages the VMs during their entire lifetime. For example, when a new VM request arrives in the system, the cloud operating system looks for a physical server that has enough resources to host the new VM. However, in a datacenter whose servers are enhanced with S_z, VMs can also rely on remote memory, i.e. memory provided by other physical servers than the one hosting the VM. Thereby, our ZombieStack can place

a VM even on a server that is unable to provide the entire amount of requested memory; the missing part will be filled with remote memory. Even if the networking technology evolved such that one can access remote memory with sub-microsecond latency, this is still much larger than any local memory access. Thereby, our system should find the most optimal ratio of local vs. remote memory that increases the resource efficiency and minimizes the VM performance impact. However, the optimal ratio is not static but depends on the VM memory activity (i.e. the *working set size* of the VM). The working set size (WSS) is defined as the amount of memory actively used by a VM at a given time. In Chapter 5, we present Badis, a system that is able to estimate a VM's WSS with high accuracy and no VM codebase intrusiveness. In short, WSS estimation increases the memory allocation efficiency for two main reasons. First, by finding out how much memory is actively used by a VM, we can reclaim the unused memory and look for more optimal ways to reallocate it. Second, WSS estimation allows us to find out the performance impact induced by a given ratio of remote memory on a VM.

Publications that constitute this thesis:

1. Vlad Nitu, Boris Teabe, Leon Fopa, Alain Tchana, Daniel Hagimont: StopGap: elastic VMs to enhance server consolidation. Softw., Pract. Exper. 47(11): 1501-1519 (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. SIGMETRICS 2018: 62-63.

3. Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, Daniel Hagimont: Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. EuroSys 2018: 16:1-16:12.

Other publications:

1. Alain Tchana, Vo Quoc Bao Bui, Boris Teabe, Vlad Nitu, Daniel Hagimont: Mitigating performance unpredictability in the IaaS using the Kyoto principle. Middleware 2016: 6

2. Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, Binoy Ravindran: Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. VEE 2017: 1-14

3. Vlad Nitu, Boris Teabe, Leon Fopa, Alain Tchana, Daniel Hagimont: StopGap: elastic VMs to enhance server consolidation. SAC 2017: 358-363

4. Katia Jaffrs-Runser, Gentian Jakllari, Tao Peng, Vlad Nitu: Crowdsensing mobile content and context data: Lessons learned in the wild. PerCom Workshops 2017: 311-315

5. Boris Teabe, Vlad Nitu, Alain Tchana, Daniel Hagimont: The lock holder and the lock waiter pre-emption problems: nip them in the bud using informed spinlocks (I-Spinlock). EuroSys 2017: 286-297

1.2 Thesis organization

The rest of this thesis is organized as follows. In Chapter 2 we introduce *StopGap*, our consolidation system extension which splits/fuses VMs in order to increase the consolidation ratio. In Chapter 3, we propose a new ACPI state which keeps the memory banks completely active and ready to be used by other servers in the rack, even when all the CPUs are turned off. In Chapter 4 we present the architecture and the implementation details of *ZombieStack*, a software stack that takes advantage of our new ACPI state. In Chapter 5, we survey the state-of-the-art for WSS estimation techniques and propose Badis, a system that is able to estimate a VM's WSS with high accuracy and no VM codebase intrusiveness.

Chapter 2

StopGap: Elastic VMs to enhance server consolidation

2.1 Introduction

These days, many organizations tend to outsource the management of their physical infrastructure to hosting centers, implementing the cloud computing approach. The latter provides two major advantages for end-users and cloud operators: flexibility and cost efficiency [44]. On the one hand, cloud users can quickly increase their hosting capacity without the overhead of setting up a new infrastructure every time. On the other hand, cloud operators can make a profit by building largescale datacenters and by sharing their resources between multiple users. Most of the cloud platforms follow the Infrastructure as a Service (IaaS) model where users subscribe for virtual machines (VMs). In this model, two ways are generally proposed to end-users for acquiring resources: reserved and on-demand [63]. Reserved resources are allocated for long periods of time (typically 1-3 years) and offer consistent service, but come at a significant upfront cost. On-demand resources are progressively obtained as they become necessary; the user pays only for resources used at each time. However, acquiring new VM instances induces instantiation overheads. Despite this overhead, on-demand resource provisioning is a commonly adopted approach since it allows the user to accurately control its cloud billing.

In such a context, both customers and cloud operators aim at saving money and energy . They generally implement resource managers to dynamically adjust the active resources. At the end-user level, such a resource manager (hereinafter *AppManager*) allocates and deallocates VMs according to load fluctuations [32]. Tools like Cloudify [5], Roboconf [25], Amazon Auto-scaling [4] and WASABi [3] can play that role. At the cloud platform level, the resource manager (hereinafter *IaaSManager*) relies on VM migration [60] to pack VMs atop as few physical machines (PMs) as possible. Subsequently, it leaves behind a

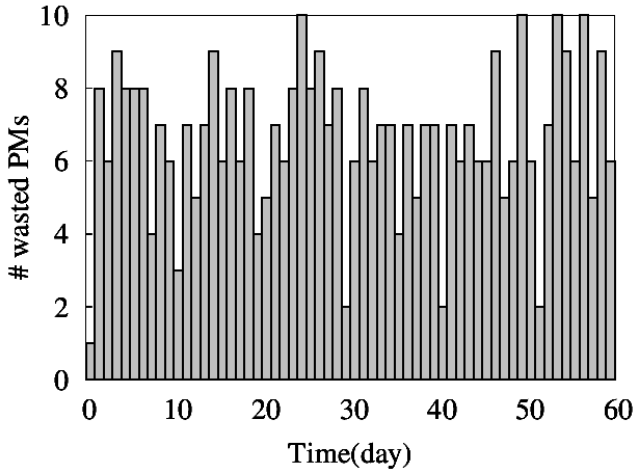

Figure 2.1: Resource wasting due to holes in a public Eolas cluster (a cloud operator) composed of 35 PMS. Holes are aggregated and represented as entire wasted PMs. We can observe that an average of 6 PMs are misspent every day.

number of "empty" PMs which may be turned-off. This process is known as server consolidation [102]. Tools like OpenStack Neat [23], DRR/DPM from VMware [7], and OpenNebula [21] can play that role.

Although VM 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 [46, 63, 113]. Due to various customer needs, VMs have different sizes (e.g. Amazon EC2[8] offers 164 VM types) which are often incongruous with the hosting PM's size. This incongruity obstructs consolidation when VMs do not fit available spaces on PMs. The data center will find itself having a set of PMs which operate with long-term unused resources (hereinafter 'holes'). The multiplication of such situations raises the issue of PM fragmentation (illustrated in Section 2.2), which is a source of significant resource waste in the IaaS. Figure 2.1 presents the waste observed in a public Eolas[11] cluster[1] composed of 35 PMs. The datacenter holes are aggregated and represented as entire wasted PMs. We can observe that an average of 6 PMs are misspent every day.

VMs which consume a low amount of resource (hereinafter "small" VMs) lead sometimes to a more efficient consolidation compared to VMs which consume a high amount of resource ("big" VMs). In order to take advantage of this fact, we introduce *StopGap*, **an extension which comes in support to any VM consolidation system**. It dynamically replaces (when needed) "big" VMs with multiple "small" VMs (seen as *VM split*) so that holes are

[1]Eolas is our cloud computing partner.

avoided. StopGap imposes a novel VM management system that deals with *elastic VMs* (i.e. VMs whose sizes can vary during execution). However, current IaaS managers handle only VMs whose sizes are fixed during execution, thus we need to extend the traditional IaaS management model. To this end, we introduce a novel management model called Hybrid Resource Negotiation Model (HRNM), detailed in Section 2.5.
The main contributions of this article are the followings:

1. We propose HRNM, a new resource allocation model for the cloud.
2. We propose StopGap, an extension which improves any VM consolidation system.
3. We present a prototype of our model built atop two reference IaaSManager systems (OpenStack[22] and OpenNebula[21]). We demonstrate its applicability with SPECvirt_sc2010 [29], a suite of reference benchmarks.
4. We show that StopGap improves the OpenStack consolidation engine by about 62.5%.
5. We show that our solution's overhead is, at worst, equivalent to the overhead of First Fit Decreasing (FFD) algorithms [67] underlying the majority of consolidation systems.

The rest of the chapter is organized as follows. In Section 2.2 we introduce some notations, we motivate our new resource management policy and we present its central idea. Section 2.4 defines the application type on which we tested our model. Section 2.5 presents in detail HRNM and its application to our reference benchmark. Section 2.6 presents StopGap while Section 2.7 evaluates both its impact and benefits. The chapter ends with the presentation of related works in Section 2.8 and our conclusions in Section 2.9.

2.2 Motivation

A data center is potentially wasting resources at a given time t if the following assertion is verified:

Assertion 1

$$\exists k \ s.t. \ \forall x \in \{CPU, memory, bandwidth\}, \\ \sum_{j=1}^{m_{P_k}} booked_x(VM_j, P_k) \leq \sum_{i=1, i \neq k}^{n} free_x(P_i) \tag{2.1}$$

where

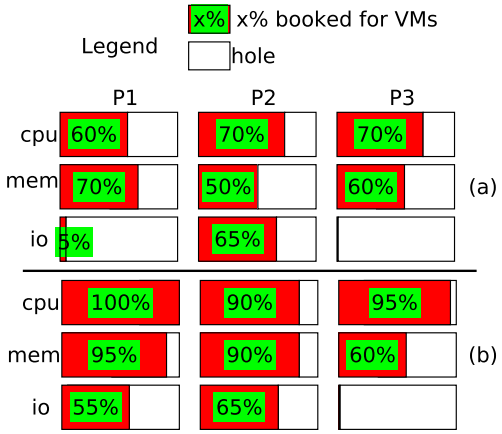

Figure 2.2: Illustration of resource waste in a data center: two states of the same data center are presented (a and b).

n	: total number of physical machines in the data center
VM_j	: VM number j
P_k	: PM number k
m_{P_k}	: total number of VMs on P_k
$booked_x(VM_j, P_k)$	: the amount of resource of type x booked by VM_j on P_k
$free_x(P_z)$	: the amount of resource of type x which is unbooked on P_z

In other words, a data center is wasting resources when there is at least one PM whose sum of booked resources by its VMs can be provided by the sum of the other PMs' holes. Figure 2.2 presents two states (top and bottom) of a data center with three PMs. According to our definition, in the first state, the data center wastes resources (k may be 1, 2 or 3). In the second case, we consider that the data center does not waste resources because Assertion 2.1 is not verified for CPU and memory.

Resource waste is a crucial issue because of the tremendous energy consumed by todays data centers. Addressing this issue is beneficial on the one hand to cloud operators (about 23% of the total amortized costs of the cloud [81]). On the other hand, it is environmentally beneficial for the planet, as argued by Microsoft [19] (which proposes the 10 best practices to move in the right direction). Several research have investigated this issue and the large majority of them [53, 122] rely on VM consolidation. The latter consists in dynamically rearranging (via live migration) VMs atop the minimum number of PMs. Thus empty PMs are suspended (e.g. in sleep mode) or switched to a low power mode for energy saving.

Even if VM consolidation has proven its efficiency, it is not perfect for two

main reasons: (1) VM consolidation is an NP-hard problem, (2) in some situations, VM relocation is not possible even if Assertion 2.1 is verified. For illustration, let us consider our data center use case introduced in the previous section. We focus on the first state (Figure 2.2(a)) where resources are potentially wasted. As mentioned in the previous section, if we aggregate the P_2 and P_3 holes we are able to provide the resources needed by all VMs which run on P_1. Therefore, one can think that by applying VM consolidation to this use case, P_1 could be freed.

Assertion 2: The efficiency of any VM consolidation algorithm depends on two key parameters: VM sizes and hole sizes.

Returning to our first data center state (Figure 2.2(a)), we may consider two VM configurations which consume the same amount of resource on P_1:

- In Figure 2.3(a) we consider two identical "small" VMs (VM_1 and VM_2). Each of them consumes 30% CPU, 35% memory and 2.5% bandwidth from P_1. In this case, VM consolidation is able to migrate VM_1 to P_2 and VM_2 to P_3. At the end, P_1 may be turned-off (Figure 2.3(b)).
- In Figure 2.3(c) we consider that P_1 runs a single "big" VM (VM'_1). VM consolidation is no longer efficient because neither P_2 nor P_3 is able to host VM'_1. It cannot fit in the available holes.

Such situations are promoted in a data center by the mismatch between VM sizes and holes. As presented in Section 2.1, Figure 2.1 shows that this issue is present in a real data center. In this chapter we propose a solution which addresses this problem.

2.3 StopGap overview

In the previous section we exposed that the regular consolidation is difficult for "big" VMs because they require big holes. A solution to this limitation could be to aggregate the holes using a distributed OS. However, the lessons learned from distributed kernels (such as Amoeba[1]) show that the reliability of these solutions is debatable. In this work, we opt for an alternative approach which relies on two assumptions.

- (A_1) the **vertical scaling** capability of VMs: this is the virtualization system's capability to resize a VM (add/remove resources) at runtime. For instance, Xen [42] and VMware [146] provide this feature.
- (A_2) the distributed behaviour of end-user applications: this is an application's capability to run atop a changeable number of VMs (**horizontal**

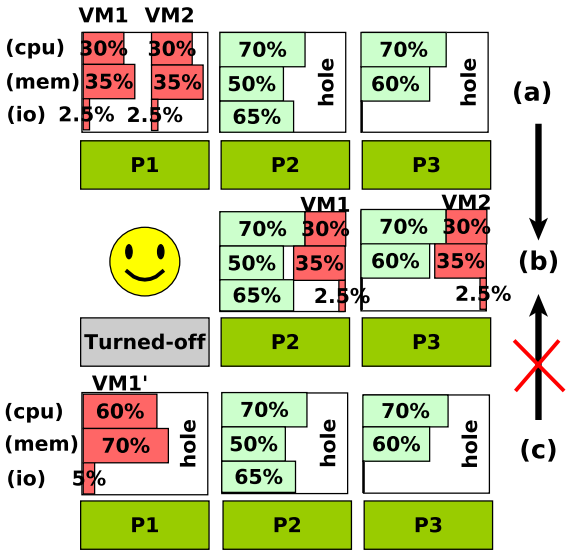

Figure 2.3: The efficiency of any VM consolidation algorithm depends on two key parameters: VM sizes and hole sizes. From state (a), there is a possible VM consolidation which releases one PM; it is evidenced by the state transition (a)→(b). The consolidation is successful because of small VM sizes. In contrast, from state *(c)*, VM consolidation is not possible because VMs are too big.

scaling). Such applications are called elastic applications. They include the large majority of applications deployed within the cloud (e.g. internet services, MapReduce, etc.). For illustration, we focus in this work on applications which follow the master-slave pattern.

Relying on these two assumptions, we propose a cooperative resource management system in which **the end-user allows the cloud manager to dynamically resize (vertical/horizontal scaling) the VMs so that a "big" VM can be replaced by multiple small VMs**. For illustration, we apply our solution to the "big" VM case in Figure 2.3(c). Firstly, we instantiate a new VM (VM_1) on P_1. Its size will be half of the "big" VM size. Secondly, we scale down the "big" VM (vertical scaling) to half of its size, resulting VM_2. Finally, we end up with the case of Figure 2.3(a) having two identical "small" VMs.

VM resizing is not a common practice in today's cloud. Therefore, we propose a novel resource allocation and management model for the cloud. Before describing this model, we first present an overview of the master-slave pattern, the application type considered in our solution.

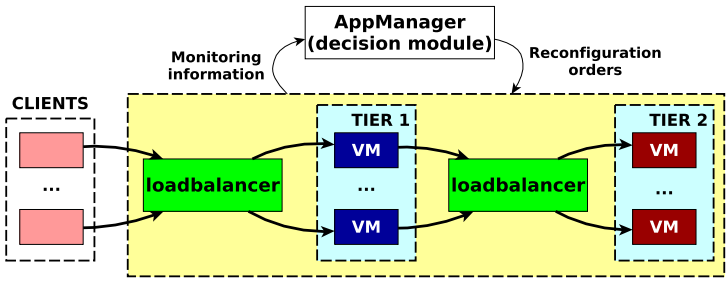

Figure 2.4: The architecture of an Multi-tier master-slave applications (MTMSA).

2.4 Multi-tier master slave applications

As mentioned earlier, our solution is suitable for Multi-tier master-slave applications (hereinafter *MTMSA*). It is important to specify that MTMSA is one of the most prevalent architecture among Internet services. For instance, most applications from SPECvirt_sc2010 [29] and CloudSuite [71], two reference benchmarks for cloud platforms, follow this pattern. In this application type (see Figure 2.4), a tier is composed of several replica (also called slaves) which all play the same role (e.g. web server, application server, database). Each replica is executed by a single VM. In front of this set of slaves, lays a master VM, responsible for distributing requests to the slaves. The master is generally called loadbalancer since it implements a load balancing policy.

The main MTMSA advantage (which justifies its wide adoption) is the high flexibility of a tier (i.e. add/remove VMs according to the workload). After any change in a tier structure, the application has to be reconfigured. This process is usually automated by an autonomic-manager component (i.e. the *AppManager*) deployed with the application. The *AppManager* is provided either by the Cloud (e.g. Amazon Auto-Scaling service), or by the customer (e.g. using an orchestration system like Cloudify [5] or Roboconf [25]). In this work we assume that the *AppManager* is provided by the cloud. Generally, an *AppManager* is responsible for:

- detecting a tier overload/underload and deciding how many VMs to add/remove (by sending instantiation/termination requests to the IaaSManager).
- invoking the loadbalancer reconfiguration in order to take into account a VM's arrival/departure.

2.5 A hybrid resource negotiation model

This work improves VM consolidation thanks to the basic idea presented in Section 2.3. **Our solution is complementary to any VM consolidation system.** However, it requires: (1) that the user VM is executing an AppManager able to reconfigure the MTMSA and (2) there is a real collaboration between the AppManager and the IaaSManager. More precisely, the AppManager exposes APIs called by the IaaSManager to reconfigure the application when the IaaSManager identifies a VM split opportunity. In a traditional IaaS model, the cloud provider sells VMs which are seen as "black boxes". Any information about the applications executed inside the VM are only known by the VM's user. Thereby, in order to take advantage of our solution, the traditional IaaS model should be enhanced with the two requirements stated above. The PaaS is a more high level service where the provider hides from users the complexity of building and maintaining the infrastructure. In this case, the two requirements can be provided without a mandatory collaboration between the provider and users. This section presents the cooperative resource management model which we propose. It can be considered from different perspectives: an extension of a PaaS or a hybrid IaaS-PaaS model. In this work we consider the latter case because it is the most general one.

2.5.1 Description of the model

In contemporary clouds, the resource negotiation model (between the customer and the provider) is based on fixed size VMs. We call it: the VM Granularity Resource Negotiation Model (hereinafter *VGRNM*). Figure 2.6 summarizes this model and illustrates its limitation in the perspective of VM consolidation. For instance, the sum of unused resources on PM_2 and PM_3 is greater than the needs of the large VM hosted on PM_4, but no consolidation system could avoid this waste.

Our model overcomes these limitations. To this end, it allows the IaaSManager to change both the number and the size of VMs, feature which is not provided by VGRNM. Thus, in addition to VGRNM, we need to define a new resource management model which allows resource negotiation at the granularity of an application tier. We call it: the Tier Granularity Resource Negotiation Model (hereinafter *TGRNM*). The HRNM (hybrid model) introduced above represents the aggregation between the traditional model (VGRNM) and our new model (TGRNM). Figure 2.5 graphically represents the negotiation phases of HRNM. They are summarized as follows:

- **(1)** Using VGRNM, the customer deploys and starts the AppManager, which exposes a web service. Through it, the AppManager is informed

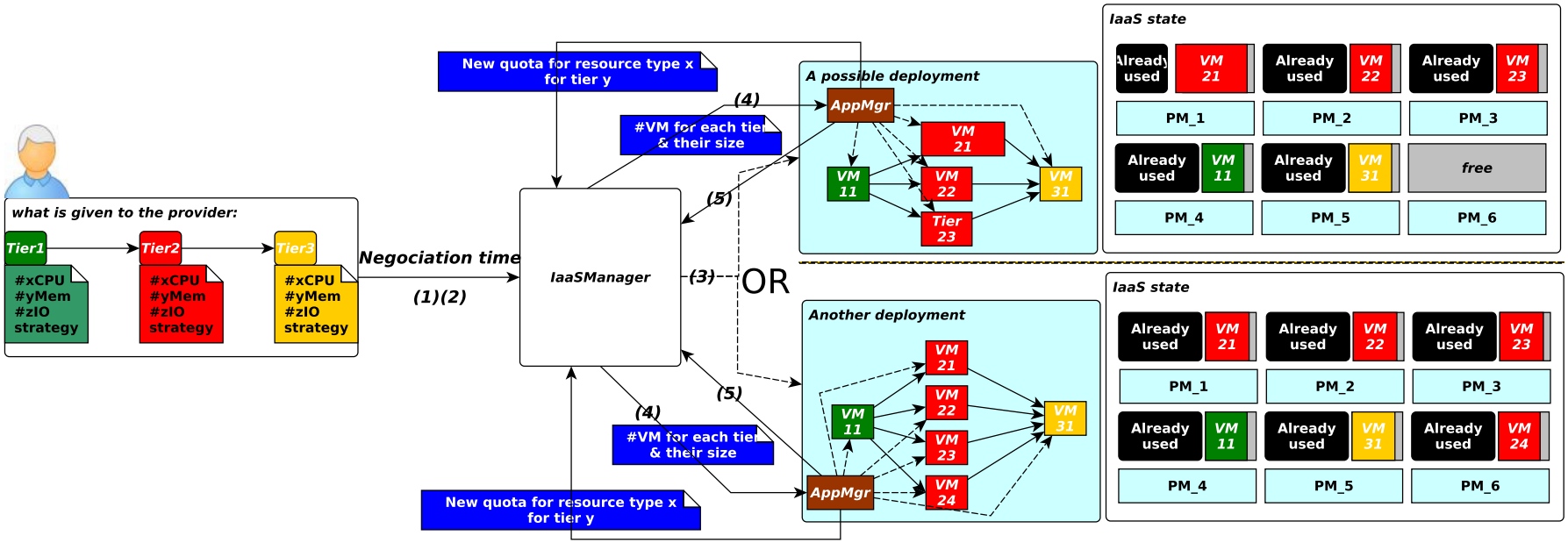

Figure 2.5: The general behaviour of our solution.

	E_SPECweb	E_SPECjAppserver	E_SPECmail
VTGRNM	HaProxy, InfraServer	HaProxy, DAS	Front, Mupdate
TGRNM	Web	Glassfish, MySQL	Imapd

Table 2.1: VGRNM and TGRNM: which model is appropriate to each E_SPECvirt tier?

about any resource changes (e.g. after a VM resize). Finally, the customer registers the AppManager endpoint with the IaaSManager.

- **(2)** The customer enters in what we call the "subscription phase". An application subscription is formalized as follows:
  $A = \{t_i(\#cpu, \#mem, \#io)\ and\ strategy | 1 \leq i \leq n\}$, where A represents a request to the provider (see Figure 2.5 left), n is the total number of tiers, t_i represents the i^{th} tier, $(\#cpu, \#mem, \#io)$ is the tier size, and strategy represents the allocation model (TGRNM or VGRNM).

- **(3)** From these information, the IaaSManager computes and starts the number of VMs needed to satisfy each tier. The first advantage (resource saving) of our solution can be observed during this step. Indeed, VM instantiation implies VM placement: which PMs will host instantiated VMs. An efficient VM placement algorithm avoids resource waste. In comparison with the traditional model in which VM sizes are constant and chosen by end-users, our model avoids PM fragmentation (the multiplication of holes). For instance, PM_2 and PM_3 in Figure 2.6 (the traditional model) have unused resources which would have been filled in our model (as shown in Figure 2.5 right).

- **(4)** When VMs are ready, the IaaSManager informs the AppManager about the number and the size of VMs for each tier so that the application can be configured accordingly (e.g. load balancing weights).

- **(5)** The AppManager informs back the IaaSManager when the application is ready. Resource changes are envisioned only after this notification.

As mentioned above, the traditional model (VGRNM) is still available in our solution because it could be suitable for some tiers. For instance, the MTMSA entry point (i.e. loadbalancer) needs a static well known IP address, thus a single VM. More generally, our solution is highly flexible in the sense that it is even possible to organize a tier in two groups so that each group uses its own allocation model (Section 2.7 presents a use case). The next section presents an application of our model to a well known set of cloud applications.

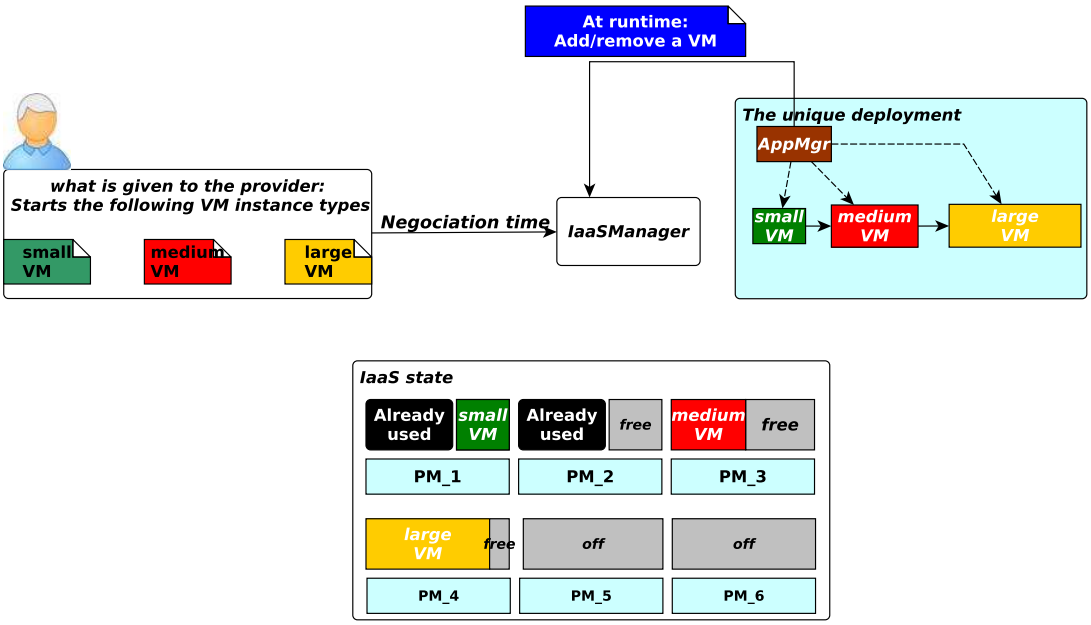

Figure 2.6: The traditional functioning of a cloud platform. The resource negotiation model is based on fixed size VMs (small, medium, large, etc.) requested by the end-user. The cloud operator has no information about the application type (its architecture) deployed within VMs. Furthermore, any modification of the application is initiated by the AppManager (VM addition or removal) according to workload fluctuations. This inflexibility is at the heart of VM consolidation limitations: see resource waste on PM_2 and PM_3.

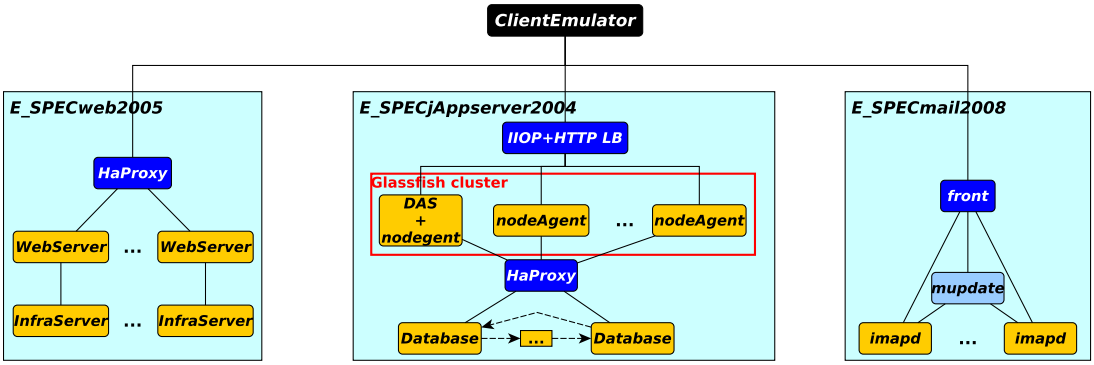

Figure 2.7: Architecture of E_SPECvirt.

2.5.2 Application of the model

SPECvirt_sc2010 [29] is a reference benchmark which has been used for evaluating the performance of the most common cloud platforms. It is composed of three main workloads which are the patched versions of more specific benchmarks: SPECweb2005[30] (web application), SPECjAppserver2004 [26] (JEE application), and SPECmail2008 [27] (mail application). SPECvirt_sc2010 also provides a test harness driver to run, monitor, and validate benchmark results. We relied on SPEC-virt_sc2010 in order to target the most popular cloud applications. For the needs of this work, we enhanced SPEC-virt_sc2010 by implementing the elasticity of each tier. This new version is called E_SPECvirt and it is composed of E_SPECweb2005, E_SPECjAppserver2004, and E_SPECmail2008. Figure 2.7 presents the new architecture. E_SPECweb2005 comprises one or more Apache[2] web servers with loadbalancing assured by HaProxy[14]. E_SPECjAppserver2004 is composed of a Glassfish[12] cluster with loadbalancing archived by both HaProxy (for HTTP requests) and Domain Administration Server (DAS) registry (for IIOP requests). HaProxy also provides loadbalancing for MySQL databases. To ensure consistency, all MySQL servers leverage a master-master replication [20]. Update requests received by a MySQL server are replicated to the others in a cyclic way. E_SPECmail2008 is achieved by Cyrus IMAP[6]. The latter provides three software types: a loadbalancer (called front), a database server which contains information about the location of all mailboxes (called mupdate), and multiple imapd slaves which serve IMAP requests.

Table 2.1 shows which HRNM submodel is suitable for each E_SPECvirt tier. VGRNM is used both for loadbalancers and for some software such as InfraServer, DAS, and Mupdate, which need to be known in advance throughout a unique static IP address (thus a single VM). All other tiers are provisioned using TGRNM.

2.6 Implementation of the model

In the cloud, a customer can request resources both at application subscription time or at runtime. There are two types of cloud actions at runtime:

- (C_1) the adjustment of both the number and the size of VMs while keeping the corresponding tier to the same size.
- (C_2) the adjustment of a tier size in response to workload variation.

C_1 operation types are initiated by the IaaSManager while C_2 operation types are initiated by the AppManager. The "subscription phase" can be seen as a C_2 operation: increase the tier size starting from zero. A runtime cloud action is taken only if the application performance insured by the provider (i.e. the Service Level Agreement) is respected. The procedure used to ensure the Service Level Agreement (SLA) is presented below.

2.6.1 SLA enforcement during VM split

One of the main goals of a cloud operator is to save resources. Thus, every time Assertion 1 is verified, it considers that there should be a better consolidation. In this respect, the IaaSManager tries to restructure application tiers by replacing "big" VMs with "smaller" VMs (VM split). The main objective of this operation is to improve VM consolidation (free as much PMs as possible). On the other hand, the customer is rather interested in the performance of its application. **There are cases where even if Assertion 1 is verified, the provider cannot split a VM**. These circumstances are promoted by two main factors. First, there is often a non-linear dependence between the performance and the amount of resource. For instance, a 2GB VM may not perform 2 times better than a 1GB VM. Second, there is always an overhead introduced by VM's operating system (OS) footprint. For an accurate VM split, we need to find a metric which exposes well the application performance. A suitable choice for our MTMSA seems to be the maximum application throughput (e.g. requests/sec for a web server). Based on this metric, we can safely split the VM without affecting the customer. For example, a customer will be satisfied with both, a single VM capable of 200 req/s or two VMs, each one capable of 100 req/s, considering that the streams are aggregated by the loadbalancer. To convert from resources to throughput, we introduce a function called $s2t_{tier}$ (size to throughput for a given tier). It takes as input a hole (#cpu,#mem,#io) and returns the throughput that a corresponding VM will deliver. The function is provided either by the customer[2] or by the provider. If the customer does not

[2]Customers may have such information since they need to predict how their applications will perform in a given VM.

```
1//This function is invoked at the end of each VM consolidation round
2void consolidationExtension (...) {
3 S1={PMs with holes}
4 thr_holes − "the sum of throughput of all holes"
5 choose P from S1 so that P has the biggest hole
6 thr_P − "the sum of throughput of all P's VMs"
7 if (thr_P > thr_holes)
8  return
9  foreach(VM v on P){
10   determine tierName of v
11   thr_v := s2t_tierName(sizeOf(v))
12   foreach (P_i ∈ S1 ∩ {P}) {
13    thr_hole := s2t_tierName(sizeOfhole(P_i))
14    if (thr_v<thr_hole){
15     resize v to thr_v
16          notify changes to AppManager
17          migrate v to P_i
18          break
19     } else {
20      enlarge or instantiate a VM in this hole
21          thr_v := thr_v − thr_hole
22     }
23     update S1
24    }
25  }
26  Switch−off PMs without VMs
27}
```

Figure 2.8: The StopGap algorithm.

have such information, the provider (IaaSManager) computes the function like Quasar[63]. The latter dynamically determines application throughput based on performance monitoring counters and collaborative filtering techniques. The estimation of $s2t_{tier}$ is beyond the scope of this work.

2.6.2 Resource management of type C_1

While HRNM can improve VMs' resource assignment at application subscription time, holes may also show up during runtime (e.g. a VM termination/migration). In order to address this issue, we introduce a VM consolidation extension called StopGap. Figure 2.8 presents in pseudo code the StopGap algorithm. It is complementary to the consolidation system which already runs within the IaaS. The only thing to do is to immediately invoke it after each VM consolidation round. For simplicity reasons, we are not presenting the pseudo code related to synchronization problems. In the real code version we

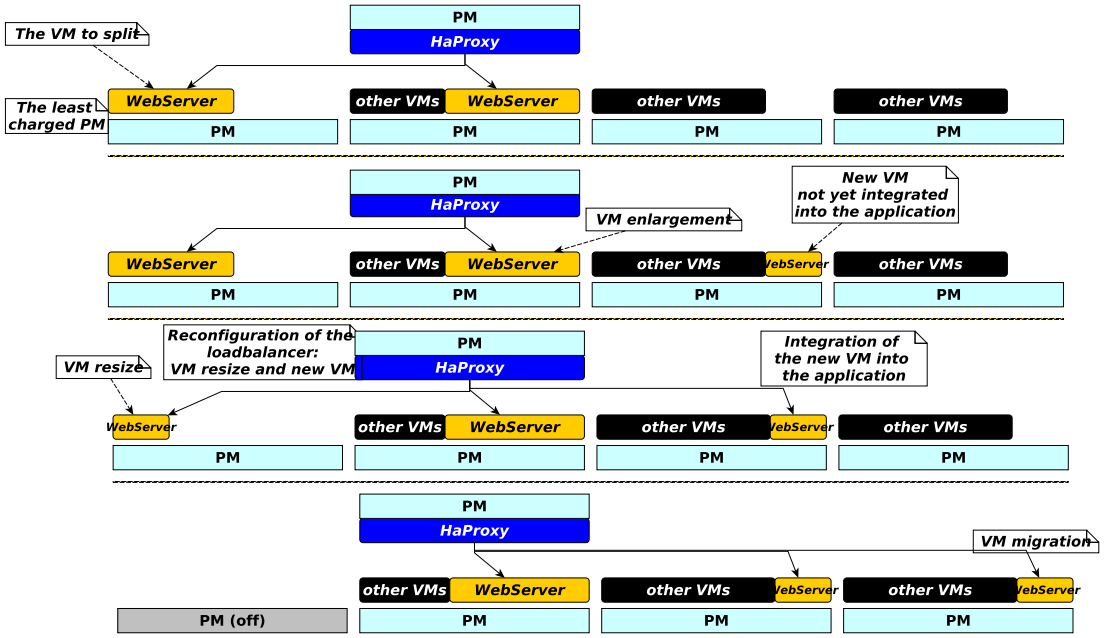

Figure 2.9: Illustration of StopGap on E_SPECweb2005.

used locks to avoid holes contention (see Section 2.7.1). Figure 2.9 illustrates the algorithm on a simple use case: the restructuring of the web server tier in Specvirt_sc2010. The StopGap algorithm is interpreted as follows. The reader can follow in parallel the illustration in Figure 2.9: top-down. First, we choose the least charged PM (line 5) (noted P). If the data center holes are unable to provide the necessary throughput for P's VMs (line 7), no application restructuring can be done without performance loss. Otherwise, we take a VM v from P (line 9). We iterate over the remaining PMs from S_1 (line 12) and we start to reconstruct v in the holes(lines 20-21). If a VM of the same tier exists on P_i we prefer to enlarge it instead of instantiating a new VM. Each time, we subtract the new VM's throughput from the throughput of v (line 21). When we find a hole which can provide the remaining throughput, we migrate v (lines 15-18). Notice that the reconfiguration of the application during VM reconstruction is only performed once all generated VMs are ready. By doing so, there is no wait time related to VM instantiation.

2.6.3 Resource management of type C_2

Due to workload variations, the AppManager may request a change in a tier's capacity/size. Figure 2.10 and 2.11 present in pseudo code the algorithms to shrink/enlarge a tier. It works as follows. The AppManager communicates to the IaaSManager the desired tier variation (Δ). Concerning the tier downscale

```
1 //decrease tierName by Δ
2 S1={PMs which run a VM belonging to the tierName}
3 thrΔ = s2t_tierName(Δ)
4 decLabel:
5 Let v be "the smallest VM of tierName in S2"
6 thrv := s2t_tierName(sizeOf(v))
7 if (thrv==thrΔ){
8         record v for termination
9 }else  if (thrv < thrΔ){
10        record v for termination
11        thrΔ := thrΔ − thrv
12        remove P from S2
13        goto decLabel
14 }else{
15        shrink v until s2t_tierName(sizeOf(v)) == thrv − tΔ
16 }
17 notify changes to the AppManager
18 when(ack is received){
19        terminate recorded VMs
20        free empty PMs
21 }
```

Figure 2.10: Tier size decrease algorithm.

```
1 //increase tierName by Δ
2 S1={PMs with holes}
3 thrΔ = s2t_tierName(Δ)
4 foreach (Pi ∈ S1){
5         thrhole := s2t_tierName(sizeOfhole(Pi))
6         enlarge or instantiate a VM in this hole
7         thrΔ := thrΔ − thrhole
8 }
9 incLabel:
10        turn−on a new PM P
11        instantiate a new VM v on P
12        thrv := s2t_tierName(sizeOf(v))
13        thrΔ := thrΔ − thrv
14        if (thrΔ > 0)
15                goto incLabel
16 notify changes to the AppManager
```

Figure 2.11: Tier size increase algorithm.

(Figure 2.10), the IaaSManager prioritizes VM termination (line 8, 10) rather than shrinking a group of VMs (line 15). Thus, the overhead caused by VM's OS footprint is minimized. Regarding the tier enlargement (Figure 2.11), the priority is placed on resizing (vertical up-scaling) the existing VMs. If at the end, the request is not completely satisfied, a set of VMs are instantiated according to available holes (line 6). If all holes are filled up and the request is still not completely satisfied, PMs are switched-on and new VMs are instantiated atop them (line 10-11). The IaaSManager always informs back the AppManager about the operations it has performed (i.e new size for old VMs, new VMs and their size, or terminated VMs). Subsequently, the AppManager answers with an ACK message. The IaaSManager only terminates VMs upon receiving that message. This prevents the termination of a VM which is still servicing requests.

2.7 Evaluations

In order to test our approach, we performed two evaluation types. The first type evaluates our solution impact on customer applications, provided by SPECvirt_sc2010 [29] (presented in Section 2.5.2). The second evaluation type focuses on VM consolidation improvements.

2.7.1 Experimental environment

The first type experiments were performed using a prototype implemented within our private IaaS. It is composed of 7 HP Compaq Elite 8300, connected with a 1Gbps switch. Each node is equipped with an Intel Core i7-3770 3.4GHz and 8GB RAM. One node is dedicated to management systems (IaaSManager, NFS server and additional networking services: DNS, DHCP). The others are used as resource pool. To show the generic facet of our solution, the prototypes have been implemented for two reference IaaSManger systems: OpenStack [22] and OpenNebula[21]. Both systems are virtualized with Xen 4.2.0. The integration of our solution with these systems is straightforward. We have implemented the resource negotiation model on top of both OpenStack and OpenNebula public APIs. Concerning VM consolidation, OpenStack relies on OpenStack Neat[23]. It is an external and extensible framework which is provided with a default consolidation system. Our solution requires a minor extension to OpenStack Neat. We only extended its "global manager" component, which implements the consolidation algorithm. Two modifications were necessary: one LoC at the end of the consolidation algorithm to invoke StopGap (Figure 2.8), and about 5 LoCs for locking PMs whose VMs are subject

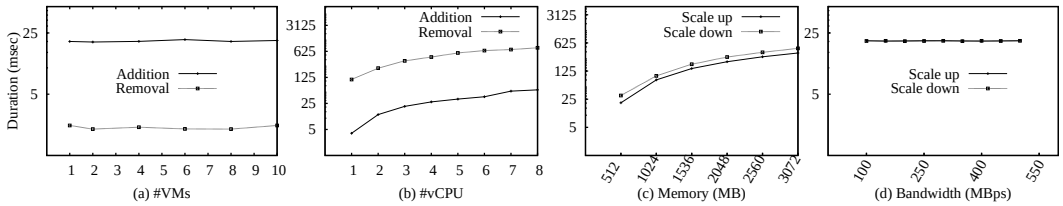

Figure 2.12: Horizontal and vertical scaling durations (N.B.: log-scale y-axes). VM instantiation or termination time is quasi constant regardless the number of VMs. vCPU addition or removal time increases almost linearly with the number of vCPUs. Similar results are observed for the main memory. Bandwidth adjustment always uses the same duration.

to resize. This prototype is used to evaluate the impact of our resource allocation model. Concerning OpenNebula, it does not implement any dynamic VM consolidation module. However, it is built so that the integration of a consolidation engine is elementary. In OpenNebula, the single component which we patched is the "Scheduler".

2.7.2 Impact on end-user's applications

In our solution, two new operation types can impact the performance of end-user applications:

- Vertical and horizontal scaling. By leveraging HRNM, the IaaSManager may combine vertical scaling (VM size adjustment) and horizontal scaling (add/remove a VM) to dynamically restructure an application tier. These operations are the basis for both VM split and VM enlargement.
- Application reconfiguration: VM spliting and enlargement require the adaptation/reconfiguration of the application level (e.g. weight adjustment).

Impact of vertical and horizontal scaling

The influence of each operations is evaluated separately. Figure 2.12 presents the experiment results. In Figure 2.12(a) we can note that the time taken to instantiate or terminate VMs is quasi constant regardless the number of VMs (about 20 msec to instantiate and 2 msec to terminate). This is due to the parallel VM instantiation/termination. Notice that neither VM instantiation nor VM termination impact applications which run on the same machine since these operations do not require high amount of resource for completion.

```
1 //AppServer: VM addition
2 Update haproxy.cfg and reload it
3 Update glassfish.env and reload it
4 Start a new Glassfish node agent
5 Start a new Glassfish server
6 //AppServer: VM removal
7 Update haproxy.cfg and reload it
8 Update glassfish.env and reload it
9 Update domain.xml and reload DAS
10 Stop Glassfish server
11 Stop Glassfish node agent
12 //AppServer: VM resize
13 Update haproxy.cfg and reload it
14 Update glassfish.env and reload it
15 //web: VM addition/removal/resize
16 Update haproxy.cfg and reload it
17 //mail: VM addition
18 Update the front server
19 Reorganize (via migration) mailboxes ↘
       according to backend's size
```

```
1 //DB: VM addition
2 Start MySQL with specdb database
3 DB pre-sync
4 Lock all the active DBs
5 Execute the final rsync
6 Unlock the DBs
7 Update the circular relationship of ↘
       MySQL slaves
8 Update haproxy.cfg and reload it
9 //DB: VM removal
10 Update haproxy.cfg and reload it
11 Update MySQL slaves relationship
12 //DB: VM resize
13 Update haproxy.cfg and reload it
14 //mail:VM removal
15 Migrate mailboxes from the removed server
16 Update the front server
17 //mail:VM resize
18 Reorganize mailboxes according to ↘
       backend's size
```

Table 2.2: E_SPECvirt *AppManager*s reconfiguration algorithms.

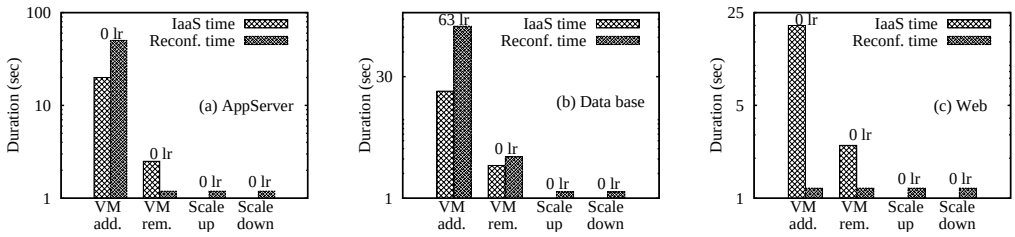

Figure 2.13: Horizontal and vertical scaling impact on (a)App Server, (b)DB Server and (c)Web Server (N.B.: log-scale y-axes). Except the addition of a new data base server, no tier suffers from our solution.

Concerning vertical scaling, we evaluated addition/removal of each resource type individually. We evaluated the time taken to make added resources (respectively removed resources) available (respectively unavailable) inside the VM. As reported in Figure 2.12(b), vCPU addition or removal time increases almost linearly with the number of vCPUs. This is explained by the fact that any adjustment in the number of vCPUs triggers the sequential execution of a set of watchers (according to the number of vCPUs). Notice that vCPU removal costs about 20 times more than addition.

Similar results have been reported for the main memory. Its shrinking corresponds to the time taken by the VM to free memory pages. This operation is triggered by a balloon driver which resides within the VM. Concerning memory addition, it corresponds to the time taken by the hypervisor to both acquire machine memory pages (which is straightforward in the context of our solution since it uses holes) and update the memory page list used by the target VM. Last but not least, bandwidth adjustment always implies a constant time. Since Xen does not manage bandwidth allocation, we relied on tc[17], a Linux tool which quickly takes into account the bandwidth adjustment. Every time a packet is sent or received, tc checks if the bandwidth limit is reached. Thus, a bandwidth adjustment is immediately taken into account.

Impact of dynamic reconfigurations

The second set of experiments evaluate both the application reconfiguration time and the consequences of this operation. The adopted impact indicator is the number of lost requests during the reconfiguration (noted lr). For each experiment, the workload is chosen so that VMs are saturated. Table 2.2 presents in a pseudo-code the reconfiguration algorithms we have implemented for each tier. Figure 2.13 and 2.14 report the results of this second set of experiments, interpreted as follows. The number of lr is shown atop each pair of histogram bars.

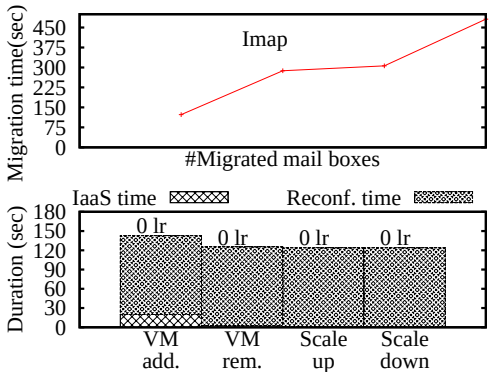

Figure 2.14: Impact of horizontal and vertical scaling on Imap Server. No request is lost during the reconfiguration. The migration time increases almost linearly with the number of migrated mailboxes.

- **Application server tier** (Figure 2.13(a)). Except the integration of a new VM which takes some time (the first two bars), the reconfiguration of the application server tier is straightforward. During this operation, no request is lost. Our solution does not incur major issues for this tier.

- **Database tier** (Figure 2.13(b)). A new database integration within the application is relatively expensive (the first two bars). During this operation, the database tier is out of service for a few moments due to synchronization reasons. This is the only situation which leads to some lost requests. Therefore, our solution could become negative for E_SPECj-Appserver2004 if the addition of new MySQL VMs occurs frequently. This problem does not appear when removing or vertically scaling a database VM since no synchronization is needed. In these cases, only the loadbalancer needs to be reconfigured.

- **Web tier** (Figure 2.13(c)). The web tier reconfiguration is straightforward since it only requires a loadbalancer update. Our solution does not impact this tier.

- **Mail tier** (Figure 2.14). The time taken at the application level to reconfigure the mail tier (Cyrus IMAP) is almost the same regardless the reconfiguration option (Figure 2.14 bottom). In any case the same number of mail boxes needs to be migrated. Due to the mailbox live migration implemented by Cyrus, no request is lost during the reconfiguration. The migration time increases almost linearly with the number of migrated mailboxes (Figure 2.14 top).

Impact of multiple reconfigurations. We tend to conclude from the above experiments that the impact of a single reconfiguration is almost negligi-

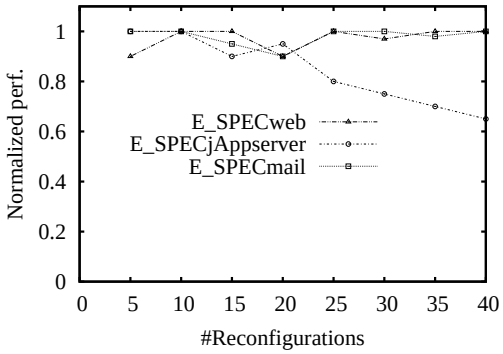

Figure 2.15: Impact of performing several reconfiguration operations. Both E_SPECmail and E_SPECweb are relatively little impacted by the multiplication of reconfiguration operations. In contrast, E_SPECjAppserver's performance starts to degenerate after about 20 reconfigurations. The degradation comes from the synchronization of the data base tier, which requires a little downtime of the application.

ble. However, the multiplication of these actions on a group of VMs belonging to the same application could be harmful. Figure 2.15 presents the normalized performance of each specific benchmark when the number of reconfiguration operations varies. We can note that both E_SPECmail and E_SPECweb are relatively little impacted by the multiplication of these operations. This is not the case for E_SPECjAppserver whose performance starts to degenerate after about 20 reconfigurations. The number of additions of new database VMs increases. To minimize this impact, the algorithm presented in Figure 2.8 has been improved for fairness. The reconfiguration operations which are performed in order to improve VM consolidation are fairly distributed among cloud applications. Thus, PMs whose VMs are subject to split are fairly chosen.

Synthesis

In comparison with horizontal scaling, vertical scaling globally provides better results regarding reconfiguration duration and performance degradation. Several reasons explained that. First of all, reconfiguration operations required to be performed at application level after a vertical scaling are most of the time less complex than those needed after an horizontal scaling (see algorithms in Table 2.2). Secondly, resource (un)plug-in is faster (in mere microseconds) than VM instantiation/termination (in mere seconds). These two options are showcased in our solution.

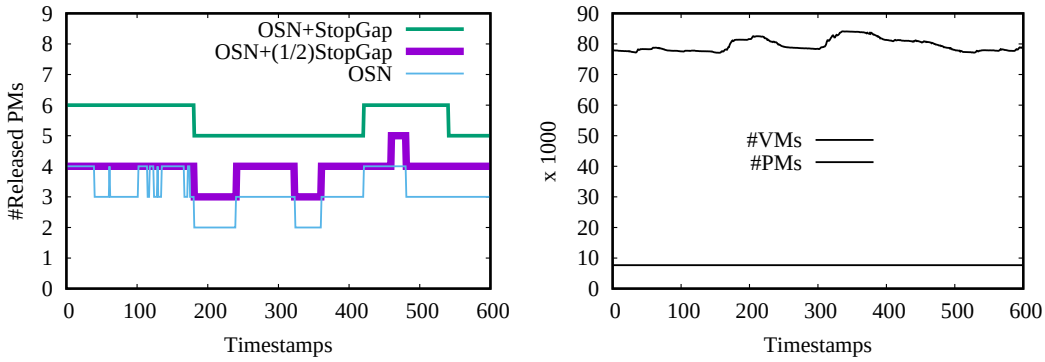

Figure 2.16: (top) The subset of Google data center traces we used. (bottom) Resource saving on google traces when our solution is used.

2.7.3 Resource saving and scalability

Resource saving The main goal of our contribution is resource saving. For this evaluation type we rely on Google data center traces obtained from [13]. Before presenting the results, let us firstly introduce how we interpreted Google traces. They represent the execution of thousands of jobs monitored during 29 days. Each job is composed of several tasks and every task runs within a container. For each container, we know the amount of resource used by the job and the PM on which it is executing. We considered a job as a customer application where its number of tasks correspond to the number of tiers. Therefore, a container is seen as the VM allocated during the first resource allocation request. The total number of PMs involved in these traces is 12583, organized into eleven types. For readability, we only present in this work the analysis of a subset of these traces. It includes up to 7669 PMs and 82531 VMs. Figure 2.16 summarizes its content. We evaluated how the StopGap extension may improve OpenStack Neat (OSN for short). The number of freed PMs is compared when OSN runs in three situations: alone (noted 'OSN'), in combination with our solution when every second tier leverages StopGap (noted 'OSN+(1/2)StopGap'), and in combination with our solution when all tiers leverage StopGap (noted 'OSN+StopGap'). Figure 2.16 (left plot) presents the results of these experiments. We can notice that both OSN+ (1/2)StopGap and OSN+StopGap perform better than the standard consolidation system (i.e. OSN). In the case of OSN+StopGap, OSN is enhanced with up to 62.5%.

Scalability StopGap complexity depends on the efficiency of the original consolidation algorithm employed by the data center. The worst case complexity corresponds to the use of StopGap as the only consolidation engine. Although this is not its main goal, StopGap can play that role in the ab-

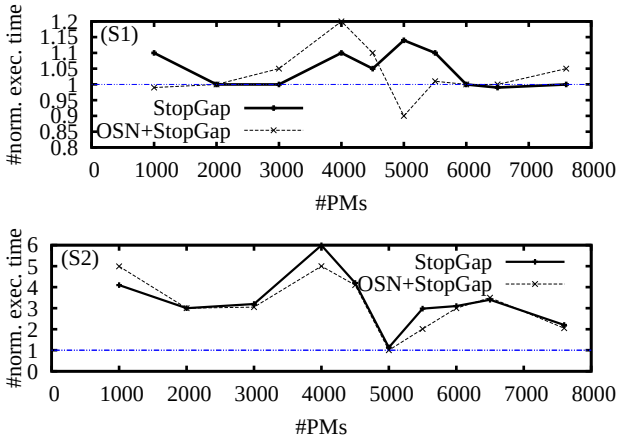

Figure 2.17: Overhead of our solution.

sence of a consolidation system. In this case, its complexity is the same as most FFD bin-packing algorithms [67]. The consolidation algorithm used by OSN has the complexity $\mathcal{O}(n \times m)$, where n is the number of PMs and m is the number of VMs to be relocated. We have also relied on Google traces to evaluate and compare both StopGap and OSN scalability. We considered two extreme datacenter states (S1 and S2) which respectively represent the highest and the lowest OSN efficiency. From Figure 2.16 left, we choose S1 and S2 to respectively be the timestamps 450 and 200. For each situation, we executed three consolidation algorithms (OSN, StopGap, and OSN+StopGap) on different subsets from the original set of PMs. Normalized execution times (against OSN) are plotted in Figure 2.17. In the most efficient case (S1), we can notice that both StopGap and OSN+StopGap are close to OSN. Conversely, both perform better than OSN when it is not efficient (S2). In this case, StopGap as well as OSN+StopGap does the entire consolidation effort. The minimum value noticed in Figure 2.17(top) represents another observation: OSN has the highest efficiency when it operates on 5000 PMs.

2.8 Related Work

Memory footprint improvements. Significant research has been devoted to improve workload consolidation in data centers [37]. Some studies have investigated VM memory footprint reduction to increase VM consolidation ratio. Among these, memory compression and memory over commitment [137, 43] are promising. In the same vein, [135] extends the VM ballooning technique to software for increasing the density of software colocation in the same VM. Xen

offers the so called "stub domain"[3]. This is a lightweight VM which requires limited memory (about 32MB) for its execution.

Uncoordinated Policies. Many research projects focus on improving resource management on client-side [131, 56, 88, 69]. They aim at improving the workload prediction and the allocation of VMs for replication. On the provider side, research mainly focuses on (1) size of resource slices, i.e. provided VM size; or (2) VM placement, i.e. VM allocation and migration across physical servers to improve infrastructure utilization ratio. Various algorithms are proposed to solve the VM packing problem [53, 122]. They take into account various factors such as real resource usage, VM loads, etc. However, in a datacenter, the resource demands of a VM are not fixed. Thus, several authors propose heuristics which address the dynamicity of this problem. Beloglazov et al [48] propose an algorithm which take consolidation decisions based on a minimum and a maximum PM utilization threshold. Since live VM migration is a costly operation, Murtazaev et al proposes Sercon [121], a consolidation algorithm that minimize not only the number of active PMs but also the number of VM migrations. Further, the state-of-the-art algorithms are leveraged in order to build dynamic consolidation systems. For example, Snooze [70] is an open-source consolidation system build on top of Sercon. Snooze implements a decentralized resource management on three layers: local controllers on each PM, group managers which survey a set of local controllers and a group leader among the group managers. However, the previous solutions operate independently either on the client side or the provider side. For this reason, their potential effectiveness may be narrowed.

Cooperative Policies. Authors in [96] describe a model to coordinate different resource management policies from both cloud actors' point of views. The proposed approach allows the customer to specify the resource management constraints, including computing capacity, load thresholds for each host and for each subnet before an allocation of a new VM, etc. The authors also describe a set of affinity rules for imposing VM collocation in the IaaS, which is a form of knowledge sharing. The authors have asserted that this model allows an efficient allocation of services on virtualized resources. This work is a first step in the direction of coordinated policies.

Nguyen Van et al. [124] describe research works closely-related to ours. The authors propose an autonomic resource management system to deal with the requirements of dynamic VMs provisioning and placement. They take both application level SLA and resource cost into account, and support various application types and workloads. Globally, the authors clearly separate two resource management levels: Local Decision Modules (LDM) and the Global

[3]http://wiki.xen.org/wiki/StubDom

Decision Module (GDM). The two are respectively similar to our AppManager and IaaSManager. These two decision modules work cooperatively: the LDM makes requests to the GDM to allocate and deallocate VMs, the GDM may request back changes to allocated virtual machines. The results reported in [124] are only based on simulations.

Christina Delimitrou et al. [63] presents Quasar, a non-virtualized cluster management solution which adopts an approach philosophically close to our. It asserts that the customers are not able to correctly estimate the amount of resource needed by their applications to run efficiently. The customers are allowed to express their needs in terms of QoS constraints, instead of low level resource requirements. The management system will allocate the appropriate amount of resource which ensures the requested QoS. Like our solution, knowledge about applications and their expected QoS is shipped to the cloud management system. This cooperation enables a smarter resource management. Contrary to our solution, Quasar manages non-virtualized clusters and does not address any dynamic consolidation issues.

Elastic workloads. Zhenhua Guo et al. [78] proposes a mechanism to split map-reduce tasks for loadbalancing reasons. Since this application type may also be split, it could be included (along with the MTMSA) in the list of suitable applications for our model.

Middleboxes represent an obstacle in the scalability of web applications. In order to address this limitation, Shriram Rajagopalan et al. [132] come up with a framework capable of splitting the middlebox VMs (e.g. loadbalancers, firewalls). Consequently, the entry point of an application (i.e. the loadbalancer) may now be distributed over multiple VMs. This work may exempt us from the need of the traditional model since the entry point of an MTMSA application could now be negotiated at the granularity of a tier (TGRNM).

2.9 Conclusion

This work proposes a way to combine cooperative resource management with elastic VMs. Knowledge about customer's applications (e.g. tier instances) is shared with the IaaS provider so that IaaSManager can better optimize the infrastructure. Based on this shared knowledge, the provider can split or enlarge VMs. Our proposed cooperative IaaS can be considered from two different perspectives: a PaaS extension or a hybrid IaaS-PaaS model. We validated the applicability of our solution through extensive experiments. Relying on Google datacenter traces, we evaluated our solution's benefits in terms of resource saving. It improves OpenStack consolidation engine by about 62.5%, without any additional overhead.

Chapter 3

Towards memory desagregation with the zombie state

3.1 Introduction

In recent years, we have witnessed some tectonic shifts in the computing landscape. First, with virtualization and containerization technologies becoming mainstream, we were finally able to decouple applications and their operation environments from the underlying hardware. Then, with cloud computing democratizing access to compute infrastructures and platforms, we have transformed these into services that can be provisioned and consumed on demand. These advances not only changed how we design software and build systems today, but also opened up many new opportunities for improving computing efficiency and cost.

With virtualization came the simplified multitenancy of operating systems, consolidation, virtual machine (VM) migration, distributed, dynamic resource and power management [86]. These were aimed at improving the notoriously-low data center (DC) server utilization [46], reducing cost, and dramatically improving power efficiency. With the cloud, we were able to push the boundaries further. Economies of scale, advents of software-based availability enables us to keep compute devices simple, cheap and designed for perfect efficiency meeting observed demands. By continuously placing thousands, if not millions, of requests on these nodes we can keep them busy, highly-utilized, and working at their optimal point of energy efficiency. Essentially, with cloud and virtualization, we could consider the compute infrastructure as one giant computer that theoretically has infinite resources, yet operates nimbly, with almost perfect efficiency based on demand.

Unfortunately the reality has been far from this. After myriad projects, papers, products and services, we now have giant computers at our fingertips on demand that are fast, easy to use, yet still highly inefficient in their resource

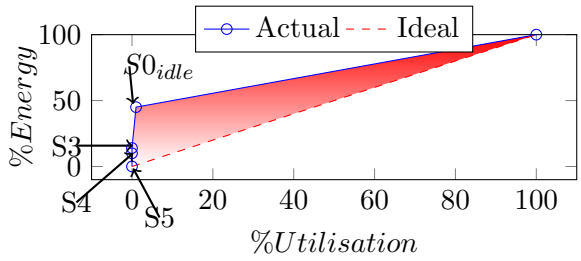

Figure 3.1: Energy consumption depending on server utilization. The solid line shows the usual server energy consumption, while the dashed line plots energy-proportional behavior.

utilization and energy efficiency. The average compute node utilization in most cloud offerings is well below 50% [46, 63, 113]. So where has this gone wrong? One main reason behind the mismatch between our expectation and the reality is our inability to efficiently pack multidimensional application needs to the underlying bundled compute resources such as CPU, memory and network. And this is because what the infrastructure offered in its evolution did not meet what software demanded in its evolution. Over the last several years we have seen new applications emerge with vastly growing memory demands, while platform evolution continued to offer more CPU capacity growth than memory, referred to as *memory capacity wall* [98]. Therefore, we are unable to leverage consolidation, efficient packing and balanced utilization of resources in the cloud as memory demand direction saturates before the other dimensions.

This observation is actually one of the underpinnings of another significant shift that has been gaining momentum, namely *disaggregated computing* [98], which aims to change the server-centric view of the infrastructure to a resource-centric view. In this model, each resource dimension can evolve and expand independently, and thus respond to evolving application demands. Disaggregated computing has the potential to lead us to our desired computing model that is nimble, boundless and highly resource and energy efficient. However, it is a solution for the long term that requires fundamental changes to compute hierarchy and operations.

In our work we explore a short-term solution that can have the benefits of disaggregation, yet that can be applied by introducing small changes to general-purpose computing hardware. Our solution targets the immediate problem at hand, disaggregating memory resources and unbundling them from other compute resources (e.g. CPU). We propose a new *Zombie* (S_z) ACPI state that is similar to suspend-to-RAM (S3) state in latency and power efficiency, but keeps the memory resources of a server active and usable by other nodes. In other words, a server in S_z state is a *Zombie* as it is brain-dead (CPU-

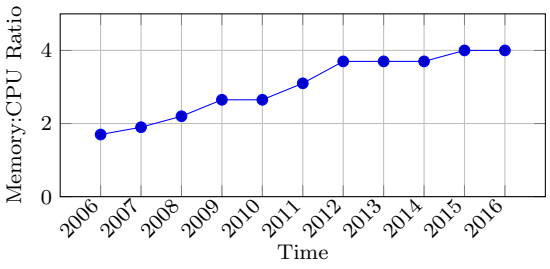

Figure 3.2: The $memory(GiB)$: $cpu(GHz)$ ratio for all introduced `m<n>.<size>` instances in AWS over the last ten years.

dead), limps along consuming minimal resources (low-energy), but still has basic motor functions such as serving memory (memory-alive).

Even if the mainstream hardware does not currently support the S_z ACPI state, its implementation is fairly simple. S_z only requires completely independent power domains for CPU and memory. In order to evaluate the benefits of the Zombie technology, in chapter 4 we present *ZombieStack*, the software stack needed to leverage the S_z state. Even if we do not own an S_z compatible hardware, we estimated the energy consumption of a server in S_z state based on a model. Our experimental evaluations demonstrate that the *Zombie* technology improves energy efficiency on datacenter workloads by up to 67%, which is 86% better than state-of-the-art consolidation techniques.

In the rest of this chapter, we first present some related background and motivation for this work. We introduce the *zombie (S_z)* state and its design in section 3.4. Section 3.5 presents the related works. Then, we present the experimental evaluation in section 3.6, highlighting the significant improvements with this approach. Last, section 3.7 offers my conclusions.

3.2 Motivation

As we have discussed in the introduction, we have seen substantial opportunity and effort in improving resource utilization and energy efficiency with virtualization and cloud. As presented in section 3.5, there have been myriad efforts at the hardware, virtualization and the ensemble to attack this problem on multiple fronts, improving energy efficiency and overheads of low-power states and driving up server utilization and consolidation. The motivation behind driving server utilization has been to improve consolidation ratios to reduce cost, while also benefiting from the widely-known observation that servers are more energy efficient (or *energy proportional*) at higher utilizations as depicted in Figure 3.1.

While these prior techniques have improved utilization numbers significantly

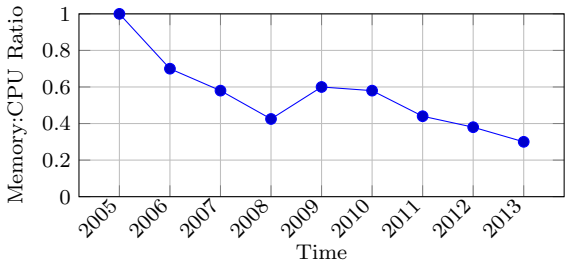

Figure 3.3: Normalized *memory* : *cpu* capacity ratio for multiple server generations.

and improved energy-efficiency of systems, it is still difficult to actually reach server loads near 50% in even the most advanced implementations. Some works demonstrated that one main reason for this is a growing mismatch between platform resources and growing application demands [152, 98]. This is due to the combination of two opposite trends. First, we observe that emerging applications such as search, in-memory data stores, and analytics have developed a fast-growing appetite for memory resources to minimize request latencies, in response to real-time needs. This results in **a growing gap between memory and CPU demand** as memory demand has been growing much more rapidly. To validate this, we looked back at the historical instance sizes in AWS, and the observations were quite telling. As expected, AWS has gradually introduced newer-generation and bigger-size instances over time, as compute demands grew. However, when we look at the growth trend among different resources, we see that the memory configuration growth substantially outpaced that of compute. Figure 3.2 shows the ratio of memory size to CPU size for all AWS instances of family `m<n>.<size>`, where `n` is the generation and `size` the size attribute. The figure shows the general trend that while demand on both resources has grown substantially, the rate of growth for memory demand has been approximately $2X$ of CPU demand.

The second trend we observe is that **there is a growing gap between Memory and CPU supply in the reverse direction.** On the one hand, the International Technology Roadmap for Semiconductors (ITRS) estimates that the pin count at a socket level is likely to remain constant [16]. As a result, the number of channels per socket is expected to be near-constant. In addition, the rate of growth in DIMM density is starting to wane (2X every three years versus 2X every two years), and the DIMM count per channel is declining (e.g. two DIMMs per channel on DDR3 versus eight for DDR) [18]. On the other hand, another trend points the increased number of cores per socket, with some studies predicting a two-fold increase every two years [40]. If the trends continue, the memory capacity per core will drop by 30% every

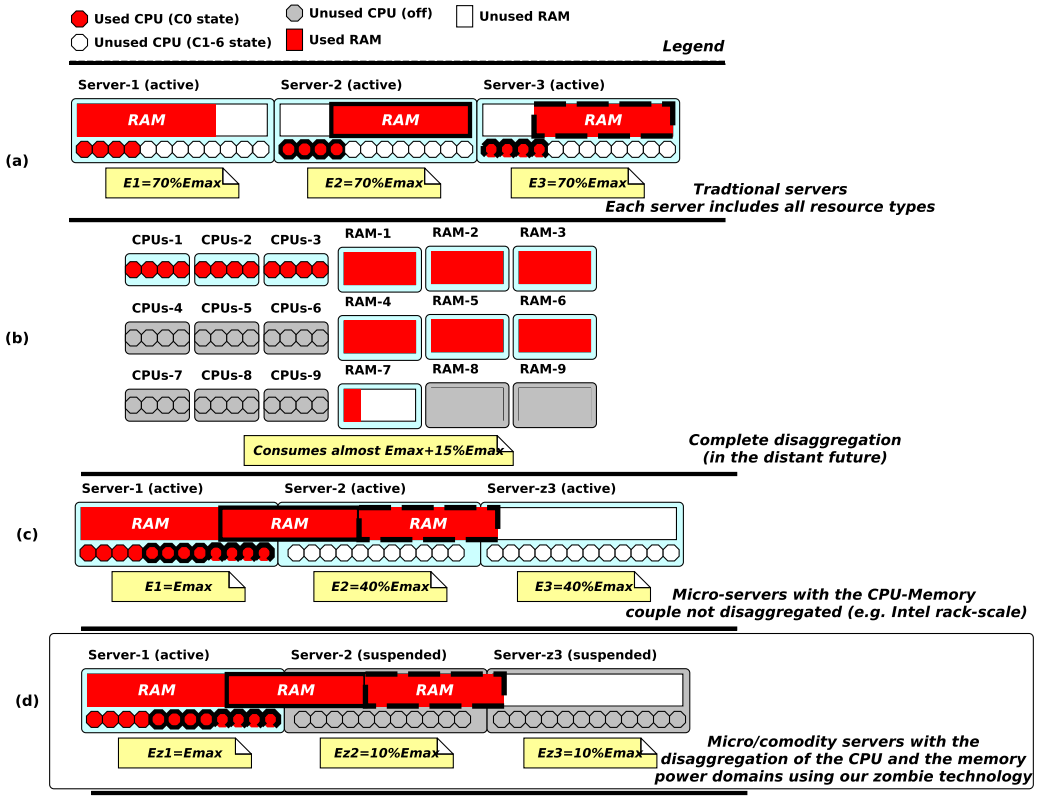

Figure 3.4: Resource disaggregation: summary of existing solutions. We illustrate each solution at the rack-level, considering a rack composed of three serves. We estimate the energy consumed by the rack in each solution. We can see that our proposition (d) results in the optimal energy proportionality while requiring less hardware and software modification.

two years, as depicted in Figure 3.3 [152].

These two opposing trends show that applications have been evolving in the direction where they require more memory than CPU, while servers are evolving to provide more CPU than memory. This situation leads to poor VM consolidation ratio [82, 98], thus energy waste as illustrated in Figure 3.4(a).

3.3 Background

Core vs. Uncore The current trend in modern processors is to reorganize the functions critical to the core, making them physically closer to the core on-die, thereby reducing their access latency. Many of this functions come from the historical northbridge. Figure 3.5 presents the architecture of a modern Intel processor. The uncore (or 'system agent') subsystem regroups all modules which are not directly related to data processing. Among them we can mention

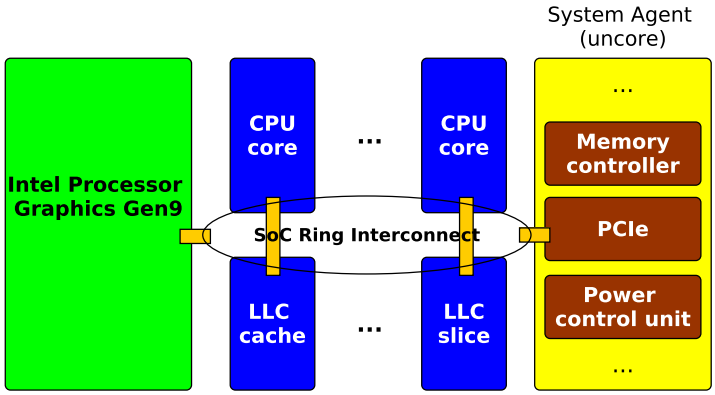

Figure 3.5: The architecture of a modern processor. All modules which are not related to data processing belong to 'uncore'.

the memory controller, the PCIe controller, the package control unit (which includes the power management logic and controller firmware), etc.

ACPI. The Advanced Configuration and Power Interface (ACPI) is a standard that allows an OS to perform power management on individual components (e.g. CPU cores, network adapters, storage devices, etc.) or the system as a whole. The global (system level) power states are named from $S0$ to $S5$. $S0$ represents the most active state (i.e. the CPU is running and executes instructions) while $S5$ is the most inactive one (i.e. the machine is turned off without saving any system state). $S3$ is an intermediate state also called Suspend-to-RAM (see Figure 3.6). It cuts power to most of the components except the RAM memory (which is in self-refresh mode and stores the system state), the network adapter (which is used to wake-on-LAN the machine) and a part of the PCI/PCIe circuitry.

3.4 *Zombie* (S_z): A Sleep State for Servers

In this section we describe our new ACPI sleep state (S-state) called *zombie* or S_z state (see Figure 3.7). The S_z state is similar to S3 state, with one key difference. It keeps the memory banks of the platform active and remotely accessible even when the server is suspended. Our main motivation in introducing this new S_z state is to address the growing gap between the memory demand vs. supply and the CPU demand vs. supply discussed earlier. With the S_z state, an application running on one platform can "borrow" memory from another, otherwise suspended, platform. This feature is provided neither by the ACPI specification nor by existing hardware or OS distributions.

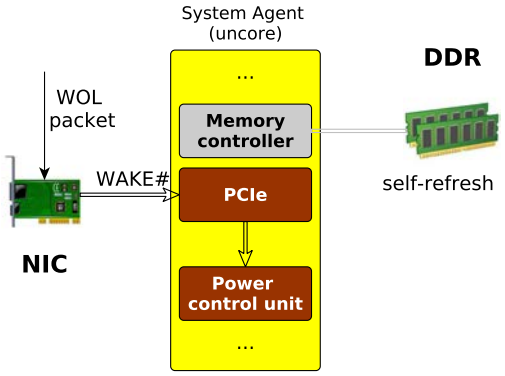

Figure 3.6: The state of the uncore subsystem in $S3$. The memory controller is powered off and the memory banks are in self-refresh mode. However, some components are powered-on and ready for the wake-on-LAN. Following a WOL magic packet, the NIC sends a WAKE signal to the power management controller which perform the system wake-up.

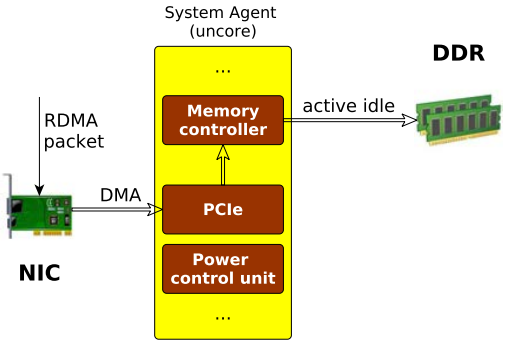

Figure 3.7: The state of the uncore subsystem in S_z. The memory controller, the memory banks and the entire PCIe circuitry are now active. In this state, the RDMA NIC is able to address the entire memory.

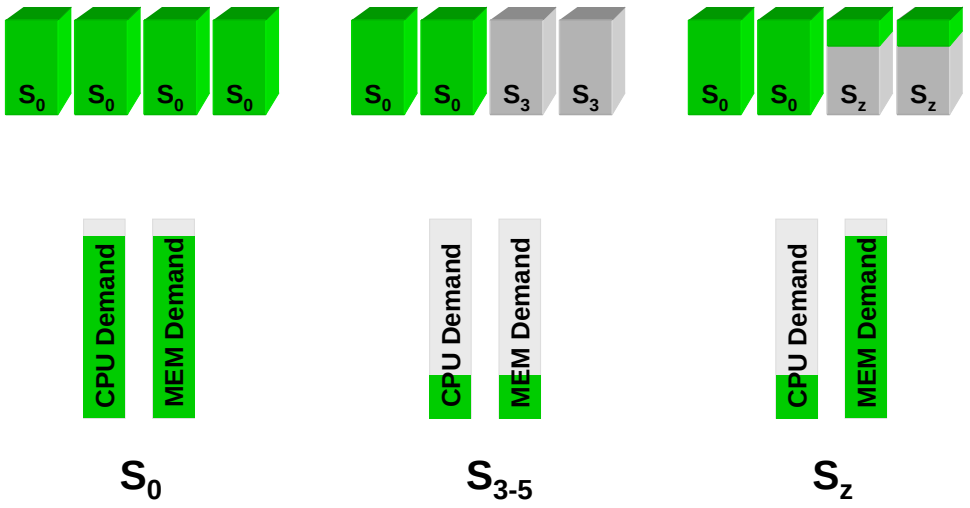

Figure 3.8: S_z state operation compared to S3 and S0.

The S_z state operates similarly to the S3 state for the most part. All components are turned off except the main memory and part of the network is kept active to serve remote memory requests. The memory behavior of S_z mimics that of *Si0x* state specifications, where the memory is kept in active idle, unlike the low-power self refresh mode of S3. The S_z State enables a nice compromise for a practical step towards disaggregated computing for memory. A general-purpose compute node can be used as a full-fledged platform when demand on resources is high, can be suspended to S3, S4 or S5 when demand is low, and can be kept in S_z state when compute demand is low, but the aggregate memory demand still requires the node to serve memory. Figure 3.8 shows the operation mode of S_z in comparison to the traditional S-states.

3.4.1 S_z State Design

The implementation of the new S_z state needs support from the manufacturer since it requires modifications across the stack from hardware and firmware to the OS, as well as to the ACPI specifications. At the hardware level, when a server enters ACPI S states, it follows a sequence to shutdown several power rails to the board components. As the memory and the networking logic for remote memory access need to remain active, power lines for these components require additional switches and control signaling for S_z enter/exit. State management hardware needs modifications to include the new S state and additional signals for triggering the right power state change actions for S_z. System management hardware needs additional signals from the participating chips for reporting and idempotence of actions. These signals are used to determine the state of the devices, when a state transition is active and to report the power state of the server. Firmware is involved in S-state transitions during boot up

```
1@+echo zom > /sys/power/state@
2@+pm_suspend@
3 enter_state
4suspend_prepare
5suspend_devices_and_enter
6suspend_enter
7acpi_suspend_enter
8x86_ acpi_suspend_lowlevel
9do_suspend_lowlevel
10 x86_acpi_enter_sleep_state
11@+acpi_hw_legacy_sleep@
12 acpi_os_prepare_sleep
13@+tboot_sleep@
```

Figure 3.9: The execution path to transition to the zombie state. It is similar to the S3 execution path, except the modifications on red functions (lines 1, 10 and 12).

and during each S_z enter and exit. During boot up the firmware initialises S_z chipset configurations. During S_z enter and exit the firmware transitions individual devices to their corresponding S-states. The additional work required for the actual steps is minimal for S_z as most of the board is still transitioned to S3. Additional logic is required to transition memory and network to their active-idle states to enable their operation while the system is in S_z state. During S_z exit, once the chipset state is reinitialised, the firmware passes the control back to the OS to transition to general-purpose computation in S0 state.

We prototype the OS components of S_z state with the Linux Operating System Power Management (OSPM) framework. OSPM is the kernel component in charge of power management and shares this responsibility with the device-drivers. S_z state implementation in the kernel requires the modification of both the OSPM and the Infiniband device driver (*MLNX_OFED* in the prototype). This implementation starts from the *S3/S4* execution path, to which we applied slight modifications as presented in Figure 3.9. We introduce a new keyword (*zom*) for triggering the transition to S_z when setting */sys/power/state*. We identify the set of devices which should be kept up during the S_z state (e.g., Infiniband card and its associated PCIe devices). The `pm_suspend()` call for these devices has been modified in order to prevent them from transitioning to the sleep state. The real activation of the transition is done by setting `PM1A` and `PM1B` ACPI registers. In the case of S3, `SLP_TYP` and `SLP_EN` are respectively set into these registers. Once set, `PM1A` and `PM1B` are read by the platform in order to know which state to transition to. Since this registers have unused values, we consider new ones for triggering to zombie.

3.5 Related works

Energy is certainly an important percentage of the total cost of ownership as datacenters are huge energy consumers. For a 10 MW datacenter, an energy reduction of only 1% would result in almost $3.4 million of savings over a period of 10 years [83]. Thereby, an important research effort focuses on improving the energy efficiency of computing infrastructure.

Component level techniques. Several studies have investigated solutions to reduce the energy consumption of server components. Most components generally implement some or all power states specified by the ACPI standard. Thereby, the operating system can switch them to lower-power states when that is possible. Concerning the CPU cores, the low-power ACPI states are named C-states. ACPI defines only the first 4 states (C0-C3) but some CPU manufacturers added additional states which can go up to powered-off cores (e.g. C6 on Intel Core i7). One of the most popular techniques that is used to reduce the energy consumption in low-power states is the Dynamic Voltage and Frequency Scaling (DVFS) and, over time, multiple works focused on refining and improving its efficiency. DVFS is implemented using *voltage regulators* which convert the noisy input voltage into one or more voltage levels applied to the processors. The energy conversion efficiency of on-chip regulators is typically much lower than off-chip regulators but the latter can only support coarse-grained power control. In this context, Yuxin Bai et al. [41] propose a framework that relies on a hierarchy of off-chip switching regulators and on-chip linear regulators to facilitate fine-grained power control and a high regulator efficiency.

Qingyuan Deng et al. [65] propose a scheme which applies dynamic voltage and frequency scaling to the memory controller and dynamic frequency scaling to the memory channels and DRAM chips. Mike OConnor et al. [127] propose a new DRAM architecture which, through a better chip-level parallelism and a shorter wiring distance between the cell array and the local I/O, improves the bandwidth of traditional DRAM by 4x and the energy efficiency by 2x. Several works [101, 104] focus on reducing the energy consumption of DRAM self-refresh. They propose different refresh rates for the DRAM rows according to the leakiness of the memory cells [101] or according to the criticality of the stored data [104]. Another technique used to decrease the servers' energy consumption is to include low-power cores in the package. For example, Tegra 3 [126] and ARM big.LITTLE [76] can switch to low-power efficient cores during the low-utilization intervals. Ganesh Venkatesh et al. [144] introduce *c-cores* which are specialized processors focusing on reducing energy, especially

in advanced architectures where DVFS becomes less effective.

Modern CPUs also adopt a technique called *power gating* in order to shut down unnecessary parts of the chip. This can naively be extended to shut down portions of the clock distribution system (*clock gating*) that were only used to feed the power gated resources. G. S. Ravi et al. [133] argue that the reconfiguration (gating) mechanism should be aware of the clock tree and the power consumption distribution over the nodes. Under these circumstances, it can take more informed decisions that result in better energy savings. M. Taram et al. [141] propose *context-sensitive decoding*, a technique that enables customization of the micro-op translation depending on the current execution context. Context-sensitive decoding allows to scalarize vector instructions in order to power gate vector units during phases of minimal vector activity. WiDGET [149] decouples the instruction engines (IEs) from the execution units (EUs). By varying the number of EUs, WiDGET affords a wide power range, from the low-power Atom-like processors to the high performance Xeon-like ones. Most of the components, taken individually, have already reached high levels of energy efficiency. Thereby it becomes more and more difficult to accomplish high datacenter energy savings by exclusively relying on component level techniques.

Server level techniques. The fundamental goal of server level techniques is to increase the energy proportionality. The latter simply means that the energy consumption of a server should be proportional to its utilization for the entire utilization interval (i.e. 0%-100%). Some research effort [114, 106, 130, 143] focus on improving the energy proportionality for latency-critical workloads. David Meisner et al. [114] found out that, for this type of workloads, an acceptable response time can only be achieved in active states so the challenge is to reach high energy proportionality in these conditions. PEGASUS [106] monitors the end-to-end latency and dynamically adjusts servers' power management so that they run just fast enough to meet the objectives. TimeTrader [143] exploits the observation that the large majority of replies are 3-4x faster than the tail. Thereby, slowing down these fast replies can be an opportunity for energy savings. As a result of all these efforts, the servers are becoming more and more energy proportional.

Chao Xu et al. [153] propose a structural change to the current Linux runtime power management (PM) framework, centralizing the PM code from device drivers to a single kernel module. Several works [138, 105, 160] introduce more sophisticated power management frameworks. T. S. Muthukaruppan et al. [138] propose a distributed power management framework for heterogeneous many-core systems based on the supply-demand market mechanism. The

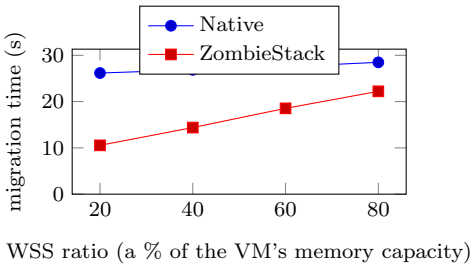

Figure 3.10: Comparison of the vanilla live VM migration solution with ZombieStack.

framework incorporates and coordinates various power management techniques like DVFS, load balancing and task migrations. SleepScale [105] dynamically selects the most appropriate combination of frequency and low power modes necessary to satisfy the QoS requirements of a workload, based on the workload's predicted behavior. They also demonstrated that naive prediction techniques are sufficient for choosing the most suitable power state. Wenli Zheng et al. [160] propose a framework that integrates the thermoelectric cooler, the fan and DVFS to improve the overall energy efficiency of chip multiprocessors (CMPs). They formulate the CMP energy optimization with temperature constraint as a nonlinear optimization problem and design a novel heuristic algorithm to solve it with low overhead.

PowerNap [113] aims to reduce the energy consumption by switching the servers to a low-power state during idle periods. It focuses on improving two things: the energy consumption of this low-power state and the transition speed. DreamWeaver [115] and Somnoloquy [34] extend PowerNap to support certain services (such as download and instant messaging) during the low-power state. KnightShift [151] proposes a datacenter where each traditional server is paired with a low-power server (called memory server). During the low utilization intervals the traditional server is turned off. Thereby, the energy efficiency of the pair is better than the case where the traditional server would have operated alone. The fundamental observation of the KnightShift paper is that many servers in the datacenter are only used for their memory (their CPUs are idle). However, C. Jiang et al. [89] shown that idle servers have the worst energy efficiency. In this context, our new zombie state substantially improves the energy efficiency of idle servers only powered on for their memory. Our work can transform a commodity server in an efficient[1] memory server without any additional capital investment.

[1]In the KnightShift paper, the memory servers are equipped with Intel Atom processors. Even if they are quite low-power, the processors are only powered on to enable access to memory. In contrast, all CPUs are powered off when a server is in S_z.

	S0WOIB	S0WIBOff	S0WIBOn	S3WOIB	S3WIB	S4WOIB	S4WIB	S_z
HP	46.16%	52.20%	53.84%	4.23%	11.03%	0.19%	6.81%	12.67%
Dell	35.35%	42.33%	44.77%	1.97%	8.71%	1.12%	8.31%	11.15%

Table 3.1: Energy consumption of our two experimental machines in different configurations. Each value is the percentage of the machine's maximum energy.

3.6 Evaluations: the S_z energy consumption

Given that we don't have a HW prototype, we estimated the amount of energy that a machine would likely consume in the S_z state. To this end, we consider two machine types available in our lab: an HP compaq Elite 8300 (noted HP) and a Dell precision Tower 5810 (noted $Dell$). Using PowerSpy2, a power analyzer device, we measured the energy consumed by each machine in several configurations: *S0* without the Infiniband card (noted $S0WOIB$), *S0* with the Infiniband card not in use (noted $S0WIBOff$), *S0* with the Infiniband card in use (noted $S0WIBOn$), *S3* without the Infiniband card (noted $S3WOIB$), *S3* with the Infiniband card (noted $S3WIB$), *S4* without the Infiniband card (noted $S4WOIB$), and *S4* with the Infiniband card (noted $S4WIB$). Notice that a server in a sleep state usually keeps at least one of its network card (the Infiniband card here) in a power state which allows the Wake-on-LAN (WoL). This corresponds to $S3WIB$ or $S4WIB$. Table 3.1 presents the results. Knowing that S_z is a kind of *S3* in which the RAM and the circuitry from the Infiniband card to the RAM are kept functioning, the energy consumed in S_z can be estimated[2] as follows:

$$\begin{aligned} E(Sz) = (E(S0WIBOn) - E(S0WIBOff)) + \\ (E(S3WIB) - E(S3WOIB)) + E(S3WOIB) \end{aligned} \tag{3.1}$$

$(E(S0WIBOn) - E(S0WIBOff))$ is the energy induced by the Infiniband card activity; $(E(S3WIB) - E(S3WOIB))$ is the energy consumption which allows the WoL (i.e. the low-powered Infiniband card, PCIe, root complex, etc.). Using equation 3.1, we estimated the energy consumed by our testbed machines in S_z (see the last column of Table 3.1).

[2]This is an optimistic estimation since it considers the memory in self-refresh mode. However the S_z can be optimized to get the energy consumption close to the estimation. For example, some (or all) memory banks can be kept in self-refresh and switched to active-idle only when the NIC performs memory operations.

3.7 Conclusion

This work proposes a simple and practical way towards memory disaggregation which can be introduced with only small changes to a commodity server. To this end, we introduced a new ACPI low-power state called the zombie state (noted 'S_z') which is similar to Suspend-to-RAM with the key difference that in S_z, the memory resources of a server are active and usable by other nodes. Since we dont have a hardware prototype, we modeled the amount of energy that a machine would likely consume in the S_z state. The results prove that the S_z state considerably improves the energy proportionality of servers and thereby, the energy efficiency of the entire datacenter.

Chapter 4

ZombieStack

While chapter 3 focuses on the low-level features the S_z state, this chapter presents a practical approach to rack-level memory disaggregation based on the S_z state. To this end, we introduce ZombieStack, a complete cloud software stack able to manage a virtualized datacenter enhanced with our new Zombie technology.

4.1 Memory Disaggregation Using S_z State

In our rack-level management implementation, servers are either *active* in *S0* state or *zombie* in S_z state. An active server can use its own memory or memory from other zombie servers. While our main contribution is on utilizing S_z state for energy-efficient memory disaggregation, our implementation also allows for serving and using remote memory from other active servers. If an active server requires more memory it can become a *user* of available remote memory. If there is capacity slack, workload is consolidated to fewer hosts to save energy, which then become zombies pushed into S_z. We implement two remote memory functions as (i) *RAM extension* (*RAM Ext* for short), and (ii) *Explicit swap device* (*Explicit SD* for short).

***RAM Ext*:** An ideal implementation of disaggregated memory as RAM extension would require special hardware interconnect for remote memory access, similar to NUMA [35]. Instead, we design a practical, simple solution based on commodity server and network architecture, and addressing the complexity in software. We implement a hypervisor-level swap mechanism, where the remote memory is presented as swap to the hypervisor. It keeps the frequently accessed pages in local memory and excess pages are simply swapped to the remote memory. One key advantage of our approach is that we simply build upon all the existing page promotion, relegation, hot page determination policies which are already built into the hypervisor. As a result, with a small set of tweaks and by leveraging hypervisor paging, we can transparently present

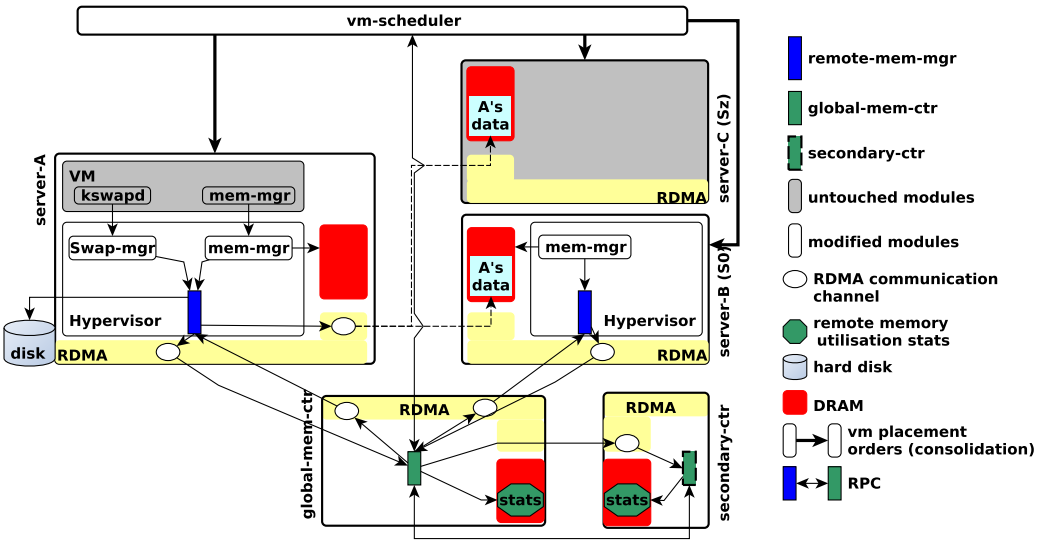

Figure 4.1: The architecture of a disaggregated rack provided by the zombie technology.

remote memory to VMs running on the host. As demand decreases, pages are naturally swapped in, requiring no custom implementation for releasing remote memory.

***Explicit SD*:** As a natural extension of our remote memory design, a server may also use remote memory to implement swap devices for VMs. These memory-backed swap devices perform substantially faster than disk-based swap. Our implementation is similar to InfiniSwap [77].

An interesting difference between these two remote memory functions is that, the VMs and applications are completely oblivious to the former function, which is hypervisor-managed, while the latter is fully-visible to those. Application behavior can be significantly different (particularly more aggressive regarding memory management) as it knows that fewer local pages are allocated to the VM (see the evaluation section).

4.1.1 Implementation

Fig. 4.1 presents our implementation architecture of a virtualized rack with the zombie technology. A general-purpose server in the rack plays one of the following five roles:

1. **Global Memory Controller** (*global-mem-ctr*) manages the memory for the whole zombie pool. It is responsible for allocating/deallocating remote memory to servers.

2. **Secondary Memory Controller** (*secondary-ctr*) enforces transparent high availability of the global controller. It monitors the main controller's state (periodic heart beat) and synchronously mirrors all operations.

3. **User Server** (*server-A*) uses remote memory from other servers.

4. **Zombie Server** (*server-C*) serves remote memory to other servers, while suspended in S_z state.

5. **Active Server** (*server-B*) serves remote memory to other servers, while in active state.

All user servers execute a **Remote Memory Manager** (*remote-mem-mgr*) agent, which interacts with the *global-mem-ctr* to request and release remote memory. The communication framework implements RPC over RDMA [72, 139]. In our implementation, the clients poll for the RPC results as RDMA inbound operations are cheaper than outbound operations. *Remote-mem-mgr* relies on low-level RDMA primitives instead of RPC calls to directly access remote memory and to implement *RAM Ext* and *Explicit SD* functions.

4.1.2 Initialisation

At startup, *global-mem-ctr* initialises various data structures for state keeping such as the list of zombie nodes. Initially all servers are designated active, and state is updated as they are pushed to S_z. Next *global-mem-ctr* starts a daemon serving the requests from *remote-mem-mgr* agents. Finally, it starts the mirroring and heartbeat processes for mirroring and high availability. *Secondary-ctr* spawns two processes to periodically monitor *global-mem-ctr* heartbeats and to establish the RPC over RDMA communication with the *global-mem-ctr* in order to receive the mirrored operations. Each *remote-mem-mgr* establishes an RPC over RDMA communication channel with the *global-mem-ctr* and initialises state to request and use remote memory.

4.1.3 Delegating and Reclaiming Server Memory

Here we first describe how servers can *delegate*, i.e., *lend*, their memory to *global-mem-ctr* via *remote-mem-mgr*. Then we explain how they can *reclaim* their memory when it is needed locally. As discussed previously, we have patched the OS of each server to implement the S_z state transition. When a server's OS receives the *suspend to* S_z signal, it signals its *remote-mem-mgr* to trigger memory delegation. *Remote-mem-mgr* computes free memory and organizes it in buffers. Their size (noted $BUFF\_SIZE$) is uniform across the entire rack. It then notifies *global-mem-ctr* of its intention to go to S_z state via

the `GS_goto_zombie(buffers)` function and communicates the list of zombie memory buffers it is lending via *buffers*. *Global-mem-ctr* uses an in-memory database to manage the allocation state of these buffers. Each remote buffer is characterized by an identifier, offset, size, its type (active/zombie), the host serving the buffer, and the server currently using this buffer (*nil* if it is not yet allocated to a server).

A zombie server can reclaim its memory once it becomes active again. Its *remote-mem-mgr* determines the amount of memory it wishes to reclaim (at buffer granularity) and informs the *global-mem-ctr* via `GS_reclaim(nbBuffers)`. *Global-mem-ctr* has to choose from its database which of the buffers belonging to this server will be returned. It first uses unallocated buffers and then chooses buffers allocated to other servers and reclaims them using the `US_reclaim(buff_IDs)` function. This function only informs the corresponding *remote-mem-mgrs* that `buff_IDs` are no longer available. As a result, the *remote-mem-mgrs* start transferring the backup copy of the data[1] to other remote locations. Last, *global-mem-ctr* returns the buffer identifiers to the reclaiming server. Once in possession of these buffers, the *remote-mem-mgr* of the server destroys the communication channels to these buffers and frees them.

4.1.4 Requesting and Allocating Remote Memory

Here we describe how a *user* server can request and allocate available remote memory from *global-mem-ctr* using the following functions:

`GS_alloc_ext(memSize)` requests a RAM Extension memory allocation of *memSize* that the *global-mem-ctr* must fulfill. This allocation is guaranteed by the cloud provider via admission control to avoid rack-level memory overcommitment. Thereby, `GS_alloc_ext(memSize)` is called once at the VM creation time and returns a list of n_b buffers such that $n_b \times BUFF\_SIZE == memSize$

`GS_alloc_swap(memSize)` requests a VM Swap memory allocation of *memSize*. The full allocation is not guaranteed as it depends on the available memory in the rack. This allocation is best-effort because using a fast swap device is not included in the VM's SLA, contrary to RAM Extension. Therefore, this allocation is such that $n_b \times BUFF\_SIZE \leq memSize$. This function is periodically called (i.e. every 1 hour) in order to take advantage of unused remote buffers.

Memory from zombie servers have always higher priority than memory from active servers. Thereby, *global-mem-ctr* first attempts to allocate the requested memory from available free buffers. Next, it tries to get more re-

[1]Each write to a remote buffer (backing either a RAM Extension or an Explicit SD) is asynchronously mirrored to the local storage.

mote memory from active and user servers with the `AS_get_free_mem()` and `US_reclaim(buff_IDs)` calls. For both, `GS_alloc_ext()` and `GS_alloc_swap()`, the *memSize* allocation is backed by memory from multiple remote servers. This approach minimizes the performance impact caused by a remote server failure. By default, all inactive servers are pushed into S_z. If the *global-mem-ctr* holds huge amounts of free memory (e.g. more than the total memory of a rack server), the cloud manager may decide to transition zombie servers to *S3* for further reducing the energy consumption.

4.1.5 Using Remote Memory

Here we describe how *user* servers use remote memory and our actual implementation for the KVM hypervisor [80]. As we previously discussed, user servers can utilize remote memory via two functions: (i) *RAM Ext*, and (ii) *Explicit SD*. Our *RAM Ext* implementation is a practical approximation to disaggregated memory, which operates transparently to VMs via our modified hypervisor-level swapping mechanism.

The ideal case of memory disaggregation requires fundamental changes to hypervisor memory virtualization implementation, where remote endpoints and page addresses need to be in shadow or extended page tables, and enabling direct access to these remote addresses. Such an implementation requires an important hardware evolution (i.e. MMUs that can understand and access remote addresses) [98]. In contrast, our solution relies on commodity, general-purpose servers[2], standard RDMA networking and a software-based solution with our modified KVM hypervisor, and unmodified VMs and applications.

Our modified virtualized memory management system within the hypervisor works as follows. Let $VMMemSize$ be the amount of memory reserved by a VM. At VM startup, the hypervisor allocates a part of the server's local RAM (noted $LocalMemSize$) to the VM. If $LocalMemSize$ is less than $VMMemSize$, the rest of the memory is provided by other remote servers as Extension memory. From VM perspective, all the memory is local and allocated in its pseudo-physical memory. From hypervisor perspective, the actual machine memory can be distributed between local physical and remote physical RAM.

We implement our solution in KVM's page fault handler, extending hypervisor paging to use remote physical memory buffers similar to swap devices. VMs are given pseudo-physical frames and the hypervisor manages their association with host-physical (machine) frames. KVM allocates physical frames on demand, which means when a VM modifies its guest page table and traps to

[2]This servers are not yet for sale since they should implement our new S_z state as described in Section 3.4.1.

the hypervisor, a physical frame is allocated and associated with the pseudo-physical page. In our solution, we provision both local and remote page frames to a VM. When a page fault is caused by a VM attempt to modify a guest page table, if a physical frame is available (free), the handler follows the traditional code execution path. Otherwise, it frees a physical frame to satisfy the page fault, using a page replacement policy.[3] Indeed, it asks the *remote-mem-mgr* for a remote page frame, transfers the content of the local frame to the remote frame, registers the information allowing its eventual reclaim and clears the present bit in the corresponding page table entry. When the page fault is caused by the non-presence of a page, we first check whether it is a page sent to a remote memory. If this is the case, a local page is allocated as above and the remote page is reloaded in the local page. Our paging policy keeps hot pages closer in local memory, and as local memory becomes scarce, demotes cold pages to remote buffers.

Our implementation of the second function, *Explicit SD*, is relatively simpler as no guarantee is offered to the VMs. Swap remote memory is obtained with the dedicated `GS_alloc_swap()` function. This function has the same prototype as `GS_alloc_ext()`, but the amount of returned memory may be less than requested as it depends on remote memory availability. Our *Explicit SD* implementation is based on the split-driver model [150]. When a VM is swapping-out a page to remote memory, the backend driver first contacts the *remote-mem-mgr* for allocating remote memory if available. It also asynchronously swaps to local storage for fault tolerance. When the *global-mem-ctr* reclaims this memory, the pages are still available on local storage and *remote-mem-mgr* uses this slower path to serve page requests.

4.2 Cloud Management with *ZombieStack*

In the previous sections we presented the hardware implementation of S_z state (Section 3.4), and how we leverage S_z state for energy-efficient, practical memory disaggregation at the hypervisor level (Section 4.1). Here, we discuss the final layer of the compute stack, the cloud operating system. We describe how we leverage memory disaggregation with *zombie* servers for energy-efficient and practical cloud computing. We build a prototype cloud management platform, *ZombieStack*, based on OpenStack and our modified KVM hypervisor. We explain below the key cloud capabilities we introduce and the changes we did to the OpenStack components in our prototype.

[3] We evaluated three policies (see Section 4.4.2)

4.2.1 Remote Memory Aware VM Placement

`Nova` is the OpenStack component responsible of VM placement on physical nodes. It operates in two phases. First, it filters the servers which are able to host the VM(s) and returns a list of suitable hosts. Second, it sorts these hosts based on certain placement criteria such as available resources and placement strategy (VM stacking or spreading). In our *ZombieStack* implementation we modify `Nova` to allow more relaxed filtering to account for remote memory availability. One trade off we explore in our implementation is the minimal amount of local memory needed for a host to be included in the list of suitable hosts. We answer this question with empirical evaluation (see Section 4.4). For the benchmarks we evaluated, the results show that down to 50% of the VM's working set size[4] in local memory is a good, conservative compromise.

4.2.2 VM Consolidation with *Zombie* Servers

Our VM consolidation implementation is based on OpenStack `Neat`. The consolidation algorithm employed by `Neat` can be outlined in four main steps [23]: Determine the underloaded hosts (all their VMs should be migrated and the hosts should be suspended); Determine the overloaded hosts (some of their VMs should be migrated in order to meet QoS requirements); Select VMs to migrate from overloaded hosts; Place the selected VMs to other hosts (wake up suspended hosts if necessary).

Vanilla `Neat` places a VM on a server only if the latter holds all the resources booked by the VM. In the same vein as VM placement, we modify this constraint to only check if 50% of the VM's working set size is available on the target server. If there is no host that satisfies this requirement, we choose and wake up a *zombie* host. We modified `Neat` so that it prefers zombie servers with the least amount of shared buffers. Neat calls `GS_get_lru_zombie()` which returns the *hostID* corresponding to the Zombie server having the minimum number of allocated zombie buffers. By this way, we minimize the amount of zombie memory which has to be reclaimed.

4.2.3 VM Migration Protocol

The vanilla pre-copy VM migration consists of only source and destination hosts that hold the VM's current and future memory state. As part of a VM's memory may be located remotely in our *zombie* implementation, the migration protocol of *ZombieStack* is more complex than traditional migration. In our implementation, the active part of VM memory is mostly local to the source

[4]The working set size is computed by the system presented in Section 5.6.

server due to the replacement policy behavior. Any remote memory used for the VM consists of cold pages.

Our migration protocol implementation first creates a *listening VM* on the target host, similar to traditional migration. However, instead of iteratively pre-copying dirty VM memory pages, we follow an approach similar to post-copy migration [85]. We stop the VM and we copy its local active memory part (hot pages) to the destination host. The newly created VM can be resumed as soon as its active part is copied on the target host. An interesting side benefit of *zombie* servers is that the VM's remote memory needs no migration. Once started on the destination host, the active part can address its remote part in the same way as before. We just need to update the ownership pointers for the remote memory components. Overall, our disaggregated memory implementation with *zombie* servers somewhat complicates the orchestration of live migration. However, in addition to the energy savings benefits, disaggregation also improves migration performance by both reducing the migration overhead and by providing a natural decoupling of hot vs. cold VM pages.

4.3 Related works

A great deal of works focus on increasing the energy efficiency of cloud datacenters. Xiaoqiao Meng et al. [117] consolidate together virtual machine (VM) pairs with strong negative correlations (i.e. the resource demand change in opposite directions). Thereby the peak resource demand of a VM can be satisfied by the valleys in the other VM. Canturk Isci et al. [87] introduced low-latency low-power states for enterprise servers and demonstrated that, in the case of a peak resource demand, a workload can be quickly deconsolidated with negligible performance impact. Oasis [161] adopts the concept of partial VM migration to densely consolidate the idle VM working sets on energy efficient memory servers. The accessed memory pages are pulled back on demand from the remote memory server. Faraz Ahmad et al. [36] address two problems caused by intensive consolidation: (1) the higher cooling power due to the hot spots created by concentrating the datacenter load and (2) the performance degradation due to power state switching. For the first issue, they propose a solution that jointly optimizes the idle and cooling power while, for the second issue, they propose to overprovision the number of active servers based on a two-tier scheme. Heracles [107] enables the *safe* collocation of best-effort tasks alongside a latency-critical service. In order to achieve the perfect performance isolation of latency-critical jobs, Heracles leverages two hardware mechanisms (shared cache partitioning and fine-grained power/frequency settings) and two software mechanisms (scheduling and network traffic control).

Yanpei Chen et al. [58] focus on increasing the server utilization for MapReduce with Interactive Analysis (MIA) workloads at Facebook. They observed that even if MIA clusters host huge data volumes, the interactive jobs operate on a small fraction of the data. Thereby, interactive jobs can be served by a small pool of dedicated machines, while the less time-sensitive jobs can run on the rest of the cluster in a batch fashion. Some works [90, 158, 159] consider several schemes of network traffic consolidation. Hao Jin et al. [90] propose a joint optimization scheme that simultaneously optimizes VM placement and network flow routing to maximize energy savings. Other works [158, 159] optimize the power consumption of the data center network but dynamically control the flow completion time of delay sensitive traffic flows.

Some works [120, 99] propose small-scale clouds to cooperate and share resources among themselves. Both works focus on designing a system that finds out the market equilibrium (i.e. the resource price that satisfies both sides) and a smart cloud scheduler aware of external resources. Prateek Sharma et al. [136] propose a resource management framework for transient servers (also known as 'Spot VMs') inspired from the concept of portfolio in financial market investment. Depending on the application risk tolerance and sensitivity, customers can create portfolios with configurable costs and availabilities, composed of a mix of transient server types. Many large-scale companies such as Yahoo [134], Google [75] and Facebook [68] use "free" cooling systems that rely on the outside environment to decrease the rack inlet temperature. However, small and medium-scale datacenters which are responsible for 49% of U.S. datacenter electricity consumption [62] are generally not suitable for these innovative technologies. Thereby, several research works [74, 47, 155, 112] focus on reducing the cooling costs of small and medium-scale cloud datacenters.

One fairly extensive research axis focus on efficient resource management and provisioning. Christina Delimitrou et al. [63, 64] observed that in the cloud, users are generally not interested in the amount of allocated resources but in the performance of their applications. Thereby, they introduce Quasar [63], a cluster management system which uses classification techniques to determine and adjust the amount of resources that satisfy the performance constraints of a given workload. In a subsequent paper, they propose HCloud [64], a hybrid provisioning system that chooses the best provisioning policy (fixed vs on-demand) for each workload and determines the optimal instance size needed to satisfy a given QoS. Liuhua Chen et al. [57] shown that the utilization curves for different VMs of the same job may be misaligned in time and they propose three refinement algorithms that improve the efficiency of resource provisioning. Eli Cortez et al. [61] monitored Microsoft Azures VM workloads and identified certain behaviors that are consistent over time. Further, they introduce Resource

Central, a system that collects various metrics during the VM execution and identify the potential recurrent patterns. Further, this information is exploited online to smartly oversubscribe the servers that host some specific VM types. Nikita Mishra et al. [119] formalize the problem of allocating available resources to meet the current performance demand as a constrained optimization problem and apply machine learning techniques to estimate the application-dependent power/performance parameters. Ning Liu et al. [103] propose a hierarchical framework that comprises a global tier for VM placement to the servers and a local tier for power management of local servers. The global tier problem is solved using the deep reinforcement learning (DRL) technique. Each decision epoch coincides with the arrival time of a new VM request thereby the action space is significantly reduced. At each local tier, a model-free RL-based power manager relies on workload predictions to decide the suitable server power state. Hao He et al. [84] adopt the Linear Temporal Logic (LTL) to resolve the conflicting objectives faced by a cloud resource manager and further, the LTL-based constraints are integrated with reinforcement learning.

Similar to ZombieStack, Juncheng Gu et al. [77] focus on efficient memory management. They developed a system called InfiniSwap whose goal is to balance the memory demand over all datacenter servers. InfiniSwap is composed of a daemon which allocates free memory and lends it over RDMA to remote servers having high memory demand. The remote memory is exposed as a swap device which may introduce useless overheads since the operating systems suppose that swap storage is slow so they try to optimize and batch accesses. Second, InfiniSwap is effective only if the global datacenter memory demand is comparable with the CPU demand. In contrast, ZombieStack relies on zombie memory which is completely decoupled from the CPU so both resources can vary independently.

The fundamental way of decoupling resources in the cloud is introduced by disaggregated computing which changes the server centric view of datacenters to a resource centric view. In a disaggregated datacenter, resources (CPU, memory, networking, etc.) are physically decoupled and can evolve independently. One of the most prominent resource disaggregation projects is The Machine[15] from HPE. In contrast, ZombieStack introduces a simple and short term solution until resource desegregation will become prevalent in the cloud.

4.4 Evaluations

This paper introduces ZombieStack, a framework that exploits the S_z state at rack level. ZombieStack includes two utilisation modes namely *RAM Ext* and

Explicit SD. Since the latter has been widely investigated in previous work [108, 37, 97, 77, 51], our evaluations focus on *RAM Ext* while comparing it with *Explicit SD*. Notice that each presented result is an average of ten executions. We do not show the standard deviation results because we observed stable results.

4.4.1 Experimental environment

Hardware. We used two environment types. First, we evaluated the effectiveness of ZombieStack using a real rack in our lab. This rack is composed of four HP compaq Elite 8300 machines (Intel Xeon Intel(R) Xeon (R) CPU i7, 16GB RAM, running Linux kernel 4.4) organized as follows: two machines for hosting the *global-mem-ctr* and the *secondary-ctr*, one machine services as a user server while the last machine plays the role of a zombie server. Having not yet S_z enabled boards, the zombie server is provided by an idle server in *S0*. The four servers are linked altogether with Mellanox Infiniband SB7800 switch. Each machine uses a Mellanox ConnectX-3 as the network card.
The second environment type is a simulator, used for the evaluation of ZombieStack in a large scale environment.
Software. We evaluated ZombieStack with both micro and macro benchmarks. The former is an application which performs random read/write operations on the entries of an array whose size is configured at start time. Each entry represents a 4KB memory page. The performance metric of this benchmark is the execution time. Regarding the macro-benchmarks, we chose the following applications: Data Caching[5] from CloudSuite [71]; Elasticsearch nightly benchmarks [9][6]; and Spark SQL [38] with BigBench [73] (we used a 100GB data set and focused on query 23[7]). The performance metric of these benchmarks is the number of operations performed per second. Otherwise specified, every VM uses 8 processors.

4.4.2 RAM Ext's page replacement policy

The efficiency of *RAM Ext* depends on the replacement policy which selects the page that should be transferred to a remote memory when the local memory becomes scarce. We compared three common replacement policies:

[5]Data Caching uses the Memcached data caching server, simulating the behavior of a Twitter caching server using a Twitter dataset.

[6]We only present the results for the NYC taxi benchmark whose 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.

[7]BigBench includes more than 30 queries and query 23 is one of the longest.

- *FIFO*. The hypervisor records to a list (called FIFO list) the pages which generate page faults. The page to transfer is the one which has generated the oldest page fault.

- *Clock*. The hypervisor iterates through the FIFO list and chooses the first page whose "accessed" bit is zero. The "accessed" bit of all pages is periodically cleared.

- *Mixed*. The Clock policy is applied to the first x elements of the FIFO list (e.g. x=5). If no page is obtained, the FIFO policy is applied to the rest of the list. This policy is designed to reduce the costs associated with "accessed" bits' management and list iterations.

We relied on the micro-benchmark to evaluate the above policies. The benchmark runs inside a VM having 7GB reserved memory while its working set size (WSS) is configured to 6GB. The VM is launched on the user server. We performed several experiments while varying the proportion of its memory in that server. Its remaining memory is provided by the zombie server using *RAM Ext*. Fig. 4.2 presents the evaluation results. The collected data are: the execution time (top curve), the number of page faults caused by the replacement policy (middle curve), and the time taken by the replacement policy in the page fault handler (bottom curve). We can see that *Mixed* is the best replacement policy. This is explained by the fact that it minimizes the page list iteration time (which is fairly important, see the gaps in Fig. 4.2 bottom) while avoiding the replacement of a page which may be used in a near future (by checking the "accessed" bit, see the gaps in Fig. 4.2 middle). As a result, Mixed outperforms both FIFO (by up to 30%) and Clock (by up to 36%), see Fig. 4.2 top. Thereby, the remaining experiments rely on *Mixed*.

4.4.3 RAM Ext limitations

We investigated to what extent a portion of a VM's RAM can be provided by a remote server. To this end, we relied on both micro and macro-benchmarks. Our micro-benchmark represents a worst-case scenario. The evaluation procedure for the macro-benchmarks is as follows. Given a benchmark, we first ran it with vanilla KVM in order to determine its maximum WSS that does not generate swap activities. This size will serve as the VM's reserved memory in *RAM Ext*. Afterwards, we ran the benchmark with ZombieStack-*RAM Ext* while varying the proportion of the VM's reserved memory in the local RAM. Table 4.1 presents the evaluation results in terms of performance penalty. We can see that providing down to 50% of the VM's reserved memory with local RAM is a good compromise. It leads to an acceptable penalty, less than 6.5%

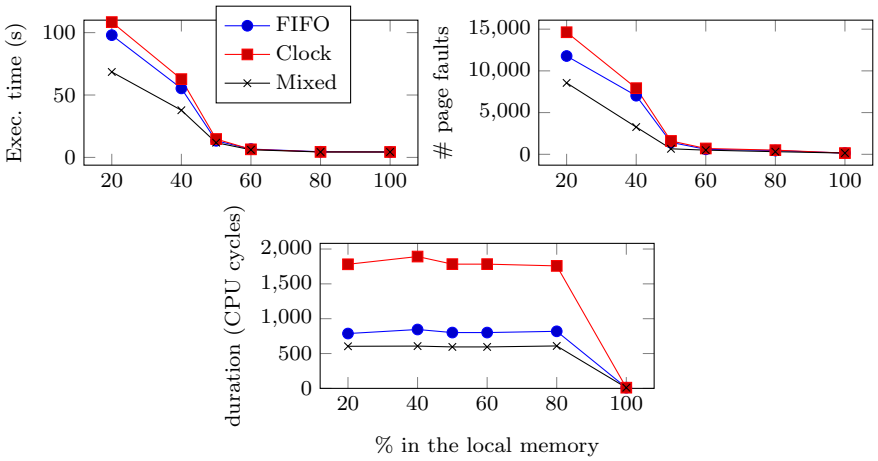

Figure 4.2: Comparison of three replacement policies (FIFO, Clock, and Mixed) for *RAM Ext*. (top) The micro-benchmark execution time, (middle) # page faults and (bottom) time taken by the policy to perform a page fault. Mixed is the best policy.

% in local mem	micro-bench.	Elastic search	Data caching	Spark SQL
20%	1400%	15.6%	9.6%	27%
40%	770%	6%	3.16%	6.5%
50%	171%	4.2%	1.35%	5.34%
60%	41%	3.01%	0.35%	2.04%
80%	1%	0.01%	0.32%	0.2%

Table 4.1: Performance penalty evaluation when a proportion of the VM's reserved memory is provided by a remote server. 50% is a good compromise.

(excepting our synthetic workload). We can observe that this proportion is also appropriate for macro-benchmarks. Our results are consistent with the ones in [72]. ZombieStack is configured with 50% local memory in order to take into account worse case applications like our micro-benchmark, even if such applications are rare.

4.4.4 RAM Ext compared with Explicit SD

Let us consider two VMs (noted v_1 and v_2) configured as follows. v_1's reserved memory is m and v_2's reserved memory is $m - x$, $x \leq m$. v_1 runs in ZombieStack-*RAM Ext* with $m - x$ memory provided by the local server. v_2 runs in ZombieStack-*Explicit SD* with a mounted swap device backed by zombie memory. The size of this swap device is x. Let us consider that v_1 and v_2 run the same application. One may think that the performance of that application will be the same in both cases. To clarify the situation, we compared the two utilization modes while extending the analysis to other swap device technologies including: a local fast swap device (provided by an SSD,

	Micro benchmark			
% in local mem	v_1-*RE*	v_2-*ESD*	v_2-*LFSD*	v_2-*LSSD*
20%	1400%	6k%	∞	∞
40%	770%	3k%	∞	∞
50%	171%	910%	302k%	∞
60%	41%	232%	3k%	429k%
80%	1%	5.4%	10.3%	21%
	Elastic Search			
% in local mem	v_1-*RE*	v_2-*ESD*	v_2-*LFSD*	v_2-*LSSD*
20%	15.6%	43.2%	85.12%	∞
40%	6%	38.6%	68%	307%
50%	3.4%	17.1%	45.04%	105.8%
60%	0.2%	12.3%	17.4%	55.3%
80%	0.01%	0.8%	1.6%	3%
	Data caching			
% in local mem	v_1-*RE*	v_2-*ESD*	v_2-*LFSD*	v_2-*LSSD*
20%	9.6%	15.7%	140.8%	∞
40%	3.16%	6.4%	41.7%	∞
50%	1.35%	3.1%	18.2%	∞
60%	0.35%	1.1%	3.04%	∞
80%	0.32%	0.35%	0.68%	13.2%
	Spark SQL			
% in local mem	v_1-*RE*	v_2-*ESD*	v_2-*LFSD*	v_2-*LSSD*
20%	27%	31.64%	122%	∞
40%	6.5%	18.39%	63.23%	∞
50%	5.34%	13%	35%	∞
60%	2.04%	2.9%	11.45%	185.36%
80%	0.2%	0.3%	3.2%	4.78%

Table 4.2: The performance penalty (i.e. how much longer the execution takes?) depending on the local/remote memory ratio. *RAM Ext* (RE) vs *Explicit SD* (ESD) and other swap technologies (LFSD=Local fast swap device; LSSD=Local slow swap device).

Samsung MZ-7PD256), and a local slow swap device (provided by a HDD, Seagate ST12000NM0007). Table 4.2 presents the evaluation results in terms of performance penalty. The following observations can be made. (1) v_1 outperforms v_2, see Table 4.2 column 2-3. In fact, v_2 generates much more swap activities on the remote server than v_1. For instance, v_2 generates more than 122% traffic than v_1 in the case of Elastic search. This comes from the fact that most applications and operating systems are configured according to the RAM size [135]. (2) Using a remote RAM as the swap space through Infiniband is better than using a local storage, even if the latter is fast (see Table 4.2 column 3-5). In addition, fast storages require additional costs, leading to an unacceptable performance per dollar for data center operators [123].

4.4.5 VM Migration

We compared our VM migration implementation with the vanilla live VM migration. To this end, we ran the micro-benchmark inside a VM with different

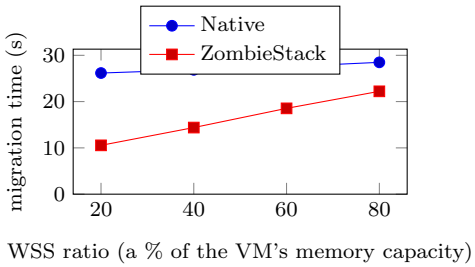

Figure 4.3: Comparison of the vanilla live VM migration solution with ZombieStack.

	S0WOIB	S0WIBOff	S0WIBOn	S3WOIB	S3WIB	S4WOIB	S4WIB	S_z
HP	46.16%	52.20%	53.84%	4.23%	11.03%	0.19%	6.81%	12.67%
Dell	35.35%	42.33%	44.77%	1.97%	8.71%	1.12%	8.31%	11.15%

Table 4.3: Energy consumption of our two experimental machines in different configurations. Each value is the percentage of the machine's maximum energy.

WSS. We are interested in the time taken by the migration process. Fig. 4.3 presents the evaluation results. We can see that in the vanilla implementation, the migration time is almost not affected by the WSS. This is explained by the fact that the number of iterations performed by the hypervisor for transferring dirty pages is fixed (independent of the memory activity). In ZombieStack, only the memory pages within the local memory (about 50% of the WSS - see Section 4.2) are transferred. Thus, our implementation outperforms the native one, especially when the WSS is small.

4.4.6 Energy gain in a large scale DC

We evaluated the energy gain that can be achieved using ZombieStack in a DC. To this end, we relied on Google datacenter traces [13] which record the execution of thousands of jobs monitored during 29 days. Each job is composed of several tasks and every task runs within a container (seen as a VM is this paper). The total number of servers involved in these traces is 12583. The traces contain, among other information, for each task: its start time and termination time, its booked resource capacity (CPU and memory), its actual resource utilization level (gathered periodically). From these traces, we built a second set in which the memory demand is twice the CPU demand as the actual trends reveal (see Section 3.2). Relying on these two set of traces, we simulated a DC which is equipped with the OpenStack consolidation system (i.e. Neat [23]).

We compared ZombieStack with Oasis [161], a consolidation approach oriented to energy-efficient cluster management. Oasis works as follows. After

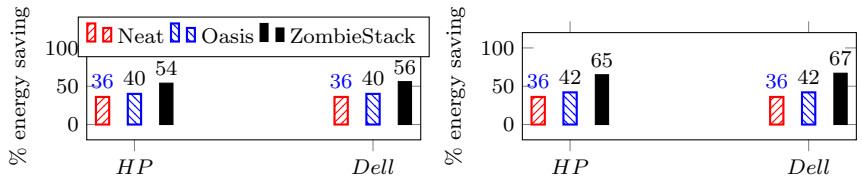

Figure 4.4: Energy saving: comparison with other resource management systems using both original (top) and modified (bottom) Google DC traces.

the execution of the consolidation plan, Oasis selects all underused servers (i.e. CPU utilization level lower than a threshold - 20% in this paper). Let us note S this set of underloaded servers. All S's VMs which are idle (e.g. CPU utilization level lower than 1%) are partially migrated [50] to other servers. A partial VM migration consists in transferring only the working set of the VM. The remaining memory pages are relocated to a low power memory server so that the initial server can be suspended for energy saving. We assume that an Oasis memory server consumes about 40% of a regular server's total energy consumption, as stated in the original paper [161]. We performed experiments while considering that servers are either *HP* or *Dell* (see Section 4.4.1). Fig. 4.4 presents the evaluation results. We can observe that ZombieStack outperforms Neat and Oasis. The best results are obtained on *Dell* servers with the modified traces (Fig. 4.4 bottom), where ZombieStack outperforms Neat and Oasis respectively by about 86% and 59%.

4.5 Conclusion

This work prototypes a cloud management platform (*ZombieStack*) based on OpenStack and a modified KVM hypervisor. We performed intensive experiments using micro-benchmarks, macro-benchmarks and real DC traces (from Google clusters) and compared our solution with existing ones (Neat and Oasis). The evaluation results showed that our solution is viable (acceptable performance degradation), leads to both high and balanced resource utilization and high energy efficiency.

Chapter 5

Working Set Size Estimation Techniques in Virtualized Environments: One Size Does not Fit All

5.1 Introduction

Energy consumption is a primary concern for datacenter (DC) management. Its cost represents a significant part of the total cost of ownership (about 80% [45]) and it is estimated that in 2020, US DCs will spend about $13 billion on energy bills [62].

A majority of DCs implements the Infrastructure as a Service (IaaS) model where *customers* buy (from *providers*) 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. Such colocation can be achieved at deployment time by starting as many VMs as possible on each physical machine, or at runtime by dynamically migrating VMs on a reduced set of physical machines, thus implementing a consolidation strategy [140].

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 [46, 63, 113]. As presented in section 3.2, the main reason is that VM collocation is memory bound, as memory saturates much faster than the CPU. 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; they provide more CPU capacity than

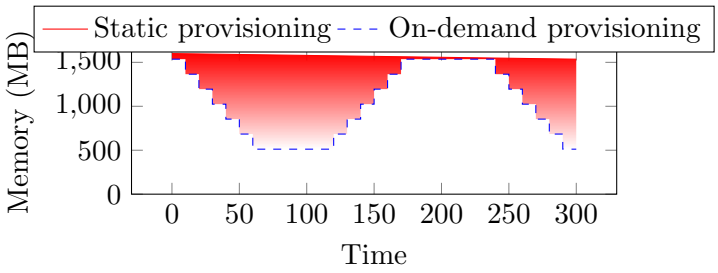

Figure 5.1: Static provisioning vs on-demand provisioning.

physical memory.

However, the existing consolidation systems [23, 70] 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. 5.1). 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.

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 [31], so that our experiments can be reproduced.

The rest of this work is structured as follows: Section 5.2 covers a quick background overview. Section 5.3 presents the general functioning of a WSS estimation solution. Section 5.4 presents the existing WSS estimation techniques that we analyze and evaluate in this article. Section 5.5 reports the evaluation results for the main studied techniques. Section 5.6.1 exposes the details of Badis while Section 5.6.2 presents the way we integrated Badis in an OpenStack cloud. Section 5.6.3 evaluates our solution. After a review of related works in Section 5.7, we present our conclusions in Section 5.8.

5.2 Background on virtualization: illustration with Xen

5.2.1 Generalities

The main goal of virtualization is to multiplex hardware resources between several guest operating systems also called Virtual Machines (VMs). Xen [42] is a well-known virtualization system employed by Amazon [8] 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 paper.

5.2.2 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 5.3).

5.2.3 Ballooning

Memory ballooning [42, 146] is a memory management technique that allows to dynamically reclaim memory from a VM to the hypervisor. Most of the modern hypervisors implement this technique in order to reclaim unused memory from VMs, thus avoiding resource waste. In such systems, every VM is equipped with a balloon driver which can be inflated or deflated from the hypervisor/dom0. Fig. 5.2 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 page to the hypervisor so that it can use them for other purposes (e.g. assigned them to a VM which is lacking memory). In the case of a balloon deflation order, the balloon driver reclaims the pinned pages from the hypervisor and deallocates them. Thereby, the pages reenter under the control of the VM's OS. In Xen, the command *xl mem-set VM_id memory_size* can be used to adjust the balloon target size from the dom0.

5.3 On-demand memory allocation

5.3.1 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.

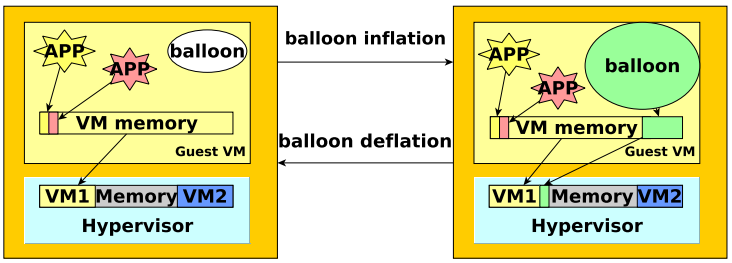

Figure 5.2: Memory ballooning principles.

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 paper, the expression "WSS estimation technique" is used to represent a solution to both Q_1 and Q_2.

5.3.2 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 paper 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 5.6.2).

5.4 Studied techniques

This section presents the main WSS estimation techniques proposed by researchers up to the writing time of this paper. We have thoroughly studied them both qualitatively and quantitatively. This section focuses on the former aspect while Section 5.5 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 5.5.

5.4.1 Self-ballooning

Description. Self-ballooning [110] 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` [10] 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. 5.3. 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 [59]. Second, this technique could lead to resource waste since the committed mem-

```
1void main(void){
2        char* tab=(char*)malloc(2*1024*1024*1024);
3        do{
4                tab[1]=getchar();
5        }while(tab[1]!='a');
6        free(tab);
7}
```

Figure 5.3: The `Committed_AS` value increases with the amount of malloc-ed memory even if it is not backed by physical memory.

ory 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.

5.4.2 Zballoond

Description. Zballoond [59] 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 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).

5.4.3 The VMware technique

Description. The VMware technique [146] is an improvement of the naive method presented in Section 5.3. 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 depending 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.

5.4.4 Geiger

Description. *Geiger* [91] 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 [128]. 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 [128]. 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.

5.4.5 Hypervisor Exclusive Cache

Description. The *Exclusive Cache* technique [109] 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 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.

5.4.6 Dynamic MPA Ballooning

Description. The Dynamic Memory Pressure Aware (MPA) Ballooning [94] 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

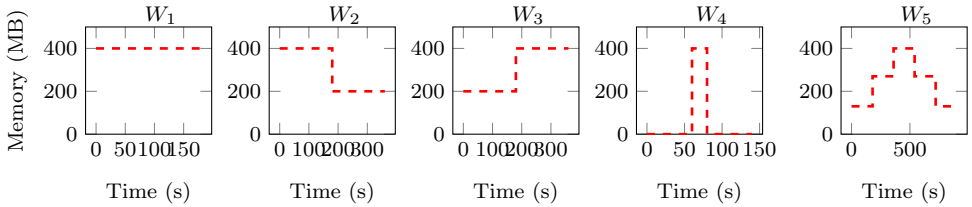

Figure 5.4: The set of synthetic workloads.

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.

5.5 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.

5.5.1 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. 5.4 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 [52], CloudSuite [71], and LinkBench [39]. DaCapo is a well known open source java benchmark suite that is widely used by memory management and computer architecture communities [157]. 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.

5.5.2 Evaluation with synthetic workloads

As stated above, these evaluations focus on the *accuracy* metric. Fig. 5.5 and Fig. 5.6 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. 5.5 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. 5.5 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*.

From now on (Fig. 5.6), 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. 5.6 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. 5.6 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. 5.6 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. 5.6 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.

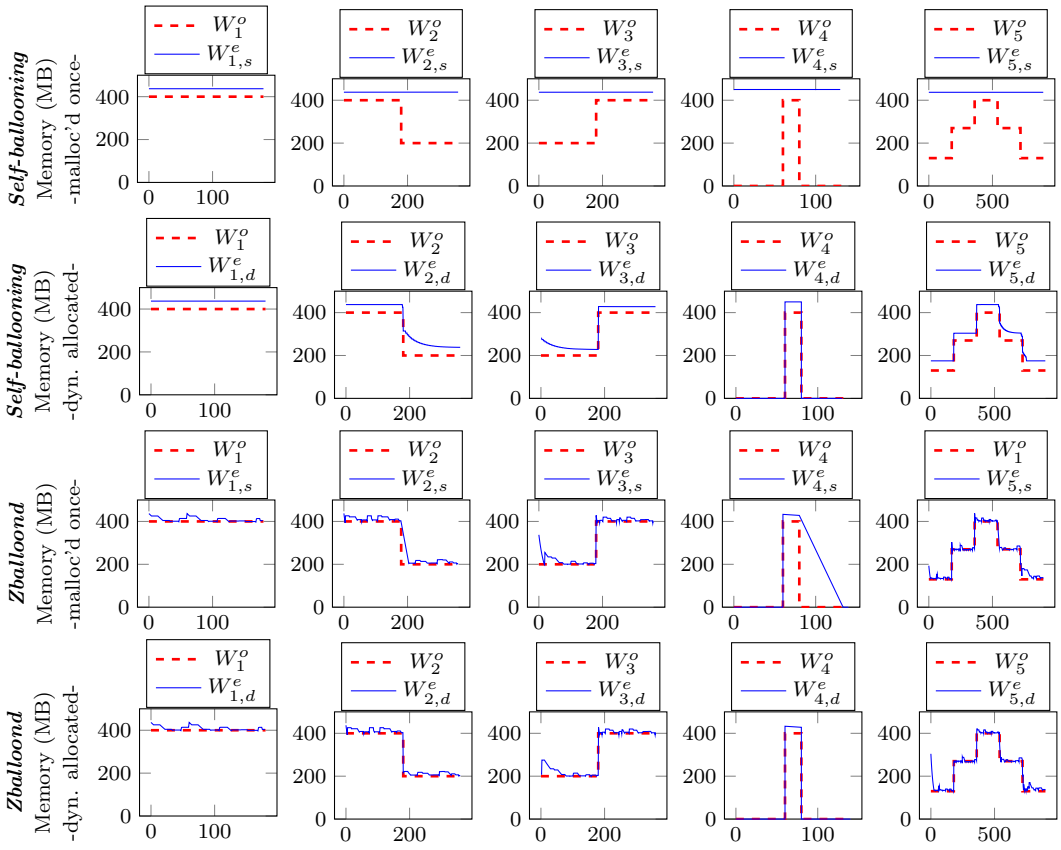

Figure 5.5: Evaluation results of *self-ballooning* and *Zballoond* with synthetic workloads. The original workload is noted W_i^o while the actual estimated WSSs are noted W_{ij}^e. "j" is s (the static implementation) or d (the dynamic implementation).

		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 5.1: Evaluation results of each technique with macro-benchmarks.

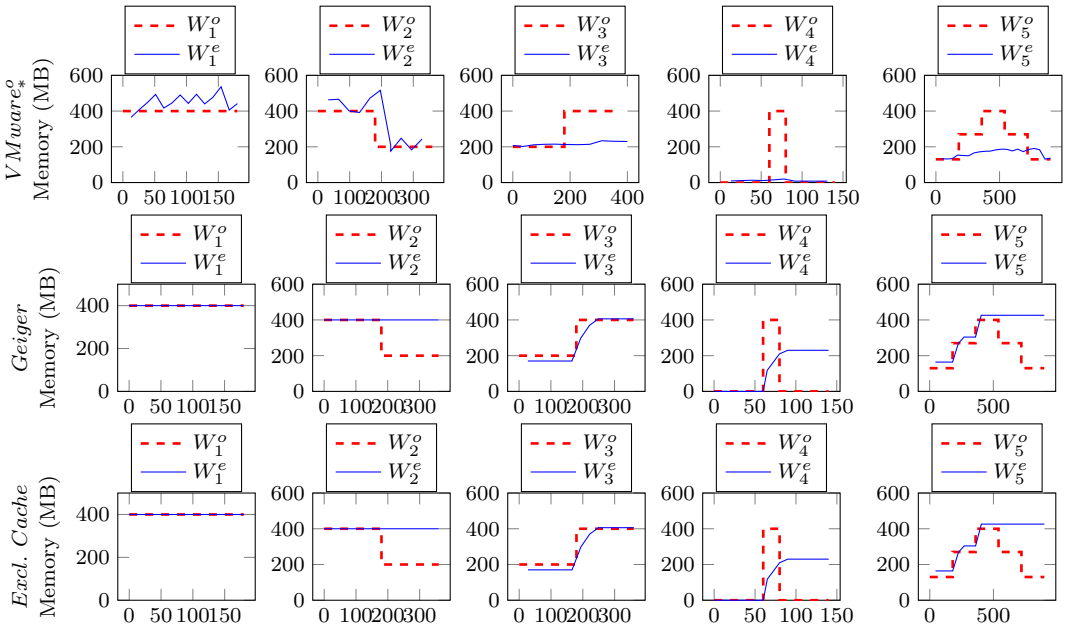

Figure 5.6: Evaluation results of *VMware*[3], *Geiger*, and *Exclusive cache* with synthetic workloads.

5.5.3 Evaluation with macro-benchmarks

Table 5.1 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 5.1 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

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

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

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).

5.5.4 Synthesis

Table 5.2 summarizes the characteristics of each technique according to both qualitative and quantitative criteria presented in Section 5.3.2. 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 [146] 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.

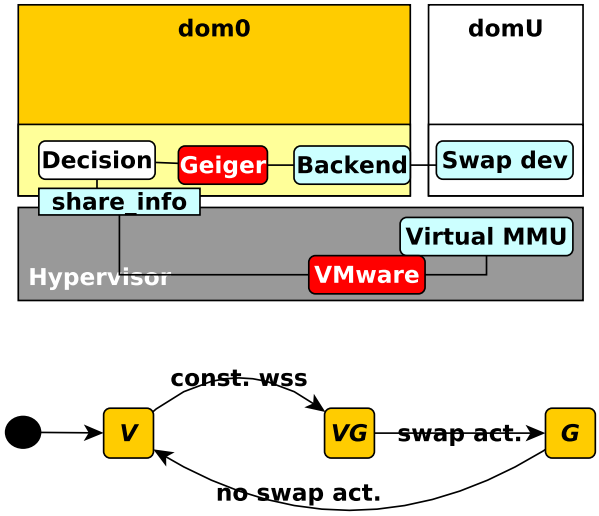

Figure 5.7: (top) The architecture of Badis. (bottom) The finite-state machine used to track a VM's WSS in Badis.

5.6 Badis

5.6.1 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 [11]. 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 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. 5.7 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.

For each VM, the system implements a 3-state finite state machine (FSM), as shown in Fig. 5.7 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.

5.6.2 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 [22]) 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 Section 3.2. 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. 5.8). **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 [100] and impact the hosted applications' performance [145]. 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 [23]. The latter takes consolidation decisions when a server is (1) underloaded or (2) overloaded. In the first case it relocates

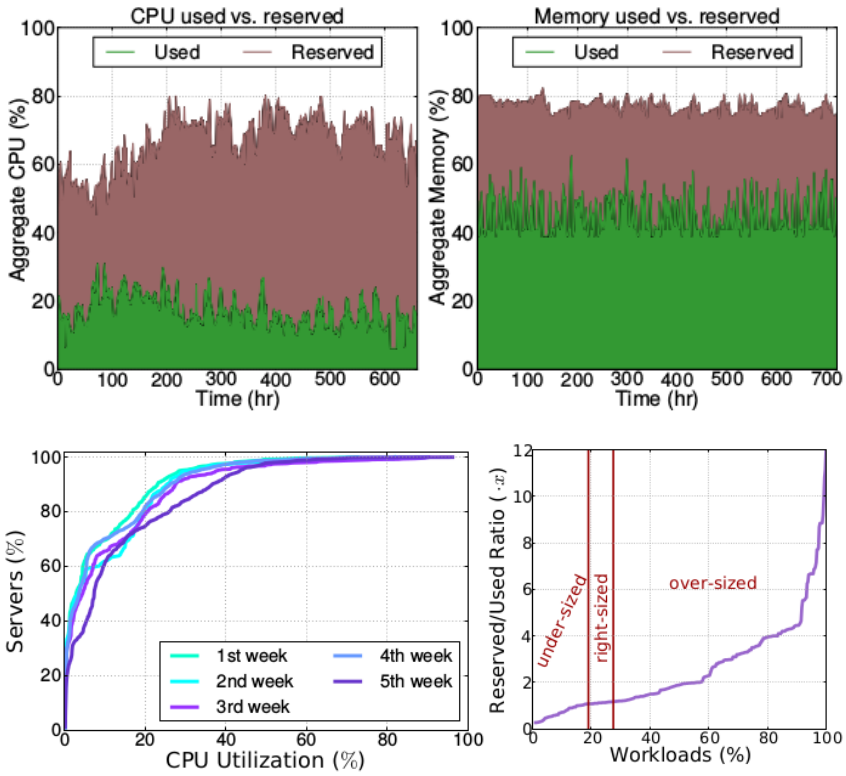

Figure 5.8: "Resource utilization over 30 days for a large production cluster at Twitter managed with Mesos. (a) and (b): utilization vs reservation for the aggregate CPU and memory capacity of the cluster; (c) CDF of CPU utilization for individual servers for each week in the 30 day period; (d) ratio of reserved vs used CPU resources for each of the thousands of workloads that ran on the cluster during this period." [63]

Benchmark and app.		*Self-ballooning*	*Zballoond*	Badis	
		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 5.3: Evaluation of our solution with macro-benchmarks, and comparison with two existing solutions.

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 last time slice*, noted $WSS_{id\_vm}^{max\_avg}$. The latter value is also transmitted to the data collection module (see Fig. 5.9). 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 5.6.3).

5.6.3 Evaluations

The experimental environment is the same as presented in Section 5.5. We evaluated our solution with both micro and macro benchmarks.

Micro-benchmark based evaluations. We first validated the effective-

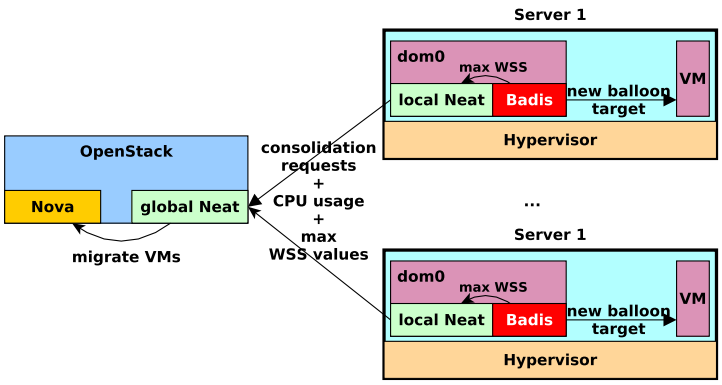

Figure 5.9: The integration of Badis in OpenStack. Badis estimates the WSS and sets the id_vm's allocated memory to $WSS_{id\_vm}^{max\_avg}$. It also transmits $WSS_{id\_vm}^{max\_avg}$ values to the local Neat. The latter collects these values along with the CPU loads and sends them in batches to the 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.

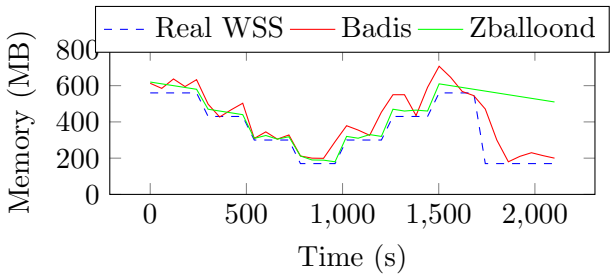

Figure 5.10: Badis and Zballoond evaluated with a synthetic workload.

ness of our solution using a synthetic workload, see the dashed blue curve in Fig. 5.10. This workload includes situations a WSS estimation technique should 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. 5.10 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 5.3. 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 [13]. 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[4]. By relying on GloudSim [66] (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 [31].

5.7 Related work

The reader should refer to Section 5.4 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 [116] could require large data collection and complex processing. Weiming Zhao et al. [156] 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.

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

Pin Zhou et al. [162] 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 [147] 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 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 [55, 118, 147, 79]. These techniques are often combined with memory compression tools to achieve better optimization rates [142, 129, 154]. Another memory optimization tool is the transcendent memory [111] 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 [157]. In a latter work, Zhao et al. [148] leverages inexpensive working set tracking systems to correctly estimate the working set size for the Memory Balancer (MEB) [157]. Xiaoqiao Meng et al. [117] 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, [135] 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 [93]. Thereby, numerous papers came up with heuristics for this problem [95, 49, 33, 92]. However, few of these projects provide real implementations to the proposed algorithms [23, 70]. 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.

5.8 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 6

Conclusion

6.1 Synthesis

This dissertation has presented resource-management techniques and hardware improvements that increase the energy efficiency of virtualized datacenters. We propose solutions that cover the entire software stack ranging from the middleware level down to the server hardware. The first solution proposed is StopGap, a middleware-level consolidation extension which dynamically replaces "big" VMs with smaller ones in order to increase the overall cloud consolidation ratio. This solution turns out to improve the OpenStack consolidation ratio by about 62.5% but it is only effective for elastic applications (see Section 2.4). Moreover, by analyzing the consolidation ratios on each resource (i.e. CPU and memory) individually, we found out that StopGap leads to a satisfactory consolidation ratio for the memory but nevertheless, the CPU utilization still remained very low.

By analyzing the resource demand trends of cloud applications vs. the resource supply trends of physical servers, we found out that, over time, physical servers could not keep up with the high memory demand requested by cloud applications. These days, applications demand about two times more memory per CPU core than a decade ago but physical servers provide two times less. Then, the natural question is why not add more memory to physical servers? Even more as the memory has become very cheap. However, it turns out that the underlying problem is the memory bandwidth which cannot keep up as more and more cores are introduced; this problem is known as the *memory wall*. For example, the memory bandwidth for a dual-core processor (i.e. Intel E5-2637) was 7.3 bytes per CPU cycle but it decreased to only 1 byte per CPU cycle in 15-core processors (e.g. Intel E7-8890v2). Thereby, this thesis claims that memory should be managed efficiently since it is a scarce resource and it is probably becoming even scarcer. The International Technology Roadmap for Semiconductors (ITRS) predict that from 2015 to 2020 the number of transis-

tors in CPUs will triple, the pin count will increase by 27% but the memory bus frequency will remain fairly constant. In this context, we claim that the static way in which cloud operating systems manage the memory is way too inefficient. The memory should be dynamically allocated to VMs based on the demand (i.e. the working set size). However, WSS estimation is not very popular since the state-of-the-art solutions are either inaccurate or very intrusive. Thereby, we proposed Badis, a system that is able to estimate the WSS of a VM with high accuracy and no VM codebase intrusiveness.

Furthermore, we focus on the traditional architecture of datacenters. They are built of commodity servers which were not designed for large-scale cloud datacenters. The memory and the computing resources are strongly coupled inside a power domain entity which is a physical server. Thereby, we proposed a practical way to decouple the memory from the CPU which can be easily implemented to a commodity server. Our solution boils down to a new ACPI state (called *zombie*) and a new cloud management software stack (called *ZombieStack*). A server in the zombie ACPI state keeps the memory banks active and accessible by the other servers in the rack. Our solution is built on top of modern networking techniques (e.g. RDMA) that can bypass the operating systems and access RAM memory without any CPU involvement. The fundamental hardware modification requested by our zombie state is separate power domains for CPU and memory so that memory can be kept active and addressable over PCIe even when the CPU is powered-off. The ZombieStack is, in a nutshell, a cloud management system that can take advantage of the memory exposed by the servers in zombie state. ZombieStack includes both the data and the control planes for accessing remote memory, as well as a memory management subsystem.

In conclusion, all these solutions aim to increase the energy efficiency of virtualized datacenters. We claim that nowadays, datacenters operate ineffectively because the system architectures and policies are not suitable for such an environment. Most of them were simply inherited from standalone or small networks of physical machines and not specifically designed for warehouse-scale computing. The solutions presented in this paper focus on enabling more efficient resource management (especially memory) by improving resource allocation policies and hardware architecture. Based on the evaluation results, we proved that a datacenter employing our solutions can seamlessly operate at a much higher efficiency, which will result in lower management costs for the cloud provider and in turn, lower prices for cloud customers.

6.2 Perspectives

One of the problems we addressed in this thesis is the memory waste due to its rigid and static allocation to VMs. In order to employ a dynamic memory allocation based on the demand, one needs to compute the working set size (WSS). The existing WSS estimation techniques (including our proposed technique) are fairly heavyweight and incur large overheads especially for systems with many virtual machines. A potentially better solution, that our team is now exploring, is to use hardware features (e.g. the Page Modification Logging [24] from Intel) for estimating the WSS. This approach will reduce the overhead and may provide a perfect accuracy. We are convinced that such a solution holds all necessary requirements to be adopted in the mainstream cloud.

Further, we showed that because of the memory wall, the memory supply could not keep up with the higher and higher application demand. The straightforward solution here may be to break the memory wall and several research initiatives [125, 54] already explored that. However, it doesn't seem that the research has found a viable way to solve this problem yet. In this context, the solution proposed in this thesis bypasses the memory wall problem and yet is able to increases the datacenter efficiency. We develop a new ACPI state that considerably increase the energy proportionality of physical machines acting as memory (zombie) servers and a cloud system that both manages and takes advantage of this remote memory. However, the commodity server CPUs are not able to directly address remote memory and this significantly reduces our system's effectiveness. In this context, a potential perspective may be to design hardware able to access remote memory. More precisely, the memory management unit (MMU) should be able to address pages in a global (e.g. rack-level) address space. On 64-bit architectures, this can be implemented by storing a machine ID tag in the most significant address bits which are often unused. For low-latency communication, the rack servers may be interconnected using a PCIe transparent bridge.

However, this kind of solutions represents only the first step of datacenter architecture metamorphosis towards a complete resource disaggregation. Many large tech companies (such as Intel, HPE, IBM, Huawei) directly investigate the implementation of a completely disaggregated datacenter. However, latency requirements impose hard limits on distances over which certain resources (especially CPU and memory) can be disaggregated. A potential solution to get low latency with relatively long interconnects is to replace electrical cables with fiber-optic interconnects. In this context, the most difficult obstacle is the system chip-level integration of optical communication since the traditional

optical fiber cables require fairly complex transceivers for electro-optical conversion. The technology that seems promising for onboard chip-level optical functions is silicon photonics. However, this technology is not ready yet and the complete resource disaggregation is mainly waiting for silicon photonics to hit the required metrics of cost, performance, and size.

Bibliography

[1] Amoeba. https://www.cs.vu.nl/pub/amoeba/amoeba.html. Accessed on 30/05/2018.

[2] Apache. http://httpd.apache.org/. Accessed on 30/05/2018.

[3] The Autoscaling Application Block. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh680892(v=pandp.50). Accessed on 30/05/2018.

[4] AWS Auto Scaling. https://aws.amazon.com/fr/autoscaling/. Accessed on 30/05/2018.

[5] Cloudify. https://cloudify.co/. Accessed on 30/05/2018.

[6] Cyrus. https://cyrusimap.org/. Accessed on 30/05/2018.

[7] Distributed Resource Scheduler, Distributed Power Management. https://www.vmware.com/products/vsphere/drs-dpm.html. Accessed on 30/05/2018.

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

[9] elastic/elasticsearch. https://github.com/elastic/elasticsearch/tree/master/client/benchmark. Accessed on 30/05/2018.

[10] */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.

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

[12] GlassFish. https://glassfish.java.net/. Accessed on 30/05/2018.

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

[14] HaProxy. http://www.haproxy.org/. Accessed on 30/05/2018.

[15] HPE - The Machine. https://www.labs.hpe.com/the-machine. Accessed on 30/05/2018.

[16] ITRS Reports - International Technology Roadmap for Semiconductors 2007. http://www.itrs2.net/itrs-reports.html. Accessed on 30/05/2018.

[17] Linux Traffic Control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html. Accessed on 30/05/2018.

[18] Memory technology evolution: an overview of system memory technologies. https://support.hpe.com/hpsc/doc/public/display?sp4ts.oid=372786&docId=emr_na-c01552458&docLocale=en_US. Accessed on 30/05/2018.

[19] Microsofts Top 10 Business Practices for Environmentally Sustainable Data Centers. https://blogs.technet.microsoft.com/msdatacenters/2009/04/21/microsofts-top-10-business-practices-for-environmentally-sustainable-data-cent Accessed on 30/05/2018.

[20] The MySQL Master-Master Replication. https://www.digitalocean.com/community/tutorials/how-to-set-up-mysql-master-master-replication. Accessed on 30/05/2018.

[21] Open Nebula. https://opennebula.org/. Accessed on 30/05/2018.

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

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

[24] Page Modification Logging for Virtual Machine Monitor White Paper. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/page-modification-logging-vmm-white-paper.pdf. Accessed on 30/05/2018.

[25] Roboconf. http://roboconf.net/en/index.html. Accessed on 30/05/2018.

[26] SPECjAppServer2004. https://www.spec.org/jAppServer2004/. Accessed on 30/05/2018.

[27] SPECmail2008. https://www.spec.org/mail2008/. Accessed on 30/05/2018.

[28] SPECpower_ssj 2008. https://www.spec.org/power_ssj2008/. Accessed on 30/05/2018.

[29] SPECvirt_sc2010. https://www.spec.org/virt_sc2010/. Accessed on 30/05/2018.

[30] SPECweb2005. https://www.spec.org/web2005/. Accessed on 30/05/2018.

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

[32] Abdelsalam, H., Maly, K., Mukkamala, R., Zubair, M., and Kaminsky, D. Towards energy efficient change management in a cloud computing environment. In *IFIP International Conference on Autonomous Infrastructure, Management and Security* (2009), Springer, pp. 161–166.

[33] 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.

[34] Agarwal, Y., Hodges, S., Chandra, R., Scott, J., Bahl, P., and Gupta, R. Somniloquy: Augmenting network interfaces to reduce pc energy usage. In *NSDI* (2009), vol. 9, pp. 365–380.

[35] Aguilera, M. K., Amit, N., Calciu, I., Deguillard, X., Gandhi, J., Subrahmanyam, P., Suresh, L., Tati, K., Venkatasubramanian, R., and Wei, M. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 121–127.

[36] Ahmad, F., and Vijaykumar, T. Joint optimization of idle and cooling power in data centers while maintaining response time. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 243–256.

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

[38] 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.

[39] 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.

[40] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., et al. A view of the parallel computing landscape. *Communications of the ACM 52*, 10 (2009), 56–67.

[41] Bai, Y., Lee, V. W., and Ipek, E. Voltage regulator efficiency aware power management. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ACM, pp. 825–838.

[42] 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.

[43] Barker, S. K., Wood, T., Shenoy, P. J., and Sitaraman, R. K. An empirical study of memory sharing in virtual machines. In *USENIX Annual Technical Conference* (2012), pp. 273–284.

[44] Barroso, L. A. Warehouse-scale computing: Entering the teenage decade.

[45] 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.

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

[47] Bash, C., and Forman, G. Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *USENIX Annual Technical Conference* (2007), vol. 138, p. 140.

[48] Beloglazov, A., Abawajy, J., and Buyya, R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems 28*, 5 (2012), 755–768.

[49] 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.

[50] Bila, N., de Lara, E., Joshi, K., Lagar-Cavilla, H. A., Hiltunen, M., and Satyanarayanan, M. Jettison: efficient idle desktop consolidation with partial vm migration. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 211–224.

[51] Binnig, C., Crotty, A., Galakatos, A., Kraska, T., and Zamanian, E. The end of slow networks: it's time for a redesign. *Proceedings of the VLDB Endowment 9*, 7 (2016), 528–539.

[52] 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.

[53] Bobroff, N., Kochut, A., and Beaty, K. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on* (2007), IEEE, pp. 119–128.

[54] Boncz, P. A., Kersten, M. L., and Manegold, S. Breaking the memory wall in monetdb. *Communications of the ACM 51*, 12 (2008), 77–85.

[55] 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.

[56] Chaisiri, S., Lee, B.-S., and Niyato, D. Optimal virtual machine placement across multiple cloud providers. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific* (2009), IEEE, pp. 103–110.

[57] Chen, L., and Shen, H. Considering resource demand misalignments to reduce resource over-provisioning in cloud datacenters. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE* (2017), IEEE, pp. 1–9.

[58] Chen, Y., Alspaugh, S., Borthakur, D., and Katz, R. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 43–56.

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

[60] 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.

[61] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 153–167.

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

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

[64] Delimitrou, C., and Kozyrakis, C. Hcloud: Resource-efficient provisioning in shared cloud systems. *ACM SIGOPS Operating Systems Review 50*, 2 (2016), 473–488.

[65] Deng, Q., Meisner, D., Ramos, L., Wenisch, T. F., and Bianchini, R. Memscale: active low-power modes for main memory. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 225–238.

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

[67] Dósa, G. The tight bound of first fit decreasing bin-packing algorithm is ffd (i) 11/9opt (i)+ 6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 2007, pp. 1–11.

[68] Facebook. Prineville, OR Data Center. https://fbpuewue.com/prineville. Accessed on 30/05/2018.

[69] Farahnakian, F., Pahikkala, T., Liljeberg, P., Plosila, J., and Tenhunen, H. Utilization prediction aware vm consolidation approach for green cloud computing. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on* (2015), IEEE, pp. 381–388.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] Goiri, Í., Nguyen, T. D., and Bianchini, R. Coolair: Temperature- and variation-aware management for free-cooled datacenters. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 253–265.

[75] Google. Efficiency: How we do it. https://www.google.com/about/datacenters/efficiency/internal/. Accessed on 30/05/2018.

[76] Greenhalgh, P. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper 17* (2011).

[77] Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., and Shin, K. G. Efficient memory disaggregation with infiniswap. In *NSDI* (2017), pp. 649–667.

[78] Guo, Z., Pierce, M., Fox, G., and Zhou, M. Automatic task reorganization in mapreduce. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on* (2011), IEEE, pp. 335–343.

[79] 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.

[80] Habib, I. Virtualization with kvm. *Linux Journal 2008*, 166 (2008), 8.

[81] Hamilton, J. Cooperative expendable micro-slice servers (cems): low cost, low power servers for internet-scale services. In *Conference on Innovative Data Systems Research (CIDR09)(January 2009)* (2009), Citeseer.

[82] Han, S., Egi, N., Panda, A., Ratnasamy, S., Shi, G., and Shenker, S. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013), ACM, p. 10.

[83] Handlin, H. Using a Total Cost of Ownership (TCO) Model for Your Data Center. http://www.datacenterknowledge.com/archives/2013/10/01/using-a-total-cost-of-ownership-tco-model-for-your-data-center. Accessed on 30/05/2018.

[84] He, H., Hu, J., and Da Silva, D. Enhancing datacenter resource management through temporal logic constraints. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International* (2017), IEEE, pp. 133–142.

[85] Hines, M. R., Deshpande, U., and Gopalan, K. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review 43*, 3 (2009), 14–26.

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

[87] Isci, C., McIntosh, S., Kephart, J., Das, R., Hanson, J., Piper, S., Wolford, R., Brey, T., Kantner, R., Ng, A., et al. Agile,

efficient virtualization power management with low-latency server power states. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 96–107.

[88] Jayasinghe, D., Pu, C., Eilam, T., Steinder, M., Whally, I., and Snible, E. Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement. In *Services Computing (SCC), 2011 IEEE International Conference on* (2011), IEEE, pp. 72–79.

[89] Jiang, C., Wang, Y., Ou, D., Luo, B., and Shi, W. Energy proportional servers: Where are we in 2016? In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 1649–1660.

[90] Jin, H., Cheocherngngarn, T., Levy, D., Smith, A., Pan, D., Liu, J., and Pissinou, N. Joint host-network optimization for energy-efficient data center networking. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013), IEEE, pp. 623–634.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] Konstanteli, K., Cucinotta, T., Psychas, K., and Varvarigou, T. Admission control for elastic cloud services. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (2012), IEEE, pp. 41–48.

[97] Li, X., Zhang, P., Chu, R., and Wang, H. Optimizing guest swapping using elastic and transparent memory provisioning on virtualization platform. *Frontiers of Computer Science 10*, 5 (2016), 908–924.

[98] 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.

[99] Lin, S.-H., Pal, R., Paolieri, M., and Golubchik, L. Performance driven resource sharing markets for the small cloud. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 241–251.

[100] 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.

[101] Liu, J., Jaiyen, B., Veras, R., and Mutlu, O. Raidr: Retention-aware intelligent dram refresh. In *ACM SIGARCH Computer Architecture News* (2012), vol. 40, IEEE Computer Society, pp. 1–12.

[102] Liu, L., Wang, H., Liu, X., Jin, X., He, W. B., Wang, Q. B., and Chen, Y. Greencloud: a new architecture for green data center. In *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session* (2009), ACM, pp. 29–38.

[103] Liu, N., Li, Z., Xu, J., Xu, Z., Lin, S., Qiu, Q., Tang, J., and Wang, Y. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 372–382.

[104] Liu, S., Pattabiraman, K., Moscibroda, T., and Zorn, B. G. Flikker: saving dram refresh-power through critical data partitioning. *ACM SIGPLAN Notices 47*, 4 (2012), 213–224.

[105] Liu, Y., Draper, S. C., and Kim, N. S. Sleepscale: runtime joint speed scaling and sleep states management for power efficient data centers. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 313–324.

[106] Lo, D., Cheng, L., Govindaraju, R., Barroso, L. A., and Kozyrakis, C. Towards energy proportionality for large-scale latency-critical workloads. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 301–312.

[107] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News* (2015), vol. 43, ACM, pp. 450–462.

[108] Lorrillere, M., Sopena, J., Monnet, S., and Sens, P. Puma: pooling unused memory in virtual machines for i/o intensive applications. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015), ACM, p. 1.

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

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

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

[112] Manousakis, I., Goiri, Í., Sankar, S., Nguyen, T. D., and Bianchini, R. Coolprovision: Underprovisioning datacenter cooling. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 356–367.

[113] 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.

[114] Meisner, D., Sadler, C. M., Barroso, L. A., Weber, W.-D., and Wenisch, T. F. Power management of online data-intensive services. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 319–330.

[115] Meisner, D., and Wenisch, T. F. Dreamweaver: architectural support for deep sleep. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 313–324.

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

[117] 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.

[118] 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.

[119] Mishra, N., Zhang, H., Lafferty, J. D., and Hoffmann, H. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *ACM SIGARCH Computer Architecture News* (2015), vol. 43, ACM, pp. 267–281.

[120] Mukherjee, T., Dutta, P., Hegde, V. G., and Gujar, S. Risc: Robust infrastructure over shared computing resources through dynamic pricing and incentivization. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International* (2015), IEEE, pp. 1107–1116.

[121] Murtazaev, A., and Oh, S. Sercon: Server consolidation algorithm using live migration of virtual machines for green computing. *IETE Technical Review 28*, 3 (2011), 212–231.

[122] Nakada, H., Hirofuchi, T., Ogawa, H., and Itoh, S. Toward virtual machine packing optimization based on genetic algorithm. In *International Work-Conference on Artificial Neural Networks* (2009), Springer, pp. 651–654.

[123] Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A. Migrating server storage to ssds: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), ACM, pp. 145–158.

[124] Nguyen Van, H., Dang Tran, F., and Menaud, J.-M. Autonomic virtual resource management for service hosting platforms. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing* (2009), IEEE Computer Society, pp. 1–8.

[125] Nowatzyk, A., Pong, F., and Saulsbury, A. Missing the memory wall: The case for processor/memory integration. In *Computer Architecture, 1996 23rd Annual International Symposium on* (1996), IEEE, pp. 90–90.

[126] NVIDIA. Tegra 3. http://www.nvidia.com/object/tegra-3-processor.html. Accessed on 30/05/2018.

[127] O'Connor, M., Chatterjee, N., Lee, D., Wilson, J., Agrawal, A., Keckler, S. W., and Dally, W. J. Fine-grained dram: energy-efficient dram for extreme bandwidth systems. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017), ACM, pp. 41–54.

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

[129] 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.

[130] Prekas, G., Primorac, M., Belay, A., Kozyrakis, C., and Bugnion, E. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 342–355.

[131] Quiroz, A., Kim, H., Parashar, M., Gnanasambandam, N., and Sharma, N. Towards autonomic workload provisioning for enterprise grids and clouds. In *Grid Computing, 2009 10th IEEE/ACM International Conference on* (2009), IEEE, pp. 50–57.

[132] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI* (2013), vol. 13, pp. 227–240.

[133] Ravi, G. S., and Lipasti, M. H. Charstar: Clock hierarchy aware resource scaling in tiled architectures. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ACM, pp. 147–160.

[134] Robison, A., Page, C., and Lytle, B. Yahoo! compute coop (ycc). a next-generation passive cooling design for data centers. Tech. rep., Yahoo! Inc., Sunnyvale, CA, 2011.

[135] 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.

[136] Sharma, P., Irwin, D., and Shenoy, P. Portfolio-driven resource management for transient cloud servers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 1*, 1 (2017), 5.

[137] Sharma, P., and Kulkarni, P. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* (2012), ACM, pp. 15–26.

[138] Somu Muthukaruppan, T., Pathania, A., and Mitra, T. Price theory based power management for heterogeneous multi-cores. *ACM SIGPLAN Notices 49*, 4 (2014), 161–176.

[139] 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.

[140] 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.

[141] Taram, M., and Tullsen, A. V. D. M. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency.

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

[143] Vamanan, B., Sohail, H. B., Hasan, J., and Vijaykumar, T. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on* (2015), IEEE, pp. 585–597.

[144] Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor,

M. B. Conservation cores: reducing the energy of mature computations. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 205–218.

[145] 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.

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

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

[148] 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.

[149] Watanabe, Y., Davis, J. D., and Wood, D. A. Widget: Wisconsin decoupled grid execution tiles. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 2–13.

[150] Williams, D., Jamjoom, H., and Weatherspoon, H. Software defining system devices with the" banana" double-split driver model. In *HotCloud* (2014).

[151] Wong, D., and Annavaram, M. Knightshift: Scaling the energy proportionality wall through server-level heterogeneity. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), IEEE Computer Society, pp. 119–130.

[152] Wulf, W. A., and McKee, S. A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news 23*, 1 (1995), 20–24.

[153] Xu, C., Lin, F. X., Wang, Y., and Zhong, L. *Automated os-level device runtime power management*, vol. 43. ACM, 2015.

[154] 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.

[155] Yeo, S., Hossain, M. M., Huang, J.-C., and Lee, H.-H. S. Atac: Ambient temperature-aware capping for power efficient datacenters. In

Proceedings of the ACM Symposium on Cloud Computing (2014), ACM, pp. 1–14.

[156] 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).

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

[158] Zheng, K., and Wang, X. Dynamic control of flow completion time for power efficiency of data center networks. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 340–350.

[159] Zheng, K., Wang, X., and Wang, X. Powerfct: Power optimization of data center network with flow completion time constraints. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International* (2015), IEEE, pp. 334–343.

[160] Zheng, W., Ma, K., and Wang, X. Tecfan: Coordinating thermoelectric cooler, fan, and dvfs for cmp energy optimization. In *Parallel and Distributed Processing Symposium, 2016 IEEE International* (2016), IEEE, pp. 423–432.

[161] Zhi, J., Bila, N., and de Lara, E. Oasis: energy proportionality with hybrid server consolidation. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 10.

[162] 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.