Autonomic and Energy-Efficient Management of Large-Scale Virtualized Data Centers

Eugen Feller

ANNÉE 2012

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention: INFORMATIQUE

Ecole doctorale MATISSE

présentée par

Eugen Feller

préparée à l'unité de recherche n^{o} 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
ISTIC

Autonomic and Energy-Efficient Management Of Large-Scale Virtualized Data Centers

Thèse soutenue à Rennes
le 17 decembre 2012

devant le jury composé de:

Ricardo BIANCHINI / Rapporteur
Professeur, Rutgers University, NJ, USA

Jean-Marc PIERSON / Rapporteur
Professeur, Université Paul Sabatier, Toulouse

Jean-Marc MENAUD / Examinateur
Maître de Conférences, HDR, Ecole des Mines de Nantes

Jean-Louis PAZAT / Examinateur
Professeur, INSA Rennes

Michael SCHÖTTNER / Examinateur
Professeur, Heinrich-Heine-University Düsseldorf, Germany

Christine MORIN / Directrice de thèse
Directrice de Recherche, Inria Rennes - Bretagne Atlantique

Phantasie ist wichtiger als Wissen, denn Wissen ist begrenzt. Phantasie umgibt die ganze Welt.
Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.
– Albert Einstein

Acknowledgements

Foremost, I would like to thank my advisor Christine Morin for all her enthusiasm and outstanding support without which this thesis would not have been possible. Despite having many responsibilities she has always found time to discuss about new ideas and provide detailed feedbacks and suggestions on my work. Thank you Christine, for being an excellent person and researcher. It has been a great honour and pleasure for me to work with you during the last three and a half years and I hope to continue in the future.

I would like to gratitude the members of my committee. Thank you Ricardo Bianchini and Jean-Marc Pierson for making me the honour to review my work. Thank you Jean-Marc also for supporting my activities at the COST Action IC804. I also thank Jean-Marc Menaud, Jean-Louis Pazat, and Michael Schöttner for taking your time to evaluate my work. A very special thanks to Michael for his excellent advising during my stay at the distributed systems group of the Heinrich-Heine-University Düsseldorf (Germany) and giving me the opportunity to work with Christine. I will never forget your and Christine's support.

I also want to acknowledge all the people who have contributed to this work. Special thanks to Louis, for all our discussions and his detailed reviews of my work from the very beginning of the thesis. Cyril for the adaption of his elastic web application which helped to evaluate this work and resulted in a joint conference publication. Armel, my master research student for his excellent work which also resulted in a conference publication. Piyush and Roberto for always being open to discuss and comment on my work. Ivona for the great joint work at the COST Action IC804. Finally, I am very thankful to Dan, Deb, and Lavanya for hosting me during my three month summer internship at the Lawrence Berkeley National Laboratory. In this context, I would like to thank all the members of the Advanced Computing for Science department for making my stay a pleasure. Especially, Devarshi, Elif, You-Wei, Taghrid, Valerie, and Keith for all the great time in and outside the lab.

I express my sincere gratitude to all my fellow labmates in the Myriads (former PARIS) project-team. First of all my office buddies André, Amine, Rémy, and Surbhi for all the pleasant time we spent together. Maryse, our secretary for handling most of the paper work. Françoise for giving me the opportunity to participate in the COST Action IC804. Cédric for the tough tennis matches and just being a wonderful person to interact with. Bogdan for all our nice discussions, coffee breaks, and lunches. David, Ghislain, Pascal, and Pierre for their support with the Grid'5000 experimentation testbed. Ancuta, Anne-Cécile, Alexandra, Chen, Djawida, Guillaume, Gabriel, Héctor, Jérôme, Katarzyna, Marko, Matthieu, Nikos, Peter, Sylvain, Sajith, Thomas, Yvon, Yann, and all other team members which contributed to the warm atmosphere which I have experienced during my stay in France.

This thesis is devoted to my parents, Alexandra and Aron Feller. Your love, patience, and support has always been a great motivation for me to ever move forward. I can't thank you enough for everything you did for me. Huge thanks to my brother Alexander Feller and uncle Oleg Sternberg for always being open minded and ready to discuss about my work.

The contributions presented in this thesis were funded by the French Agence Nationale de la Recherche (ANR) project EcoGrappe under the contract number ANR-08-SEGI-008-02. They were evaluated using the Grid'5000 experimentation testbed, being developed under the Inria ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

Autonomic and Energy-Efficient Management of Large-Scale Virtualized Data Centers

Abstract

Large-scale virtualized data centers require cloud providers to implement scalable, autonomic, and energy-efficient cloud management systems. To address these challenges this thesis provides four main contributions. The first one proposes Snooze, a novel Infrastructure-as-a-Service (IaaS) cloud management system, which is designed to scale across many thousands of servers and virtual machines (VMs) while being easy to configure, highly available, and energy efficient. For scalability, Snooze performs distributed VM management based on a hierarchical architecture. To support ease of configuration and high availability Snooze implements self-configuring and self-healing features. Finally, for energy efficiency, Snooze integrates a holistic energy management approach via VM resource (i.e. CPU, memory, network) utilization monitoring, underload/overload detection and mitigation, VM consolidation (by implementing a modified version of the Sercon algorithm), and power management to transition idle servers into a power saving mode. A highly modular Snooze prototype was developed and extensively evaluated on the Grid'5000 testbed using realistic applications. Results show that: (i) distributed VM management does not impact submission time; (ii) fault tolerance mechanisms do not impact application performance and (iii) the system scales well with an increasing number of resources thus making it suitable for managing large-scale data centers. We also show that the system is able to dynamically scale the data center energy consumption with its utilization thus allowing it to conserve substantial power amounts with only limited impact on application performance. Snooze is an open-source software under the GPLv2 license.

The second contribution is a novel VM placement algorithm based on the Ant Colony Optimization (ACO) meta-heuristic. ACO is interesting for VM placement due to its polynomial worst-case time complexity, close to optimal solutions and ease of parallelization. Simulation results show that while the scalability of the current algorithm implementation is limited to a smaller number of servers and VMs, the algorithm outperforms the evaluated First-Fit Decreasing greedy approach in terms of the number of required servers and computes close to optimal solutions. In order to enable scalable VM consolidation, this thesis makes two further contributions: (i) an ACO-based consolidation algorithm; (ii) a fully decentralized consolidation system based on an unstructured peer-to-peer network. The key idea is to apply consolidation only in small, randomly formed neighbourhoods of servers. We evaluated our approach by emulation on the Grid'5000 testbed using two state-of-the-art consolidation algorithms (i.e. Sercon and V-MAN) and our ACO-based consolidation algorithm. Results show our system to be scalable as well as to achieve a data center utilization close to the one obtained by executing a centralized consolidation algorithm.

Keywords: Autonomic computing, Cloud computing, Scalability, Self-configuration, Self-healing, Energy efficiency, Ant Colony Optimization, Consolidation, Virtualization

Gestion autonome et économique en énergie des grands centres de données virtualisés

Résumé

Les grands centres de données virtualisés nécessitent que les fournisseurs de nuages informatiques mettent en œuvre des systèmes de gestion de machines virtuelles passant à l'échelle, autonomes et économiques en énergie. Pour répondre à ces défis, cette thèse apporte quatre contributions principales. La première est la proposition d'un nouveau système de gestion de nuages IaaS, Snooze, qui a été conçu pour gérer plusieurs milliers de serveurs et de machines virtuelles (VMs) tout en étant facile à configurer, hautement disponible et économique en énergie. Pour le passage à l'échelle, Snooze gère les VM de manière distribuée sur la base d'une architecture hiérarchique. Pour offrir la facilité de configuration et la haute disponibilité, Snooze met en œuvre des mécanismes d'auto-configuration et d'auto-réparation. Finalement, pour l'efficacité énergétique, Snooze est fondé sur une approche globale à travers la surveillance de la consommation de ressources (i.e. CPU, mémoire, réseau) des VMs, la détection et la résolution des situations de sous-charge et de surcharge, la consolidation de VMs (par la mise en œuvre d'une version modifiée de l'algorithme Sercon) et la gestion de la consommation d'énergie en faisant passer les serveurs inactifs dans un mode de faible consommation énergétique. Un prototype modulaire du système Snooze a été développé et a fait l'objet d'une évaluation approfondie à l'aide d'applications réalistes sur la plate-forme Grid'5000. Les résultats montrent que (i) la gestion distribuée des VMs est sans impact sur le temps de soumission, (ii) les mécanismes de tolérance aux fautes n'ont pas d'impact sur les performances des applications, et que le système passe à l'échelle avec le nombre de ressources, ce qui fait qu'il est approprié pour les grands centres de données. Nous montrons également que le système est capable d'adapter la consommation énergétique du centre de données par rapport à sa charge permettant donc de substantielles économies d'énergie avec seulement un impact limité sur les performances des applications. Snooze est un logiciel libre sous licence GPLv2.

La seconde contribution est un nouvel algorithme de placement de VMs fondé sur la méta-heuristique d'optimisation par colonies de fourmis (ACO). L'ACO est intéressante pour le placement de VMs en raison de sa complexité dans le pire cas polynomiale, de ses solutions proches de l'optimal et de sa facilité de parallélisation. Les résultats de simulation montrent que le passage à l'échelle de la mise en œuvre actuelle de l'algorithme est limité à un petit nombre de serveurs et de VMs. Cependant, l'algorithme se comporte mieux que l'approche gloutonne *First-Fit-Decreasing* pour le compactage des VMs et qu'il calcule des solutions proches de l'optimal. Pour une consolidation de VMs passant à l'échelle, cette thèse apporte deux contributions supplémentaires : (i) un algorithme de consolidation fondé sur l'ACO, (ii) un système de consolidation totalement décentralisé fondé sur un réseau pair-à-pair non structuré. L'idée clé est d'appliquer la consolidation dans de petits groupes de serveurs formés aléatoirement. Nous avons évalué notre approche par émulation sur la plate-forme Grid'5000 en utilisant deux algorithmes de consolidation existants (i.e. Sercon et V-MAN) ainsi que notre algorithme fondé sur l'ACO. Les résultats montrent que notre système passe à l'échelle et permet d'obtenir un taux d'utilisation du centre de données proche de celui qui serait obtenu avec un algorithme de consolidation centralisé.

Mots clés: Système autonome, cloud computing, passage à l'échelle, auto-configuration, efficacité énergétique, optimisation par colonies de fourmis, consolidation, virtualisation

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ACO	Ant Colony Optimization
AC	Alternating Current
AE	Application Environment
AIS	All-In Strategy
ALR	Adaptive Link Rate
AM	Autonomic Manager
API	Application Programming Interface
BEEMR	Berkeley Energy Efficient MapReduce
BF	Best Fit
BIP	Binary Integer Programming
BMC	Baseboard Management Controller
CC	Cluster Controller
CDSP	Cost-driven Scheduling Policy
CLC	Cloud Controller
CLI	Command Line Interface
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
CS	Covering Subset
DCiE	Data Center infrastructure Efficiency
DC	Direct Current
DFS	Distributed File System
DPM	Dynamic Power Management
DRAC	Dell Remote Access Controller
DRPM	Dynamic Rotations Per Minute

DRR	Dynamic Round Robin
DVFS	Dynamic Voltage and Frequency Scaling
DVMS	Distributed VM Scheduler
EC2	Amazon Elastic Compute Cloud
EC	Efficiency Controller
EM	Enclosure Manager
EP	Entry Point
FFD	First Fit Decreasing
FF	First Fit
FLOPS	Floating Point Operations Per Second
GA	Genetic Algorithm
GLPK	GNU Linear Programming Kit
GL	Group Leader
GM	Group Manager
GPFS	General Parallel File System
HaaS	Hardware-as-a-Service
HA	High Availability
HDFS	Hadoop Distributed File System
IaaS	Infrastructure-as-a-Service
IPC	Instructions Per Cycle
IPMI	Intelligent Platform Management Interface
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LAN	Local Area Network
LC	Local Controller
LP	Linear Programming
MAPE-K	Monitor, Analyze, Plan, Execute, and Knowledge
MCDA	Multiple Criteria Decision Analysis
MCP	Maximum Correlation Policy
MDBPP	Multi-Dimensional Bin Packing Problem
ME	Managed Element
MILP	Mixed-Integer-Linear-Programming

MMAS	MAX-MIN Ant System
MMS	Min, Max, and Share
MMT	Minimum Migration Time
MPC	Memory Per Cycle
MPI	Message Passing Interface
MPL	Memory Power Limiting
NC	Node Controller
NF	Next Fit
NIC	Network Interface Card
OCCI	Open Cloud Computing Interface
ODBPP	One-Dimensional Bin-Packing Problem
OF	Objective Function
OS	Operating System
P2P	Peer-to-Peer
P2V	Physical-to-Virtual
PaaS	Platform-as-a-Service
PADS	Power-Aware Domain Distribution
PCPG	Per-Core Power Gating
PCU	Power Control Unit
PDU	Power Distribution Unit
PE	Packing Efficiency
PM	Physical Machine
PSU	Power Supply Unit
PUE	Power Usage Effectiveness
QEMU	Quick EMUlator
QoS	Quality-of-Service
RAPL	Running Average Power Limit
RC	Random Choice
SaaS	Software-as-a-Service
SLA	Service Level Agreement
SLURM	Simple Linux Utility for Resource Management
SM	Server Manager

SNMP	Simple Network Management Protocol
SPM	Static Power Management
SPOF	Single Point of Failure
SSAP	Static Server Allocation Problem
SSD	Solid State Disk
TCD	Threshold Crossing Detection
TCO	Total Cost of Ownership
TCP	Transmission Control Protocol
UID	Unique Identifier
VC	Virtual Cluster
VLAN	Virtual Local Area Network
VMCS	Virtual Machine Control Structure
VMC	Virtual Machine Controller
VMM	Virtual Machine Monitor
VMPP	Virtual Machine Placement Problem
VM	Virtual Machine
WAN	Wide Area Network
WF	Worst Fit
WOL	Wake-on-LAN
WWS	Writable Working Set
XML	Extensible Markup Language

Chapter 1

Introduction

Contents

THIS chapter provides the motivation for this thesis, introduces its objectives, discusses its contributions and presents the document outline.

1.1 Motivation

Cloud computing has recently emerged as a new computing paradigm in which services are offered based on the pay-as-you-go model. Customers consuming those services are charged only for as much as they have used. One particularly cloud service model which has gained a lot of attraction over the past years is commonly refereed to as Infrastructure-as-a-Service (IaaS). In IaaS clouds, resources such as compute and storage are provisioned on-demand by the cloud providers. Thereby, compute capacity is typically provided in the form of virtual machines (VMs). VMs appear to the customers as if they were real physical machines (PMs). VMs were made possible through latest advances in server virtualization technologies which allow to efficiently multiplex PM resources (e.g. CPU, memory, I/O devices) between multiple VMs.

Since the introduction of cloud computing, many cloud providers (e.g. Amazon, Google, Rackspace) have appeared and are now offering a tremendous amount of services such as compute capacity and data storage on demand. In order to support the customers growing service demands, cloud providers have recently started to deploy an increasing number of large-scale data centers. Managing such data centers, requires the cloud providers to solve a number of challenges. Particularly, cloud providers now must design and implement novel

IaaS cloud computing systems [1] which are capable of operating at *large scale*. More precisely, the cloud computing systems must remain *scalable* in order to support the increasing number of customers and resources (i.e. VMs and PMs). In addition, besides beeing scalable, the probability for hardware and software failures increases at scale. Consequently, cloud management systems must be designed to behave *autonomically*. This will allow them to automatically detect failures and initiate a recovery. Moreover, configuring cloud management systems to operate at large scale requires a substantial amount of highly skilled IT experts. In order to automate the configuration efforts, IaaS cloud management systems must be designed with self-configuration aspects in mind thus enabling system configuration with minimal human intervention. Last but not least, data centers are now hosting equipment (e.g. storage and compute servers, cooling) requiring huge amounts of energy. For instance, Google which is a major internet search engine and cloud computing services provider alone accommodates over 900,000 servers which have consumed approximately 2 billion kWh of energy in 2010 [195]. While Google's data centers energy requirements are still less than 1% of the worlds data center energy demands, reducing the energy consumption during periods of low utilization in data centers is crucial in order to lower the data centers Total Cost of Ownership (TCO) and carbon footprints in a time where most of the data centers are still powered by either coal or nuclear power plants [180]. Given the importance of energy savings, energy-efficient IaaS cloud management systems must be designed.

Several attempts have been made over the past years to design and implement IaaS cloud management systems to facilitate the creation of private IaaS clouds. Given the increasing data center scales, such systems are faced with challenges in terms of scalability, autonomy, and energy-efficiency. However, many of the existing attempts to design and implement IaaS cloud systems for private clouds are still based on centralized architectures, have limited autonomy, and lack of energy saving mechanisms. Consequently, they are subject to *Single Point Of Failure (SPOF)*, *limited scalability*, and *low energy efficiency*.

1.2 Objectives

The goal of this thesis is to design, implement, and evaluate an IaaS cloud management system for large-scale data centers. To achieve its main goal this thesis investigates the following four sub-goals:

- **Scalability:** Data centers are now hosting many thousands of servers. For instance, Rackspace which is a well known Infrastructure-as-a-Service (IaaS) provider accommodated approximately 78 000 servers in 2011 [248]. Managing such amount of servers requires highly scalable IaaS cloud management systems. Consequently, our goal is to design a system that *scales with increasing number of servers (PMs and VMs)*.
- **High Availability:** With increasing number of servers, the probability for system component (hardware and software) failures increases. In order to support continuous system operation the IaaS cloud management system should be *highly available*. Consequently, high availability mechanisms must be implemented. Our goal is to design

1. Cloud computing systems are defined as a software frameworks capable of managing the physical data center resources.

a system which integrates such mechanisms.

- **Ease of Management:** Managing large scale data centers can be a tremendous task which requires many highly experienced IT experts. Providing an easily configurable system can significantly reduce the costs and ease the system management. One of our goals is to design a system which *requires minimal human intervention to be configured*. Moreover, once the system is deployed and configured, it becomes increasingly important to perform updates and/or add new servers. In such scenarios servers will be required to brought offline and added back later. Our goal is to design a system which is flexible enough to allow for *dynamic addition and removal of servers*. Finally, as system components can fail at any time, it is desirable for a system to *heal* in the event of failures without human intervention. Consequently, we aim at designing a system using *self-healing* mechanisms to enable high availability.
- **Energy Efficiency:** Over the past years, rising energy bills have resulted in energy efficiency to become a major design constraint for data center providers. Given that traditional data centers are rarely fully utilized, significantly energy savings can be achieved during periods of low utilization by transitioning idle servers in a power saving state. However, as servers are rarely fully idle, first idle times need to be created [81]. Our goal is to design an IaaS cloud management system and VM management algorithms which are capable of creating idle times, transitioning idle servers in a power saving state and waking them up once required (e.g. when load increases). This will allow to scale the data center energy consumption proportionally to its load.

1.3 Contributions

To tackle the introduced sub-goals this thesis makes the following three contributions:

Snooze: A Scalable, Autonomic, and Energy-Efficient IaaS Cloud Manager. We propose an autonomic and energy-efficient IaaS cloud management system for large-scale virtualized data centers called Snooze. For scalability Snooze is based on a self-configuring hierarchical architecture and performs distributed VM management. Distributed VM management is achieved by splitting the data center into independently managed groups of PMs and VMs. Moreover, autonomy features are provided at all levels of the hierarchy, thus allowing the system to self-configure upon bootup and provide high availability via self-healing in case of failures. To save energy, Snooze provides a holistic energy-efficient VM management solution. Particularly, it integrates a power management mechanism which automatically detects idle PMs, transitions them into a power-saving state (e.g. suspend), and wakes them up once required. However, before this can be achieved, idle times need to be created as VMs are typically load balanced across the PMs. To create idle-times, underload detection and mitigation are performed along with VM consolidation. Both mechanisms aim at releasing lightly utilized PMs (resp. pack VMs on the least number of PMs). To evaluate Snooze, a prototype has been implemented and extensively evaluated using realistic applications on the Grid'5000 experimentation testbed. The experimental results have proven our system to be scalable, autonomic, and energy-efficient. The core system principles of Snooze were published in [131, 135, 136]. The scalability and autonomy evaluation was published

in [134]. Finally, the description and evaluation of the energy management mechanisms was published in [137].

VM Placement via Ant Colony Optimization. One traditional approach to favour idle times starting from the VM submission in IaaS cloud management systems, is to allocate a set of VMs to PMs such that the number of required PMs to accommodate the VMs is minimized. This is achieved by implementing the so-called VM placement algorithms. However, many of the traditional VM placement algorithms consider only a single resource (e.g. CPU) to evaluate the PM load and VM resource demands. Moreover, they rely on centralized algorithms such as First-Fit Decreasing (FFD) [304] which are known to be hard to distribute/-parallelize [76]. To solve these limitations, we investigate the use of Ant Colony Optimization (ACO) for the VM placement problem and propose an ACO-based VM placement algorithm. ACO is especially attractive for the VM placement problem due to its polynomial time worst-case complexity and the ease of parallelization. We evaluate the ACO-based approach by comparing it with the FFD algorithm and the optimal solution as computed using the IBM ILOG CPLEX optimizer [33]. Simulation results demonstrate that ACO outperforms the FFD algorithm as it achieves superior energy gains through better PM utilization and requires less PMs. Moreover, it computes solutions which are close to optimal. This work was published in [133].

VM Consolidation via Ant Colony Optimization. The previous contribution has shown that even though ACO computes near optimal solutions, the designed algorithm still had scalability issues in terms of its computing time when considering a large number of PMs and VMs. Moreover, while addressing the VM placement problem is important in order to favour idle-time creation starting from the VM submission, VM consolidation algorithms are required in order to enable *continuous consolidation of already placed VMs* on the least number of PMs. This is particularly important in order to avoid resource fragmentation and further increase the data center resource utilization. To address both aspects this thesis makes the following two contributions: (1) we adapt our previously proposed ACO-based VM placement algorithm to enable continuous VM consolidation; (2) to tackle the scalability issues we propose a fully decentralized VM consolidation system based on an unstructured peer-to-peer (P2P) network of PMs. The key idea of the proposed system to achieve both scalability and high data center utilization is to apply VM consolidation only in the scope of randomly formed neighbourhoods of PMs. Considering the computational complexity of dynamic VM consolidation limiting its application to the scope of the neighbourhoods greatly improves the system scalability. In addition to that, the randomness of the neighbourhoods facilitates the convergence of the system towards a global packing efficiency very close to a centralized system by leveraging traditional centralized VM consolidation algorithms. Packing efficiency is defined as the ratio between the number of released PMs to the total number of PMs. We have implemented a distributed system emulator and validated it using two well known VM consolidation algorithms: Sercon [224], V-MAN [216], and the ACO-based VM consolidation algorithm. Extensive experiments performed on the Grid'5000 experimentation testbed show that once integrated in our fully decentralized VM consolidation system, traditional VM consolidation algorithms achieve a global packing efficiency very close to a centralized system. Moreover, the system remains scalable with increasing number of PMs and VMs. Finally, the ACO-based VM consolidation algorithm outperforms Sercon in the

number of released PMs and requires less migrations than V-MAN. This results were published in [132].

1.4 Outline of the Thesis

This thesis is organized as follows:

- Chapter 2 covers the state of the art. Particularly, it first presents the context of this dissertation by giving a brief introduction to server virtualization, autonomic computing, and cloud computing. Then, existing energy management approaches in computing clusters are reviewed. Understanding the energy saving approaches is mandatory to position the energy management contributions of this work.
- Chapter 3 describes our first contribution: Snooze, an autonomic and energy-efficient IaaS cloud management system based on a self-configuring and healing hierarchical architecture. We first provide a high-level system architecture overview. Then, we detail how the hierarchy and energy-efficient VM management are achieved. This involves the description of the self-configuration and healing mechanisms as well as the energy-efficient VM management algorithms. Finally, important implementation aspects are presented and the results from the experimental evaluation are analyzed.
- Chapter 4 is devoted to VM management via Ant Colony Optimization. We first provide an introduction to the ACO. Then, the ACO-based VM placement algorithm is proposed and its evaluation results are presented. Afterwards, the VM placement algorithm is adapted to enable VM consolidation. Moreover, in order to improve its scalability, a fully decentralized VM consolidation system based on an unstructured P2P network of PMs is proposed. Finally, the evaluation results of the VM consolidation algorithm as well as the fully decentralized VM consolidation system are presented.
- Chapter 5 concludes this manuscript by summarizing our contributions and presenting future research directions.

Chapter 2

State of the Art

Contents

THIS PhD thesis proposes a novel autonomic and energy-efficient IaaS cloud management system for large-scale virtualized data centers. To provide the necessary background for our work, in this chapter we presents the state of the art in related fields which include server virtualization, autonomic computing, cloud computing, and energy-efficient management of computing clusters. First, the concept of server virtualization is

detailed. Server virtualization is a fundamental technology which can be used to enable efficient data center resources utilization. Then, autonomic computing and cloud computing are presented, two complementary concepts emerged as the result of the massive adaption of Internet and large-scale distributed systems over the past years. Finally, related work on energy management in computing clusters is presented.

2.1 Server Virtualization

This section gives a brief introduction into server virtualization. Note, that for the sake of ease of explanation we focus on CPU virtualization only. Obviously, other hardware subsystems (e.g. memory and I/O devices) need to be virtualized as well to enable complete server virtualization. First, the history behind server virtualization is presented. Afterwards, privilege levels of the x86 CPU architecture and their application in traditional Operating Systems (OS's) as well as virtualized environments are briefly reviewed. Understanding the CPU privilege levels and their role in virtualized environments is crucial in order to differentiate between the virtualization techniques. After introducing the privilege levels, the state of the art virtualized techniques are presented. Finally, VM live migration is introduced as a basic mechanisms allowing to move VMs between PMs with ideally no service downtime [110].

2.1.1 What is Server Virtualization?

The history of server virtualization goes back to the time of IBM mainframes in the mid 1960s [254]. IBM mainframes were large tightly-coupled multi-processor computer systems designed for high I/O throughput, security, and reliability. They were typically installed in large enterprises and governmental organization. During that time IBM was faced with the challenge of efficiently partitioning their mainframe systems hardware between multiple OS's (i.e. kernel, libraries, applications). Hosting multiple OS's on a single mainframe was mandatory in order to support a broad range of customers applications which were developed for different OS's [77]. IBM's solution to the problem was a software layer between the physical hardware and the OS's, the so-called hypervisor/Virtual Machine Monitor (VMM) [153]. OS's run on top of the VMM which gives them the illusion of beeing able to control the physical hardware (e.g. CPU, memory, storage, I/O devices) much like the processes are given the illusion of having the entire CPU and a large amount of virtual memory by the OS. In other words, a VMM can be seen as an intermediate OS which instead of switching the physical hardware between processes does it for entire OS's (see Figure 2.1).

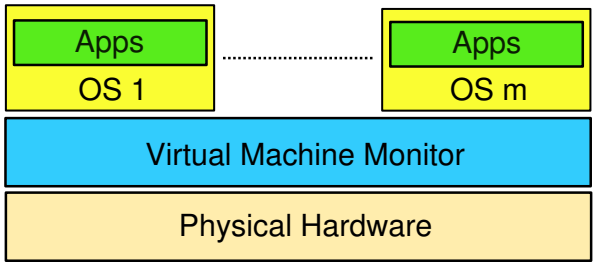


Figure 2.1: Virtual Machine Monitor example

While IBM's server virtualization efforts finally did not find a broad adaptation due to the emergence of distributed computing and low cost x86 server/desktop hardware in the early 1980s, they found their renaissance starting from the 1990s when companies were suddenly faced with the issue of having too many underutilized, power-hungry servers. Underutilized servers were the result of the adaption of more powerful hardware (i.e. multi-core machines) and the best practice of providing application isolation by running one application per server. In order to provide both, application isolation and efficient server hardware utilization, VMware introduced the concept of x86 server virtualization in the 1990s [58] which enabled to create and consolidate multiple virtual servers, the so-called VMs on a single PM. Each VM is defined by its meta-data describing the VM resource requirements (e.g. number of cores, memory) and a disk image where the actual OS (i.e. kernel, libraries, applications), the so-called guest OS resides [1]. VMs can be controlled (e.g. started, stopped, suspended) in the same manner as processes on a non-virtualized system. Similarly, to the OS's on the mainframes, guest OS's are managed by a VMM which controls the PM and provides guest OS's the illusion of beeing run on real hardware. This is achieved by multiplexing servers hardware between the guest OS's. To provide isolation between the guest OS's, VMM integrates security mechanisms which prevent multiple guest OS's kernels from modifying each others and VMM own memory. Since its introduction, server virtualization has become an ubiquitous technology in today's data centers to enable service isolation and efficient data center resource utilization.

Server virtualization is not to be confused with *emulation*. In contrast to virtualization which involves multiplexing the physical hardware between multiple guest OS's thus creating the illusion for the guests to be running on real hardware, emulation provides fully in-software emulated hardware (e.g. CPU, memory, I/O devices) to the guest OS's. For example, in a virtualized environment the guest OS must support the processor architecture (e.g. x86) of the host PM as its processor instructions are finally executed on it. In contrast, emulation enables to run guest OS's which have been developed for a specific processor architectures (e.g. VxWorks for PowerPC) on PM which have a distinct processor architecture and host OS (e.g. Linux on x86). This is typically achieved by a service, the so-called emulator which runs as an application on top of the PMs OS and emulates the desired hardware for the guest OS's (e.g. by implementing every CPU instruction in software) (see Figure 2.2). Emulation is very flexible as any hardware can be emulated for the guest OS's. This makes it

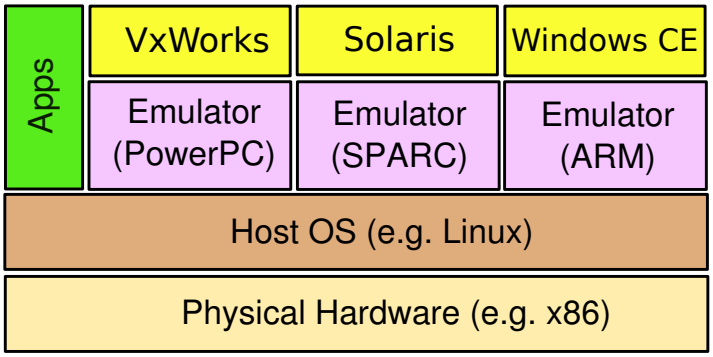


Figure 2.2: Emulation example

1. In this document we use the terms VM and guest OS interchangeably.

very convenient for developers who would like to debug specific guest OS's on their desktop computers (e.g. x86-based systems). However, the flexibility comes at the cost of decreased performance as all the guest OS hardware commands need to be interpreted and handled in software. Popular emulators include Quick EMUlator (QEMU) [82] when run in non-hardware accelerated mode and Bochs [202]. In the following sections we focus on *server virtualization techniques*.

2.1.2 Privilege Levels and Virtualization

In this section, we first briefly introduce the concept of privilege levels, also known as protection rings [261] which is implemented on all the modern x86 CPUs. Then, we discuss how privilege levels are used in traditional OS's and what extensions have been recently made by the major CPU vendors (i.e. Intel and AMD) to support virtualization technologies.

Privilege levels are a fundamental concept integrated into the x86 CPUs to enable security in modern OS's. Particularly, privilege levels allow the OS to prevent users and processes with different privileges from obtaining uncontrolled access to the shared physical resources such as CPU, memory, and I/O. For instance, a user process running on the OS should not be able to gain direct access to the disk as he then could initiate malicious activities such as deleting other users file or manipulate the hardware thus harming the system stability. Indeed, it is the job of the OS to grant/deny access to physical hardware and enforce the user process requests (e.g. sending data over the network) by interacting with the hardware. Figure 2.3 visualizes the privilege levels of the x86 architecture.

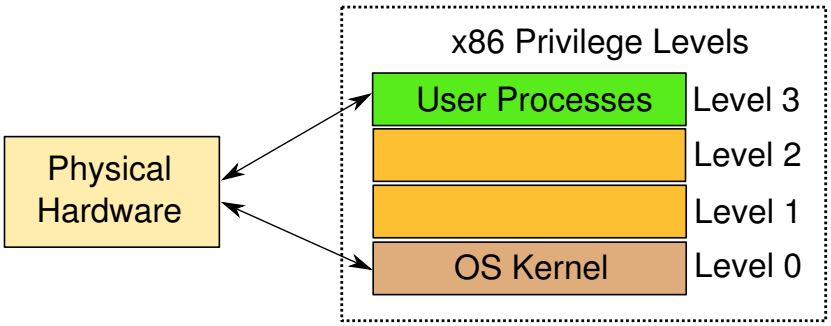


Figure 2.3: x86 architecture privilege levels overview

As it can be observed, there exist four privilege levels which are organized in increasing order form 0 to 3. Processes running on the level 0 and 3 have the highest (resp. lowest) privileges to access the physical hardware. As of today, the OS kernels of most of modern OS's (e.g. Linux, Windows) run in the highest privilege level, while user processes are typically executed in the lowest privilege level. The intermediate privilege levels remain unused. In other words, if a user process runs on the CPU, the CPU is in the lowest privilege level. On the other hand, when the user process instructs the OS (via software interrupt) to do some privileged instruction (e.g. I/O device access), the CPU switches into the highest privilege level and hands control over to the OS kernel so that it can perform the privileged instruction (e.g. by interacting with the OS network driver to send data) on behalf of the user process. After the OS finishes the critical task execution, the CPU switches back to the lowest

privilege level and hands control over to the user process. This procedure is also known as the *context-switch*.

When a VMM is used to control the physical hardware (i.e. CPU, memory, I/O) it must operate in the highest privilege level (i.e. 0) in order to: (1) support physical hardware multiplexing between the guest OS's; (2) secure the VMM and provide guest OS isolation. In that case the guest OS's kernel are obviously not allowed to use the highest privilege level. However, OS kernels were not designed to operate at any privilege level lower than 0. Indeed, they need to perform privileged instructions as well. One solution to this problem is to let the guest OS kernels run in a higher privilege level (e.g. 1) and let the VMM intercept and handle all the instructions requiring higher privilege level. However, traditional x86 CPU architecture was *not designed for virtualization*. Particularly, for an x86 architecture to be virtualizable all instructions which go beyond the scope of a certain privilege level must be cause a trap into the highest privilege level (i.e. 0). For example, when a guest OS kernel runs in privilege level 1 and issues an instruction requiring level 0 access it must trap into level 0. This way, a VMM running at privilege level 0 could handle the privilege instruction on behalf of the guest OS kernel. Unfortunately, as the x86 architecture was not designed for virtualization, some privileged instructions *do not cause traps into level 0* when executed from a lower privilege level (e.g. 3). Such instructions are also sometimes referred to as *non-virtualizable instructions*. This issue requires either to use emulation or complex mechanisms inside the VMM (e.g. binary translation [264]) to identify non-virtualizable instructions at run-time thus allowing the VMM to emulate their behaviour [237].

In order to provide a clean solution to the privilege level issue, recently both major CPU vendors Intel and AMD have introduced Intel VT-x [282] (resp. AMD-V [13]), extension to their CPU privilege levels hierarchy by adding one more level, called -1 [237]. This additional privilege level can be used to run the VMM thus allowing the guest OS kernels to continues operating in privilege level 0.

2.1.3 Virtualization Techniques

We now present the state of the art server virtualization techniques. Particularly, we discuss solutions to the previously introduced issue of x86 architecture virtualization. Server virtualization techniques can divided into three categories: full virtualization, paravirtualization, and OS-level virtualization [225, 290, 295, 215].

2.1.3.1 Full Virtualization

Full virtualization enables to run guest OS's on top of the existing host OS without the need to do any modifications to the host OS or guest OS's kernels. Full virtualization can be either achieved by means of: *binary translation* or *hardware acceleration*. In the following two paragraphs we will discuss both approaches.

Binary Translation. The main objective of full virtualization with binary translation to is leverages the host OS I/O device support while providing close to native CPU performance by executing as many CPU instructions on bare hardware as possible. The architecture of this virtualization technique is shown in Figure 2.4. When the virtualization solution is installed it first loads a driver into the host OS kernel. This driver

is required by the user space (i.e. privilege level 3) application to gain control over the physical hardware when needed. For example, when the user space application is used to start a guest OS it contacts the driver to reconfigure the host OS in order to start the guest OS kernel in privilege level 1. Moreover, the driver hooks a VMM below the host OS kernel in order to trap privileged instructions (e.g. network device access) issued by the guest OS's. Finally, the VMM integrates binary translation to detect non-virtualizable instructions at run-time and replace them with VM-safe code. This is achieved by examining the guest OS kernel binary stream using binary translation.

The main benefit of full virtualization with binary translation is that no modifications to the host and guest OS kernel need to be done. Indeed, the guest OS kernel drivers leverage the original host OS kernel driver interfaces. However, the flexibility comes at the cost of decreased performance due to the need to perform binary translation and emulation of privileged CPU instructions. Probably the most popular full virtualization solutions supporting binary translation are Microsoft Virtual PC [39], Parallels Workstation / Desktop [47], VMware Workstation/Fusion/Player [62, 59, 60], and VirtualBox [234]. Note that in contrast to emulation which emulates the entire hardware in software (e.g. each CPU instruction), full virtualization takes advantage of the physical hardware to run the guest OS's. For example, guest OS user processes execute CPU instructions directly on the physical CPU. On the other hand, privilege instructions are emulated by the VMs among other hardware components (e.g. Graphics Processing Unit). Ultimately, fully virtualization with binary translation can be seen as a hybrid between emulation and virtualization.

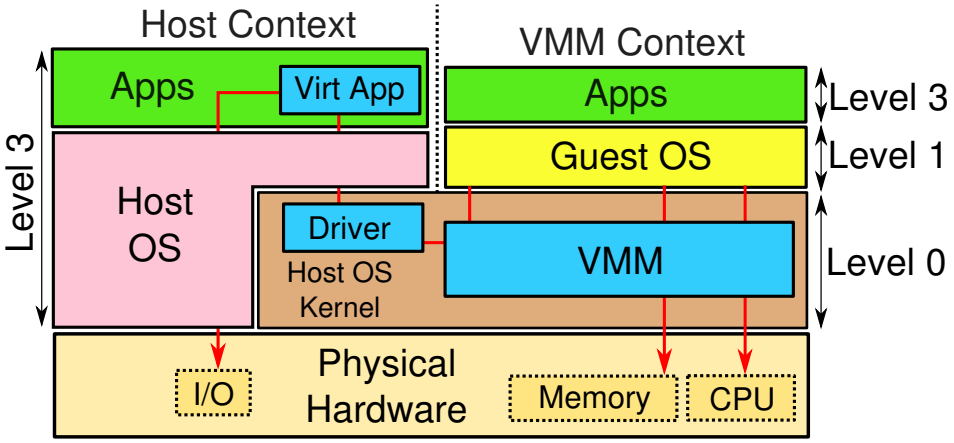


Figure 2.4: Full virtualization with binary translation example

Hardware Acceleration. When full virtualization with hardware acceleration is used, the Intel VT-x (resp. AMD-V) technology is leveraged to enable physical hardware virtualization without the binary translation overheads. For example, Intel-VT-x introduces two new CPU modes, namely: root and non-root. Each of the modes provides its own four privilege levels. The root mode is equivalent to the well-known x86 privilege level mechanism. On the other hand, the non-root mode introduces a new structure called Virtual Machine Control Structure (VMCS) which allows to provide fine grained control over the CPU instructions. The non-root mode therefore has more privileges than the root mode. It can be seen as one more privilege level, referred to as -1. In

this new system the guest OS kernel runs in the non-root mode on the privilege level 0, while the VMM runs in the root mode on the privilege level 0 (see Figure 2.5). This allows the guest OS kernel to operate unmodified without the need for binary translation for non-virtualizable instructions. The context switch between the root to non-root modes is called VMEntry (resp. VMExit). VMEntry typically happens when the guest OS attempts to run a non-privileged instruction. In that case the information (e.g. instruction name, exit reason) describing the root of the problem is saved in the VMCS structure. This information is used by the VMM in the root mode to resolve the issue. The main drawback of full virtualization with hardware acceleration is that it requires hardware support which is not available in many of the older servers. Moreover, context switches between the root and non-root modes can be expensive. Most of the modern VMMs such as Kernel-based Virtual Machine (KVM) [193], Microsoft Hyper-V [38], VMware ESX/ESXi [292], Xen Hardware Virtual Machine (HVM), VMware Workstation/Fusion/Player, VirtualBox, and Parallels Workstation / Desktop support this technique.

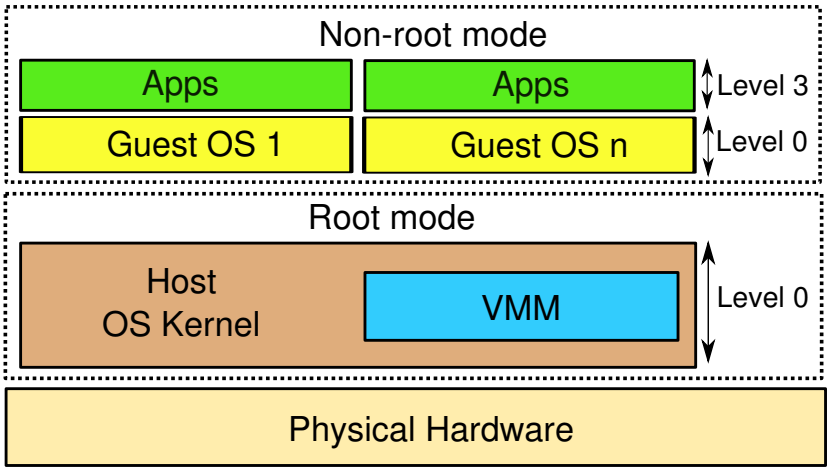


Figure 2.5: Full virtualization with hardware acceleration example

2.1.3.2 Paravirtualization

In contrast to full virtualization which does not require any guest OS kernel modifications, paravirtualization involve running a lightweight kernel, the so-called VMM on top of the bare hardware. This eliminates the need of a host OS and thus increases the guest OS performance as the guest OS's can now almost naively interact with the bare hardware. However, this comes at the cost of flexibility as paravirtualization requires the guest OS kernel to be modified to support the VMM. The most prominent paravirtualization solution is called Xen Paravirtualization (PV) [80]. Its high-level architecture overview is shown in Figure 2.6. In Xen, the VMM runs in the highest privilege level (i.e. 0). Note, that in Xen the VMM is really a lightweight kernel and thus does not implement any complex management decisions (e.g. admission control, inter-VM CPU scheduling decisions). Consequently, it only exports a low-level control interface which can be used to enforce such decisions. In order to provide a management layer, a *control VM*, also known as Dom0 (Domain 0) is

started as the first guest OS in the privilege level 1 during the VMM boot process. It is the job of the Dom0 to implement the appropriate policies and use the VMM control interface to enforce them. Dom0 is also used to perform other management decisions such as controlling the guest OS life-cycle (e.g. boot, reboot, shutdown) and physical memory allocations. As the VMM occupies the highest privilege level, guest OS's are unable to run without further modifications (see Section 2.1.2). To solve this issue, Xen moves the guest OS's to privilege level 1 (same as Dom0) and requires guest OS's kernels to be modified in order to enable the delegation of all the privileged (including non-virtualizable) instructions (e.g. creation of page tables) to the VMM using Xen specific software interrupts, also known as *hypercalls*. The VMM traps the hypercalls and executes the privileged instructions on behalf of the guests OS's either by translating them into the native hardware instructions or using emulation. Despite its near native performance, the requirement to patch the guest OS kernel limits the application of paravirtualized VMMs to either open-source OS's (e.g. Linux) or proprietary OS's which are tailored towards using a specific VMM interface. Other examples of paravirtualization-based VMM solutions using similar techniques are Microsoft Hyper-V and VMware ESX/ESXi.

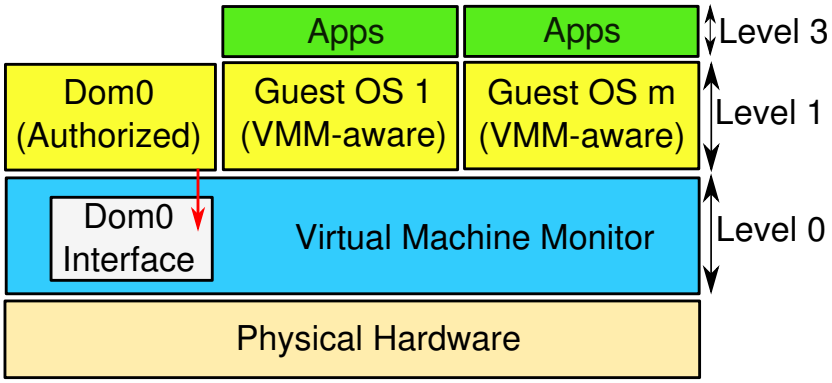


Figure 2.6: Paravirtualization example

2.1.3.3 OS-level Virtualization

OS-level virtualization creates multiple containers on the same OS which are managed by a *virtualization layer within the host OS kernel* (see Figure 2.7). Containers have their own resources (i.e. root file system, users, applications, networking settings, and firewall) which are isolated from each other by the host OS kernel. OS-level virtualization provides close to native performance. Containers provide near native performance to their applications and ease the system management. Moreover, they can be migrated in the same manner as traditional VMs (e.g. in KVM). However. the major drawback of OS-level virtualization is that the containers are limited to a single host OS kernel. For instance, no modifications to the kernel can be done (e.g. loading additional modules). Examples of well known OS-level virtualization solutions are OpenVZ [44] and Linux Containers [37].

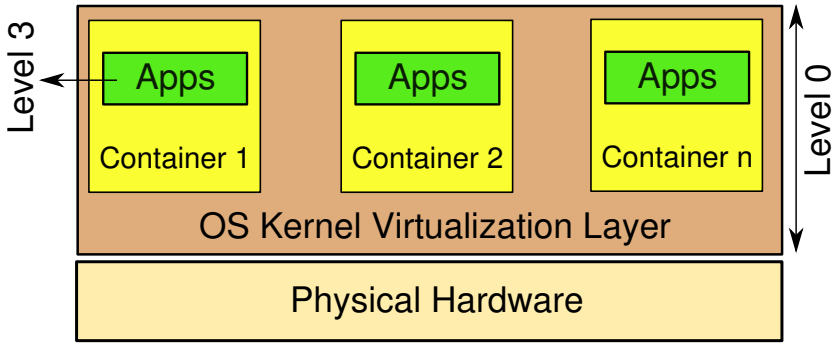


Figure 2.7: OS-level virtualization example

2.1.4 VM Live Migration

Among the traditional VM control operations (e.g. start, reboot, stop, suspend), most of the aforementioned virtualization techniques offer a feature known as *live migration*. VM live migration allows to seamlessly move VMs between PMs (source and destination) either over local or wide area network (LAN resp. WAN). Seamless live migration refers to a VM move which is either not noticeable or barely noticeable (i.e. in the order of milliseconds) to the VMs users. VM live migration can be either triggered manually by the system administrator or automatically by a cloud management system. In case it is triggered manually it is up to the system administrator to decide which VMs and to which PMs they have to be migrated. Otherwise, the cloud management system is in charge of taking live migration decision based on its high-level objectives. For example, a cloud management system could decide to consolidate multiple VMs on a fewer number of PMs for energy saving reasons. In that case it would automatically instruct the PMs to do the appropriate VM live migrations.

In order to achieve seamless live migration, VMs resources (i.e. CPU state, memory and disk content, networking connections) must be transparently moved from the source to the destination PM. Independent of the considered resources, VM live migration can be categorized into three approaches: *pre-copy*, *post-copy*, and a *hybrid* of both [110, 168]. In this section we focus on the VM memory content live migration approaches and assume a shared storage to be available in order to avoid VM disk migration (see Figure 2.8). Live migration without shared storage migration is still an ongoing research topic and is not fully supported by many of the open-source VMMs (e.g. KVM). Related work on VM disk and networking resources migration over both, LAN and WAN can be found in [94, 169, 230, 94, 121, 251].

Pre-copy Live Migration. In the pre-copy live migration approach the VMs memory pages are iteratively copied by the hypervisors over the network from the source to the destination PM while the VM is running. First all VM memory pages are transferred to the destination PM. Then, the hypervisor examines the VMs memory in order to detect pages which were modified during the previous copy round. In case modified pages were detected, they are copied to the destination PM in another copy round. This process (i.e. examine and copy) continues until no memory modifications can be observed. In that case the VM is stopped by the hypervisor on the source, its CPU state is transferred to the destination PM and the VM is resumed on the destination PM. One issue arises when memory modifications are performed faster than memory

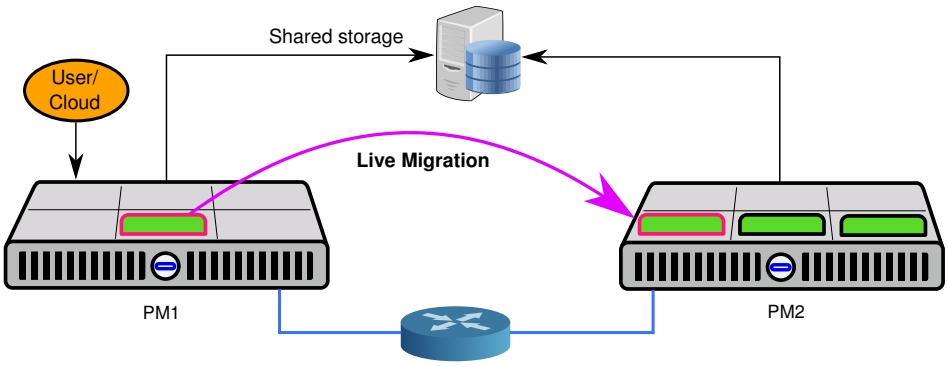


Figure 2.8: Live migration example

pages can be transferred over the network. In that case a situation happens when VM live migration can last forever (i.e. unable to converge). One solution to mitigate this issue is to terminate the migration only when a Writable Working Set (WWS) has been identified and transfer this set only. WWS is defined as a small set of pages which are constantly modified. If such a WWS exists it is a very good candidate to be transferred as it would incur a low service downtime. However, a WWS is not guaranteed to be identified [169]. Another, solution to the convergence issue is to put an upper bound on the number of copy rounds. In case the upper bound is reached the VM on the source is stopped by the hypervisor and its CPU state is transferred to the destination PM along with the modified memory pages. The VM on the destination is then resumed from the new state. The service downtime in this approach depends on a number of parameters such as the amount of modified memory pages to be transferred and network throughput. Both approaches are implemented in Xen hypervisor [110]. Finally, the convergence issue can be moved to the application level by letting the user or a management application observe the VM live migration duration and suspend the VM on the source PM when it reached a predefined timeout. In that case the hypervisor finishes the migration as no memory pages are modified anymore. This approach is hypervisor agnostic and is suggested in the KVM community [90].

Post-copy Live Migration. In the post-copy live migration [168] approach the VM is first suspended on the source PM. Then, its CPU state is transferred to the destination PM and the VM is resumed on the destination PM. As memory pages still reside on the source PM, page faults are generated on the destination PM when the VM attempts to access any not-yet available memory page. Each page fault is intercepted by the hypervisor which instructs the source PM to sent the page involved in the page fault to the destination PM. This way post-copy live migration guarantees that each page will be transferred to the destination PM at most once thus reducing the network overhead. The major drawback of post-copy live migration is the on-demand memory page transfer. During this time services inside the VM can experience serious performance degradation. The performance of post-copy live migration can be greatly improved by proactively pushing pages to the destination PM using adaptive pre-paging [168].

Hybrid Live Migration. The hybrid live migration [231] approach as its name implies brings together concepts of pre-copy and post-copy live migration. The key idea of hybrid live migration is to start with VM pre-copy live migration and switch to post-

copy directly after the first memory pre-copy iteration. First all the VM memory is copied to the destination PM by the source PM while the VM continues to run on the source PM. After the first copy phase the VM is suspended on the source PM and its CPU state including the modified pages is transferred to the destination PM. Then, the VM is resumed on the destination PM and the post-copy mechanism is enabled to push memory pages on-demand (i.e. when a page fault occurs) from the source to destination PM. For the time beeing, no known implementation and evaluation of this approach for VMs exists. Some work has been started by the authors in [168].

In this thesis we leverage the pre-copy VM live migration mechanism of the KVM hypervisor and implement the previously introduced application-level VM live migration convergence enforcement approach.

2.2 Autonomic Computing

This section gives a brief introduction to autonomic computing. We start our discussion with a history of autonomic computing. Then, the self-management properties of autonomic systems are reviewed. Afterwards, the concept of autonomic manager is introduced, a core architectural component which implements the self-management properties. Finally, selected autonomic computing systems are reviewed.

2.2.1 What is Autonomic Computing?

Autonomic computing [171] was first introduced by IBM in 2001 as a vision of computing environments which can automatically observe and adapt themselves according to high-level objectives (e.g. maximize profit). The driving motivation behind the autonomic computing initiative was the fact that, scale as well as the complexity of today's large-scale distributed systems makes it increasing hard to develop, deploy, configure, and maintain them even for the most experienced system administrators. The term autonomic computing is inspired from the human nervous system which is capable of autonomously observe and adapt the human body to its environment without the need for us to concentrate on it. For example, it autonomously controls our heart rate, body temperature, and blood sugar level. Similarly, self-managing autonomic computing systems are envisioned to allow users to focus on *what* instead of *how* something is to be done.

2.2.2 Self-Management Properties

According to [191] the building block of any autonomic system is self-management. Self-management is the ability of an autonomic system to automatically adapt to changes in its environment without the need of an human intervention. For example, a self-managed autonomic system is able to update itself and transparently handle failures in its Managed Elements (MEs) [2]. In order to achieve self-management, closed control loops are used. They are implemented by Autonomic Managers (AMs) which are used to manage one or multiple

2. A managed element can be any hardware (e.g. compute/storage server, router) or software (e.g. operating system, database, web server) resource [177].

MEs. Control loops are organized in categories based on their responsibilities. Particularly, the authors in [177] identify the following four control loop categories: self-configuration, self-healing, self-optimization and self-protection. They are visualized in Figure 2.9 and reviewed in this section.

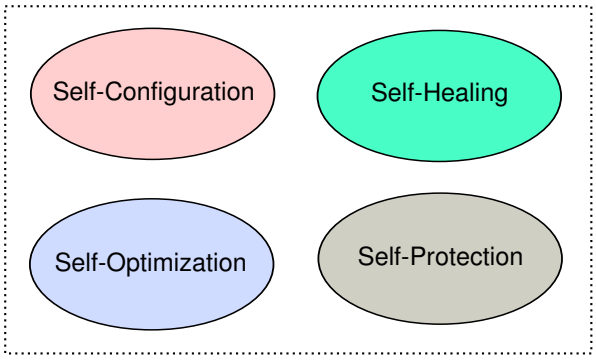


Figure 2.9: Self-management properties overview

Self-Configuration. Large-scale computing infrastructures now require substantial amount of IT experts (e.g. developers, administrators) to be configured, integrated, and maintained. Self-configuration refers to the ability of a system component to seamless integrate into the system. This involves two steps: (1) gathering of information about the system to be joined; (2) registering itself with the new system. For example, in a traditional large-scale web cluster farm, for scalability reasons requests are load balanced across multiple backend web servers. Registering additional backend servers typically requires modifications to the load balancer in order to make it aware of the new backend servers. A self-configuration enabled backend web server has the capability to automatically learn about the load balancer and register with it.

Self-Optimization. One of the core issues during the deployment of large-scale systems (e.g. databases, web servers) is the tuning of their performance parameters. For example, setting a certain cache size might be a good idea during the initial system deployment. However, at scale the cache size needs to be typically adjusted in order to keep pace with increasing load. In this context self-optimization refers to the ability of a system to monitor itself, learn from the past experience, and automatically adjust the systems parameters in order to satisfy the high-level performance goals (e.g. minimum energy consumption).

Self-Healing. With the probability for software and hardware failures increasing at scale, entire departments of developers and system administrators start investing a large amount of time in the debugging and fixing of problems. For example, complex software systems such as the ones involved in the management of entire companies (e.g. SAP) are composed of many inter-dependent services (e.g. database, printing, billing) which can be programmed and managed by different developer teams within a company. Fixing even the smallest bugs or failures in such complex ecosystems of services can be a very challenging task. On the other hand, hardware (e.g. HDDs) fails constantly and requires system administrators a substantial amount of work to replace it and reconfigure the system accordingly. Self-healing refers to the ability of the system to automatically detect, analyze, and resolve internal problems. For example, in

the most basic case the system could automatically detect a software bug, download a patch and apply it.

Self-Protection. Security management is crucial in today's large-scale systems in order to protect them and their users against malicious activities (e.g. intrusion attempts, code injection). Many technologies (e.g. SSL, RSA, firewalls) have been developed in the past to enable secure user authentication, data transfer, and network communication. However, the implementation, integration, and maintenance of those technologies in large computing infrastructures is still a complicated and error prone task which requires a substantial amount of highly educated IT experts. Self-protection refers to the ability of a system to proactively detect malicious activities from its past experience and automatically enforce the appropriate protection mechanisms.

2.2.3 Autonomic Managers

Autonomic Managers (AMs) [177] are software agents which implement the previously introduced self-management properties. They are in charge of managing one or multiple MEs. AMs can be either embedded into the MEs or run externally. In order for an AM to manage its MEs it first must be able to collect and store their monitoring information. The monitoring information gathering is supported by sensors which are exported by the MEs. Once gathered, monitoring information is stored in a knowledge base. It is then analyzed in order to decide whether actions need to be taken or not. In case actions need to be taken a plan must be created, which will generate a set of desired changes to the MEs. Therefore, a problem specific planning policy is used by the AM. Finally, the plan must be executed by issuing management commands to the MEs. In order to accept management commands from AMs, MEs must expose the appropriate actuator interfaces which will enforce the requested changes. To summarize, while the implementation of an AM is certainly control-loop category dependent and problem specific, all AMs share the following five functional components: *Monitor, Analyze, Plan, Execute, and Knowledge* (see Figure 2.10). Together they form the so-called MAPE-K control loop [173].

To ease the AM understanding we finish this section with a brief example of an AM, able to self-optimize the server power consumption. The key idea behind our AM is to slow down the CPU of a server once the servers power consumption exceeds a system administrator predefined power threshold. Slowing down the CPU inevitably reduces the overall server power usage. In our example the AM runs as a daemon on the server it manages. The AM monitoring module periodically observes the servers power consumption by querying the appropriate power sensor through the server OS sensors Application Programming Interface (API). The AM stores the sensor information in a knowledge base implemented as a local in-memory repository and instructs the analyze module to determine whether an action (i.e. slowdown CPU) needs to be taken. The analyze module estimates the current server power consumption by taking the average of the 20 most recent power values as received from the knowledge base. The estimated power value is compared with the threshold. In case the threshold is exceeded the planning module is called to derive by *how much the CPU must be slown down in order to bring the server power consumption under the threshold* according to a planning policy. For example, a naive policy could be used which would slow down the CPU by a constant factor (e.g. 1000 MHz). Finally, the planning module instructs the execution module to call the OS actuator API which will enforce the CPU slowdown.

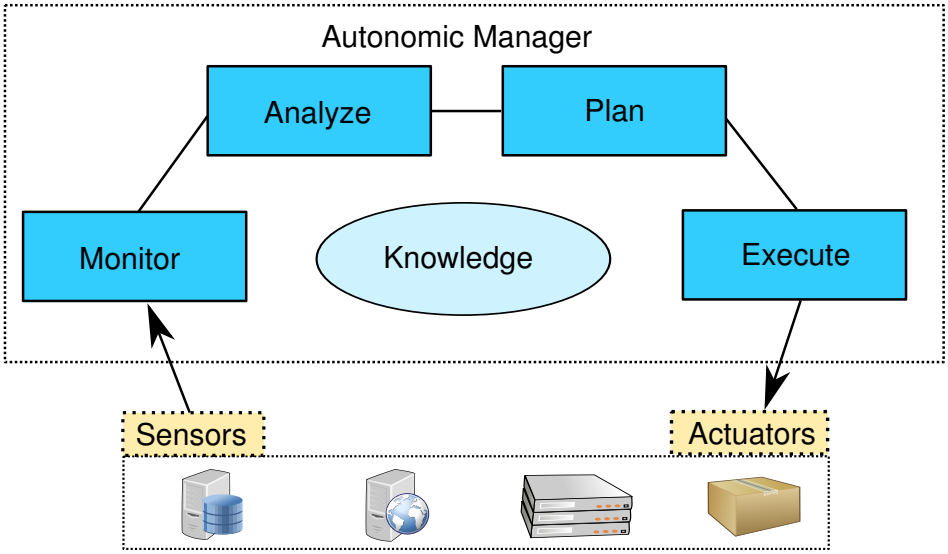


Figure 2.10: Autonomic manager example

2.2.4 Autonomic Computing Systems

Since the introduction of autonomic computing much research has been done to design and implement autonomic computing systems. Autonomic computing systems are software infrastructures following the vision of autonomic computing. This section briefly reviews some of these works. In [238], the authors introduce Hasthi, a generic framework for managing large-scale distributed systems according to user-defined management logic. Hasthi is designed around a self-configuring and healing hierarchical architecture in which one coordinator oversees multiple managers. Each manager is in charge of managing a subset of resources. Through simulations the authors show that their system is able to provide self-management for up to 100,000 resources. In [299] the authors present the design and implementation of a decentralized autonomic system for processing user requests and managing application execution in the context of the In-VIGO grid-computing system [68]. The proposed system follows the previously introduced principles of autonomic computing and integrates multiple AMs which each AM beeing in charge of managing a number of MEs (i.e. applications and resources). When an AM joins an administrative domain of the grid computing system it contacts its central repository and receives contact information about a subset of the other AMs in this domain, the so-called neighbourhood. Monitoring information is exchanged between the AMs in the same neighbourhood. Each time AMs leave and join new domains they propagate their previously collected monitoring information to the new neighbours thus constructing a global view of the system. The authors evaluate the efficiency, scalability, and robustness of the system on 10 PMs by comparing it with the previously developed centralized version of the system. They conclude that the new system is scalable and robust. The major drawback of the system is that at the end each AM requires to have a global system knowledge thus reducing its scalability. In [274], the authors present Unity, a self-optimizing multi-agent system for dynamic management of compute resources between different Application Environments (AEs) based on utility functions. Each AE is associated with a service-level utility function. AEs send their resource-level utility functions

to a single resource arbiter. Based on this functions the arbiter decides on the AEs resource allocations. Small scale experimental results show that utility functions are viable for enabling systems self-management. Finally, in [93], the authors present JADE, a general purpose architecture-based autonomic system for managing distributed systems. The key idea of JADE is to provide self-management properties to any loosely-coupled legacy system (e.g. web services). In order to achieve this JADE introduces the concept of *wrappers*. A wrapper is an abstraction providing a uniform interface to manage any legacy system component (e.g. tomcat server). Wrappers are collocated with the legacy components. More, one *node component* exist per PM. Node component is a software agent which knows all the wrappers available on a PM and exports interfaces in order to gather sensor information from the wrapped legacy components. Finally, multiple autonomic managers exist which implement control-loops and manage one or multiple node components. The management involves performing autonomic manager specific self-management tasks (e.g. self-optimization). JADE is experimentally evaluated using a three tier web architecture made of an web server, application server, and a database. It is shown to provide self-healing and self-protection properties for the evaluated scenario.

2.3 Cloud Computing

This section briefly introduces cloud computing, a computing paradigm which borrows ideas from the vision of self-managing autonomic computing systems and complements autonomic systems with a business model which enables to rent resources on-demand. First, the basic principles behind cloud computing are defined. Afterwards, the cloud characteristics, service models, and deployment models are presented. Finally, existing attempts to design and implement IaaS cloud management systems are reviewed.

2.3.1 What is Cloud Computing?

The cloud computing paradigm evolved as the result of the massive adaption of the Internet as well as major advances in the areas of virtualization (e.g. server, storage, network), grid computing, utility computing, and autonomic computing. Cloud computing borrows some concepts from autonomic computing in the sense that cloud providers implement autonomic managers in order to automate the management of their systems and provide on-demand services. However, both of them have different origins and goals. While autonomic computing was an IBM initiative aiming at reducing the management complexity of large-scale distributed systems, the primary goal of cloud computing is to reduce the costs of managing own infrastructures [305]. Many cloud computing definition have been proposed over the past years [286]. However, as of today still no standard definition exists. In this work we rely on the definition presented in [221], where Peter Mell and Tim Grance define cloud computing as:

> a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

2.3.2 Characteristics

According to [221] the five main cloud characteristics are: *broad network access, on-demand self-service, resource pooling, rapid elasticity, and measured service*.

Broad network access. Cloud providers services are available over the network (e.g. Internet) and thus can be accessed from any networked device (e.g. laptop, cell phone, desktop computer, server).

On-demand self-service. Customers can rent resources without the need of personal negotiation with the cloud providers. This can be achieved by the use of cloud providers APIs accessed over the network.

Resource pooling. Cloud resources are transparently provisioned by the cloud provider and hosted on its infrastructure which is commonly shared between multiple customers. Thereby, the internal structure of the cloud providers infrastructure is unknown to the customer. Consequently, customer are not aware where the provisioned resources are exactly running (e.g. on which rack of the cloud data center). For instance, when a customer rents compute capacity in the form of VMs he might know in which country his VMs are running. However, the knowledge on which rack yet PM the VMs are hosted is typically not exposed to the customers.

Rapid elasticity. Customers can automatically provision and release resources (e.g. compute or storage capacity) whenever required. For instance, the cloud provider by utilizing the cloud providers APIs resources can be requested in order to deal sudden resource utilization spikes.

Measured service. Cloud providers monitor the customers resource usage and charge customers for the used resources based on a selected business model (e.g. pay-as-you-go).

2.3.3 Service Models

The cloud computing stack categorizes services based on the following three types of service models [78]: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). In addition to the three well known cloud service models, Hardware-as-a-Service (HaaS) is probably the oldest service model which already existed long before cloud computing arrived. Figure 2.11 visualizes the introduced service models.

HaaS. HaaS offering allows customers to *lease* hardware resources (e.g. compute, storage, laptops, screens, desktops) on-demand. This model is particularly interesting either for private or business customers which do not want to invest in their own hardware.

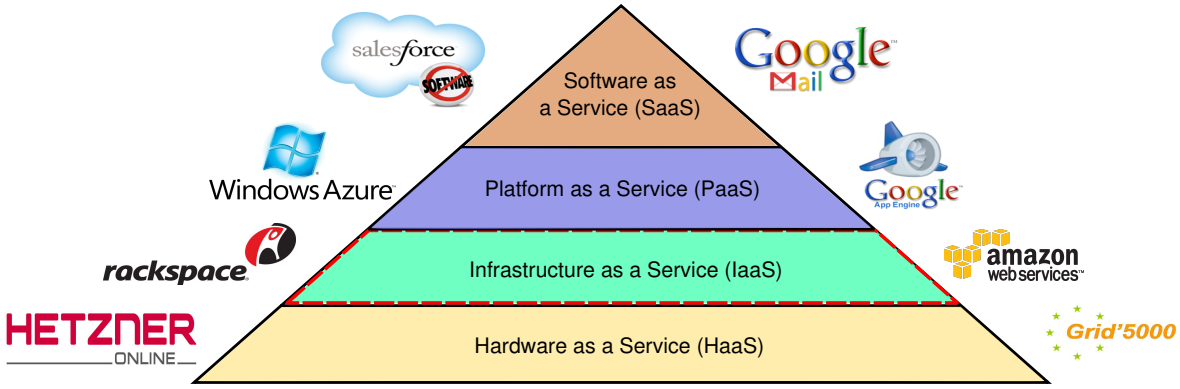


Figure 2.11: Cloud service models

For instance, in the most basic example a business customer can simply lease entire office computer equipment (i.e. desktop computers, screens) from a HaaS provider. This es especially beneficial for start-ups which do not want to make huge up-front investments in hardware. One example of an HaaS provider allowing to lease office computer equipment for small and medium-sized companies is GRENKE [26]. On the other hand, HaaS providers exist which allow to lease dedicated servers. Examples of such providers are Hetzner [30], LeaseWeb [36], and OVH [45]. Finally, several scientific experimentation testbeds have been established over the past years which allow researches to request hardware resources once experiments need to be performed. Some prominent examples of HaaS scientific experimentation testbed providers are Grid'5000 [97] in France and the National Energy Research Scientific Computing Center [42] in US.

IaaS. IaaS clouds allow customers to lease and manage virtual resources (e.g. server, storage, networks) over the Internet. The customers are provided total delegation over the resources thus being able to install, configure, and operate own software (e.g. OS, applications) without the need to worry about the underlying cloud computing system. Indeed, the customers are not given any control of the cloud computing system. For example, customers can easily provision servers in the form of VMs without the need to worry on which PMs they are running. Some well known public IaaS cloud computing systems include Amazon Elastic Compute Cloud (EC2) [71], Amazon S3 [11], Google Compute Engine [23], and Rackspace [50]. Moreover, a number of open-source IaaS cloud management systems have been developed over the last years to facilitate the creation of private clouds (see Section 2.3.4). They include CloudStack [275], Eucalyptus [19], Nimbus [190], OpenNebula [223], and OpenStack [279].

PaaS. PaaS clouds provide a cloud computing system for the deployment of applications (e.g. servlets, web services) developed using programming languages and libraries supported by the cloud provider. This allows customers to focus on application development by releasing them from the burden of deploying, managing and scaling own run-time environments (e.g. application servers). Prominent commercial PaaS offers include Google App Engine [20], RedHat OpenShift [51], and Windows Azure [64] which provide scalable environments for the development and deployment of web applications. Note, that PaaS offers are not necessarily limited to web applications.

For instance, Amazon Elastic MapReduce [10] allows customers to efficiently process a large amount of data. Similarly to IaaS clouds a number of open-source PaaS projects have been started such as AppScale [109], ConPaaS [241], Cloud Foundry [17], and Cloudify [18].

SaaS clouds. In contrast to PaaS clouds which allow customers to deploy custom applications, SaaS clouds typically provide a set of already hosted business applications (e.g. accounting, customer relationship management). Applications are managed by the cloud providers computing system on behalf of the customer. Applications can be accessed over the Internet by either using a web browser or another software capable of accessing their API. Some examples of public SaaS offers are Google Apps (e.g. mail, sheets, calender) [21], iCloud [32], and Salesforce [52].

2.3.4 Deployment Models

In [78] the authors distinguish between the following four cloud deployment models: private, public, community, and hybrid clouds (see Figure 2.12).

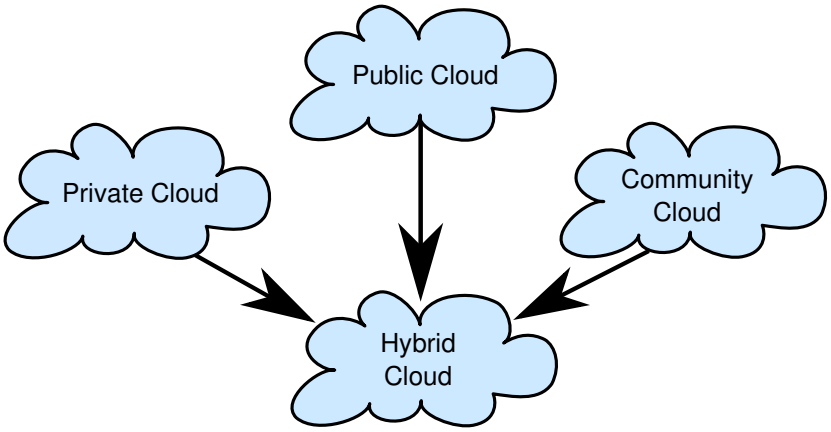


Figure 2.12: Cloud deployment models

Private clouds. Private clouds are cloud computing systems which are deployed on compute and storage infrastructures belonging to a single institution data center and network. Their usage is typically restricted to the scope of the institution. Private clouds are either managed by the institutions own IT department or a external IT provider (e.g. IBM). Private clouds can be based either on cloud computing systems developed in-house or third party commercial (e.g. VMware vCloud [61]) and open-source solutions (e.g. OpenStack, CloudStack, Eucalyptus).

Public clouds. Public clouds are commercial cloud computing systems available to everyone on the Internet. They are typically operated by a public cloud provider and allow customers (either individuals or institutions) to easily provision services (e.g. VMs) without the need to operate their own infrastructure. Thereby, customers are charged only for what they use. Examples of public cloud providers include Amazon Web Services [72], Rackspace [50], Google Cloud Platform [22], and Microsoft Azure [64].

Community clouds. Community clouds are cloud computing systems which allow infrastructure (e.g. compute, storage) sharing among different individuals or institutions with common interests. In contrast to public clouds, the access to community clouds is typically limited to the community members only. For example, a health care community cloud could be used by hospitals to exchange patients medical information. Moreover, community clouds can be also established between scientific institutions to collaborate on projects, share data, and exchange latest research results.

Hybrid clouds. Hybrid clouds are cloud computing systems which allow institutions to leverage infrastructures from private, public, and community clouds. For example, a hybrid cloud computing system enables intuition to offload less sensitive data into the public cloud while preserving sensitive data on its private cloud. Alternatively, hybrid clouds allow institutions to use their own infrastructure during periods of low service (e.g. web) load and scale their services during periods of high load by accessing public clouds.

The contributions presented in this thesis focus at cloud computing systems implementing the *IaaS cloud service model*. While we do not restrict ourselves to any particular deployment model, the primary targets of our work are *private clouds*.

2.3.5 IaaS Cloud Computing Systems

A lot of work has been done in the past on the design and implementation of IaaS cloud computing systems in order to facilitate the creation of private clouds. Such systems aim at providing users with VM execution environments while relieving them from the burden of manually managing those systems yet knowing where individual VMs are running. Thereby, virtualization technologies such as Xen, KVM, or VMware ESX/ESXi serve as building block to enable server virtualization and thus efficient data center resource utilization. Nevertheless, to enable the creation of VM execution environment IaaS cloud management systems are faced with a number of challenges such as: *(1) Scalability; (2) Autonomy; (3) VM life-cycle, storage, and network management; (4) Interoperability*.

Scalability is particularly important for IaaS cloud computing systems in order to enable the management of a large number of PMs, VMs, and users. Autonomy allows IaaS cloud computing systems to provide *self-configuration, optimization, healing, and protection*. Particularly, configuring an IaaS cloud computing system can require a substantial amount of highly skilled IT experts. Self-configuration refers to the ability of an IaaS cloud computing system to configure itself with minimal human intervention. On the other hand, IaaS cloud data centers now can host many thousands of servers and VMs. This vast amount of resources needs to be managed efficiently in order to reduce the costs (e.g. energy) and ease the system management. Self-optimization enables IaaS cloud computing systems to provide efficient resource management using algorithms and mechanisms able to dynamically reconfigure the IaaS cloud computing systems according to the given high-level objectives (e.g. energy management). Moreover, given that the probability for hardware and software failures increases at scale, IaaS cloud computing systems must automatically detect system component (e.g. compute nodes) failures and take the appropriate actions (e.g. inform the management components about failures, perform recovery) in order to enable continues operation. Self-healing refers to the ability of an IaaS cloud management system to automatically perform

these tasks without human intervention. Besides beeing able to self-heal, security is a crucial aspect in order to provide authenticated access to the IaaS cloud computing system, enable user data isolation, and protect the system against malicious activities (e.g. code injection). Self-protection refers to the ability of a IaaS cloud computing system to provide the security mechanisms and automatically protect itself against malicious activities.

Finally, VM life-cycle, storage, and networking management are three fundamental blocks of any IaaS cloud management system. VM life-cycle management enables users to control (i.e. boot, reboot, suspend, shutdown) their VMs. VM storage management provides a common repository for users to store and instruct the system to use VM disk images during VM deployment. Moreover, storage management implements mechanisms to enable efficient VM disk image propagation to the compute nodes. This is particularly important in large IaaS cloud computing where VM disk image need to be made available on hundreds of compute nodes. Several technologies such as SCP Tsunami [53] and TakTuk [111] have been developed over the past years which can be leveraged by the storage management mechanism to support efficient VM disk image propagation. Last but not least, interoperability is an important aspects in IaaS clouds which aims at allowing users to seamlessly transition between different IaaS cloud providers and reuse their tools. In order to interoperability IaaS cloud computing systems must expose standard interfaces. Several efforts have been made over the past years to design such interfaces. For instance, EC2 [9] and Open Cloud Computing Interface (OCCI) [43].

This section reviews the current efforts on the design and implementation of private IaaS cloud computing systems with respect to their scalability, autonomy, and interoperability. In the following discussion we distinguish between four types of IaaS cloud computing systems: *centralized, hierarchical, fully decentralized*.

2.3.5.1 Centralized Systems

In this section we review some of the recently proposed centralized IaaS cloud management systems. In [223], the authors introduce the OpenNebula IaaS cloud computing system. OpenNebula architecture follows the traditional frontend/backend model where an agent (e.g. cloud controller) runs on the frontend node, accepts users VM life-cycle requests and delegates them to the backend nodes. Each backend node runs an agent (e.g. node controller) which receives requests from the frontend node and enforces them by interacting with the hypervisor (see Figure 2.13). Moreover, the agent on the backend reports VM CPU and memory utilization to the frontend. A similar system can be found in [190], where the authors introduce the Nimbus IaaS cloud computing system. Neither OpenNebula nor Nimbus implement any autonomy features. With respect to interoperability, Nimbus provides the EC2 interface while OpenNebula supports both EC2 and OCCI interfaces. In [275], CloudStack is presented, a centralized system which allows to create and manage VMs. In contrast to OpenNebula and Nimbus, CloudStack supports the so-called multi-node configuration. In the multi-node configuration multiple frontend nodes/management servers can be used to avoid Single Point of Failure (SPOF). However, CloudStack does not integrate any mechanisms to load balance VMs between the management servers nor handle automatic fail-over of the management servers. Particularly, it is up to the user to decide which management server to contact for VM submission and the system administrator to implement management server fail-over mechanisms using tools such as Pacemaker [46],

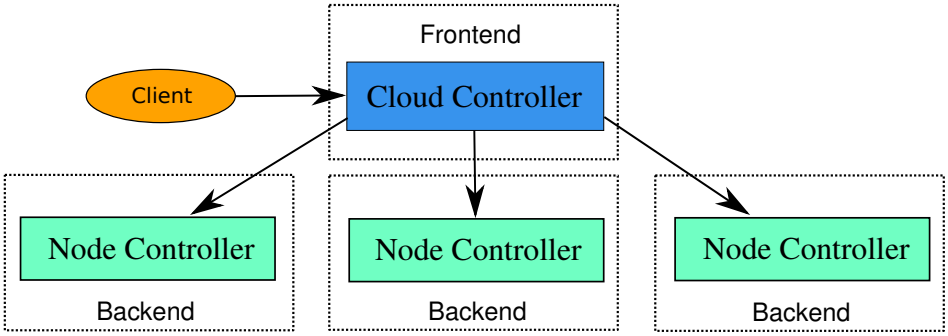


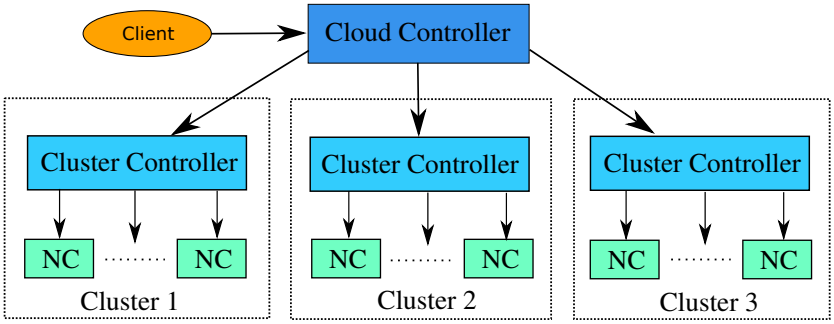
Figure 2.13: Centralized architecture example

Heartbeat [29], and Corosync [56]. In addition, as the management servers share a common MySQL database, MySQL own high availability (HA) solution (e.g. MySQL replication [3]) must be configured to provide database fault-tolerance. In other words, CloudStack does not provide *integrated self-healing mechanisms*. Moreover, it also lacks self-configuration, optimization, and protection features. Regarding its interoperability, CloudStack implements the EC2 interface. In [279], the OpenStack an open-source IaaS cloud computing system is presented. OpenStack services (e.g. database) are designed to support HA. However, similarly to CloudStack it is up to the system administrator to setup them in a fault tolerant manner using the appropriate third party tools. Consequently, no integrated self-healing mechanisms exists. OpenStack does not provide self-configuration, optimization, and protection features. For interoperability OpenStack implements the EC2 and OCCI interfaces. In [167], Entropy is presented. In contrast to the previously introduced systems Entropy supports self-optimization via for energy conservation. No self-configuration, healing, or protection mechanisms are provided. Entropy integrates its own user interface using the btrScript language [222]. Two similar system can be bound in [287, 122], where the authors introduce pMapper (resp. vGreen). Both systems focus on self-optimization and do not provide standard interfaces.

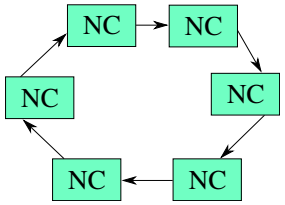
Finally, in [289], the design and implementation of the VMware Distributed Resource Scheduler (DRS) is discussed, a commercial VM management system. VMware DRS provides self-optimization. Moreover, in its commercial version HA features are available. VMware DRS does not provide any self-configuration and protection mechanisms. With respect to its interoperability VMware DRS implements its own interface called vCloud API. Given the centralized nature of the presented systems they share common drawbacks: SPOF and/or limited scalability. For example, in [157] VMware DRS is shown to have a limited scalability for approximately up to 32 PMs and 3000 VMs.

2.3.5.2 Hierarchical Systems

One way to improve the system scalability is to utilize a hierarchical architecture. Only a few works have investigated the use of hierarchical architectures for VM management. In [233] the authors introduce Eucalyptus. Eucalyptus is based on static hierarchical architecture. It is composed of three software components: node, cluster, and cloud controllers (see Figure 2.14(a)). The Node Controller (NC) runs on each PM. NCs interact with the hypervisor in order to discover the available PM capacity (e.g. CPU, memory), control VMs (e.g.



(a) Hierarchical architecture example



(b) Ring architecture example

start, stop), and learn about the VM status (e.g. running, terminated) on behalf of the Cluster Controller (CC). The CC runs on the frontend PM of a cluster. It places VMs on the NCs a simple round-robin algorithm. VM placement is based on the PM information collected from the NCs. The Cloud Controller (CLC) manages the CCs and implements the web services-based user interface. The CLC also interacts with the CCs in order to support monitoring information retrieval and resource allocation/deallocation enforcement. In its enterprise version Eucalyptus provides HA features which rely on replicated software components. For instance, in order to achieve CLC fail-over a redundant CLC must be installed which will monitor the primary CLC and take over in case of its failure. No evaluation of the HA mechanisms is publicly available. Finally, Eucalyptus does not support self-optimization, configuration, and protection. For interoperability Eucalyptus implements the EC2 interface. In [185], the authors introduce Mistral. Mistral is made of multiple controllers, the so-called Mistral controllers which each controller managing a subset of PMs. The authors argue that Mistral can be organized in a hierarchical manner to manage allow the management of large-scale systems. However, only small scale experiments on 8 PMs and 20 VMs are conducted to demonstrate the viability of the system. Indeed, no evaluation targeting its scalability is presented. Finally, the system is limited to self-optimization and does not implement any of the standard interfaces. No source code is publicly available.

2.3.5.3 Fully Decentralized Systems

Recently several research attempts have been made to design fully decentralized IaaS cloud computing systems. In [247], the authors introduce Distributed VM Scheduler (DVMS), a fully decentralized VM manager. PMs are organized in a ring (e.g. Chord [272]) with each PM being controlled by a software agent. For the sake of consistency with the previous graphs in this work we refer to this software agent as NC (see Figure 2.14(b)) DVMS targets self-optimization and has been validated by means of simulation only. Another system based on a ring architecture is proposed in [258]. Similarly to DVMS it focuses on self-optimization and has been evaluated by simulation only.

In [216], the authors introduce V-MAN, a fully decentralized VM management system based on an unstructured P2P network of PMs. Unlike, the previously introduced works V-MAN is built on top of peer sampling service [183] which periodically constructs randomized system topologies in which each PM only knows a subset of other PMs, the so-called neighbourhood. VM management is applied only within the scope of the neighbourhoods.

V-MAN is limited to self-optimization. Moreover, it has been validated by simulation only.

2.3.5.4 Summary

Table 2.3.5.4 summarizes the results from our study. Particularly, it presents the systems, their architectures, autonomy features, user interfaces, and implementation status. As it can be observed, despite the ambitious vision of autonomic computing, some of its concepts such as self-configuration, healing, and protection still did not find their way into today's cloud computing systems while others (i.e. self-optimization) are now slowly starting to get adapted. Moreover, only a few systems implementing the self-management properties are publicly available.

System	Architecture	Autonomy	User Interface	Implementation
OpenNebula [223]	Centralized	None	EC2, OCCI	Open source
OpenStack [279]	Centralized	None	EC2, OCCI	Open source
Nimbus [190]	Centralized	None	EC2	Open source
CloudStack [275]	Centralized	None	EC2	Open source
VMware DRS [289]	Centralized	Optimization, Healing	vCloud API	Closed source
Entropy [167]	Centralized	Optimization	btrScript	Open source
pMapper [287]	Centralized	Optimization	-	Simulation
vGreen [122]	Centralized	Optimization	-	Closed source
Rouzaud-Cornabas [257]	Structured P2P	Optimization	-	Simulation
DVMS [247]	Structured P2P	Optimization	-	Simulation
V-MAN [216]	Unstructured P2P	Optimization	-	Simulation
Mistral [185]	Static Hierarchy	Optimization	-	Closed source
Eucalyptus [233]	Static Hierarchy	Healing	EC2	Open source

Table 2.1: Comparison of the IaaS cloud computing systems

2.4 Energy Management in Computing Clusters

We now review the related work on energy management in computing clusters. Energy management in computing clusters can be achieved either by means of static or dynamic power management (SPM resp. DPM) [104]. SPM, sometimes also referred to as low-power computing is applied at design time of a system. For instance, by improving the CPU microarchitecture and/or using low-power CPUs. Most recent examples of systems following this approach are the BlueGene/Q [105] supercomputers which are among the most energy efficient computing systems available today [57]. On the other hand, DPM techniques are used to save system power at *run-time*. This is typically achieved by leveraging low-power states available on modern server components (e.g. CPUs). The contributions presented in this thesis belong to the category of DPM. Consequently, we concentrate our discussion on related DPM techniques.

This section is organizes as follows. First, the terminology is introduced. Then, well-known server power measurement techniques are reviewed. The ability to measure the server power consumption is the first step to identify the most power demanding server components. After identifying the power measurement methods we present a typical server consumption breakdown by its components and review traditional DPM techniques at

server and cluster-level in non-virtualized environments. Server-level techniques typically target power savings on an individual server, while cluster-level approaches focus on DPM across multiple servers. Finally, DPM techniques in virtualized environments are presented. Note, that while the traditional DPM available in non-virtualized environments were not explicitly evaluated in a virtualized environment they could be used to complement the DPM techniques in virtualized environments.

2.4.1 Terminology

We now introduce the terminology used in this section. Particularly, we define the two fundamental terms, namely power and energy. Beeing able to distinguish between the two terms is essential in order to understand the ultimate objectives and differences of (resp. between) power management and energy management mechanisms in computing clusters.

Electrical power is defined as the rate at which electrical energy is transferred by a circuit. It is measured in *Watt* or *Joules per second*. Electrical power is computed by multiplying the Current (I) with Voltage (V) (see Eq. 2.1).

$$P = I \times V \tag{2.1}$$

Current represents the amount of electrical charge (i.e. number of coulombs) flowing over the wire per second, referred to as Amperes (Amps). Voltage represents the change in electrical potential energy per unit of charge on the wire. It is measured in joules per coulomb. There is a direct relationship between I and V in the sense that, the greater the voltage the more current will flow. On the other hand, energy is a quantity typically measured in Watt-seconds (Ws). It is defined as power consumed over a period of time (see Eq. 2.2).

$$E(T) = \int_0^T P(t)\,\mathrm{d}t. \tag{2.2}$$

Given that power is an instant value while energy is the integral of power over a period of time, computing clusters can either implement power management or energy management mechanisms. The primary objective of power management is the ability to *cap the system power usage at any discrete point in time*. This is typically achieved by defining a power usage upper bound and integrating mechanisms able to enforce this upper bound by either slowing down or turning off system components or the entire system. The power capping ability allows data center providers to deploy more servers at a given data center power budget (e.g. 1 MW) without risking to exceed the data center infrastructure (e.g. power distribution, cooling) capabilities during periods of high server utilization. Note, that power efficient systems can significant degrade application performance to achieve their goal. For instance, an abrupt CPU slowdown on the servers could result in a significant performance (i.e. execution time) degradation of running applications [283] thus increasing the overall energy consumption. On the other hand, the primary objective of energy management technique is to *reduce the energy consumed*. Consequently, they aim at reducing the total power consumed over a period of time without significant applications performance degradation. In order to achieve this they often rely on low-level mechanisms (e.g. turning off servers) involved in power management. While the definition of power management and energy management appears clear in theory, in practise the distinction between the mechanisms is often blurry as both terms are used interchangeably. The works presented in this section can be used for both, power and energy management.

2.4.2 Power Measurement Techniques

This section reviews the state of the art in data center power consumption measurement. Beeing able to measure the power consumption is the first step towards designing new energy conservation techniques. We distinguish between power measurement techniques at three levels: data center, server, and OS level. Data center power measurement techniques allow to account for the data center IT infrastructure power consumption usage and efficiency. Server level power measurements provide power usage information of the entire server and its components. OS-level measurements provide the power usage at the level of individual processes running on a server. Independent of the level power usage can be accounted by means of direct, indirect, or hybrid power measurements. Direct power measurements are typically performed using hardware which is either embedded into the equipment (e.g. server, servers components) or externally attached. On the other hand, indirect power measurements are performed by estimating the power usage using power models. Indirect power measurements are especially beneficial due to their ease of integration (i.e. no additional hardware is required). Finally, hybrid power measurements combine both, indirect and direct power measurements.

2.4.2.1 Data center level

One major contributor to today's data center power consumption is their physical infrastructure (e.g. power and cooling equipment) which is used to support the IT equipment (e.g. compute, storage, network). Studies have shown that physical infrastructure alone can amount to more then 50% of the total data center power usage [128] (see Figure 2.14).

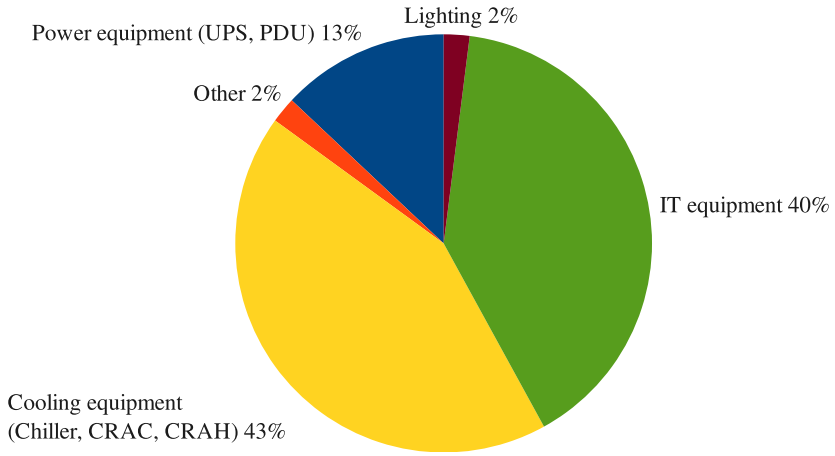


Figure 2.14: Data center power consumption breakdown [126]

As it can observed, while power efficiency optimization of the IT equipment are certainly important, physical infrastructure is a very good candidate for power usage optimization (e.g. cooling equipment update). However, the first step towards beeing able to optimize the physical infrastructures power usage and compare it with other data centers is the ability to *measure its power efficiency*. In 2007, The Green Grid [156] proposed two metrics to account for the physical infrastructure power efficiency, namely Power Usage Effectiveness (PUE) and its reciprocal Data Center infrastructure Efficiency (DCiE) (see Eq. 2.3 resp. Eq. 2.4).

$$PUE := \frac{\text{Total DC power}}{\text{IT equipment power}} \quad (2.3) \qquad DCiE := \frac{\text{IT equipment power}}{\text{Total DC power}} \times 100 \quad (2.4)$$

PUE is defined as the ratio of the total power entering the data center to the power used by the IT equipment. In the ideal world a PUE of 1 is desirable. This would imply that all power which goes into the data center is consumed by its IT equipment. Obviously, the reality looks different as some power is required to support data centers physical infrastructure. Moreover, the actual PUE heavily depends on the current IT infrastructure load and physical infrastructure conditions [228]. For example, when the IT infrastructure is fully utilized ($\sim$ 99%) it will typically imply a higher IT equipment power usage thus decreasing the data center PUE. This implies that instant PUE measures are not necessary the same as the ones over a period of time (e.g. daily, weekly, monthly, or yearly). Modern well-utilized data centers are able to achieve a PUE close to 1.12 (e.g. Google in the 2nd quarter of 2012 [24]).

DCiE can be used to capture what percentage of the power entering the data center was consumed by the IT equipment. It is is computed as the ratio of IT equipment power to the total data center power usage and the result is multiplied by 100 to get the percentage. For example, a PUE of 1 corresponds to a DCiE of 100% meaning that 100% of data center power usage was spent to power the IT equipment (a pure hypothetical example). To enable the computation of PUE and DCiE, power which is entering the data center and the one used by the IT infrastructure must be measured. Power used by the data center can be captured the utility meter. IT infrastructure power can be measured at the output of an Uninterruptible Power Supply (UPS). UPS provides backup power to the IT infrastructure during periods of power grid outages. It is plugged in between the power grid and the IT infrastructure.

2.4.2.2 Server level

Server level power measurements can be divided into two categories: (1) entire server; (2) server components. The former approaches target power measurements of the entire server while the letter aim at taking fine-grained power measurements of the individual server components. In the following two paragraph both methods are detailed.

Entire server. One way to measure the power consumption of the entire server is to use a metered Power Distribution Unit (PDU). Metered PDUs are used to power the servers in most of the modern data centers. They can be easily accessed using the Simple Network Management Protocol (SNMP) protocol [102]. In case a metered PDU is not available, power meters such as Watts up PRO [63] can be used. Finally, Advanced Configuration and Power Interface-enabled Power Supply Units (PSUs) can be leveraged to obtain the server power consumption. Alternatively, in case external hardware is not available, the Intelligent Platform Management Interface (IPMI) [179] can be used to access the power sensors available on modern servers. IPMI is a standard which is typically supported by a Baseboard Management Controller (BMC), a hardware chip which is embedded into most of the modern servers. BMC can be accessed either locally through the OS or remotely using the dedicated BMC network card. Note, that a BMC operates independently of the server OS and thus can be accessed despite of OS failures and without the need of the server to be powered on (i.e. connection to the

power grid is enough). Finally, server remote management cards such as Dell Remote Access Controller (DRAC) [218] can be used to obtain the power consumption data. Figure 2.15 provides a graphical overview of the introduced techniques.

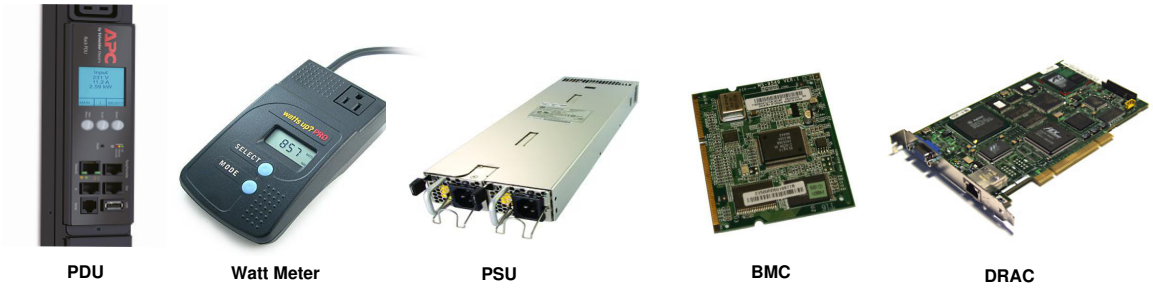


Figure 2.15: Examples of server-level power measurement techniques

Server components. Often measuring the power usage of the entire server is not accurate enough. Indeed, in order to get a better understanding where most of the power in a server is spent, a power consumption breakdown by server components is required. One naive approach to obtain the server components power usage is to look at the components data sheets. However, data sheets only provide information regarding the components designed peak power usage. Moreover, data sheets neither account for the correlation between server utilization and power consumption nor the average power consumption. Another, more accurate approach is to plugin a digital meter between the servers Direct Current (DC) lines and the server components [146]. However, it is not always possible to separate all the components as many of them share the same power plane (e.g. some multicore processors and memory chips). Separating DC lines is at most appropriate in a research environment.

As we have observed, the lack of integrated power measurement hardware can make the server components power usage measurement a complicated task. Another, more promising power measurement approach is instead of relying on specific hardware, to take indirect power measurements. Particularly, most of the m modern CPUs include additional registers, the so-called hardware performance counters that can be used to monitor various low-level system events (e.g. TLB misses, cache hits). These events are exported by the OS and can be leveraged by software (i.e. applications, libraries) which integrates power models to estimate the CPU and even the entire server power consumption [263]. Moreover, the actual servers components utilization can be used as input for power models [130]. Estimations based on performance counters and other metrics (e.g. utilization) do not require additional power measurement hardware. However, their accuracy heavily depends on the underlying power model and the choice of its parameters (e.g. counters, weights) [114].

After being able to measure the server and its components power consumption, metrics and benchmarks are needed to compare the energy efficiency of different servers. One metric which has gained a lot of attention during the last years is *FLOPS (Floating Point Operations Per Second) per watt*. FLOPS is particularly a good measure for scientific applications which generate many floating-point numbers. Consequently, the metric itself is particularly interesting in the area of supercomputing. For example, it is used by The Green500 List [138] to

rank the most energy efficient supercomputers. Probably the most known benchmark today implementing the FLOPS per watt metric is SPECpower [200], an industrial effort to develop a benchmark able to evaluate the performance and energy efficiency of server. Another interesting benchmark is JouleSort [252].

2.4.2.3 OS level

While the power measurements at the server level help to understand the servers hardware power consumption, they are unable to account for the application [3] power usage. Understanding the application power usage is crucial in order to facilitate application power efficiency improvements and enable power-aware application scheduling. Power measurements at the OS-level are either based on indirect power measurements using power models or rely on hybrid approaches thus requiring a combination of indirect and direct measurements. In [124], the authors introduce pTop, a process-level profiling tool. pTop is implemented as a service at the kernel-level and provides application power usage estimations based on power models which rely on the applications resource (e.g. CPU, memory) utilization. pTop does not require additional hardware. A similar work can be found in [232], where the authors introduce PowerAPI, an application power consumption profiling library. In contrast to pTop, PowerAPI is more modular and implemented in user-space. Closely related works based on indirect application power measurements include PowerTop [48], JouleMeter [187], and Intel Energy Checker SDK [34]. Considering hybrid approaches, in [141] PowerScope is proposed, a tool supporting applications energy consumption profiling. In contrast to the previously introduced works, PowerScope requires additional hardware instrumentation. It is therefore more complicated and less flexible than approaches which are based on power models. Nevertheless, thanks to hardware instrumentation, hybrid approaches are able to learn an accurate server power model and thus provide more accurate predictions. For example, JouleMeter is known to output better results once complemented with an external power meter.

2.4.3 Energy Management in Non-Virtualized Environments

We now discuss the DPM techniques available in non-virtualized environments. First, we investigate on which hardware subsystem (e.g. CPU, memory) most of the power savings can be achieved. Being able to understand where most of the power is spent is crucial in order to design energy saving techniques yielding the most energy savings. Then, we introduce DPM approaches available at the server and cluster-level. DPM techniques at the server-level can be divided into two categories: *fine-grained and course-grained*. Fine-grained approaches focus at either the server components (e.g. CPU, memory, disk, Network Interface Card (NIC)) or software (e.g. file system, compiler). DPM techniques targeting the server components either slow down (i.e. do less work) or turn off (e.g. shutdown, suspend) the server hardware. On the other hand, DPM targeting the software attempt to perform optimizations at the software level. For instance, by integrating energy-aware data layout policies in the file systems or optimizing the compilers to perform less CPU instructions. Finally, course-grained server-level DPM approaches attempt to design servers able to en-

3. We define an application as a set of one or multiple processes.

tirely and rapidly transition between active and low-power states thus avoiding the need of fine-grained DPM techniques.

2.4.3.1 Power Breakdown by Hardware Subsystem

Figure 2.16 visualizes the peak power usage breakdown by hardware subsystem of a Google data center (∼ 2007).

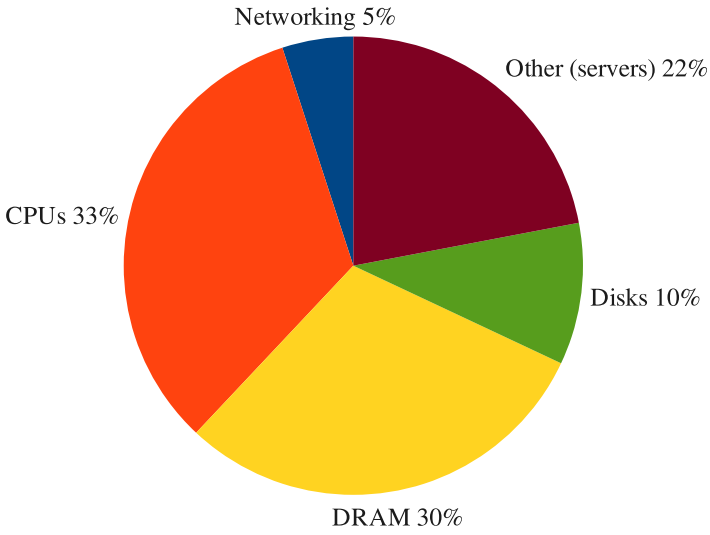


Figure 2.16: Google data center peek power breakdown by hardware subsystem [170]

As it can be observed CPU (33%) and memory (30%) require substantial amount of power and this numbers are most likely to increase in the future. For instance, studies have shown that in order to power a 2 terabyte in-memory database up to 57% of server power is required in a server which integrates 128 DDR3 DIMMS [303]. Moreover, given the current trend of Big Data and the associated need to process large amount of data while achieving high throughput, the need to save disk power consumption will become more and more important. Particularly, storage requirements are now taking new dimensions. Yahoo! which is a major search engine and internet services provider as of today alone accommodates over 170 petabyte of storage [189]. Amazon S3 [11] which is one of the major cloud storage service providers hosted over 762 billion objects and processed over 500 000 requests per second on those objects during peek periods of load at the end of 2011 [88]. While the Amazon power consumption statistics are not publicly available it is clear that hosting such large-scale storage infrastructures requires tremendous amounts of power. Finally, while not explicitly mentioned on the Figure 2.16, Power Supply Units (PSUs) also contribute to the overall server power consumption. Especially low-end PSUs consume a significant amount of the total server power due to their low Power Efficiency (PE) of approximability 70-75% [280]. On the other hand, high-end PSUs can achieve power efficiency which goes beyond 90% [176]. The reason for the PE variations lies in the power losses experienced by the PSUs while converting high-voltage Alternating Current (AC) from the power grid to a low-voltage DC as required by the servers components. Particularly, in order to switch from higher AC voltage (e.g. 220 V) to lower DC voltage (e.g. 12 V) additional circuits are required inside the

PSU. These circuits convert the voltage and inevitably loose some energy in the form of heat, which then needs to be dissipated by additional fans inside PSU. PE is defined as the ratio between the DC output to the AC input power. Consequently, for a 90% power efficient labeled PSU to produce 400 W DC output, approximability 444 W AC input are required, resulting in 44 W beeing lost in the form of hear. Note, that the PE of a PSU heavily depends on the power load imposed on the PSU, number of circuits, and other conditions (e.g. temperature). Indeed, a PSU which is labeled as 90% efficient is not necessarily that efficient at all power loads. For instance, when operated at 50-60% power load a PSU could achieve a PE of 83.9%. However, when operated at power loads below 30% the PE could degrade to 69% and even more [280] depending on the PSU. For the best PE it is therefore important to choose PSUs according to the expected load. To reward highly power-efficient PSUs the 80 PLUS Certificate [7] was established which is handed out to PSUs with a minimum power efficiency of 80% at 20%, 50% and 100% of their rated load.

We can conclude that it is clear that focusing on CPU and memory as the only subsystems for power-optimizations is not enough anymore. Indeed, a holistic power saving approach covering all subsystems is desirable in order to achieve the maximum energy savings.

2.4.3.2 Fine-grained server-level DPM

This section reviews the fine-grained server-level DPM techniques with an emphasis on the server hardware. Particularly, we focus on DPM mechanisms available at the CPU, memory, disks, and the NIC as they can be leveraged by the contributions presented in this thesis. Note, that the discussion of server-level DPM approaches targeting the software (e.g. file systems, compilers) is out of the scope of this work.

CPU. We start our discussion with the power saving techniques available for the CPU. They can be divided into three categories: *Dynamic Voltage and Frequency Scaling (DVFS), Core Off/On, and Turbo Boost*.

DVFS. DVFS is probably the most known CPU DPM techniques available since the early days of mobile devices (e.g. smart phones, tablets, laptops), desktops, and high-end servers. The key idea of DVFS is to reduce the CPU frequency and voltage during periods of low utilization. Reducing frequency and voltage inevitably yields to decreased power consumption due to the nature of today's Digital CMOS circuits (e.g. CPUs) [239]. Particularly, CMOS circuits power consumption is composed of two parts: *static and dynamic*. Static part is mostly the result of leakage current. It can be improved during the CMOS circuit design. On the other hand, dynamic part dominates the CMOS circuit power consumption as every charge and discharge of its components (e.g. gates) requires additional power. The dynamic power consumption can be approximated by the following equation [181]:

$$P = C \times f \times V^2 \tag{2.5}$$

where C is the switching capacitance, f the switching frequency, and V the supply voltage. According to this equation a linear reduction in voltage yields quadratic power savings. However, decreasing the voltage also lowers the transistors switching speed which results in a reduced maximum CPU frequency. As a result, to guarantee proper

CPU functionality the CPU frequency must be lowered to the reduced frequency. Lowering the frequency increases the application execution time which has a negative impact on the energy consumption. Moreover, additional energy is required to rise the frequency and voltage level back when required (e.g. during increased utilization). To support different frequency/voltage settings CPUs expose a set of performance and operating states (P-states resp. C-states) to the OS. P-states are the CPUs supported frequency/voltage pairs when the CPU is turned on. Higher P-states lead to lower power usage. For example, P3 requires less power than P0. C-states allow to turn on/off internal CPU components. For example, a CPU in C-state 0 is considered as turned on (all internal components are active) while a CPU in a higher C-state (e.g. 3) could have all its internal clocks turned off. Note, that DVFS operates on P-states only and requires the CPU to be in C0 state. Given the possibly negative effects of frequency/voltage reduction on application performance (i.e. executing time) it is clear that deciding on the appropriate P-state is crucial to achieve both, good performance and decreased energy consumption. Consequently, a lot of research has been done over the past years to design and evaluate DVFS algorithms able to determine the appropriate CPU frequency based on the applications CPU utilization. DVFS algorithms can be divided into two categories: interval-based and task-based [213]. Interval-based algorithms divide the application execution time in intervals and attempt to find the optimal CPU frequency for the upcoming interval based on CPU utilization from the past intervals. This is typically achieved by keeping track of the applications CPU utilization data and using prediction algorithms. Interval-based algorithms are simple, work transparently to the applications and can be implemented in a real system as they do not require future CPU utilization knowledge. Such algorithms were first studied in [293, 155]. On the other hand, task-based algorithms [300, 243] assume a system which is composed of a set of applications with known deadlines (e.g. execution time and required number of CPU cycles). Once the deadlines are known they attempt to set the CPU frequency just as high to meet the deadlines. Task-based algorithms are often more energy-efficient than interval-based algorithms as they can provide a better energy vs. performance trade-off [140]. However, their strong assumptions makes them hard to implement and limits their application scope. Consequently, most of today's OS's (Linux, Windows) implement interval-based DVFS algorithms. For instance, in Linux the *CPUfreq* [4] infrastructure implements interval-based DVFS algorithms at the kernel level using different governors. Recently, in [203] the authors evaluate the effectiveness of interval-based DVFS algorithms on multiple generations of AMD Opteron CPUs using the SPEC CPU2000 benchmark suite [164]. The results show that while interval-based DVFS algorithms can yield energy savings when used on older CPUs, *energy consumption is increased* when modern CPUs are used *even with memory-bound workloads*. The authors conclude that saturation of CPU frequencies, large static power, small dynamic power ranges, and improved sleep-states will further lower the benefits of DVFS in the future.

Core Off/On. Given the limitations of DVFS, in [207] the authors propose Per-Core Power Gating (PCPG). The key idea of PCPG is to allow the deactivation of individual cores by cutting down their voltage supply during periods of low utilization. PCPG

4. CPUfreq - http://www.kernel.org/doc/Documentation/cpu-freq/

is shown to save 30% more energy than DVFS. Moreover a hybrid approach implementing both techniques resulted in approximability 60% energy savings. PCPG is now supported by Intel Nehalem microarchitecture based CPUs and can be used by the OS to turn off/on individual cores. In order to achieve this, CPUs provide an extended set of C-states. Particularly, a new Deep Power Down state known as C6 is introduced. When a core enters this state its cache is flushed and the core state is saved into the shared Last Level Cache (LLC). Power gates are used to completely shutdown the core. In Linux, the *CPUidle* infrastructure [235] and its governors are in charge of detecting idle cores and turning them off, similarly to what CPUfreq does for P-states. However, before, cores can be turned off, idle times need to be created. In [269], the authors show that in practise creating idle times on individual cores is hard to achieve due occurring interrupts and timers. Interrupt, timer, and process consolidation are used in the Linux kernel scheduler to maximize CPU cores sleep times once enabled.

Turbo Boost. Finally, another promising technique which can be used to save CPU energy is Intel Turbo Boost [178]. Turbo Boost is integrated in the recent Intel CPUs (e.g. Core i7). The key idea of Turbo Boost is to opportunistically increase the CPU frequency and voltage when the basic conditions such as power consumption, temperature, and current draw permit it. Turbo Boost is automatically activated by the CPU while operating at the lowest performance-state (i.e. P0). In order to achieve this a dedicated Power Control Unit (PCU) is embedded into the CPU. The PCU adjusts the frequency/voltage of the processor by monitoring the CPU power consumption, temperature, and current draw within a closed-loop feedback control. Thereby, given that the constraints are not exceeded the frequency of all the cores can be either increased or decreased in steps (e.g. 133 MHz for Nehalem architecture) when they are active. In the case when only one core is active, its frequency can be modified in 266 MHz steps. Turbo Boost can be seen as a method to overclock the CPU in a controlled manner. It enables to operate the CPU beyond the base operating frequency thus increase the performance for some applications (e.g. CPU-bound). Increasing the performance allows the applications to finish faster thus enabling the CPU to enter low-power sleep modes. A very similar technique is also provided by AMD and is called Turbo Core [12].

Memory. Memory (e.g. DRAM) power management can be achieved either by power-cycling (e.g. standby) memory chips or slowing them down. In [129], the authors propose an analytical model for idle time approximation of DRAM chips. They evaluate the model using a trace-driven simulation and conclude that the best approach is to transition the DRAM chips directly into a power-saving state instead of performing idle-time predictions. Close work can be found in [118], where the authors study techniques for detecting memory module idleness and perform memory power management actions (e.g. standby modules). In [204], the authors study the effects of different page allocation policies on the energy consumption. Simulation results show that power-aware page allocation can achieve substantial energy savings. In [123], the authors study dynamic memory power management approaches. Particularly, they propose a number of techniques which automatically adjust memory module power states depending on the load. Simulation results show that the proposed techniques can limit the memory power consumption without significant performance degradation thus able to save substantial amount of energy. More recently, several works at-

tempt to exploit techniques able to slowdown the memory modules instead of entering sleep states. In [114], the authors propose a novel method called Running Average Power Limit (RAPL) for enforcing memory power limits. The key idea of RAPL is to maintain an average memory power limit over a sliding time window. To enforce the limit, Memory Power Limiting (MPL) states are used. Each MPL state results in a different memory bandwidth and power consumption. Experimental results show that RAPL is able to enforce a memory power limits with minimal performance degradation. In [120], the authors present MemScale, a system which saves energy by applying DVFS on the memory controller and dynamic frequency scaling on the memory channels and DRAM modules. Simulation results show that MemScale yields significant energy savings.

Disk. One way save disk power is to spin down the device during periods of low utilization. Particularly, most of the available disks provide power modes such as: active, idle, sleeping. When a disk is active its disk platters are spinning and it can serve I/O requests. On the other hand, when a disk is idle, it is spinning but not serving requests. Finally, when a disk is sleeping the disk platters are not spinning thus the disk is unable to serve I/O requests. In [210], the authors study the costs of spinning down disks on portable computers and conclude that spinning down the disks can eliminate almost all the consumed energy. In [163], the authors study the problem of deciding when to spin down the disk. Particularly, they design a spin down prediction algorithm based on machine learning. Using simulations the authors show that the proposed algorithm can reduce the disk power consumption by half. In [214], the authors assume a disk which is either spinning or sleeping and propose an adaptive disk shutdown algorithm to save energy. The algorithm predicts batches of requests and performs disk shutdown between the batches. Simulation results show that the algorithm can save disk energy with limited performance degradation. In [161], the authors present Dynamic Rotations Per Minute (DRPM). The key idea of DRPM is to dynamically control the speed at which the disk rotates in order to provide fine-grained power control thus avoiding the need to completely spin down and up the disks. Simulation results have shown that this technique is especially beneficial with short idle times. A very similar approach can be found in [101] where the authors propose to use disks supporting two speeds. Emulation results show that 20 to 30% of energy can be conserved depending on the load and speed transition overheads. Finally, Solid State Disks (SSDs) can be used to replace traditional Hard Disk Drives (HDDs) in order to conserve energy. SSDs are much more performant and energy efficient then HDDs as they do not integrate any mechanical parts. Indeed, according to [236] the average power consumption of an SSD is 1.2 W only.

Networking Interface Card. In [227], the authors study the issue of power management in network equipment (e.g. routers, switches, network interface cards) and argue for two energy saving mechanisms: (1) Sleep states (e.g. shutdown) for network elements (e.g. links) during idle times (i.e. when no packets are processed); (2) Rate adaptation (i.e. slowdown) of network elements depending on the network load, an approach also known as Adaptive Link Rate (ALR) [158]. Simulation results using real-world traces of network topology and traffic data the authors show that both techniques can cut the energy consumption by half for underutilized networks (10-20%). Other works targeting ALR can be found in [160, 196, 159] where the authors study policies to de-

termine when to change the link rate and approaches to combine both, sleep states and ALR. Finally, in [69], the authors propose a system called Somniloguy. The key idea of Somniloguy is to augment the NIC of desktop computers with a second low-power microprocessor which is able to serve requests of Internet-enabled applications (e.g. BiTorrent) even when the host system is transitioned to a sleep state (e.g. shut-down). In order to achieve this an OS is embedded into the microprocessor which can run either entire or parts of applications (e.g. file download). In case when the host initiates a sleep request its network state as well as the state of the selected applications is synchronized with the microprocessor. Experimental results conducted in a real environment have shown significant energy savings ranging from 60% to 80%.

2.4.3.3 Course-grained server-level DPM techniques

While a lot of work has been done over the past years to design fine-grained server-level DPM approaches, not much research has been done on the design of course-grained server-level DPM techniques. Indeed, this is a challenging task as it typically involves novel server designs which are capable of rapidly transition between fully active and sleep states. One prominent example of such an approach can be found in [220], where the authors introduce PowerNap, a concept which aims at designing Blade servers able to *rapidly transition between full power active and ultra low-power nap state*. Particularly, when a server is in an active state (e.g. receives network packets) it is fully available at its highest performance. On the other hand, when a server enters the nap state it only activates as much logic as needed to allow a wake-up when new workload (e.g. network packets) arrives. This design greatly simplifies the energy management mechanisms of individual server components as they only require two states: active and nap. The authors show by simulations that servers designed following the PowerNap principle can reduce average power consumption by 74%. Finally, in [75], the authors argue that turning off entire servers can have a negative impact on the response times once they need to be woken up in order to handle traffic spikes. They propose, a new power-state, called *barely-alive*. Servers in such state have most of their components (e.g. all cores) turned off. However, their memory is still reachable using a remote interface and thus can be leveraged to perform cooperative caching. A middleware is designed to resize the cooperative cache just as much to guarantee Service-Level Agreements (SLAs). Preliminary results using a trace-driven simulation show that promising energy savings can be achieved.

2.4.3.4 Cluster-level DPM

In the previous section we have introduced the server-level DPM approches. Server-level DPM approaches are basic mechanisms which can be used to save energy on an individual server. However, before such mechanisms can be applied at cluster-level, first *idle times need to be created*. Indeed, servers are rarely fully idle thus unless idle times are created, significant performance degradation is to be expected for the hosted services when server-level DPM approaches are applied. This section reviews the cluster-level DPM approaches. Cluster-level DPM approaches aim at creating idle times necessary for the adaptation of server-level DPM mechanisms. In this section we discusses the energy saving efforts for clusters of web servers, batch systems, distributed file systems, and cluster computing frameworks (i.e. MPI and MapReduce).

Clusters of Web Servers Some work has been done to save energy in the context of clusters of web servers. In [127], the authors study ways to save energy on clusters of web servers. In this context they propose coordinated voltage scaling (CVS). The key idea of CVS is to keep the same average frequency on all the cluster nodes. In order to do so, a dedicated node is used to periodically collect the CPU frequencies of the cluster nodes, compute an average frequency and instruct the cluster nodes to adjust their own frequency to the newly computed one. In [103], the authors present Muse, an market-based approach for managing resources (e.g. compute, storage) in hosting centers. In Muse, customers *bid* for resources depending on the delivered performance. Through experimental results the authors show that such an dynamic environment can decrease the energy requirements by 29%. In [162], the authors have designed a web server cluster which distributes the user requests such that the power, energy, throughput, and latency are optimized. Particularly, the requests are distributed such that the maximum number of nodes can be turned off. Nodes are automatically turned on when more resources are needed. The authors show that their approaches requires 42% less energy than a traditional web server with only 0.35% loss in throughput. Finally, in [246] the authors propose a cluster configuration and load distribution algorithm which turns off and on servers based on the expected performance and power consumption. The algorithm is implemented at the application level in a web server as well as OS-level by modifying the Nomad [244] single system image OS. The results show that significant power and energy savings can be achieved.

Batch Systems Batch schedulers are job scheduling systems which are commonly used to manage the resources (i.e. compute and storage) of most of the worlds supercomputers. While historically power management was not the primary concern in such systems, it has gained a lot of attraction over the past years as supercomputers power requirements are now starting to hit the data centers power budget constraints. A few works have studied power management in batch systems. In [113], an energy-efficient framework for grids called GREEN-NET [25] is introduced. GREEN-NET extends the OAR [96] batch scheduler with a prediction module which makes use of the OAR future resource (e.g. physical machine) reservation agenda in order to predict when next reservations start. An advanced reservation is a certain amount of resources reserved over a period of time. This way the framework can avoid turning off servers which will be required in the near future. Moreover, in order to avoid frequent turning off and on of resources due to fragmentation's in the reservation agenda, the framework employs an reservation aggregation mechanisms. The key idea of this mechanism is to place reservations as close as possible to each other. The prediction part is now integrated in the production version of OAR which is deployed on the Grid'5000 experimentation testbed [98]. In [302], the Simple Linux Utility for Resource Management (SLURM) batch scheduler is introduced. SLURM integrates two power saving mechanisms: DVFS and node power down [54]. When node power down is used SLURM will power down the servers after a system administration configured idle time interval. In [149], authors introduce GreenSlot, a parallel batch job scheduler able to leverage green energy (i.e. solar). Particularly, the key idea of GreenSlot is to schedule the jobs such that the amount of green energy is maximized while meeting the job's deadlines as specified by the users. The authors have implemented GreenSlot as an extension of the SLURM batch scheduler and evaluated it using realistic applications.

The results show that GreenSlot can conserve up to 39% of energy and increase the green energy usage by 117%. Finally, in [182], the Maui scheduler algorithms are presented. Moab Energy-Aware Resource Management [40] is a commercial version of the Maui scheduler and integrates power management mechanisms such as workload consolidation and turning off idle servers.

Distributed File Systems Distributed File System (DFSs) such as Hadoop DFS (HDFS) [276] and General Parallel File System (GPFS) [259] have gained a lot of attraction over the past years and are now used to manage petabytes of data distributed across thousands of servers [189]. Some works have investigated the problem of power management in DFS. In [189], the authors propose GreenHDFS. The key idea of GreenHDFS is to split the physical nodes of the Hadoop DFS cluster into two logical zones: hot and cold. Data which is frequently accessed is moved to the hot zone while data with less frequent access patterns is moved to the cloud zone. This separation allows to transition physical nodes from the cold zone into a power saving state (e.g. suspend). Simulations using Yahoo! traces indicate 26% energy savings while performing cold zone energy management. A very similar idea can be found in [245] where the authors have studied energy management in disk array-based servers. Particularly, the main objective was to concentrate most frequently accessed data (e.g. files) on as few disks as possible in order create enough idle time for transitioning a large number of disks into a power saving state, a technique known as *Popular Data Concentration*. Through simulations the authors have shown that energy savings are only possible under very low load. In [206, 73], the authors investigate the energy-efficiency of Hadoop DFS and extend it with a new data layout and load balancing policy which distributes the data block replicas such that the maximum number of nodes can be transitioned into a low-power state while still guaranteeing data availability. This is achieved by maintaining one replica of each data block in a subset of nodes called *Covering Subset (CS)*. Such a CS has the nice property that it guarantees data availability even if all nodes which are not part of it are turned off. Experimental results show that CS can yield energy savings at the cost of decreased performance. The same idea has been adapted by the authors in [73]. In [74], the authors argue for a fine-grained power management approach. Particularly, the key idea is to *keep the disks always on* and instead power-cycle individual node components (e.g. slow down CPU, turn off individual cores and/or memory banks) depending on the current system load. Experimental results show that such an approach does not require significant changes in the DFS and still achieves power-efficiency.

Cluster Computing Frameworks Some work has been done on power management in cluster computing frameworks. In this document we review the power management research targeting two prominent parallel computing models: *MPI* and *MapReduce*.

MPI. Studies have shown that CPU is not always the primary bottleneck of scientific Message Passing Interface (MPI) application [145]. The key idea to save energy in such applications is to lower the CPU frequency during memory and/or network I/O intensive execution periods. For example, given that some phase of an application is memory bound it is possible to reduce the CPU frequency during the execution of this phase. Some research has been done in order to exploit this behaviour in order to conserve energy. In [144, 188, 267] the MPI application is profiled and divided into

phases. In each phase a different CPU frequency is set depending on its boundness. For example, during a memory-bound phase the CPU frequency is decreased. A similar work can be found in [172] where the authors propose a new DVFS algorithm with on the fly CPU-boundness detection and CPU frequency setting. In [256], the authors state that the problem of deciding when to change the CPU frequency is NP-hard. To determine how close existing heuristics are from the optimal solution, a Linear Programming (LP) approach is proposed to compute the bound on the achievable energy savings for MPI applications. Experimental results show that heuristics which utilize DVFS work well for certain applications while yield less energy savings for others.

MapReduce. MapReduce [117] has recently appeared as a promising parallel programming model which allows to efficiently process large amounts of data (e.g. sets of log files). The key idea behind the MapReduce programming model is to split the data to be processed into equally sized (e.g. 128MB) chunks and process them (e.g. filter data) simultaneously on clusters of commodity servers. Recently, several attempts have been made to improve the energy efficiency of the parallel data processing frameworks implementing the MapReduce model (e.g. Hadoop MapReduce [278]). In [107], the authors have studied the performance and energy efficiency of Hadoop MapReduce jobs and proposed quantitative models to facilitate the development and administration of Hadoop MapReduce clusters. Such models are required in order to answer questions such as how much energy a job consumes and how many nodes must be assigned to a MapReduce cluster to handle that job. In [199], the authors propose All-In Strategy (AIS), a new MapReduce cluster energy management approach. The key idea of AIS is to transition entire MapReduce cluster into a low-power sleep state when it is idle and turn it on only when new workload is to be processed. The authors compare their strategy with the CS approach presented in [206] and conclude that AIS is often the better choice. In [296], the authors study the MapReduce performance and energy efficiency in two cases: (1) varying number of worker nodes; (2) DVFS-enabled worker nodes to scale the frequency and voltage depending on the worker node load. Experiments conducted on eight power-aware nodes show that substantial energy savings are possible. However, the energy savings depend on the workload characteristics, number of worker nodes, and the selected DVFS policy. In [100], the authors investigate the energy-efficiency of MapReduce in a virtualization environment. Particularly, they propose a VM placement algorithm which aims at collocating MapReduce VMs with similar run-times as well as complementary resource (e.g. CPU) demands. Collocating MapReduce VMs with similar run-times allows to turn off servers directly once all VMs have finished executing. On the other hand, exploiting complementaries between the VM resource demands improves the server utilization and avoids VM performance problems. Indeed, VMs with complementary resource demands mitigate performance degradation by avoiding bottlenecks on shared server subsystems (e.g. CPU caches, memory busses). Simulation results show that the proposed VM placement algorithms perform 20-35% better than traditional ones (e.g. Random First-Fit) in terms of energy savings. In [151], the authors propose GreenHadoop, an extended version of the Hadoop MapReduce [278] for data centers with green energy (i.e. solar) available. GreenHadoop first estimates future green energy availability and jobs approximate energy requirements. The estimations are then used to guide MapRe-

duce job scheduling decisions. Particularly, the jobs are scheduled such that the use of green energy is maximized while still preserving job execution time boundaries. Jobs are either delayed until green energy is available or scheduled during periods of cheap brown energy by respecting their deadlines. Experimental results achieve substantial energy savings while maximizing the use of green energy as compared to native Hadoop MapReduce. A closely related work aiming at maximizing the use of green energy based on predictions in MapReduce clusters can be found in [70]. Finally, in [106], the authors propose Berkeley Energy Efficient MapReduce (BEEMR). BEEMR leverages the fact that interactive jobs often operate only on a subset of data. Consequently, they can be processed by a small set of servers while transitioning the remaining servers into a power saving state. On the other hand, less critical jobs can be queued and executed in batches on the remaining cluster servers using the AIS strategy. This work combines concepts introduced in [189, 199, 206]. Using real workload-traces from Facebook the authors show that BEERM can achieve between 40-50% energy savings.

2.4.4 Energy Management in Virtualized Environments

DPM techniques can be categorizes in two types: application-aware and application-agnostic. Application-aware approaches considered the applications high-level QoS requirements (e.g. response time) while performing the VM management tasks. Application-agnostic approaches consider VMs as black-boxes and thus perform VM management decisions solely based on VMs low-level QoS requirements (e.g. requested number of cores, RAM, memory, networking resources). Application-agnostic techniques are typically unable to provide high-level application QoS guarantees. However, especially in the context of IaaS clouds (e.g. Amazon EC2) this is not always desirable as the users applications are typically unknown. In this work we focus on the application-agnostic DPM techniques and briefly review the application-aware DPM approaches at the end of the section.

2.4.4.1 Problem Statement

Application-agnostic DPM in virtualized environments can be decomposed into three parts: VM placement, underload and overload management, VM consolidation, and power management. The former three parts integrate algorithms to favour the creation of idle PMs and resolve overload situations. The latter part is used to detect idle PMs and transition them into a low-power state. Idle PMs are PMs which do not accommodate any VMs for a predefined amount of time. As the major challenges appear in the former three algorithmic parts we focus our discussion on those.

The problem of VM placement arises when users of a cloud management system attempt to submit VMs. The cloud management system then must find the appropriate PMs to accommodate the VMs such that the number of used PMs is minimized. This is achieved by considering the VMs static resource requirements (e.g. number of VCORES, memory). Indeed, at the time of VM submission no historical VM resource utilization data is available. In a cloud management system which considers the actual VM resource utilization after the VM placement, PM underload and overload situations can occur due to inefficient VM resource utilization (resp. overcommitted PMs). For example, when a PM is underloaded it is beneficial to move all its VMs to other PMs in order to transition the underloaded PM into a

power-saving state (e.g. suspend). This is typically achieved by estimating the VMs resource utilization based on the previouvly collected VM monitoring data, finding PMs with enough capacity to accommodate the VMs, and finally moving the VMs (see Figure 2.17).

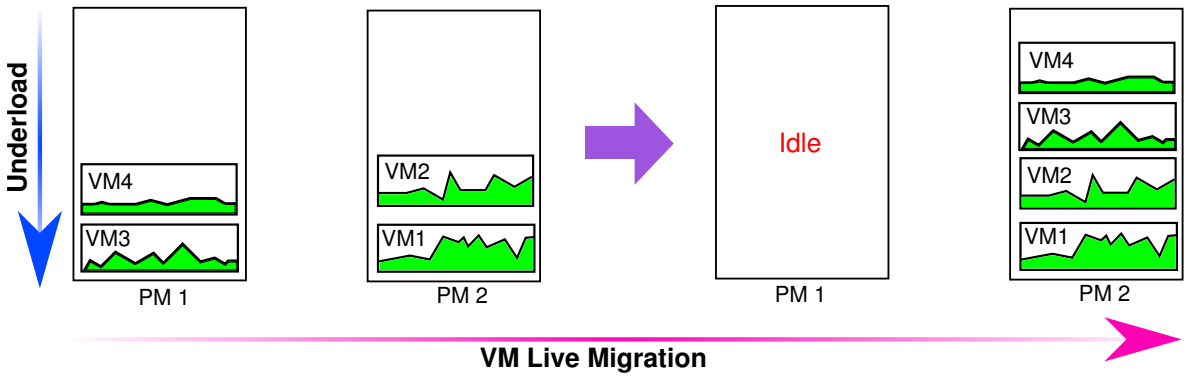


Figure 2.17: Underload mitigation and VM live migration example

However, as the result of VM movements, other PMs can become overcommited as they would be hosting more VMs than their physical resources can accommodate. This can result in an overload situation during periods of high resource utilization when the aggregated VM resource utilization exceeds the total PMs physical capacity. Consequently, overload management mechanisms are required in order to detect overload situations and decide which VMs and to which PMs they should be migrated from the overload PMs.

Complementary to the VM placement as well as the overload and underload management algorithms, VM consolidation can be used to periodically (e.g. daily, weekly) repack *already placed VMs* on the least number of PMs. This is especially useful in order to mitigate resource fragmentation on PMs which are neither underloaded nor overloaded. Resource fragmentation can prevent new VMs from beeing submitted to the system despite available capacity. For instance, in the most basic example on a system with two compute nodes A and B that are equally loaded to 60%, when a user submit a VM requesting 50% of capacity not enough resources are available place the VM on any node. However, given that one VM can be migrated from node B to A in order to decrease node B load to 50%, the users VM can be placed on node B. Moreover, repacking already placed VMs on the least number of PMs also facilitate the creation of idle times which are required in order to power down over-provisioned PMs.

In contrast to the VM placement problem which does not manipulate already placed VMs, VM consolidation requires to migrate existing VMs. Particularly, given that VMs are already assigned to PMs, a new solution (i.e. VM to PM assignment) must be computed which minimized the number of used PMs. Thereby, depending on the VM consolidation algorithm multiple solutions could exist which all *yield the same number of used PMs*. However, in order to arrive to the new solutions starting from the current VM to PM assignment, a *different number of migrations is required*. This adds another dimension to the problem as now the *number of migrations needs to be minimized* along with the number of PMs. Minimizing the number of migrations required to reach the newly computed consolidated state is mandatory as every useless VM live migration consumes additional resources (e.g. CPU capacity, network bandwidths) and yields unnecessary application performance degradation. Conse-

quently, VM consolidation algorithms must be designed to take the current VM placement into account while computing a solution. Depending on the system used, VM consolidation algorithms can be based on either *estimated VM resource utilization information* or on the *static VM resource requirements* as specified by the user during VM submission. Note, that depending on the cloud management system design, VM consolidation can be also used to resolve underload/overload situations by simply triggering it in the event of underload or overload situations. Obviously this would have a cost in terms of the number of migrations as potentially VMs from non underloaded/overloaded would need to be migrated.

In the following sections we present the related works targeting VM placement, underload & overload management, and VM consolidation. We classify the related works based on their provided properties and features. First, we present the *algorithm* used. Different types of algorithms can be utilized to solve the aforementioned problems. For instance, greedy, metaheuristic, or mathematical programming (i.e. linear programming and constraint programming) algorithms. Then, we study the considered resources. Many algorithms base their decisions either on a single VM resource (e.g. CPU) or multiple VM resources (e.g. CPU, memory, network). After knowing which resources are considered it is important to know how *the VM resource utilization* is taken into account. In other words, do the algorithms operate on *static* or *dynamic* VM resource demands. Another important aspect is *heterogeneity*. Particularly, PMs can be either homogeneous or heterogeneous in terms of their hardware. It is therefore important to know whether the proposed approaches target homogeneous or heterogeneous PMs. While the works presented in this section aim at facilitating the creation of idle times, ultimately *power management mechanisms* (e.g. DVFS, PM on/off/suspend) are required in order to save energy. Consequently, we indicate whether power management mechanisms are available or not in the presented works. Last but not least two aspects are important: *evaluation* and *workload*. Evaluation indicates whether experiments or simulations have been performed to evaluate the algorithms. Workload indicates whenever real applications, synthetic benchmarks, or application traces were used in the evaluation.

2.4.4.2 VM Placement

The VM Placement Problem (VMPP) can be reduced to an instance of the Multi-Dimensional Bin Packing Problem (MDBPP) [268] which is known to be NP-hard [284, 266]. When mapped to the MDBPP the PMs represent the bins and the VMs the items to be packed. Each PM has a predefined static total (e.g. CPU, memory, network) capacity vector and each VM is assigned with a static requested capacity vector. The goal of VM placement is to assigned the VMs to PMs such that the number of PMs is minimized. Thereby, PMs can be either empty or already have some VMs assigned. In this section, we first give a formal VMPP definition by presenting a Binary Integer Programming (BIP) model. Then, we review the available algorithms to solve the introduced model.

Formal Problem Definition We now present Binary Integer Programming (BIP) model for the VMPP problem. The BIP model introduced in this section assumes empty PMs. However, it can be generalized for the case of pre-filled PMs by simply assuming that PMs already have VMs assigned. Let P denote the set of PMs and V the set of VMs, with $n = |P|$ and $m = |V|$ representing the number of PMs and VMs. PMs are represented by d-dimensional total capacity vectors $\mathbf{TC_p} := \{TC_{p,k}\}_{1 \leq k \leq d}$. In this work

three dimensions are considered ($d = 3$). Each dimension k represents the PMs capacity of resource $R_k \in R$ with R being defined as $R := \{CPU, Memory, Network\}$. VMs are represented by static requested capacity vectors $\mathbf{RC_v} := \{RC_{v,k}\}_{1 \leq k \leq d}$ with each component specifying the requested VM capacity of resource R_k. Finally, in order to complete the BIP model, we define the following two decision variables:

1. PM allocation variable y_p, equals 1 if the PM p is chosen, and 0 otherwise.
2. VM allocation variable $x_{v,p}$, equals 1 if the VM v is assigned to the PM p, and 0 otherwise.

The objective is to place the submitted VMs such that, the number of PMs used is minimized. This is reflected in our objective function (2.6).

$$\textbf{Minimize } \mathbf{f}(\mathbf{P}) = \sum_{p=0}^{n-1} y_p \tag{2.6}$$

Subject to the following constraints:

$$\sum_{v=0}^{m-1} \mathbf{RC}_{v,k} x_{v,p} \leq \mathbf{TC_{p,k}} y_v, \forall p \in \{0, \ldots, n-1\}, k \in \{0, \ldots, d-1\} \tag{2.7}$$

$$\sum_{p=0}^{n-1} x_{v,p} = 1, \ \forall v \in \{0, \ldots, m-1\} \tag{2.8}$$

Constraint (2.7) ensures that the capacity of each PM is not exceeded and constraint (2.8) guarantees that each VM is assigned to exactly one PM.

Greedy algorithms. We now first discuss the traditional greedy algorithms to solve the VMPP. Greedy algorithms construct a solution step by step by taking *local best decision*. Thereby, already taking decisions are never reverted. Such algorithms are very good candidates for the VMPP due to their low-degree polynomial-time worst-case complexity and ease of implementation. However, because of the local decision taking procedure greedy algorithms do not necessarily yield global optimal solution (i.e. VM to PM assignment). Greedy algorithms for VMPP can be decomposed into online and offline algorithms. Online algorithms assign VMs to PMs as they arrive. In other words, they have no prior knowledge of all the VMs which will be submitted in the future. On the other hand, offline algorithms do have the knowledge about all the VMs to be assigned thus they are able to sort them beforehand. One well known online algorithm is First-Fit (FF). The offline version of it is called First-Fit Decreasing (FFD) [304]. In FF, VMs are assumed to arrive sequentially and are placed on the first PM which can accommodate them, starting from the first PM sorted according to a predefined metric (e.g. available resources, power efficiency). For example, when a cluster spans three PM having respectively 30, 50 and 40% of free CPU capacity and a new VM is to be placed which requires 20% of capacity, this VM is placed on the first PM. Afterwards, the algorithm will try to place subsequent VMs starting again from the first PM. In FFD, when a set of VMs is to be placed the algorithm is improved by presorting the VMs in decreasing order according to their resource demands prior assignment. Similarly, PMs can be sorted in decreasing order according to their power efficiency.

Note, that since VM's resource demands and PM's capacities are represented as multi-dimensional vectors, sorting VMs and PMs requires to choose an ordering function like the L1 norm [294]. Previous works have shown that FFD requires no more then $11/9\,OPT + 1$ bins with OPT beeing the optimal number of bin [304]. Other examples of online algorithms are Best-Fit (BF), Worst-Fit (WF), and Next-Fit (NF). Similarly to FF they can be easily transformed in offline versions by presorting the VMs. A lot of work has been done over the past years to apply such algorithms to the VMPP.

In [112], the authors present the worst-case and average-case complexity of the FF, BF, WF, and NF algorithms. A similar work can be found in [284] where the authors reduce the VMPP to the MDBPP and prove that it is NP-hard.

In [205], the authors investigate the MDBPP and propose two new greedy algorithms, namely Choose Pack and Permutation Pack. Simulation results using synthetic benchmarks show that both algorithms outperform the FF algorithm. This work considers CPU only and targets homogeneous PMs.

In [209], a framework called EnaCloud is introduced. In contrast to algorithms such as FF which do not migrate existing VMs during the VM placement, EnaCloud attempts to displace existing VMs in favour of the to be placed VM. The key idea is that less utilized VMs are more likely to fill the resource gaps available on the destination PMs. Particularly, when a new VM is to be placed, EnaCloud attempts displaces less utilized VMs with the new VM. The displaced VMs are then reinserted into the system using an algorithm such as FF. Experimental results using synthetic benchmarks show that proposed algorithm achieves approximability 10% and 13% more energy savings as FF (resp. BF). This work assumes homogeneous PMs and considers CPU utilization only.

In [99], the authors argue that all existing VM placement algorithms fail to leverage the Min, Max, and Share (MMS) parameters available on modern hypervisors (e.g. Xen) thus preventing them from providing differentiation between VM priorities. Min parameter allows to set a minimum amount of required VM resources. On the other hand, max allows to set an upper bound on resources a VM is allowed to use. Finally, share can be used to guide the VMM scheduling decisions. Differentiation between applications is particularly useful in enterprise data centers where VMs by nature have different priorities. For instance, the web request load balancer VM is of a higher priority as a private development server. In that case the load balancer should be assigned more resources (e.g. CPU) as the development server. However, traditional VM placement approaches ignore theses facts by treating all VMs equal. This can result in a significant performance degradation depending on the system load. Particularly, the authors show that MMS parameters become increasingly important at high loads. Consequently, they develop a suite of techniques taking into account the MMS parameters while performing the VM placement and come up with a novel VM placement algorithm called PowerExpandMinMix. Simulation results using randomly generated data as well as small scale real data center experiments show that MMS parameters can improve the data center utilization by 47% and more. This work considers CPU utilization only and is limited to homogeneous PMs.

In [288], the authors analyze the characteristics of enterprise workloads and attempt to find correlations between their resource demands. They find that while minimizing the number of PMs as part of VM placement can yield significant power savings unless correlations are not taken into account performance degradation can limit the potential

energy savings and degrade VM performance. Based on their workload characteristics analysis they propose new algorithms in order to avoid placing positively correlated VMs on the same PM. Instead, the algorithms collocate negatively correlated VMs. Negatively correlated VMs have the property that their probability to exhibit a peak resource utilization at the same time is low ($\sim$1%). Experimental results using CPU utilization traces prove the viability of the proposed algorithms. The major drawback of this work is that it considers CPU only and is limited to homogeneous PMs.

In [271, 270], the authors present simulation results for many of the well-known greedy algorithms (e.g. FF, Choose Pack, Permutation Pack) applied on the MDBPP in the context of virtualized environments. They find the Choose Pack algorithm to be the fastest one. This work is further extended in [242, 91] where the authors take into account DVFS capabilities and power consumption constraints. Simulations using a mix of application traces and randomly generated input data sets were used to verify the algorithms. This work targets homogeneous PMs and considers CPU and memory. It is extended in [270] to consider heterogeneous PMs.

In [212], two new VMPP algorithms are proposed: Dynamic Round Robin (DRR) and a hybrid algorithm which combines DDR with FF. Simulations are used to compared both algorithms with the greedy, round robin, and power save algorithms of the Eucalyptus cloud management framework. DDR and the new hybrid algorithm are reported to decrease the power requirements by 56.5% (resp. 55.9%) compared to the traditional round robin algorithm. This work is evaluated using experiments as well as simulations. It considers CPU utilization only and is limited to homogeneous PMs.

Finally, in [150], the authors propose a new VM placement policy called Cost-driven Scheduling Policy (CDSP). CDSP considers the energy efficiency, virtualization overheads, and SLA violations during the placement. Simulation results based on Grid'5000 as well as web traces show that the proposed algorithm outperforms simple algorithms such as round-robin and backfill by 30% in all the mentioned aspects. This work supports heterogeneous PMs. However, it focuses only on the CPU utilization.

Meta-heuristics. Another category of algorithms suitable for solving the VMPP are meta-heuristics. Meta-heuristics are probabilistic algorithms which are able to compute near optimal solutions to complex optimization problems (e.g. bin packing). Examples of such algorithms include ant colonies [125] and genetic algorithms [152]. In [208], the authors propose an Ant Colony Optimization (ACO)-based algorithm for solving the One-Dimensional Bin-Packing Problem (ODBPP). Through simulations and randomly generated data the authors show that combined with a local search their algorithm could outperform the evaluated Genetic Algorithm (GA). This work has been further refined in [95] by proposing an algorithm called AntPacking. AntPacking was shown to perform at least as good as the best genetic algorithm. In [306], another ACO-based algorithm for solving the ODBPP is introduced. Simulation results show that the algorithm achieves better solutions than FFD. Finally, in [265] the authors define a generic ACO algorithm for solving subset selection problems. As all the mentioned algorithms target the ODBPP they all focuses on a single resource dimension. Finally, in [217], the authors propose an probabilistic VM placement algorithm. A master PM which accepts the VM submission request, broadcasts the request to all the PMs in the data center. Once a PM receives the request it computes an probabilistic assignment function which is based on the current and maximum allowed PM utilization. Servers for

which the function returns a value greater zero respond to the master as beeing available to accommodate the VM. The master then randomly chooses a PM among the ones which have responded. The major drawback of this algorithm is that it does not scale due to the need to broadcast VM submission requests. Moreover, as the PMs do not consider the requested VM capacity during the computation of the probabilistic assignment function, many rounds are needed to assign the VMs. This work is evaluated using simulations and targets only one resource (i.e. CPU). Finally, in [253], the authors design a genetic algorithm for the VMPP. Using simulations they show that it can compute better (i.e. utilize less PMs with moderate performance degradation) solutions (i.e. VM to PM assignments) than traditional greedy algorithms. This work considers heterogeneous PMs. However, it is focuses on the CPU resource only.

Mathematical programming. Mathematical programming techniques such as Linear Programming (LP) [260] and Constraint Programming (CP) [255] can be used to compute the *optimal solution* to the VMPP by leveraging LP (resp. CP) solvers. Examples of LP solvers are IBM ILOG CPLEX [33], Gurobi Optimizer [27], LP Solve [1], and GNU Linear Programming Kit (GLPK) [4]. The previously introduced BIP model can be implemented with such solvers. On the other hand, prominent CP solvers include IBM ILOG CPLEX CP Optimizer [31] and Choco [16]. The main advantage of mathematical programming techniques over greedy algorithms and meta-heuristics is that they allow to easily add additional constraints such as VM collocation and anti-collocation. Moreover, knowing the optimal solution can provide good insights on the quality of solutions obtained using greedy algorithms or meta-heuristics. However, mathematical programming approaches require exponential time to solve the VMPP optimally. Consequently, they scale only to a small number of PMs and VMs. Moreover, the solution time highly depends on the number of constraints and decision variables [271, 266].

LP, in particular Mixed-Integer-Linear-Programming (MILP) derives an optimal solution to the VMPP using a branch-and-bound [115] algorithm. One example of using the LP approach for a problem related to VMPP can be found in [219] where the authors introduce a server consolidation planning tool called ReCon. Server consolidation is a process of transforming older servers into VMs and packing the VMs on less but powerful servers, a procedure also known as Physical-to-Virtual (P2V). ReCon allows system administrators to estimate the benefits from server consolidation. In order to achieve P2V, ReCon collects CPU utilization traces from servers. It then treats existing servers as VMs and issues VM to target server migration suggestions. For instance, it can recommend to transform two servers into VMs and collocate them on single PM. ReCon also supports VM placement constraints such as collocation and anti-collocation. This is particularly useful when two VMs must be collocated on the same PM or to prevent certain VMs from beeing collocated together. For instance, for performance reasons one might want to collocate two VMs. On the other hand, legal obligations might prevent two VMs from beeing collocated. In [253], the authors solve the VMPP using LP and compare their results with a genetic algorithm. Simulation results show that the genetic algorithms achieves close to optimal results and requires significant less amount of time than LP. This work considers homogeneous PMs. Moreover, it targets the CPU resource only. In [87, 266], the authors formulate the Static Server Allocation Problem (SSAP) which is related to VMPP and compare its LP solution with the one from FFD. Simulation results show that FFD computes close

to optimal solutions. Moreover, the authors prove that SSAP is strongly NP-hard. The proposed BIP model for SSAP supports CPU, memory, and networking resources as well as heterogeneous PMs. However, its evaluation considers CPU traces only and targets homogeneous PMs.

CP is an alternative approach to LP. Similarly to LP, decision variables, objective functions to minimize or maximize, and constraints must be defined, which are then solved using a branch-and-bound algorithm. However, in contrast to LP, CP supports logical constraints and provides arithmetical expressions (e.g. integer division). Moreover, CP does not assume mathematical properties of the solution space (e.g. linearity) while LP requires the model to fall in a well-defined category (e.g. MILP). Nevertheless, both approaches are orthogonal and can be combined in order to achieve better results. For example, Constraint Satisfaction Problems (CSPs) [281] have been efficiently solved by such hybrid methods [240]. Not much research has been done so far in order to apply CP on the VM placement problem. The most prominent work can be found in [167] where the authors model the VMPP as an instance of the CSP and solve it using CP. Experimental results show that the CP approach outperforms the FFD heuristic in the number of used PMs. This work assumes homogeneous PMs and considers CPU and memory resources.

Table 2.2 summarizes the properties of the presented VM placement algorithms.

Approach	Algorithm	Considered resources	Heterogeneity	Evaluation	Workload
[112, 284]	Greedy	ODBPP	No	Analytical proof	None
[205]	Greedy	CPU	No	Simulations	Synthetic benchmarks
[209]	Greedy	CPU	No	Experiments	Synthetic benchmarks
[99]	Greedy	CPU	No	Experiments	Randomly generated
[288]	Greedy	CPU	No	Experiments	CPU traces
[271, 270]	Greedy, GA, LP	CPU, memory	Yes	Simulations	Mix of Google traces and randomly generated
[91, 242]	Greedy, LP	CPU, memory	No	Simulations	Randomly generated
[212]	Greedy	CPU	No	Experiments and simulations	Randomly generated
[150]	Greedy	CPU	Yes	Simulations	Grid'5000 and web traces
[306, 208, 95]	ACO	ODBPP	No	Simulations	Randomly generated
[265]	ACO	Subset selection	No	Simulations	Randomly generated
[217]	Bernoulli trial	CPU	No	Simulations	Randomly generated
[253]	LP, GA	CPU	Yes	Simulations	Application traces
[219]	LP	CPU	No	Simulations	Application traces
[87, 266]	LP	CPU, memory	No	Simulations	Application traces
[167]	CP	CPU, memory	No	Experiments	Synthetic benchmarks

Table 2.2: Comparison of the VM placement approaches

2.4.4.3 Underload and Overload Management

A lot of work has been done on underload and overload management over the past years. In [297], the authors introduce Sandpiper a VM management system which is able to detect and resolve overload situations. Sandpiper overload detection mechanisms flags a PM as overloaded only if its aggregated resource (i.e. CPU, memory, network) demand exceeds a predefined threshold for a long enough period of time. This way the system can avoid performing useless migrations due transient resource demand spikes. Once overload situations are detected the resolution mechanism is triggered. This mechanism works in two steps: (1) local VM resource adjustments: (2) VM migrations. The former step attempts to resolve the overload situations by changing the resource allocation of the VM locally. For example, by adding more virtual CPUs, network interfaces, or memory depending on the overloaded resource dimension. Otherwise, if not enough physical resources are available, a load balancing algorithm is triggered which attempts to migrate the VMs from the most overloaded PM to the least overloaded ones. Sandpiper resorts on a heuristic which first sorts the PMs in decreasing order according to their utilization. It then takes the most loaded PM, sorts its VMs in decreasing order according to their utilization and attempts to assign this VMs to PMs starting from the least loaded one. If no suitable PM can be found the algorithm proceeds with the next most loaded VM. This process continues for all PMs until their utilization falls below a predefined threshold. The authors have evaluated Sandpiper in a realistic environment and shown that VM live migration is a feasible technique for resolving overload situations. The major drawback of Sandpiper is that it does not consider migrating VMs away from underload PMs for the purpose of energy savings. Moreover, it targets homogeneous PMs. Finally, no power management mechanisms (e.g. node off/on) exist.

In [192], the authors detect underload/overload situations based on static thresholds (low resp. high) and trigger greedy algorithms to resolve them. For instance, when an PM overload situation is detected, an overload mitigation algorithm attempts to move some VMs away in order to resolve the overload situation. This is achieved by first sorting VMs on the overloaded PM in increasing order according to their utilization. Moreover, destination PMs are sorted in decreasing order according to their utilization. The algorithm then attempts to move the least utilized VM to one of the destination PMs starting from the most loaded PM which still has enough capacity to accommodate the VM. If a destination PM could be found the VM is placed on the PM and the destination PMs are resorted in decreasing order again according to their utilization. This process continues with the next least loaded VM until the overload situation on the PM is resolved. In case no destination PM for a VM can be found the authors assume that a new PM can be started to accommodate the VM. Underload situations are handled as follows. The key idea of the underload mitigation algorithm is to first sort all VMs (i.e. across all PMs) in increasing order according to their utilization. Moreover, similarly to the overload mitigation algorithm PMs, are sorted in decreasing order according to their utilization. Then, the algorithm starts with the least loaded VM and attempts to move the VM to a destination PM which has enough capacity to accommodate it starting from the most loaded PM. A VM is only moved to a PM if this move increases the variance [154] across the utilizations of all PMs. In case a VM move does not increase the variance the algorithm continues with the next least utilized VM. The algorithm terminates when the variance starts decreasing. PMs which do not host VMs after this procedure are powered off. Experimental results prove the viability of the system. The major limitations of this work are the restriction to the CPU resource and homogeneous PMs.

Moreover, no power management mechanisms are integrated.

In [148, 147], the authors present a system able to detect and react to underload/overload situations. Particularly, a PM is considered as overloaded if either its CPU or memory utilization crosses a predefined threshold. Then, a fuzzy controller is used to select the VMs to be migrated as well as the candidate destination PM to accommodate the VMs. Candidate PM is chosen such that it corresponds to the least loaded PM with enough capacity for the selected VM. In case such candidate PM does not exist a new PM is powered on and the VM is migrated to it. Finally, underload situations are detected by computing the average resource utilization of all PMs and checking whether it falls below a given threshold or not. Taking the average utilization of all PMs instead of performing underload detection based on individual PMs, helps to prevent turning off PMs which have been recently powered on and thus do not accommodate much load. Once the underload situation was identified, the fuzzy controller chooses the least loaded PM and attempts to move all its VM away in order to transition it into a power saving state (i.e. shutdown). In case it is not possible to move all the VM then no power state transition is performed. Simulation results show that the best CPU and memory overload thresholds are 85% resp. 95%. Concerning, underload thresholds best choices are 50% and 80%. This system considers CPU and memory resources only. Moreover, no power management mechanisms are integrated. Finally, only homogeneous PMs are considered.

In [307], the authors introduce 1000 island, an integrated resource management system for virtualized data centers. In 1000 islands, a PM is considered as overloaded if the aggregated VM resource (e.g. CPU) utilization including the VMM resource demand exceeds a given threshold. Particularly, a PM is considered as overload if its aggregated resource CPU and memory utilized exceeds 99% (resp. 95%). On the other hand, a group of PMs is considered as underloaded if its CPU and memory utilization falls below 40% (resp. 60%). To resolve the overload (resp. underload) situations this work builds upon the algorithms presented in [147]. Simulations and small scale experimental results show that the system is able to efficiently manage the data center while mitigating SLAs. Similarly to the previous work only CPU and memory resources are considered. Finally, no power management mechanisms are integrated. Moreover, the system targets homogeneous PMs.

In [211], the authors introduce the Power-Aware Domain Distribution (PADS) scheme. In PADS each PM has a reserved local buffer of CPU resources in order to handle transient VM CPU utilization spikes. PMs are considered overloaded as soon as their hosted VMs start using resources from the PM local buffer. PADS is triggered periodically and detects overloaded PMs based on the local buffer usage. In order to decide which VMs must be moved to resolve the overload situations PADS supports three candidate VM selection schemes: maximum average demand, minimum average demand, and minimum standard deviation of resource demands. Simulations show that up to 70% of energy can be saved with less than 1% SLA violations by using PADS. The main drawback of PADS is that it considers CPU only. Moreover, it assumes homogeneous PMs and ignores power management mechanisms.

In [197], the authors introduce vManage. The key idea of vManage is to provide a framework able to coordinate system (e.g. power and/or thermal management) and virtualization management (e.g. VM placement) solutions. The authors argue for such a coordination as both system and virtualization management solutions can negatively influence each other. For instance, when a power manager scales down the CPU frequency in order to enforce given power budget limits, this typically leads to SLA violations which need to be resolved

by the virtualization manager. On the other hand, resolution of SLA violations typically requires increasing the resource capacity (e.g. CPU frequency) and thus affects the power consumption which increases the temperature. In vManage overload mitigation mechanism is triggered when CPU utilization exceeds 80%. To maintain SLAs, VM migration is performed. Particularly, the first PM which can provide the required SLA is chosen as the destination PM. vManage was evaluated on an experimental testbed and shown to reduce the number of violations by 71% and 10% power savings compared to a non-coordinated environment. Finally, the number of migrations was reduced by 54%. This system considers CPU resource only and targets homogeneous PMs.

In [301], the authors introduce an underload/overload mitigation system based on Multiple Criteria Decision Analysis (MCDA) using the PROMETHEE method [139]. The overload/underload detection is based on lower and upper level resource utilization thresholds. Once a PM is overloaded (i.e. upper threshold is crossed) it computes a set of VMs which must be migrated in order to resolve the overload situation using the proposed method. Particularly, a central manager is contacted in order to receive a list of candidate PMs able to accommodate the VMs. Finally, the VMs are migrated to the candidate PMs. This work is validated by simulations. It considers CPU, memory. and bandwidth utilization. Homogeneous PMs are assumed. No power management mechanisms are evaluated.

In [298], the authors present a cross-layer system for managing the system (e.g. power management) and virtualization layers (e.g. VM placement). A central controller is used to collect sensor information from both layers and has a global view of the system. VM live migrations are triggered in three cases: thermal emergency, resource contention, low energy efficiency. Thermal emergency happens when a PM is overheated and attempts to move the most loaded VMs away. Resource contention happens when the aggregated VM resource utilization exceeds a given threshold. Finally, low energy efficiency is the result of a PM beeing underutilized. The central controller periodically checks the PMs and decides if any of the introduced cases are present based on predefined thresholds. For instance, when a PM is overloaded some VMs need to be migrated away. In order to select the VMs to be migrated the authors first compute an average resource utilization of all the VMs on the overloaded PM. Then, VMs those resource utilization is above the average are considered as candidates to be migrated and sorted in increasing order. Finally, destination PMs are selected by considering their temperature, power, and performance. Given that these three criteria may have conflicting objectives, the authors apply a multi-objective approach which combines the objective functions of the three criteria into a single one. Experimental results show that the proposed system reduces the number of migrations for up to 80%, integrates stable PM selection, and greatly improves the application performance and the power efficiency.

In [247], the authors present Distributed VM Scheduler (DVMS), a ring-based system designed to manage underloaded and overloaded PMs. Similarly to the previous works, overload and underload conditions are detected by the PMs based on thresholds. Once detected, requests to mitigate them are forwarded to the successor PM in the ring. The successor PM attempts to resolve them by applying VM consolidation considering itself and the predecessor PMs in the chain. The system continues to forward underload/overload mitigation requests in the ring until the last PM has been reached or a solution has been found before. Simulation results show that the system is able to achieve solutions (i.e. number of used PMs) close to a centralized system. DVMS does not target power management. The considered resources are CPU and memory. The simulated PMs and VMs are homogeneous.

Workload in VMs is randomly generated.

In [92], a rule-based approach for detecting underload and overload conditions is presented. Particularly, lower and upper thresholds for each PM resource dimension (e.g. CPU) are defined. If the resource utilization in any resources falls below the lower threshold the PM is considered as underutilized. Otherwise, if the resource utilization exceeds the upper threshold the PM is marked as overloaded. Any value between the thresholds is considered as optimal. Once PM underload/overload situations are detected they are resolved using modified initial VM placement algorithms [91] which take into account the current VM placement. Using simulations the authors report 61.6% energy savings with limited impact on VM performance. To conserve energy a node off/on power management mechanism is simulated. This work targets homogeneous PMs and considers CPU and memory resources.

In [86, 83], the authors propose an adaptive approach for defining underload/overload thresholds based on historical VM resource utilization data. They argue that such an approach is necessarily for an environment in which the workloads are dynamic and hard to predict. In order to resolve overload situations the authors present a number of VM selection algorithm: Minimum Migration Time (MMT), Random Choice (RC), and Maximum Correlation Policy (MCP). MMT selects a VM which requires the least migration time compared to the others. Migration time is computed based on VMs memory requirements. RC selects a VM randomly. Finally, MCP selects the VMs with highest correlation in terms of CPU utilization. Indeed, VMs with highest resource utilization correlation have a higher probability to result in an overloaded PM. Finally, after the VMs are selected the authors apply a modified version of the Best-Fit Decreasing heuristic to place the VMs on the non-overloaded destination PMs by considering their power efficiency. Regarding, underload situations a simply algorithm is used which takes the underloaded PMs and attempts to migrate all VMs away to other PMs while respecting their upper resource utilization thresholds. Simulation results show promising results. This work has a number of strong assumptions which could prevent many of its concepts to be implemented in a real environment. First, the adaptive threshold mechanism as well as the proposed MCP assume *long running VMs* due to their requirement of historical data. Given that the authors target Amazon EC2 like clouds such data is hard to obtain as VMs are typically short-lived (e.g. spot instances). Second, the minimum migration time algorithm attempts to select VMs requiring the least migration time. VM migration time heavily depends on many parameters such as VM memory page dirtying rate, efficient storage migration mechanisms, and network conditions which are known to be hard to estimate beforehand. This work considers the CPU utilization only, targets homogeneous PMs, and lacks power management mechanisms. Finally, in [84], the same authors propose an Markov host overload detection algorithm for handling overloaded PMs and evaluate it using simulations. Ongoing efforts exist to integrated the proposed algorithms in the OpenStack cloud management system [85]. However, as of now no working implementation yet evaluation exists.

Finally, in [289] the VMware Distributed Power Manager (DPM) [289] is presented. DPM integrates proprietary algorithms to resolve underload and overload situations. The algorithms are triggered periodically and detects underload and overload PMs based on lower (resp. upper) level thresholds. Particularly, for each resource (i.e. CPU and memory), DPM attempts to maintain an resource utilization between 45% to 81% on all PMs. In the event of an underload situation DPM attempts to migrate all VMs to other PMs in order to turn off the underloaded PM. On the other hand, in the event of an overload situation, DPM attempts

to migrate some VMs away to resolve the problem. In case VMs can not be placed on any of the active PMs, DPM turns on new PMs. DPM involves power management mechanisms which turn off idle PMs and wake them up once required (i.e. when the load increases). The major drawback of DPM is that it does not consider the network utilization.

Table 2.4.4.3 summarizes the presented works by providing a classification of their properties and features.

Approach	Mitigation	Algorithm	Considered resources	Resource utilization	Heterogeneity	Power management	Evaluation	Workload
[297]	Underload	Greedy	CPU, memory, network	Dynamic	No	None	Experiments	Mix of synthetic benchmarks
[192]	Underload, overload	Greedy	CPU	Dynamic	No	None	Experiments	Websphere workload simulator
[148, 147]	Underload, overload	Greedy	CPU, memory	Dynamic	No	None	Simulations	Application traces
[307]	Overload	Greedy	CPU, memory	Dynamic	No	None	Mix of experiments and simulations	Application traces
[211]	Overload	Greedy	CPU	Dynamic	No	None	Simulation	Synthetic benchmarks
[197]	Overload, underload	Greedy	CPU	Dynamic	No	DVFS	Experiments	Web
[298]	Thermal emergency, underload, overload	Greedy	CPU, memory, network	Dynamic	No	None	Experiments	Synthetic benchmarks
[92]	Underload, overload	Greedy	CPU, memory	Dynamic	No	Node off/on	Simulation	Synthetic benchmarks, bioinformatic traces
[289]	Underload, overload	Greedy	CPU, memory	Dynamic	PMs and VMs	Node off/on	Mix of experiments and simulations	Synthetic benchmark
[86, 83]	Underload, overload	Greedy	CPU	Dynamic	No	None	Simulation	Application traces
[301]	Underload, overload	MCDA	CPU, memory, network	Dynamic	No	None	Simulations	Randomly generated
[247]	Underload, overload	Constraint programming	CPU, memory	Dynamic	No	None	Simulations	Randomly generated

Table 2.3: Comparison of the underload and overload mitigation approaches

2.4.4.4 VM Consolidation

Complementary to the overload and underload management algorithms, VM consolation can be used to continuously remove resource fragmentation on moderately loaded PMs via repacking of already placed VMs on the least number of PMs. Some works have studied the problem of VM consolidation.

In [89], the authors present a novel VM consolidation algorithm based on a modified version of the FFD heuristic. Simulation results show that the proposed algorithm requires up to 50% less physical resources to maintain SLAs compared to a VM to PM assignment which is not modified for long periods of time (e.g. several months). The major drawback of the proposed algorithm is that it does not take into account the current VM placement while computing the new VM to PM assignments. Consequently, its solutions result in a large number of VM migrations. This work considers the CPU resource only and targets homogeneous PMs. Finally, no power management mechanisms exist.

In [287], the authors introduce pMapper, a power and migration-cost aware application placement framework. pMapper is designed around three main components: performance, power, and migration manager. Performance manager has a global view of all applications in the system. It observes the application performance and suggests VM resizing actions.

Power manager monitors the current power consumption and can suggest power saving actions such as DVFS. Finally, the migration manager, interacts with the hypervisor in order to trigger VM live migrations. An arbitrator exists to coordinate the management decisions between all the components. pMapper integrates three algorithms: min Power Parity (mPP), min Power Placement algorithm with History (mPPH), and PMaP. mPP and mPPH algorithms are integrated into the power manager while the PMaP algorithm resides on the arbitrator. The mPP algorithm takes as input the VM sizes, current assignment of VMs to PMs and a power model of all PMs. It then attempts to place the VMs such that the total power consumed is minimized. This is achieved by first sorts VMs and PMs in decreasing order according to their utilization (resp. power efficiency). Then, VMs are assigned to the PMs starting from the most power efficient one using the FF algorithm. The major drawback of mPP is that it does not take into account the current VM placement thus its solutions result in a large number of migrations. To solve this problem the authors propose mPPH, an extension of the mPP algorithm which considers the current VM placement. Still, despite the fact that mPPH algorithm takes into account the current VM placement its efficiency which respect to minimizing the power usage is low. Consequently, the authors propose the PMaP algorithm which strikes to find a balance between power and migration costs while minimizing the number of migrations. Most of the pMapper components were evaluated using simulations. This work considers heterogeneous PMs and focuses on the CPU resource utilization. While pMapper architecture is designed to support many power management mechanisms (e.g. DVFS, node off/on) none of them is evaluated.

In [249], a coordinated multi-level power management system for virtualized data centers is proposed. The key idea of the system is to coordinate the power management decisions taken at different levels (i.e. node, rack, data center) of the system. The authors argue that coordinating the power management decisions is crucial in order to avoid inter-system level power management decisions interference's. Particularly, the proposed system targets average and peak power management. Its architecture is composed of five nested components implementing feedback control loops: Efficiency Controller (EC), Server Manager (SM), Enclosure Manager (EM), Group Manager (GM), and a Virtual Machine Controller (VMC). EC and SM are in charge of average and peak power management at the node level, respectively. For instance, to reduce the average node power consumption, a CPU utilization reference value can be set on the EC. In the event of a low utilization, the EC feedback control loop gradually scales down the CPU frequency thus reducing the power consumption. One key aspect of the system is the nested nature of its system components. Consequently, in the event of a server power limit violation, instead of directly manipulating the CPU frequency, the SM changes the CPU utilization reference value of the EC. This allows the EC to scale down the CPU frequency and thus lower the server power consumption. This avoids the need of a central coordinator and thus greatly reduces the system scalability and implementation complexity. Similarly, EM and GM enforce peak power capping's at the rack and data center level. Finally, in order to reduce the nodes average power consumption, VMC performs periodic, power-capping aware VM consolidation at the data center level and turns off the resulting idle nodes. VM consolidation is modeled as a binary integer program and solved using a not further specified greedy bin-packing algorithm. Using a trace-driven simulation based on enterprise workloads, the authors show that the proposed system is able to efficiently coordinate the power management actions at different system levels.

In [167], the authors propose a VM consolidation manager called Entropy. Entropy model

the VM consolidation problem as an instance of the CSP and solves it using constraint programming. VM consolidation is started at the arrival and removal of VMs. Experiments performed on 39 nodes of the Grid'5000 testbed shows that the constraint programming approach outperforms the FFD algorithm in the number of used PMs and migrations. The major advantage of this method is that it allows to easily consider additional VM placement constraints such as VM collocation and anti-collocation. Entropy treats the VMs as static boxes which are either fully utilized or not. It targets homogeneous PMs and focuses on CPU and memory. This work is extended in [165], where the authors evaluate the scalability of the constraint programming approach. Simulation results show that constraint programming can scale for up to 2000 PMs and 10 000 VMs. This work considers homogeneous PMs. It treats CPU and memory demands as dynamic (resp. static). In [166], the authors introduce Plasma, an extension of the Entropy consolidation manager targeting web applications. Simulation results show that the Plasma algorithm scales for up to 2000 PMs and 4000 VMs. Moreover, experiments on 8 compute PMs hosting 21 VMs running a synthetic benchmark are performed. Similarly, to the previous work VMs have dynamic CPU and static memory demands. Power management is not the target of these works.

In [122], an energy-efficient VM management system called vGreen is introduced. The authors argue that exploiting the VM characteristics (e.g. instructions per cycle, memory accesses) during VM consolidation is essential in order to achieve energy savings while limiting the performance degradation. Indeed, collocating VMs with similar characteristics could create contention on shared physical resources (e.g. CPU caches, memory busses) and thus degrade the performance. Consequently, in contrast to the previous works, vGreen is designed to capture the Memory Per Cycle (MPS) access and Instructions Per Cycle (IPC) metrics at the hypervisor level. Particularly, vGreen implements a greedy VM consolidation which computes its solutions based on the two metrics. The authors have implemented vGreen and evaluated it using synthetic benchmarks on a two PM testbed. The results show that taking into account VM characteristics can improve the average performance and energy consumption by 40% compared to the greedy VM scheduling policy (with and without DVFS) of Eucalyptus. vGreen considers dynamic CPU and memory demands. It targets homogeneous PMs and does not perform any power management actions (e.g. shutdown).

In [224], the Sercon algorithm is introduced. Sercon modifies the FFD heuristic in order to minimize the number of migrations, PMs are first sorted in decreasing order according to their utilization. Then, VMs from the least loaded PM are sorted in decreasing order according to their utilization. The algorithm then attempts to assign these VMs to the PMs starting from the most loaded one. In case all VMs could be assigned the algorithm repeats the procedure with the next least loaded PM. Otherwise, if some VMs could not be assigned they are left on the PM and the algorithm goes to next least loaded PM and attempts to move its VMs starting from the most loaded PM. Using simulations the authors show that the algorithm greatly reduces the number of migrations compared to FFD and requires only up to 6% more PMs. Sercon considers CPU and memory. It was evaluated in a homogeneous environment. Power management is not considered in this work.

Finally, in [216], the authors introduce V-MAN. The key idea of V-MAN is to periodically apply VM consolidation only within the scope of randomly formed subsets of nodes, the so-called neighbourhoods. V-MAN was evaluated by means of simulations. It is limited to a single VM consolidation algorithm which considers at most two PMs at a time. Moreover, it makes its decisions solely based on the number of VMs thus ignoring the actual VM resource

demands. V-MAN targets homogeneous PMs. Power management was not the target of this work. Table 2.4.4.4 summarizes the properties and features of the reviewed works.

Approach	Algorithm	Considered resources	Resource utilization	Heterogeneity	Power management	Evaluation	Workload
[89]	Greedy	CPU	Dynamic	No	None	Simulation	Application traces
[287]	Greedy	CPU	Dynamic	Yes	DVFS	Simulation	Synthetic benchmarks
[249]	Greedy	CPU	Dynamic	Yes	DVFS, node off/on	Simulation	Application traces
[122]	Greedy	CPU, memory	Dynamic	Yes	No	Experiments	Synthetic benchmarks
[224]	Greedy	CPU, memory	Static	No	None	Simulation	Randomly generated
[216]	Greedy	Number of VMs	Static	No	None	Simulation	Randomly generated
[167]	Constraint programming	CPU, memory	Static	No	None	Experiments and simulations	Synthetic benchmarks
[165]	Constraint programming	CPU, memory	Dynamic CPU, static RAM	No	None	Simulation	Randomly generated
[166]	Constraint programming	CPU, memory	Dynamic CPU, static RAM	No	None	Experiments and simulations	Synthetic benchmarks, Randomly generated

Table 2.4: Comparison of the VM consolidation approaches

2.4.4.5 Application-aware Management of Virtualized Infrastructures

We now present some works targeting application-aware VM management.

In [226], the authors argue that today's cloud providers SLA model is very limited as it only provides guarantees in terms of uptime thus ignoring the application-level Quality-of-Service (QoS) (e.g. response time) requirements. Ignoring application-level QoS requirements can become particularly critical for customers of cloud providers which attempt to minimize the energy costs by consolidating VMs hosting customers applications on the least number of PMs. Indeed, depending on the application characteristics, VM consolidation can yield significant performance degradation of the applications in collocated VMs [194]. Consequently, the authors propose Q-Clouds, an QoS-aware cloud system which aims at providing application-level QoS guarantees thus creating the illusion for the application to run in isolation. This is achieved by monitoring applications performance and adjusting the VM resource allocations to mitigate performance degradation due performance interference cause by collocated VMs. For this, Q-Clouds relies on a Multiple-Input and Multiple-Output feedback control-loop. Moreover, Q-states are introduced as a notion for the customer to specific the desired VM performance levels. For instance, the lowest Q-state translates into a minimum required VM performance (e.g. half a core) at any point in time. Higher Q-states can be used to request more resources if the customer is willing to pay more. High Q-states are enforced by Q-Clouds once the applications inside the VM demand for more resources. In order to support Q-states, Q-Clouds keeps a buffer of resources, the so-called *head room* on each PM. Experimental results show that Q-Clouds is able to entirely avoid performance degradation and improve the system utilization by 35%. The main drawback of Q-Clouds is that it is considers CPU utilization only and does not integrate any power management mechanisms (e.g. node off/on). Moreover, no VM consolidation actions were performed during the experiments.

In [185], the resource management mechanisms of the Mistral VM management system are presented. Mistrals is designed to balance power usage, application performance, and costs of system reconfiguration actions. Mistral controllers are triggered periodically and

checks if some VMs need to be migrated to meet applications performance targets. To estimate the benefits of the adaptation actions Mistral integrates four prediction modules and one optimization module. The predictions modules are: performance manager, power manager, cost manager, and a workload predictor. Performance and power managers are used to estimate the application performance (resp. power usage) of a current system configuration (i.e. VM to PM assignment). To estimate the application response time depending on the workloads, layered queuing network models [186] are used. Power consumption is estimated using a linear function of CPU utilization which was proposed by the authors in [130]. Cost manager receives the current configuration and a set of adaptation actions. Based on this two parameters it estimates the cost of an adaptation action. Finally, the workload predictor is used to estimate the stability intervals. A stability interval is defined as the time between two system reconfigurations. The outputs of all the prediction modules are fed into the optimization module which decides on the optimal set of actions using a heuristic algorithm. Mistral targets CPU utilization and is evaluated using homogeneous PMs.

In [108], the authors present a holistic data center management system which federates IT, power, and cooling management. Each application is assigned with an application-level performance target (e.g. response time). The goal of the system is to meet this target by the use of dynamic resource allocation while minimizing over-provisioned resources. Applications are composed of one or multiple application components which are hosted in VMs. There exist on application controller per application which monitors the application components QoS metrics (e.g. response time). Its task is to guarantee the applications performance target. Therefore it, periodically estimates an utilization target (e.g. 70% CPU utilization) for each application components VM based on an integrated performance model. The utilization target is then propagated to a *node controller* which makes the required VM resource adjustments to the given utilization target. Node controllers are assigned with each PM and are in charge of enforcing the applications utilization targets. To save energy a global arbitrator the so-called pod controller exists. It oversees the node controllers and performs, underload-/overload mitigation, workload consolidation, and PM power down actions. Once PMs are powered down this information is communicated to a service called *Daffy*. Daffy interacts with the cooling infrastructure in order to adjust the computer room air conditioning blower power. Experimental results in a real environment show that the solution can reduce the IT and cooling infrastructure energy consumption by 35% respectively 15%. No details of the algorithms (e.g. underload/overload management) on the global arbitrator were given. This work focuses on the CPU resource and assumes homogeneous PMs.

In [285], the authors propose a utility-based dynamic VM provisioning manager based on constraint programming which aims at balancing the application QoS and energy consumption. Moreover, a consolidation manager is used to minimize the number of PMs via live migration. Small scale experiments on 3 PMs using synthetic benchmarks are used to prove the viability of the approach. To evaluate the scalability the authors conduct a simulation considering 12 PMs and 8 applications. The system considers static CPU and RAM demands. It targets homogeneous PMs. Power management is used to turn off/on the PMs.

Finally, in [116], the authors provide a framework for managing application QoS in clouds. The key idea of the system is to assign each application with its own application manager which will be in charge of managing the applications QoS. Particularly, the application manager monitors the applications QoS metrics (e.g. response time) and takes actions such as scaling up/down the VMs or removing individual application components in or-

der to achieve the best performance vs. energy tradeoffs. Constraint programming is used to model the problem. Simulations are conducted to show its viability. Preliminary results indicate that energy savings are possible. This work considers static CPU and memory demands, focuses on heterogeneous PMs, and lacks power management mechanisms.

Table 2.4.4.5 summarizes the results from our comparison.

Approach	Algorithm	Considered resources	Resource utilization	Heterogeneity	Power management	Evaluation	Workload
[226]	Greedy	CPU	Dynamic	No	None	Experiments	Synthetic benchmarks
[185]	Greedy	CPU	Dynamic	No	None	Experiments	Synthetic benchmarks
[108]	Greedy	CPU	Dynamic	No	None	Experiments	Synthetic benchmarks
[285]	Constraint programming	CPU, memory	Static	No	Node off/on	Experiments and simulations	Synthetic benchmarks
[116]	Constraint programming	CPU, memory	Static	No	None	Simulations	Randomly generated

Table 2.5: Comparison of the application-aware virtualization approaches

2.5 Summary

This chapter has introduced the state of the art of this thesis. It has started with a brief introduction into server virtualization which is a fundamental building block enabling server consolidation in today's cloud data centers. Particularly, we have reviewed the history of server virtualization and presented existing server virtualization and VM live migration techniques. Then, autonomic computing and cloud computing were introduced, two complementary computing paradigms which emerged during the last years and form the context of this thesis. The complementaries come from the fact that cloud computing leverages some of the self-management properties (e.g. self-optimization) of autonomic computing systems. In both computing paradigms server virtualization is typically used to ease compute infrastructure management (e.g. increase security by leveraging VM isolation properties, perform server consolidation for energy savings and better resource utilization, speed up service deployment). First, autonomic computing was introduced by presenting its history, properties, architectural components, and a few selected autonomic computing systems. Afterwards, cloud computing was presented as a promising computing paradigm whose goal is to offer resources (e.g. compute, storage) on-demand based on the pay-as-you-go model. To be precise, we have first provided a cloud computing definition and introduced the cloud characteristics, service, and deployment models. Then, existing attempts to design and implement IaaS loud management systems were extensively reviewed with respect to their scalability and autonomy. Finally, we have conducted an comprehensive review of energy management approaches in computing clusters. Particularly, after defining the terminology and presenting traditional power measurement techniques, we have discussed DPM techniques in non-virtualized and virtualized environments. Our study has shown that despite the fact that a lot of efforts have been made over the past years to design and implement IaaS cloud management systems, as well energy management techniques in computing clusters, much work is still left to be done. The three key observations from this chapter are:

- Despite the vision of autonomic computing, existing IaaS cloud management systems still lack many of the self-management properties such as self-configuration, self-healing, and self-protection. Moreover, most of the IaaS cloud management systems

are based on centralized architectures thus limiting their scalability.

- Only a few works targeting energy management in IaaS cloud management systems have been experimentally validated. Indeed, the few ones which have been experimentally validated were typically limited to *one aspect of the problem* (e.g. VM consolidation). In other words, there is a clear lack of an experimentally validated *holistic energy-efficient IaaS cloud management system* which federates the introduced VM management algorithms (i.e. VM placement, underload and overload management, VM consolidation, and power management).
- A huge amount of attention has been given to the design of centralized VM management algorithms based on greedy algorithms which are known to be hard to parallelize/distribute [76]. A considerable low amount of attention has been given to the application of algorithms which are by nature of their properties very good candidates for parallelization and distribution. Examples of such algorithms include ant colonies and genetic algorithms.

Chapter 3

Snooze: A Scalable, Autonomic, and Energy-Efficient IaaS Cloud Manager

Contents

In the previous chapter we have presented the context of this work and reviewed the state of the art on the design and implementation of autonomic and energy-efficient IaaS cloud management systems. Our analysis has shown that existing IaaS cloud management systems are either based on centralized architectures or lack autonomy and energy management mechanisms. To address these limitations, the goal of our first contribution is to design, implement, and evaluate a novel IaaS cloud management system which: (1) improves the scalability and autonomy issues of centralized IaaS cloud management systems; (2) provides a holistic energy-efficient VM management solution by integrating advanced VM management mechanisms such as underload mitigation, VM consolidation, and power management. To tackle both goals we propose *Snooze*, a novel *autonomic* and *energy-efficient* cloud management system based on a *self-configuring and healing hierarchical architecture*.

This chapter presents the design, implementation, and evaluation of Snooze. It is structured as follows. Section 3.1 introduces the design principles. Section 3.2 presents the system assumptions, model, and a high-level system architecture overview. Section 3.3 describes the hierarchy management mechanisms. They involve the self-configuration and healing of the hierarchy. Section 3.4 introduces the energy-efficient VM management mechanisms and algorithms. Section 3.5 presents selected implementation aspects. Section 3.6 discusses the evaluation results targeting the scalability, autonomy, and energy efficiency of the system. Finally, Section 3.7 summarizes the contributions.

3.1 Design Principles

The main goal of this thesis is to design and implement a scalable, autonomic, and energy-efficient IaaS cloud management system. Thereby, several properties have to be fulfilled by a cloud management system in order to achieve these goals. First, the cloud management system architecture has to scale across many thousands of nodes. Second, nodes and thus framework management components can fail at any time. Therefore, the system needs to self-heal and continue its operation despite of component failures. Finally, the cloud management system has to be easily configurable. In order to achieve this, the cloud management system must satisfy the self-configuration property of an autonomic system as discussed in Chapter 2.

To achieve scalability and autonomy we have made the key design choice to design a system based on a *self-organizing and healing hierarchical architecture*. Our scalability and autonomy design choices are motivated by previous works which have proven that hierarchical architectures can greatly improve the system scalability. Particularly, the Snooze architecture is partially inspired from the Hasthi [238] autonomic system which is shown to scale up to 100 000 resources by simulation. However, in contrast to Hasthi whose design is presented to be system agnostic and utilizes a Distributed Hash Table (DHT) based Peer-to-Peer (P2P) network, Snooze follows a simpler design and does not require the use of P2P technology. Moreover, it targets virtualized systems and thus its design and implementation is driven by the system specific objectives and issues.

Organizing the system hierarchically improves its scalability as components at higher-levels of the hierarchy *do not require global knowledge* of the system. The key idea of our system

is to split the VM management tasks across multiple independent autonomic managers with each manager having only a partial view of the data center. Particularly, each manager is only in charge of managing a *subset of the data center compute nodes and VMs*. A coordinator is automatically elected among the managers during the hierarchy self-configuration and in case of a coordinator failure. The coordinator oversees the managers and is contacted by the clients to submit VMs.

To conserve energy power management mechanisms exists which detect idle compute nodes, transitions them into a power-saving state, and performs wake ups when needed (e.g. in case of not enough active nodes are available during VM placement). To favour idle times, advanced VM management mechanisms are implemented on each manager. They involve compute node underload mitigation and VM consolidation.

3.2 System Architecture

This section presents the system architecture of Snooze, a novel scalable, autonomic, and energy-efficient IaaS cloud management for private clouds. First, the assumptions and system model are introduced. Then, a high-level system overview is presented.

3.2.1 Assumptions and Model

We assume a data center whose nodes are interconnected with a high-speed LAN such as Gigabit Ethernet or Infiniband. Multicast support is assumed to be available at network level. Physical nodes (called nodes) can be either homogeneous or heterogeneous. They are managed by a virtualization solution (e.g. Xen [80], KVM [193]) which supports VM live migration. We assume that the same virtualization technology is deployed on all the nodes to enable advanced VM management mechanisms involving VM live migration. VMs are seen as black-boxes thus no application-specific knowledge is required to guide the VM management mechanisms decisions. Snooze system integrates a leader election algorithm in order to elect a coordinator among the autonomic managers. In this context network partitioning can result in a so-called split-brain situation which results the system in having multiple coordinators. Failures that partition the network and thus yield multiple coordinators are not tolerated. Nodes may fail, following a fail-stop model.

In a virtualized data center multiple VMs are typically collocated on the nodes. Despite the resource isolation properties of modern virtualization solutions *performance isolation* is not always guaranteed. In other words, collocated VMs with correlated resource demands (e.g. memory bound VMs) can experience a performance degradation as they typically share the same hardware subsystems (e.g. last level cache). In this work we assume that performance isolation is provided by the underlying virtualization solution. Consequently, the VM management algorithms presented in this work do not take into account complementaries between the VM resource demands during their decision making processes.

Finally, applications inside VMs can incur dependencies on the VMs. For instance, a web server VM will typically have a dependency on an application server VM and/or a database VM. Similarly, security obligations might prevent two VMs from beeing close to each other. In this work we assume *independent VMs*. In other words, the proposed VM management algorithms do not support the specification of VM collocation or anti-collocation constraints

which could allow to allow/forbid certain VMs to be collocated. Nevertheless, given that Snooze is flexible enough to support any VM management algorithm nothing prevents the system from integrating such algorithms in the future.

3.2.2 High-level Overview

The high-level overview of the hierarchical Snooze architecture is shown in Figure 3.1. It is partitioned into three layers: computing, management, and client. At the computing

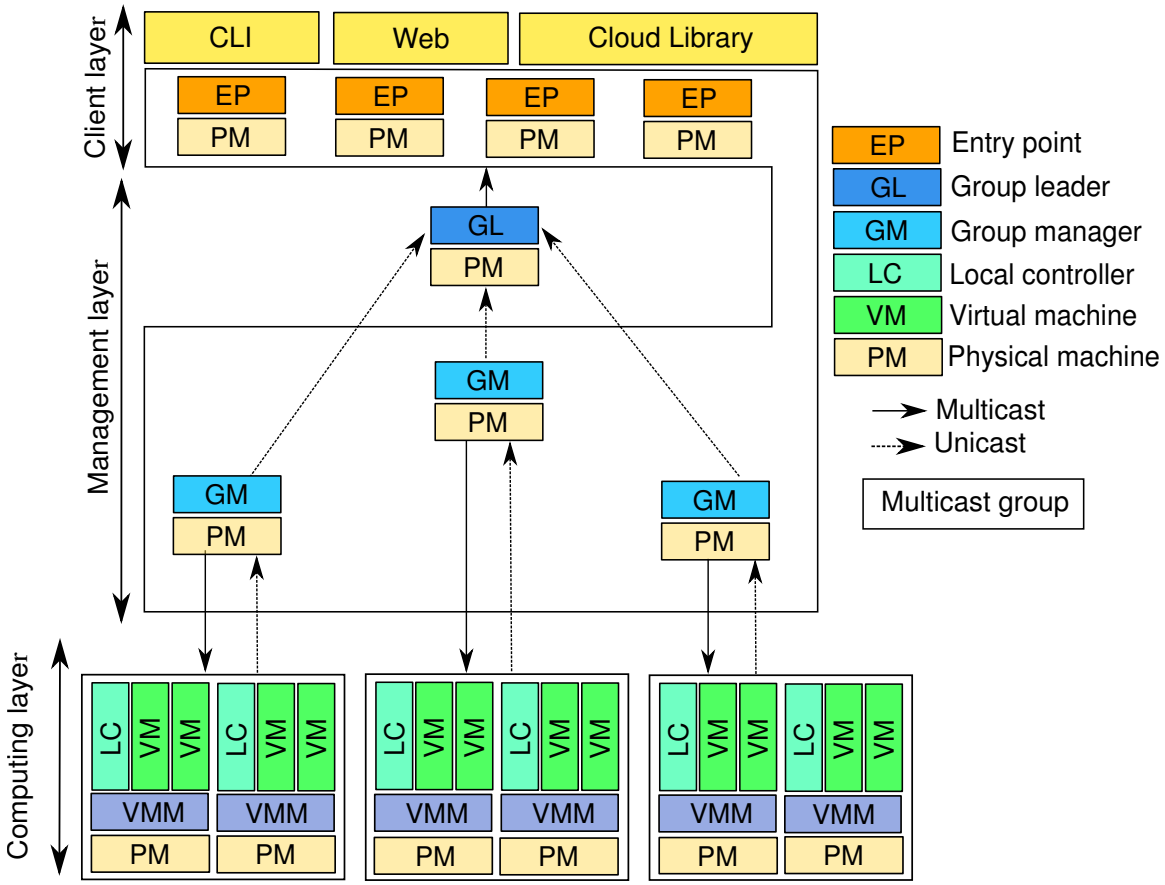


Figure 3.1: Snooze high-level system architecture overview

layer, nodes are organized in a cluster in charge of hosing the VMs. Each compute node is controlled by a system service [1], the so-called *Local Controller (LC)* [2]. A management layer allows to scale the system. It is composed of nodes hosting fault-tolerant system services: one *Group Leader (GL)* and one or more autonomic managers, the so-called *Group Managers (GMs)*. System services are organized hierarchically. GL oversees the GMs. It is elected among the GMs during the hierarchy self-configuration and in the event of a GL failure. Each GM manages a *subset* of LCs and VMs. GL receives VM submission requests from the clients and distributes them among the GMs. Once VMs are submitted clients interact

1. We define a system service as a background process running on the OS.
2. In this document we use the terms LC, compute node, and physical machines (PMs) interchangeably.

directly with the GMs to control the VMs (e.g. shutdown, reboot). As the GL can change over the time, a method is required in order for the clients to discover the current GL. This functionality is provided by the client layer. The client layer is composed of a predefined number of services, the so-called *Entry Points (EPs)* which remain updated about the current GL. All system services are accessible through a RESTful interface. Consequently, any client software (e.g. Command Line Interface (CLI), web, Cloud Library) can be implemented to interact with the EPs, GL, and the GMs.

Given a set of physical nodes it is up to the system administrator to decide on the number of LCs and GMs upon Snooze deployment. For instance, in the most basic deployment scenario two GMs and one LC are required. One of the GMs will be promoted to a GL as part of a leader election procedure. System services are flexible enough to co-exist on the same node. Consequently, it is possible to deploy the entire hierarchically on a single node.

3.2.2.1 Local Controllers

Each LC enforces the VM life-cycle and node management commands coming from its assigned GM. Examples of such commands include VM start and live migration as well as node power-cycling (e.g. suspend). LC also monitors VMs, detects overload and underload situations, and periodically sends VM resource utilization data to its assigned GM. Overload/underload indicators are piggybacked with these data. Each LC maintains a repository with information about the currently running VMs on its node.

3.2.2.2 Group Managers

Each GM is in charge of the management of a subset of LCs. It receives VM resource utilization data from LCs and stores it in a local repository. Based on this data the GM estimates VM resource utilization and takes VM management decisions involving three tasks: VM placement, LC overload/underload mitigation, and VM consolidation.

VM placement mechanisms are triggered event-based to handle VM submission requests arriving from the GL. LC overload and underload mitigation mechanisms are triggered when overload (resp. underload) events arrive from LCs and aims at moving VMs away from heavily (resp. lightly) loaded LCs. VM consolidation is performed periodically according to the system administrator specified interval. For example, it can be used to optimize the utilization of moderately loaded LCs on a weekly basis by repacking existing VMs on as few LCs as possible. Both, overload/underload mitigation and VM consolidation policies output a migration plan which specifies the new VM to LC assignments. A GM enforces the migration plans by instructing the LCs it manages to perform VM live migration.

Power management is integrated into each GM to power-cycle and wake up idle LCs. LCs are woken up in case of either not enough powered-on LCs are available on a GM during VM placement or overload situation.

GM summary information is periodically sent by each GM to the current GL in order to support high-level VM to GM distribution decisions (see Section 3.4.2 for more details on monitoring). Finally, GMs are also contacted by the client software to control (e.g. shutdown) VMs and retrieve VM information (e.g. resource utilization, status).

3.2.2.3 Group Leader

The GL manages the GMs. It is in charge of assigning LCs to GMs upon startup, accepting clients VM submission requests, VM networking management and the dispatching of the submitted VMs among the GMs. Moreover, it receives GM summary information and stores it in a repository. We now discuss these tasks in more detail.

When LCs are started they need to get a GM assigned. LC to GM assignment decisions are guided by a LC assignment policy. For example, LC could be assigned to GMs in a round-robin fashion or based on the current GM utilization. Once the LCs are assigned to GMs, VMs can be submitted by the clients to the GL. In case client VM submission requests arrive before LCs are assigned to GMs, an error message is returned. Note, that additional LCs can join the system at any time without disturbing its normal functioning.

Once a group of VMs (one or multiple) is submitted to the GL, VM to GM dispatching decisions are taken by the GL. They are implemented using a VM dispatching policy which decides on the assignment of VMs to GMs. In order to compute this assignment the GM summary information is used which contains information about aggregated GM resource utilization. Based on this assignment the GL dispatches the VMs among the GMs. However, before a VMs can be dispatched to GMs, VM networking needs to be managed in order for the VMs to become reachable to the outside world after its startup. This process involves two steps: (1) getting an IP address assigned to the VM; (2) configuring the network interface based on the assigned IP. The GL is in charge of the former step. Therefore, it maintains a system administrator configurable subnet from which it is allowed to assign IP addresses. When VMs are submitted to the GL, each of them automatically gets an IP address assigned from this subnet. The assigned IP address is embedded in the VMs MAC address. When the VM boots it decodes the IP from its MAC address and performs the network configuration.

The GL does not maintain a global view of the VMs in the system. Instead, after the VM dispatching, information about the GMs on which the VMs were dispatched is stored on the client-side thus allowing the clients to directly interact with the corresponding GMs for subsequent VM management requests. Despite the lightweight VM dispatching decisions and no global view of the VMs, the GL scalability can be further improved with replication and a load balancing layer.

3.2.2.4 Entry Points

In Snooze, the GL is automatically elected among the GMs during the system startup and in case of a GL failure. Consequently, a GL can change over time. In order for the clients to instruct a GL to start VMs, they must be provided a way to *discover the current GL*. In order to achieve this we introduce a predefined number of EPs. EPs are system services which typically reside on nodes of the same network as the GMs and have a notion to remain updated about the current GL (see the following section for more details). EPs are contacted by the clients whenever they need to submit VMs to discover the current GL.

3.3 Hierarchy Management

This section describes how the Snooze system hierarchy is constructed and maintained. First, the heartbeat mechanisms are introduced. Then, the self-configuration and healing mechanisms are presented. Self-configuration refers to the ability of the system to dynamically construct the hierarchy during startup. On the other hand, self-healing allows to automatically reconstruct the hierarchy in event of system services or node failures.

3.3.1 Heartbeats

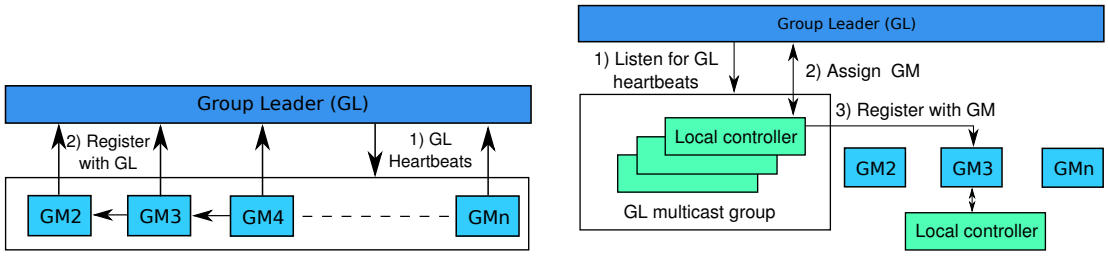
To support self-configuration and healing, Snooze integrates bi-directional heartbeat protocols at all levels of the hierarchy. The GL periodically sends its identity to a dedicated GL heartbeat multicast group containing all EPs and GMs. GL heartbeats allow the EPs to remain updated about the current GL. GL heartbeats are also required by the LCs and GMs to discover the current GL during boot time and in the event of GM (resp. GL) failures. Indeed, the current GL needs to be discovered by the LCs upon initialization in order to get a GM assigned. On the other hand, GMs must inform the current GL about their presence in order to enable VM dispatching by the GL. In the event of a GM failure, LCs need to contact the GL in order to get a new GM assigned. Finally, in the event of a GL failure, GMs need to inform the newly elected GL about their presence.

One heartbeat multicast group exists per GM on which it announces its presence to its assigned LCs. It is used by the LCs to detect a GM failure. Finally, in order for the GL and GMs to detect GM (resp. LCs) failures unicast-based heartbeats are used. They are piggybacked by the GMs and LCs along with their monitoring data which is periodically sent to the GL (resp. GM).

3.3.2 Self-Configuration

When a system service is started on a node it is statically configured to become either a LC or a GM. When the services boot, the first step in hierarchy construction involves the election of a GL among the GMs. After the GL election, other GMs need to register with it. For a LC to join the hierarchy, it first needs to discover the current GL and get a GM assigned by contacting the GL. Once it is assigned to a GM, it can register with it. We now describe all these steps in more details.

When a GM starts, the GL election algorithm is triggered. Currently, our GL election algorithm is built on top of the highly available and reliable coordination system Apache ZooKeeper [174]. It follows the recipe proposed by the ZooKeeper community in [66]. The key idea of the algorithm is to construct a chain between the GMs in which each GM watches its predecessor GM. The GM with the lowest Unique Identifier (UID) becomes the GL. GL election works as follows. Each GM contacts the ZooKeeper service upon startup to get a UID and have the ZooKeeper service create an entry associating the GM UID with its network address. The ZooKeeper service guarantees that the first GM contacting it receives the lowest UID. After the entry creation each GM first attempts to find an entry in the ZooKeeper service with the next lower UID. To do so, a list of entries is retrieved by leveraging the ZooKeeper API and sorted by the GM in decreasing order. If an entry with a lower UID exists, the GM starts watching it and joins the GL heartbeat multicast group to get informed



(a) Group manager join

(b) Local controller join

Figure 3.2: Snooze self-configuration: GM and LC join example

about the current GL, otherwise it becomes the GL and starts announcing its presence by sending GL heartbeat messages on the GL heartbeat multicast group. Once the GMs which were not promoted to be a GL receive a GL heartbeat message, they register with the GL by sending their UID and network address (see Figure 3.2(a)).

The join process of a LC works as follows. Each time a LC starts it subscribes to the GL heartbeat multicast group in order to discover the current GL. When a GL heartbeat message arrives, the actual join process is started by sending a GM assignment request to the GL. The GL distinguishes between two scenarios. In the first scenario a LC joins the system as part of its usual boot process. In the second scenario a LC joins the system as the result of a power management activity. For instance, when a GM wakes up an LC which was previously in a power saving state. To support the former scenarios the LC can be assigned to any GM. However, to enable the latter scenario, the LC *must be assigned to exactly the same GM which triggered the LC wake up*. Otherwise, the GM will not be able to start VMs on the newly woken up LC. Indeed, the LC will be out of its management scope. In order to avoid such a situation when a GL receives a LC to GM assignment request it queries the GMs for the status of the joining LC. In case any GM replies to the GL that it was previously in charge for the joining LC, the GL returns the network address of the GM to the LC, which then initiates the actual GM registration process by sending its description (i.e. UID, network address, available capacity) to the GM (see Figure 3.2(b)). Finally, the LC unsubscribes from the GL heartbeat multicast group and starts listening for its assigned GM heartbeat messages to detect GM failures. Moreover, it periodically sends VM resource utilization data to its assigned GM.

In case none of the GMs was previously in charge for the LC, the GL triggers the system administrator selected GM assignment policy which assigns the LC to a GM according to its high-level objective (e.g. round robin). The network address of the assigned GM is returned to the LC and the previously introduced GM registration procedure is started.

The aforementioned mechanisms allow Snooze to provide autonomy via self-configuration thus significantly reducing the system configuration efforts. Indeed, the hierarchy is constructed fully automatically without human intervention.

3.3.3 Self-Healing

Self-healing is performed at all levels of the hierarchy. It involves the detection of and recovery from LC, GM, and GL failures.

LC failures are detected by their assigned GM based on a unicast heartbeat timeout.

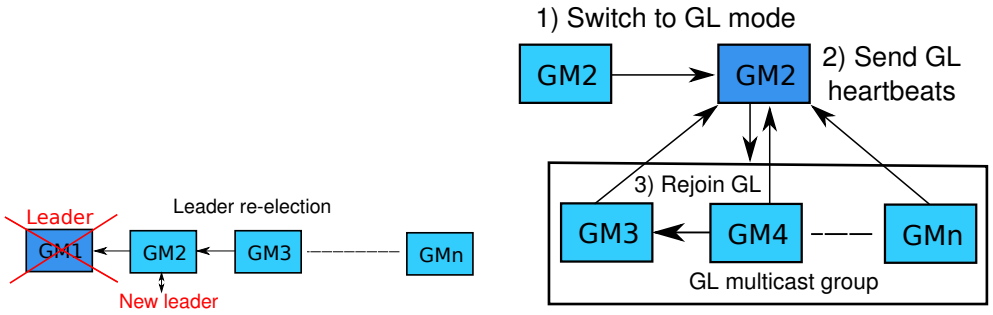
Once a LC failure is detected, the GM gracefully removes the failed LC from its repository in order for it not to be considered in future VM management tasks. Note, that in the event of a LC failure, the VMs it hosts are terminated. IP addresses assigned to the terminated VMs must be recycled by the GL in order to avoid IPs leakage. Therefore, IPs of the terminated VMs are added to a list of recyclable addresses which exists on each GM. This list is periodically sent to the GL. Upon reception of the terminated VM IPs, the GL adds the IPs back to its pool of managed addresses. Note, that for the time being Snooze does not handle the recovery of terminated VMs. However, snapshot features of hypervisors can be used by LCs in order to periodically take VM snapshots. This will allow the GM to reschedule the failed VMs on its remaining active LCs.

GM failures are detected by the GL and LCs based on unicast (resp. multicast) heartbeat timeouts. When a GM fails all its knowledge (e.g. VMs and their assigned IP addresses, LC network addresses) is lost and the GM is removed from the GL repository in order to prevent the GL from sending VM submission requests to dead GMs. LCs which were previously managed by the failed GM start the rejoin procedure. Similar to the join procedure a rejoin involves the assignment of a GM to the LC. However, during the rejoin, LCs knowledge about the currently running VMs must be transferred to the newly assigned GM. Otherwise, clients will fail to manage their VMs (see Section 3.5.2 for more details). Finally, as every GM is beeing watched by its successor GM in the ZooKeeper service, a failure event is triggered on the successor GM. Upon a GM failure the successor GM relies on the Apache ZooKeeper service in order to discover a new predecessor GM and start watching it. This is achieved by using the GM discovery procedure described in Section 3.3.2. Note, that in case this procedure is not done, due to the nature of the GL election algorithm a successor GM will never be able to become a GL in the future (see the GL failure recovery description below).

GL failure is detected by its successor GM based on a timeout triggered by the ZooKeeper service. When a GL fails, first a new GL must be elected among the GMs. The newly elected GL then must be discovered and joined by the remaining GMs. Moreover, LCs which were previously assigned to the GM becoming the new GL must rejoin the hierarchy. Note, that in the event of a GL failure all its knowledge about the existing GMs as well as the VM networking information (i.e. assigned IP addresses) is lost. Consequently, this knowledge must be rebuilt in order for the system to remain in a consistent state. This steps are achieved as follows.

Upon GL failure, the successor GM in the chain becomes the new GL as its UID is the next lowest one (see Figure 3.3(a)). When a GM is promoted to be a GL, it gracefully terminates all its tasks such as the heartbeat and summary information sending. Moreover, it cleans its internal LC and VM knowledge. Afterwards, the GM switches to GL mode, and starts sending its network address to the GL heartbeat multicast group. The switch to GL mode yields two effects: (1) the newly elected GL must be detected and joined by the remaining GMs; (2) LCs which were previously managed by the GM must rejoin the hierarchy to another GM.

In order to detect the new GL, GMs always keep listening for GL heartbeat messages. Upon reception of a new GL network address they trigger the GL rejoin procedure (see Figure 3.3(b)). In contrast to the previously introduced GM join procedure, a GM rejoin requires additional data to be sent. Particularly, in order to rebuild GLs VM networking knowledge, each time a GM registers with a new GL it attaches IPs of its active VMs to its description thus allowing the GL to reconstruct its view of already assigned IPs. Moreover, GM summary information is periodically sent back to the new GL (see Section 3.4.2 for more



(a) Group manager failure (b) Group leader switch

Figure 3.3: Snooze self-healing: GM failure and GL switch example

details) thus allowing it to rebuilt its GM resource utilization knowledge.

Finally, as a GM has been promoted to become the new GL it stopped sending heartbeat messages to its heartbeat multicast group. LCs which were previously assigned to it thus fail to receive the heartbeat messages and consider it as failed. They rejoin the hierarchy by following the GM failure recovery procedure.

The aforementioned mechanisms enable Snooze to satisfy the *self-healing property of autonomic computing systems*. Particularly, they allow Snooze to automatically detect system component failures and repair the system in order to continue normal functioning without external intervention.

3.4 Energy-Efficient VM Management

We now present the core VM management mechanisms and algorithms of Snooze. Snooze provides a holistic energy management solution for IaaS clouds by integrating VM resource utilization monitoring and estimations, underload/overload mitigation mechanisms, and VM consolidation, and finally power management within one system. Power management is used to transition idle PMs in to a power saving state during periods of low utilization. In this section first, the notations and metrics are introduced. Then, we describe the VM resource utilization monitoring and estimation, VM dispatching and placement, LC overload and underload mitigation, migration plan enforcement, and finally the power management mechanisms.

3.4.1 Notations and Metrics

Let LCs denote the set of LCs and VMs the set of VMs, with n = $|LCs|$ and $m = |VMs|$ representing the amounts of LCs and VMs, respectively.

Available resources, CPU, memory, network Rx, and network Tx are part of the set R with d = $|R|$ (d = 4). VM CPU utilization is measured in *percentage of the total LC capacity*. For example, if a LC has four physical cores (PCORES) and a given VM requires two virtual cores (VCORES), the maximum CPU requirement of the VM would be 50%. Memory is measured in KiloBytes and network utilization in Bytes/sec.

VM is represented by its requested and used capacity vectors ($\mathbf{RC_v}$ resp. $\mathbf{UC_v}$). $\mathbf{RC_v} := \{RC_{v,k}\}_{1 \leq k \leq d}$ reflects the VM resource requirements at submission time. Each vector component defines the requested capacity for resource k $\in$ R. The used capacity vector $\mathbf{UC_v} := \{UC_{v,k}\}_{1 \leq k \leq d}$ contains estimated VM resource utilization at a given time. It is computed based on the monitored VM resource utilization.

LCs are assigned with static and used capacity vectors. The static capacity vector represents the total amount of resources available on a LC l. It is defined as $\mathbf{C}_l := \{C_{l,k}\}_{1 \leq k \leq d}$. Used capacity vector represents the estimated LC resource utilization. It is defined as $\mathbf{c}_l$ and is computed by summing up the used VM capacity vectors: $\mathbf{c}_l := \sum_{\forall v \in LC_l} \mathbf{UC}_v$.

We introduce $\mathbf{M}_l := \{MID_{l,k}\}_{1 \leq k \leq d}$ as the node resource capping vector which puts an *upper bound on the maximum aimed LC utilization for each resource k* with $0 \leq MID_{l,k} \leq 1$. In other words we keep a limited amount of available resources to compensate for overprovisioning. This is required in order to mitigate performance problems during periods of high resource contention. Once M_l is in place, the static LC capacity vector is computed as $\mathbf{C_l} := \mathbf{C}_l \diamond \mathbf{M}_l$. $\diamond$ denotes elementwise vector multiplication. LC_l is considered to have enough capacity for VM_v if either $\mathbf{c}_l + \mathbf{RC_v} \leq \mathbf{C_l}$ holds during VM placement or $\mathbf{c}_l + \mathbf{UC_v} \leq \mathbf{C_l}$ during overload/underload mitigation or consolidation.

Introducing resource utilization upper bounds leads to situations where VMs can not be hosted on LCs despite enough resources being available. For example when $MID_{l,CPU} = 0.8$ and only two PCORES exist, VM requiring all of them can not be placed (i.e. 2 VCORE / 2 PCORE $\leq$ 0.8 does not hold). Therefore, we define the notion of packing density (PD) which is a vector of values between 0 and 1 for each resource k. It can be seen as the *trust given to the user's requested VM resource requirements* and allows VMs to be hosted on LCs despite existing MID capping's. When PD is enabled, Snooze computes the requested VM resource requirements as follows: $\mathbf{RC_v} := \mathbf{RC_v} \diamond \mathbf{PD}$.

LCs, GMs, and VMs need to be sorted by many VM management algorithms. Sorting vectors requires them to be first normalized to scalar values. Different sort norms such as L1, Euclid or Max exist. In this work the L1 norm [294] is used.

3.4.2 Resource Monitoring and Estimations

VM and LC resource utilization changes over time. In order to support VM management decisions such as VM dispatching, placement, underload/overload mitigation, and VM consolidation, VM and LC monitoring is performed at all layers of the system. At the computing layer VMs are monitored and VM resource utilization is periodically transferred by the LCs to the assigned GM.

At the management layer, GMs periodically send summary information to the GL. GM summary information includes the aggregated resource utilization for all the LCs a GM manages. Aggregated resource utilization captures the total active, passive, requested and used capacity of a GM. Aggregated resource utilization is used by the GL to guide VM to GM dispatching decisions. Active and passive capacity are static vectors that represents the total amount of LC resources available on active (resp. passive) LCs. Active and passive LCs are nodes which are powered on (resp. power-cycled). Active and passive capacity vectors are computed by summing up the C_l vectors of all the powered on (resp. power-cycled) LCs. Requested capacity is a vector that represent the total amount of LC resources requested by

the VM at submission time. It is computed by aggregating the $\mathbf{RC_v}$ vectors of all the VMs. Used capacity is a vector that represents the total amount of used LC resources as estimated by the LC. It is computed by summing up the $\mathbf{c_l}$ vectors of all the active LCs.

Finally, VM resource utilization estimations are essential for most of the tasks involved in VM management on a GM. For instance, a GM requires the knowledge of the used capacity of its assigned LCs when generating the GM summary information to be sent to the GL generation. The overload and underload mitigation detection decisions are based on the LC used capacity vectors. Finally, VM used capacity vectors are required by all of the VM management algorithms in order to sort LCs and VMs. VM used capacity can be either computed by simply considering the average of the n most recent VM resource utilization values for each resource k. Alternatively, more advanced algorithms (e.g. based on Autoregressive-Moving-Average) can be used. In this work the former approach is taken.

3.4.3 VM Dispatching and Placement

When a client attempts to submit VMs to the GL, a dispatching policy is used to distribute them among the GMs. For example, VMs could be distributed in a capacity-aware round-robin or first-fit fashion. A dispatching policy takes as input the submitted VMs and the list of available GMs including their associated aggregated resource utilization data and outputs a dispatching plan which specifies the VM to GM assignments. Particularly, the dispatching policy assigns sets of VMs to GMs. Priority is given to assign VMs to GMs with enough available active capacity in order to minimize the number of passive LCs to be woken up by the GMs during VM placement. Note, that aggregated resource utilization is not sufficient to take *exact dispatching decisions*. For instance, when a client submits a VM requesting 2GB of memory and a GM reports 4GB available it does not necessary mean that the VM can be finally placed on this GM as its available memory could be distributed among multiple LCs (e.g. 4 LCs with each 1 GB of RAM). Consequently, a *list of candidate GMs* is provided by the VM dispatching policy. Based on this list, a linear search is performed by the GL during which it sends VM placement requests to the GMs.

Finally, once a GM receives a request from the GL to start VMs it triggers the VM placement algorithm to compute a VM to PM allocation. At this stage any traditional VM placement algorithm such as Round-Robin or First-Fit Decreasing (FFD) can be used. The current Snooze implementation integrates both the algorithms. It is up to the system administrator to choose which algorithms will be used to perform VM dispatching and VM placement.

3.4.4 Overload and Underload Mitigation

Because of the fluctuating VM resource consumption and the systems ability to overcommit resources, resource contention can occur when the aggregated resource utilization of the VMs exceeds the total LC capacity, the so-called *overload* condition. Moreover, for energy efficiency reasons, once a LC has become idle (i.e. *underloaded*) it could be transitioned into a lower power-saving state to save energy. In order to handle both cases, Snooze integrates overload and underload mitigation mechanisms which involve the detection and resolution of overload (resp. underload) situations.

Overload and underload detection is performed locally by each LC. For each VM a system administrator predefined amount of resource (i.e. CPU, memory, network Tx, network

Rx) utilization vectors is first collected based on which the average VM resource utilization vector is computed by the LC. Computing the *average VM resource utilization* allows the system to avoid many false-positive overload/underload alerts which would have been triggered due to transient resource utilization spikes if instant values were taken. In other words, computing the average VM resource utilization attempts to smooth the VM resource utilization data. After the LC has computed the average VM resource utilization for each managed VM, it computes the used LC capacity vector by summing up the average VM resource utilization vectors and applies a Threshold Crossing Detection (TCD) mechanism on it. The TCD mechanism defines two thresholds: $0 \leq MIN_k \leq 1$ and $0 \leq MAX_k \leq 1$ for each resource k and applies them on the used LC capacity vector. If the estimated resource utilization for at least one k falls below MIN_k the LC is considered as underloaded, otherwise if it goes above MAX_k, the LC it is flagged as overloaded (see Figure 3.4).

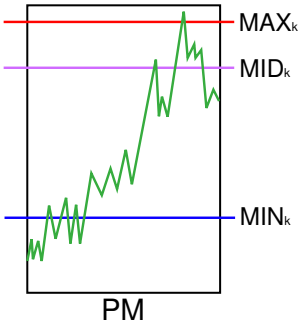


Figure 3.4: Snooze thresholds example

In the event of an overload situation VMs must be moved to a more lightly loaded nodes in order to mitigate performance degradation. In the event of an underload situation, for energy saving reasons it is beneficial to move VMs to moderately loaded LCs in order to transition the underutilized LCs into a lower power state. Consequently, two types of mitigation policies are supported at the GM level: overload and underload mitigation. In order to compute solutions, in a reasonable amount of time both policies are currently implemented using greedy algorithms with a polynomial worst-case complexity.

The Snooze VM overload mitigation algorithm is shown in Algorithm 1. The key idea of the algorithm is to move just as many VMs away as needed to resolve the overload situation. The algorithm takes as input the overloaded LC along with its associated VMs and a list of LCs managed by the GM. It outputs a migration plan. The overload mitigation policy first estimates the used LC capacity and computes the LC static capacity vector. It then generates the overloaded capacity delta (i.e. difference between used and static capacity vector). Afterwards, it gets the VMs assigned to the overloaded LC, sorts them in increasing order based on used capacity and computes a list of candidate VMs to be migrated. The routine to compute the migration candidates first attempts to find the most loaded VM among the assigned ones whose used capacity equals or is above the overloaded capacity delta. This way a single migration will be sufficient to move the LC out of the overload state. Otherwise, if no such VM exists, it starts adding VMs to the list of migration candidates starting from the least loaded one until the sum of the used VM resources equals or is above the overload capacity delta. Finally, the destination LCs are sorted in increasing order based on used capacity and their status (i.e. active, passive). The sorting procedure favours active

LCs over passive ones. This away useless PM wake ups are avoided. Migration candidates are assigned to LCs starting from the first one if enough capacity is available. Moreover, the new VM to LC mappings are added to the migration plan. In case PMs need to be woken up, Snooze first performs the PM wake ups using the appropriate system administrator specified mechanisms (e.g. IPMI [35], Wake-on-Lan (WOL) [65]). It then enforces the migration plan by performing the VM live migrations.

Algorithm 1 Snooze VM overload mitigation

1: **Input:** Overloaded **LC** with the associated VMs and resource utilization vectors **UC**, list of destination **LCs**
2: **Output:** Migrating Plan **MP**
3: **c** ← Estimate used LC capacity
4: **m** ← Compute static LC capacity
5: **o** ← Compute the amount of overloaded capacity (**c**, **m**)
6: VMs_{source} ← Get VMs from LC
7: Sort VM_{source} in increasing order
8: $VM_{candidates}$ ← computeMigrationCandidates(VMs_{source}, **o**)
9: Sort destination LCs in increasing order
10: **for all** $v \in VM_{candidates}$ **do**
11: LC_{fit} ← Find LC with enough capacity to host v (v, LCs)
12: **if** $LC_{fit} = \emptyset$ **then**
13: continue;
14: **end if**
15: Add (v, LC_{fit}) mapping to the **MP**
16: **end for**
17: **return MP**

The underload mitigation algorithm is shown in Algorithm 2. In contrast to the overload mitigation algorithm, the underload mitigation does not require to compute a VM migration candidate set. Instead, the algorithm follows an *all-or-nothing approach* in which either all or none of the VMs executing on a node are moved. Moving a subset of VMs does not contribute to the energy saving objective (i.e. create idle times) and thus is avoided. The algorithm takes as input the underloaded LC and its associated VMs along with the list of LCs managed by the GM. It first retrieves the VMs from the underloaded LC and sorts them in decreasing order based on the used capacity. Similarly, LCs are sorted in decreasing order based on the used capacity. Then, VMs are assigned to LCs with enough spare capacity and added to the migration plan. If some VMs could not be assigned the migration plan is cleaned and the algorithm is aborted. Otherwise, a non-empty migration plan is returned. Snooze enforces the migration plan via VM live migration and transitions the PM in to a lower system administrator specified power state (e.g. shutdown) once all VMs have been migrated to save energy.

Note that despite our efforts to mitigate the effect of transient resource spikes via aggregation of VM resource utilization data, the proposed mechanisms have a cost in terms of energy and performance (i.e. executing time). For instance, when an overload situation occurs, VMs need to be migrated from the overloaded node to less loaded nodes. VM live migration requires resources on the source node as well as on the destination nodes. This it can negatively impact the performance of collocated VMs on the nodes involved in live mi-

Algorithm 2 Snooze VM underload mitigation

1: **Input:** Underloaded **LC** with the associated VMs and resource utilization vectors **UC**, list of destination **LCs**
2: **Output:** Migration Plan **MP**
3: $VM_{candidates} \leftarrow$ Get VMs from underloaded LC
4: Sort $VM_{candidates}$ in decreasing order
5: Sort LCs in decreasing order
6: **for all** $v \in VM_{candidates}$ **do**
7: $LC_{fit} \leftarrow$ Find LC with enough capacity to host v
8: **if** $LC_{fit} = \emptyset$ **then**
9: Clear **MP**
10: break;
11: **end if**
12: Add (v, LC_{fit}) mapping to the **MP**
13: **end for**
14: **return MP**

gration. Moreover, VM live migration also has an impact on the performance of applications hosted on the VMs. Consequently, the benefits of the underload and overload mitigation mechanisms heavily depend on the application workloads. Nevertheless, as we our evaluation will show, energy savings can be achieved especially in the context of web applications.

3.4.5 VM Consolidation

In the previous section we have introduced the Snooze overload and underload mitigation mechanisms. While such mechanisms are certainly important to resolve underload and overload situations, resource fragmentation can still happen due to differences in VM resource demands when nodes are neither underutilized nor overloaded. In Snooze each GM integrates a VM consolidation engine which can be enabled by the system administrator to periodically perform these tasks. Consolidation is performed by GMs concurrently and independently within their set of PMs. Note, that VM consolidation has a cost in terms of both energy and performance (i.e. execution time) on the overall system (resp. applications inside the VMs). Consequently, choosing the appropriate interval is crucial in order to achieve both, energy savings and limited impact on application performance. For instance, performing VM consolidation should be avoided during periods of high utilization.

VM consolidation is a multi-objective variant of the multi-dimensional bin-packing problem which is known to be NP-hard [284, 266]. Snooze is not limited to any particular VM consolidation algorithm. Indeed, thanks to its flexible design any VM consolidation algorithm can be integrated. However, because of the NP-hard nature of the problem and the need to compute solutions in a reasonable amount of time it currently implements a simple yet efficient two-objective polynomial time greedy consolidation algorithm which minimizes the number of LCs along with the number of migrations. Particularly, a modified version of the Sercon [224] algorithm is integrated which differs from the original one in its termination criteria and the number of VMs which are removed in case not all VMs could be migrated from a LC. Sercon follows an all-or-nothing approach and attempts to move VMs from the least loaded LC to a non-empty LC with enough spare capacity. Either all VMs can be migrated from a host or none of them will be. Migrating only a subset of VMs does not yield

to a smaller number of LCs and thus is avoided. The pseudocode of the modified algorithm is shown in Algorithm 3.

Algorithm 3 Snooze VM consolidation

```
1: Input: List of LCs with their associated VMs and resource utilization vectors UC
2: Output: Migration Plan MP, nUsedNodes, nReleasedNodes
3: MP ← ∅
4: nUsedNodes ← 0
5: nReleasedNodes ← 0
6: localControllerIndex ← |LCs| − 1
7: while true do
8:    if localControllerIndex = 0 then
9:       break;
10:   end if
11:   Sort LCs in decreasing order
12:   LC_least ← Get the least loaded LC (localControllerIndex)
13:   VMs_least ← Get VMs from LC_least
14:   if VMs_least = ∅ then
15:      localControllerIndex ← localControllerIndex - 1
16:      continue;
17:   end if
18:   Sort VMs_least in decreasing order
19:   nPlacedVMs ← 0
20:   for all v ∈ VMs_least do
21:      Find suitable LC to host v
22:      if LC = ∅ then
23:         continue;
24:      end if
25:      LC_least ← LC_least ∪ {v}
26:      Add (v, LC_least) mapping to the MP
27:      nPlacedVMs ← nPlacedVMs + 1
28:   end for
29:   if nPlacedVMs = |VMs_least| then
30:      nReleasedNodes ← nReleasedNodes + 1
31:   else
32:      Remove VMs_least from LC_least and MP
33:   end if
34:   localControllerIndex ← localControllerIndex - 1
35: end while
36: nUsedNodes ← |LCs| - nReleasedNodes
37: return MP, nUsedNodes, nReleasedNodes
```

The algorithm receives the LCs including their associated VMs. LCs are first sorted in decreasing order based on their used capacity. Afterwards, VMs from the least loaded LC are sorted in decreasing order, placed on the LCs starting from the most loaded one and added to the migration plan. If all VMs could be placed the algorithm increments the number of released nodes and continues with the next LC. Otherwise, all placed VMs are removed from the LC and migration plan. The procedure is then repeated with the next loaded LC. The algorithm terminates when it has reached the most loaded LC and outputs the migration plan, number of used nodes, and number of released nodes.

3.4.6 Migration Plan Enforcement

Overload and underload mitigation as well as VM consolidation algorithms output a migration plan which specifies the new mapping of VMs to LCs. This mapping is used by Snooze to transition the system from its current to the optimized state. The migration plan is enforced only if it *potentially yields less active LCs*. Enforcing the migration plan computed is straightforward as it only involves moving VMs from their current location to the given one. Note that our algorithms do not introduce any sequential dependencies or cycles of VM migrations. Particularly, *VMs are migrated to an LC if and only if enough capacity is available on it without requiring other VMs to be moved away first*.

3.4.7 Power Management

In order to save energy, idle nodes need to be transitioned into a lower power state after the migration plan enforcement. Therefore, Snooze integrates a power management service, which can be enabled by the system administrator to periodically observe the LC utilization and trigger power-saving actions once they become idle.

The following power saving actions can be enabled if hardware support is available: shutdown, suspend to ram, disk, or both. Different shutdown and suspend drivers can be plugged in to support any power management tool. For example, shutdown can be implemented using IPMItool [201] or by simply calling the Linux native shutdown command. To enable PM power on, wake up drivers exist. Currently, two wake up mechanisms are supported in Snooze: IPMI and WOL.

Power management works as follows. Snooze can be configured by the system administrator to keep a number of reserved LCs always on in order to stay reactive during periods of low utilization. Other LCs are automatically transitioned into a lower power state after a predefined idle time threshold has been reached (e.g. 180 sec) and marked as passive. Passive LCs are woken up by the GMs either upon VM submission or overload situation when not enough active capacity is available to accommodate the VMs. In both cases when a GM triggers a passive LC wake up it must to wait for the LCs to come online and register with the GM until it can attempt to start designated VMs on them, otherwise the VM start requests will fail. Indeed, the LCs will be not reachable. In this context two aspects are important: (1) a time interval needs to be defined to inform the GM for *how long it needs to wait* until it can attempt to start VMs on the woken up LCs; (2) in order for the GM to successfully start VMs on the woken up LCs, *LCs must be assigned to exactly the same GM which triggered the wake ups*. In order to enable the former aspect a system administrator configurable wake up timeout exist on each GM. To support the latter aspect, the GL is designed to assign LCs to the same GM which triggered the wake up (see Section 3.3.2).

Finally, care must be taken during the VM submission when the GL computes a VM to GM assignment based on the aggregated GM resource utilization information. Depending on the implemented VM placement algorithm, computing the VM to GM assignment can take considerable amount of time. During this time, concurrently operating power management modules on the GMs can detect and transition idle PMs into a low-power state even though the PMs will be required shortly after the VM to GM assignment computation. In order to prevent such race conditions, a GL freezes the GMs power managers prior to computing the VM to GM assignment and unfreezes them afterwards.

3.5 Implementation

We have developed a Snooze prototype from scratch in Java. It currently comprises approximately 15 000 lines of highly modular code. This section presents a few but important implementation aspects. They involve the VM life-cycle management and monitoring, command line interface (CLI), asynchronous VM submission processing, live migration convergence enforcement, and repositories. Note, that the Snooze is programmed in a highly modular manner and thus does not depend on the choices made so far (e.g. a particular database). Since May 2012 Snooze is distributed in open-source under the GPL v2 license at http://snooze.inria.fr.

3.5.1 VM Life-Cycle Enforcement and Monitoring

Different tools can be used to control the VM life-cycle management as well as to collect monitoring information. For example, life-cycle management commands can be either enforced by directly contacting the hypervisor API or using an intermediate library such as libvirt [250], which provides a uniform interface to most of the modern hypervisors. Similarly, VM resource utilization data can be obtained from different sources (e.g. libvirt, Ganglia [5], Munin [41]). Snooze provides abstractions to integrate any VM life-cycle management and monitoring solution but currently relies on libvirt. Libvirt is particularly interesting as it can be used for both, VM life-cycle management and transparent VM resource utilization monitoring. This way dependencies on third party tools (e.g. Ganglia) can be minimized.

3.5.2 Command Line Interface

A Java-based CLI is implemented on top of the RESTful interfaces exported by Snooze system services. The CLI supports the definition and management of virtual clusters (VCs). VCs are constructed on the client side to represent collections of one or multiple VMs. VCs are used by the CLI to group VMs and perform collective VM commands such as startup of multiple VMs. Moreover, visualizing and exporting the current hierarchy organization in the GraphML format is supported by the CLI. Both, the visualization and the exporting are implemented using the Java Universal Network/Graph framework [184]. Before VMs can be submitted into the system a VC needs to be defined and filled with at least one VM by specifying a path to the VM description. Currently, the libvirt [250] templates are accepted by the CLI. The meta-information (e.g. name, VM template paths) about VCs is store locally on the client side in the Extensible Markup Language (XML) format.

When a user attempts to start a VC, the CLI first transparently discovers an active EP by walking through the EPs list specified in its configuration file. This list must be statically configured during the CLI setup. Given that an active EP exists, a GL lookup is sent in order to receive the current GL network address. Finally, the request to submit the VC is delegated to the GL which dispatches the VMs on the available GMs. The response provides the following information: assigned VM IP addresses, network addresses of the GMs managing the VMs, VM status (e.g. RUNNING) and an error code which indicates problems during the submission. Examples of problems include not enough compute resource or errors in the VM description (e.g. bad path to the VM disk image). The CLI displays the response to the user. The GM network addresses on which the VMs were dispatched are stored in the CLI

repository, thus allowing the CLI to directly contact the GM whenever VC/VM management commands need to be performed.

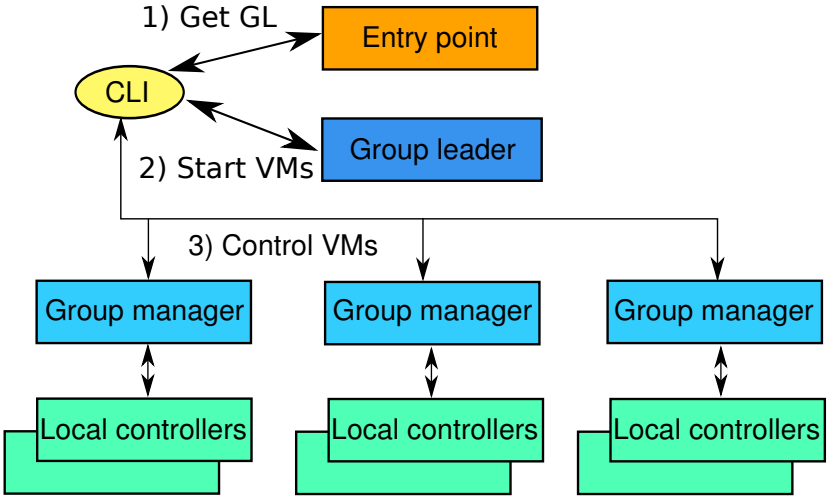


Figure 3.5: Snooze VM submission example

The following VM management commands are currently supported: start, destroy, shutdown, suspend, resume and info. All commands can be executed either for an individual VM or a VC. When applied to a VC, the command is applied to all VMs belonging to the VC. The info command allows to retrieve the VM information which includes VM UID, assigned IP address, network address of the LC and GM in charge of managing the VM, and the VM status. Finally, it is important to mention that on GM failures the CLI repository information becomes obsolete. When the CLI attempts to contact a GM managing a given VM which is not reachable, it queries the EP in order to discover the current GL. Afterwards, a GM discovery request including the VM UID is sent to the GL. Upon reception of the request, the GL queries the currently active GMs in order to find the one assigned to the VM, and returns the result to the CLI. Thus the VM management command can be performed on the new GM. On the other hand, in the event of an LC fails all the knowledge about the LC including its VMs is removed from the GM. In case the client contacts a GM which does not have knowledge about a certain VM it responds to the client that with the appropriate error code (i.e. VM information not available).

3.5.3 Asynchronous VM Submission Processing

In order to stay scalable with an increasing number of client requests, VM submissions are processed asynchronously by the GL as well as the GMs. Particularly, when any client software attempts to submit VMs to the GL, it receives a task identifier (TID) and is required to periodically poll the GL for the response (i.e. long-polling design pattern). VM submission requests are queued on the GL and processed sequentially. for each VM submission request, the GL performs the VM dispatching on behalf of the client by instructing the GMs to place the VMs. Thereby, it receives a TID from each of the GMs upon VM placement request submission and polls the GMs for responses. GMs perform the VM placement tasks on behalf of the GL. When a GL collects responses from all GMs it associates a submission

response with the client TID and the request is considered as finished.

Note that, client VM submission requests are queued on the GL and processed sequentially. Each GM implements a state machine which starts rejecting VM submission requests when it becomes busy. A GM is considered busy when it performs one of the following tasks: *GL request processing, overload/underload mitigation, VM consolidation, or power management*. The GL periodically retries sending VM placement requests to GMs according to the system administrator specified interval and number of retries. An error code is associated with the client request by the GL if after this period no VMs could be placed. This way infinite loops are avoided.

3.5.4 Live Migration Convergence Enforcement

Snooze overload/underload mitigation and VM consolidation algorithms require VM live migrations. Migrations can happen either sequentially or in parallel. In the former case only one VM is moved from the source to the destination LC at a time, while the latter allows multiple VMs to be migrated concurrently. Given that modern hypervisors (e.g. KVM) support parallel migrations there is no reason not to do so given that enough network capacity is available. This is what Snooze does.

VM live migration involves the transfer of a large amount of memory pages across the data center network links. Depending on the hypervisor, the live migration technique and the live migration termination criteria it can take a significant amount of time to finish the migration of the VMs to be consolidated. For example, in KVM live migration can last forever if the number of pages that got dirty is larger than the number of pages that got transferred to the destination LC during the last transfer period. In order to detect and resolve such situations, for each migration Snooze spawns a *watchdog* thread. Watchdog threads enforce convergence after a system administrator predefined convergence timeout given the migration is still pending. Therefore it *suspends the migrating VM* thus preventing further page modifications. The hypervisor is then able to finish the migration and restart the VM on the destination LC.

3.5.5 Repositories

Each system service implements a repository for data storage. For example, the GL stores GM descriptions, GM resource utilization and VM networking information. Each GM maintains a local view of its managed LCs and their associated VM resource utilization data. LCs store information (e.g. UID, assigned IP address) about the currently running VMs. Snooze is not limited to a particular repository implementation. Consequently, different storage backends (e.g. MySQL [2], Apache Cassandra [198], MongoDB [67]) can be integrated. However, it currently relies on an in-memory storage based on a ring buffer. In other words, the repository keeps a limited amount of data and starts overwriting the least recently used data once the limit is reached. It is up to the system administrator to set the appropriate limit during the Snooze deployment for each GM.

3.6 Evaluation

This section presents the experimental results from the Snooze evaluation. A Snooze prototype was developed and evaluated on the Grid'5000 experimentation testbed. To emphasize the features of Snooze our evaluation is structured into two parts. The first part targets the scalability and autonomy of the system. The second part is devoted to the the energy management mechanisms.

3.6.1 Scalability and Autonomy

In order to test the scalability of the system we have performed two evaluations: (1) VM submission time with increasing number of VMs; (2) amount of heartbeat and resource utilization data; (3) CPU and memory load consumption. VM submission time is an important metric to evaluate as it greatly impacts the users system experience. Heartbeat and resource utilization data are continuously exchanged between the system services (i.e. GL, GM and LC) and thus consumes additional network capacity. Finally, the GL and GM CPU and memory load scalability is evaluated in two scenarios: (1) during VM submission; (2) with increasing number of nodes hosting GMs (resp. LCs) services. The former scenario is important in order to get an insight about how the GL resource requirements scale with a large number of GMs. The latter shows how the GL and GM resource requirements scale with a large number of nodes. Finally, the impact of the autonomy (i.e. self-configuration and healing) mechanisms on the application performance is analyzed by injecting system component failures. Understanding whether the system is able to sustain system component failures and which implications such failures have on the overall application performance is important as the number for software and hardware failures increases at scale. In the following sections we first present the system setup and then discuss the results from the scalability and autonomy evaluation.

3.6.1.1 System Setup

In order to evaluate the scalability and autonomy of the Snooze framework we have deployed it on a 144 nodes cluster of the Grid'5000 experimentation testbed [98]. Each node is equipped with one quad-core Intel Xeon X3440 2.54 GHz CPU, 16 GB of RAM, and a Gigabit Ethernet interconnect. Note, that while the hardware in this experiment is homogeneous nothing prevents the system from beeing deployed in a heterogeneous environment. The operating system on each server is Debian with a 2.6.32-5-amd64 kernel. All tests were run in a homogeneous environment with qemu-kvm 0.14.1 and libvirt 0.9.6-2 installed on all machines. Each VM is using a QCOW2 [3] disk image with the corresponding backing image hosted on a NFS server. Debian is installed on the backing image and uses a ramdisk in order to speed up the boot process. Finally, the NFS server is running on one of the EPs with its file tree being exported to all LCs. During all experiments, the LC assignment, VM dispatching, and VM placement policy were set to *round robin* thus resulting in a *balanced hierarchy* in terms of LC assignments as well as VM locations. Underload (resp. overload) mitigation, VM consolidation, and powered management mechanisms were disabled. Indeed, this study is focused on the scalability and autonomy evaluation of the system.

3. The QCOW2 Image Format - http://people.gnome.org/∼markmc/qcow-image-format.html

3.6.1.2 Submission time: centralized vs. distributed

VM submission time was evaluated by starting a large number of VMs. It is is defined as the time between initiating the VM submission request and receiving the reply on the client side. VM submission involves assigning IP addresses to VMs, dispatching VMs to the GMs, placing VMs on the LCs and returning the response to the client.

Two deployment scenarios were considered: *centralized* and *distributed*. In the former the EP, GL, GM as well as the Apache ZooKeeper service were co-located on a single node while 136 other nodes were hosting the LC services. This allowed us to emulate the traditional frontend/backend-model as close as possible. Emulating the frontend/backend-model is interesting as this is how most of the existing IaaS cloud management systems are designed. Consequently, it can serve as a baseline for comparison with our more distributed approach.

To evaluate the potential overheads of being distributed, Snooze was configured in a distributed manner on 144 nodes. Two nodes served as the EPs with each of the nodes hosting a replica of the Apache ZooKeeper service. Six nodes were used as GMs and each of the remaining 136 nodes had a LC service installed. Note, that one of the GMs became the GL during startup. Finally, in both scenarios a given number of VMs was submitted simultaneously to the system, ranging from 0 to 500. Each VM required one virtual core and 2 GB of RAM. VMs were hosting the OS and basic system services (e.g. sshd). All VM templates and disk images were pre-created on the NFS-server and submissions happened sequentially directly after the predecessor VMs were terminated. 500 VMs were a good tradeoff (i.e. ∼4 VMs per LC) in order not to risk application performance degradation due to possible resource overcommit.

The experimental results of this evaluation are plotted in Figure 3.6. As it can be observed, the submission time increases approximately linearly with the number of VMs in both the centralized and distributed deployment. However, more interesting is the fact that besides minor measurement errors, submission times in both scenarios are nearly equivalent thus indicating the good scalability of the system as *no overhead of being distributed can be observed*. Finally, submission of 500 VMs were finished in less than four minutes which proves that our prototype is robust enough to manage such amounts of VMs.

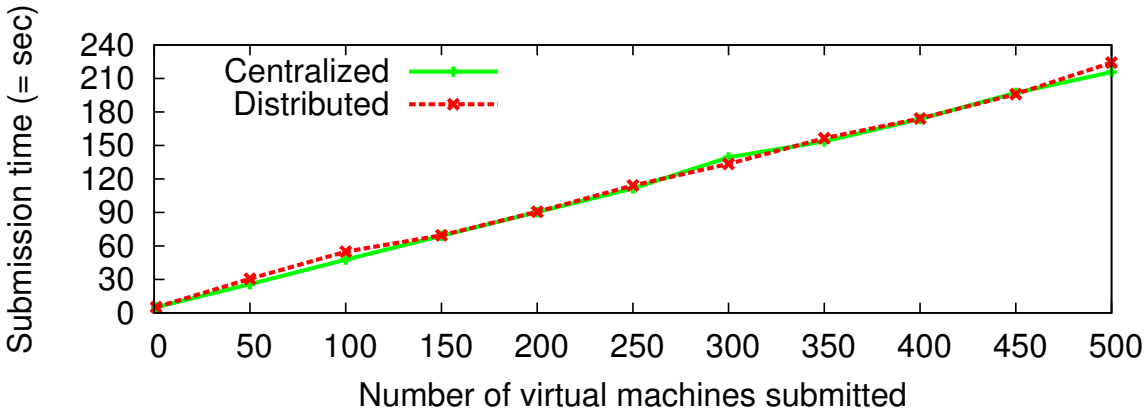


Figure 3.6: Snooze submission time: centralized vs. distributed

3.6.1.3 Heartbeat and Resource Monitoring Information Overhead

We evaluate the heartbeat and monitoring information overhead of the framework by measuring the network utilization at the nodes hosting the Snooze system services. To isolate the heartbeat and monitoring traffic the framework was deployed with one EP, one GL, one GM, and one LC. No VMs were running on the system. Each service was hosted on a dedicated node. Heartbeat intervals of the GL as well as of the GM were set to 3 seconds. Moreover, a fixed-amount of aggregated resource utilization and heartbeat data was sent by the GM (resp. LC) periodically in 10 second intervals. Time intervals were derived empirically. Accounting the monitoring information is important as it is involved in the process of failure-detection (see Section 3.3.1).

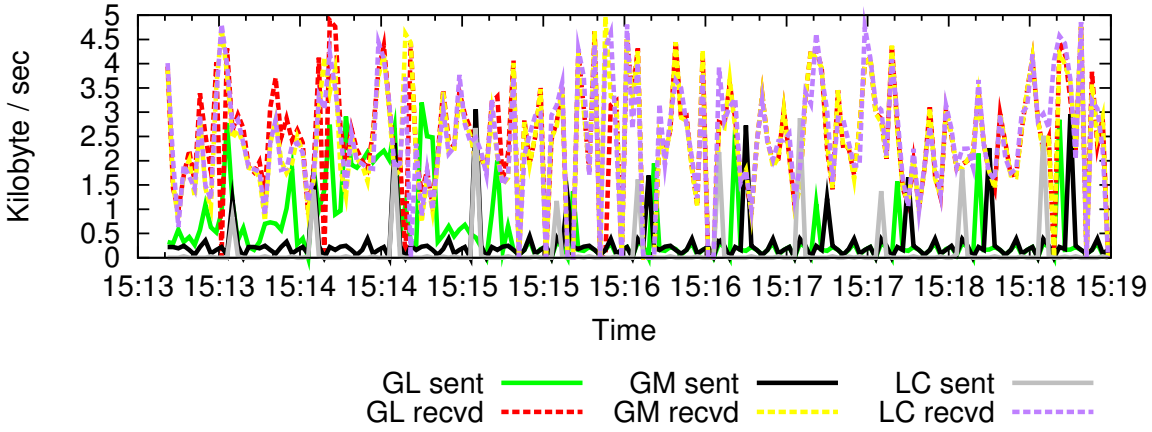


Figure 3.7: Snooze network load scalability

Figure 3.7 depicts the correlated incoming and outgoing network traffic of the nodes hosting the GL, GM and LC services. As it can be observed, the GL heartbeat multicast messages only account to approximately 2.5 kB/s thus not putting any significant pressure on the network. On the other hand, GL incoming traffic is mainly dominated by the received GM summary information which amounts to approximately 4.5 kB/s and is sent using the Transmission Control Protocol (TCP).

When considering the network load scalability of the GMs, heartbeats are sent from the each GM to its LCs and vice versa. Similarly to the GL, GM heartbeat messages are multicast based while LC monitoring information is periodically sent using TCP. For scalability and system design reasons, only one TCP connection exists per LC to its assigned GM over which all nodes, VM and heartbeat monitoring information is sequentially transmitted. Thus when no VMs are active, still a fixed amount of data (i.e. heartbeat) is periodically sent by each LC (see Figure 3.7). Particularly, as the LC monitoring information is of the same structure as the one from a GM, approximately 4.5 kB/s are arriving at the GM. Similarly, the heartbeat information sent by the GM and GL is equivalent in terms of size (i.e. $\sim$2.5 kB/s).

3.6.1.4 CPU and Memory Load Scalability

This section presents the results from the GL (resp. GM) CPU and memory load evaluation. First, the GL node CPU and memory load captured during VM submission is discussed. Afterwards, the GL (resp. GM) node CPU and memory load obtained with an increasing number of GM (resp. LC) nodes is presented.

VM Submission In this experiment we evaluate the scalability of the GL during VM submission by analyzing the CL node CPU and memory load obtained from the VM submission experiment described in Section 3.6.1.2. The GL node CPU and memory load during VM submission is shown in Figure 3.8. We notice that there is a short spike in CPU load and

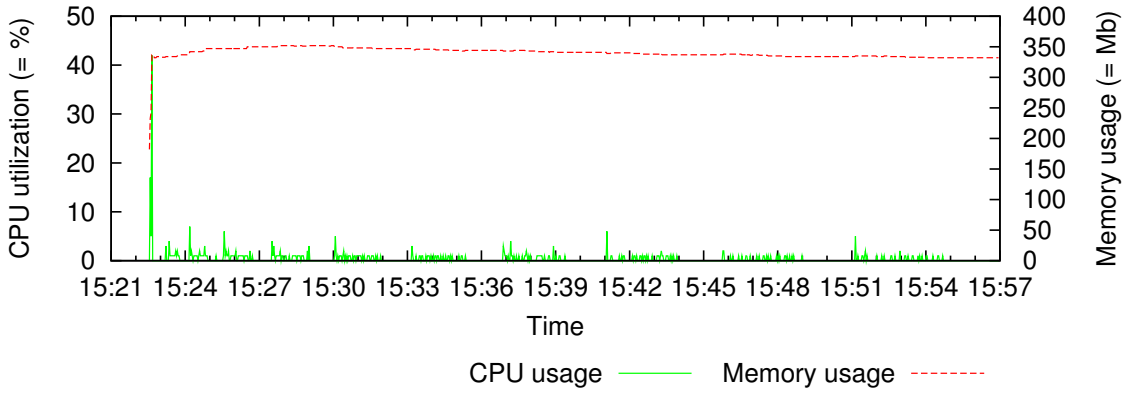


Figure 3.8: Snooze GL node CPU and memory load during VM submission

memory usage at the beginning of the experiment. Indeed the GL service needs to be started first. Afterwards, the actual VC submission is started. After the boot period the system settles at a fixed memory amount of approximately 327 MB (including OS services) which remains constant with the number of VMs submitted. Similarly, small CPU load spikes can be clearly observed during periods of VM submissions which are as well independent of the number of VMs and never exceed 10% of CPU utilization. Both results emphasize the good scalability of the GL service.

Increasing Numbers of GMs and LCs In order to get more insights into the GL as well as GM CPU and memory scalability with an increasing number of GMs resp. LCs, we have evaluated Snooze with different numbers of nodes hosting GMs and LCs. In the first experiment, the amount of nodes hosting LCs was fixed while the number of nodes hosting the GMs was dynamically doubled every minute until 128 nodes were reached. In the second experiment, one node was hosting the GM service while the number of nodes hosting LCs was increased up to 128. In both experiments one node served as the GL. Figure 3.9 depicts the results from the first evaluation. While the GL service scales well with respect to CPU utilization (i.e. small spikes during GM joins), because GM summaries are stored in-memory, the memory usage increases linearly with the number of GMs.

Figure 3.10 presents the results from the second experiment. Apart from a similar increase in memory consumption, the GM shows good CPU load scalability with increasing

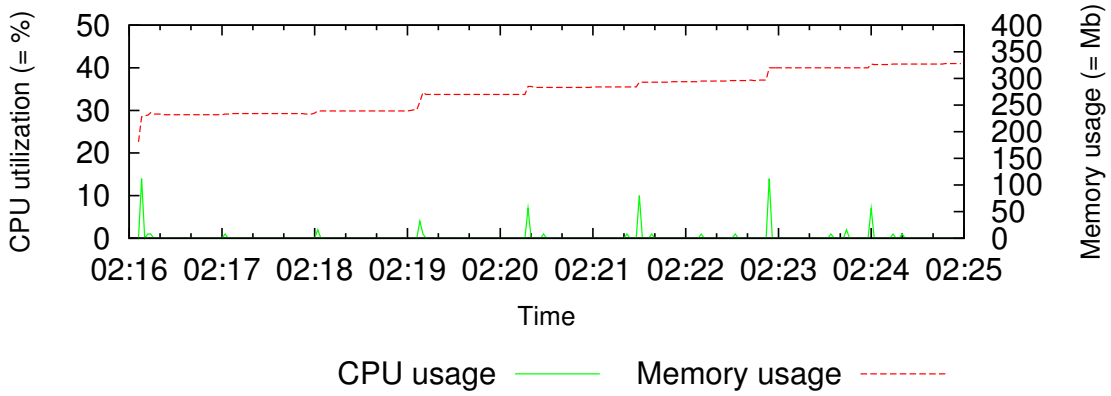


Figure 3.9: Snooze GL node CPU and memory load with increasing number of GMs

number of LCs. Note, that the in-memory storage is implemented using a ring-buffer. Consequently, after some time the least recently used summaries will be overwritten. Nevertheless, The in-memory storage can be replaced by a distributed storage (e.g. [198, 67]) thus improving its scalability.

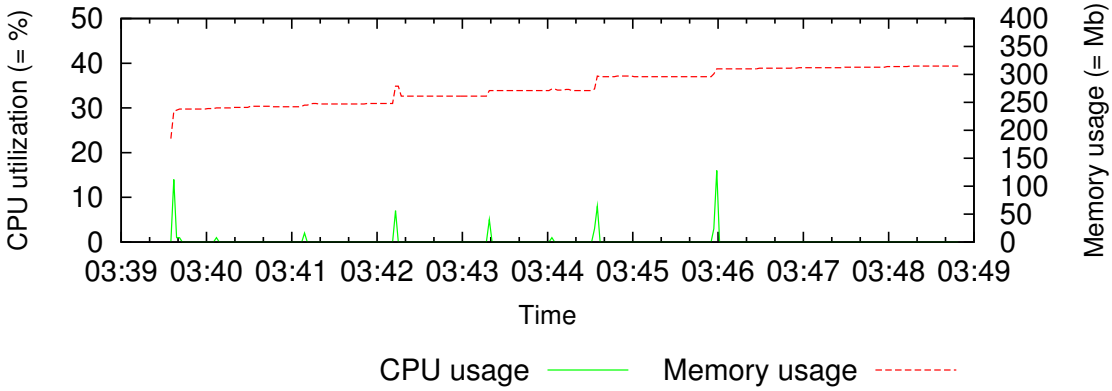


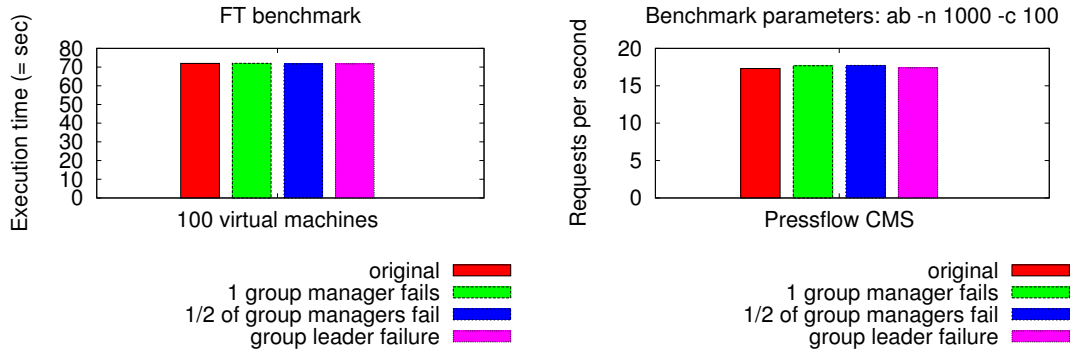
Figure 3.10: Snooze GM node CPU and memory load with increasing number of LCs

3.6.1.5 Impact of Self-Configuration and Healing on Application Performance

To evaluate the impact of the self-configuration and self-healing mechanisms on application performance, the system was configured in a distributed manner with the same configuration as described in Section 3.6.1.2. Three types of VMs with the following applications were created: (1) VMs hosting the MPI-implementation of the *NAS Parallel Benchmark (NPB) v3.3* [79], that represent high performance computing (HPC) workloads; (2) VMs hosting the Linux, Apache, MySQL, PHP (LAMP) stack running the *Pressflow v6 content management system (CMS)* [6], that represent scalable servers workloads; (3) VMs hosting the Apache Hadoop MapReduce [277] framework v0.20.2 to evaluate data analysis workloads. For MPI we have selected the FT benchmark from NPB, because of its heavy use of collective communication thus leading to high average network utilization (approximately 100 Mbit/s per

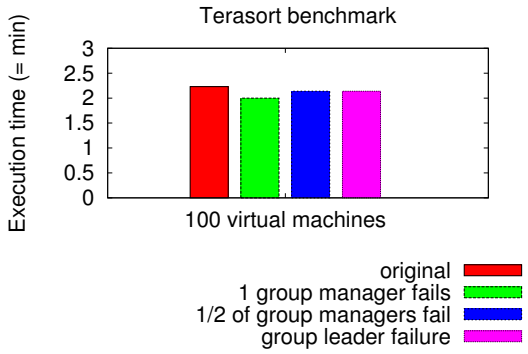
VM). The benchmark was run with the Class A problem size across 100 VMs and the total execution time was measured. For server workloads, the throughput (i.e. maximum number of requests per second) of *Pressflow v6* was analyzed. To generate load the *Apache HTTP server benchmark* [8] was used, with concurrency set to 100 and number of requests to 1000. To represent data analysis workloads we have selected the I/O-intensive Terasort benchmark from the Hadoop MapReduce benchmark suite and run it on 100 Hadoop VMs. Each Hadoop VM had two virtual cores, 4 GB of RAM, and 45 GB disk space. VM disk images were hosted locally on the LCs. Hadoop MapReduce was configured with one map and one reduce slot per VM. Hadoop Distributed File System (HDFS) served as the storage backend. It was configured with a default block size of 128 MB and a replication level of three. Terasort execution time was evaluated while running with 10 GB of input data using 1000 mappers and 500 reducers.

To get an insight in the actual impact of self-configuration and self-healing on application performance, system service failures were injected randomly during the benchmark execution. Three types of failures were injected: single GM failure, catastrophic GM failures (i.e. half of GMs fail) and a GL failure. FT and Apache benchmark measurements were repeated five times and the average values were taken while the Teragen benchmark was run twice. The results of this evaluation are shown in Figure 3.11(a), 3.11(b), and 3.11(c).



(a) FT benchmark

(b) Apache benchmark



(c) Terasort benchmark

Figure 3.11: Impact of Snooze fault tolerance mechanisms on application performance

As it can be observed, apart from small measurement errors neither in the MPI nor in the

web-based or MapReduce benchmark any performance degradation can be observed. This is not surprising as the heartbeat overhead is negligible. Moreover, due to the ZooKeeper service supporting the GL election process, the GMs required only a few dozen of second to recover from a GL failure, resulting in a low amount of network traffic. Similarly, the amount of data sent when a GM rejoins the new GL is approximately 100 bytes. Last but not least, in the event of a GM failure data needs to be transferred by LCs to the current GL and the new GM. In our experiment, a few dozen of kB were transferred thus not requiring substantial amounts of network capacity. Consequently, no overhead due to self-configuration and healing can be observed on the applications.

3.6.2 Energy Efficiency

This sections presents the energy management mechanisms and algorithms evaluation. First, the experiment setup is detailed which involves the description of the evaluation scenario and application used. Then, the system setup is presented and the Snooze configuration parameters are introduced. Finally, the results from the evaluation are discussed.

3.6.2.1 Experiment Setup

Our study is focused on evaluating the energy and performance benefits of the Snooze energy-saving mechanisms using an elastic web application. Elastic web applications are particularly interesting to evaluate as they are the most representative workloads in the cloud context. Thereby, we define an elastic web application a an web application which is able to dynamically request more VMs based on its current load. Consequently, to make the study realistic, we have set up our experiment in a way that reflects a *real-world web application deployment:* An extensible pool of VMs, each hosting a copy of a backend web application running on a HTTP server, while a load-balancer accepts requests coming from an HTTP load injector client (see Figure 3.12).

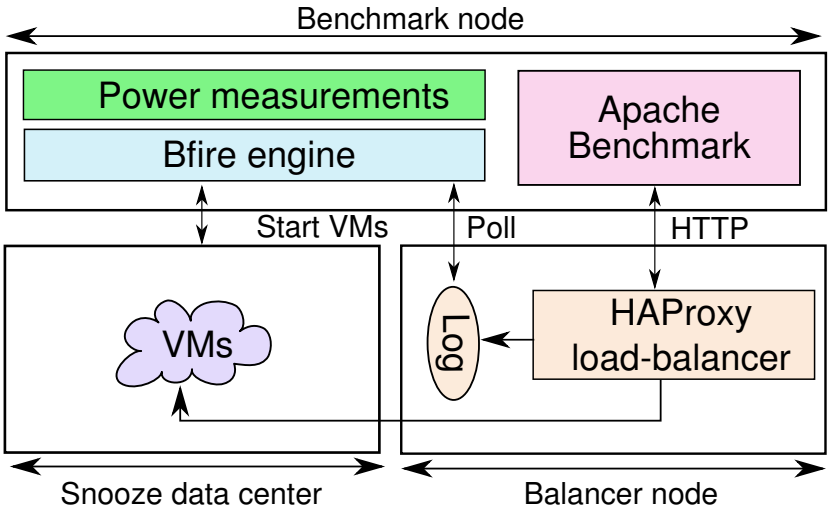


Figure 3.12: Energy management: Bfire experiment setup

The backend application consists of a single HTTP endpoint, which triggers a call to

the *stress* tool [55] upon each request received. Each stress test loads all VM cores during one second and uses 512 MB of RAM. The load-balancer tool used is HAProxy v1.4.8, which is a *state-of-the-art load-balancer used in large-scale deployments* [28]. HAProxy is configured in HTTP mode, four concurrent connections maximum per backend, round-robin algorithm, and a large server timeout to avoid failed requests. Finally, the load injector tool is the well-known Apache benchmark tool [8]. It is configured to simulate 20 concurrent users sending a total number of 15000 requests. According to our experiments these parameters provide the best trade-off between the experiment execution time and the effectiveness of illustrating the framework features. The initial deployment configuration of the backend VMs is done using the Bfire tool [14], which provides a domain-specific language (DSL) for declaratively describing the provisioning and configuration of VMs on a cloud provider. Bfire also allows the monitoring of any metric and provides a way to describe elasticity rules, which can trigger up- or down-scaling of a pool of VMs when a Key Performance Indicator (KPI) is below or over a specific threshold. This tool is currently developed by INRIA within the BonFIRE project [15]. A thin wrapper was developed to make Bfire Snooze compatible (i.e. allowing the Bfire tool to interact with the Snooze RESTful API to provision VMs). The experiment lifecycle is as follows: our Bfire DSL is fed into the Bfire engine, which initially provisions one backend VM on one of the physical nodes. At boot time, the backend VM will automatically register with the load-balancer so that it knows that this backend VM is alive. Once this initial deployment configuration is ready, the Bfire engine will start the Apache benchmark against the load-balancer. During the whole duration of the experiment, Bfire will also monitor in a background thread the time requests spent waiting in queue at the load-balancer level (i.e. before being served by a backend application). Over time, this KPI will vary according to the number of backend VMs being available to serve the requests. In our experiment, if the *average value of the last 3 acquisitions of that metric is over 600ms* (an acceptable time for a client to wait for a request), then a scale-up event is generated, which increases the backend pool by *four new VMs at once*. If the KPI is below the threshold, then nothing happens. This elasticity rule is monitored every 15 seconds, and all newly created VMs must be up and running before it is monitored again (to avoid bursting). Meanwhile, an additional background process is registering the power consumption values coming from the PDUs to which the physical nodes are attached.

3.6.2.2 System Setup

To evaluate the Snooze energy management mechanisms we have deployed it on *34 power metered HP ProLiant DL165 G7 nodes* of the Grid'5000 experimental testbed with one EP, one GL, one GM and 31 LCs. All nodes are equipped with two AMD Opteron 6164 HE CPUs each having 12 cores (in total 744 compute cores), 48 GB of RAM, and a Gigabit Ethernet connection. They are powered by six APC AP7921 power distribution units (PDUs). Power consumption measurements and the benchmarking execution are done from two additional Sun Fire X2270 nodes in order to avoid influencing the measurement results (see Figure 3.13). The load balancer and load injector are running on the Sun Fire X2270 nodes.

The node operating system is Debian with a 2.6.32-5-amd64 kernel. All tests were run in a homogeneous environment with qemu-kvm 0.14.1 and libvirt 0.9.6-2 installed on the machines. Each VM is using a QCOW2 disk image with the corresponding backing image hosted on a Network File System (NFS). Incremental storage copy is enabled during live

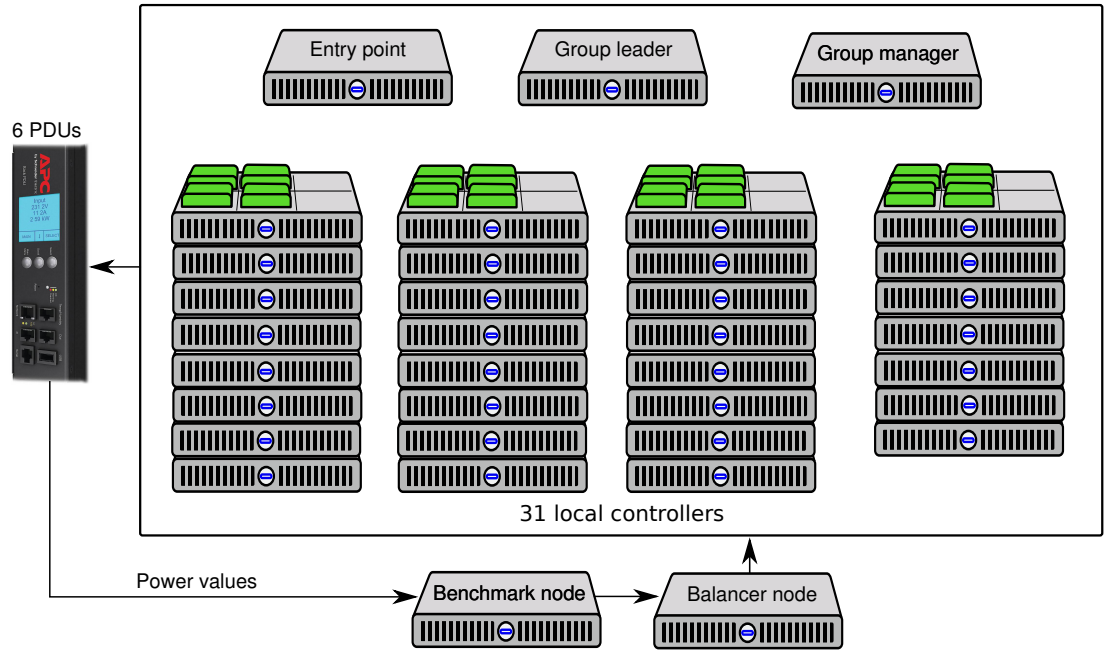


Figure 3.13: Energy management: data center setup

migration. Debian is installed on the backing image. The NFS server is running on the EP with its directory being exported to all LCs. VMs are configured with 6 VCORES, 4GB RAM and 100 MBit/sec network connection. Note that libvirt currently does not provide any means to specify the network capacity requirements. Therefore, Snooze wraps around the libvirt template and adds the necessary network capacity (i.e. Rx and Tx) fields.

Tables 3.6.2.2, 3.6.2.2, 3.6.2.2, and 3.6.2.2 show the threshold, estimator, scheduler, and power management settings used in the experiments.

Resource	MIN, MID, MAX
CPU,	0.2, 0.9, 1
Memory	0.2, 0.9, 1
Network	0.2, 0.9, 1

Table 3.1: Threshold settings

Parameter	Value
Packing density	0.9
Monitoring backlog	15
Resource estimators	average
Consolidation interval	10 min

Table 3.2: Estimator settings

Policy	Algorithm
Dispatching	RoundRobin
Placement	FirstFit
Overload	see Algorithm 1
Underload	see Algorithm 2
Consolidation	see Algorithm 3

Table 3.3: Scheduler settings

Parameter	Value
Idle time threshold	2 min
Wake up threshold	3 min
Power saving action	shutdown
Shutdown driver	system
Wake up driver	IPMI

Table 3.4: Power management settings

Our evaluation is focused on the performance (i.e. response time) of the Apache bench-

mark, the power consumption of the nodes, the number of VMs and live migrations. Moreover, we visualize all the events (i.e. Bfire, relocation, consolidation, power management) which were triggered in our system during the experiments. Two scenarios are evaluated: (1) No energy savings, to serve as a baseline; (2) Energy savings enabled (i.e. underload relocation, VM consolidation, and power management). In both scenarios overload detection is enabled. The evaluation results are discussed below.

3.6.2.3 Elastic VM Provisioner Events

The elastic VM provisioner (i.e. Bfire) events without and with energy savings enabled (red resp. green colored) are shown in Figure 3.14. Bfire distinguishes between three types of events: READY, SCALING, SCALED. The experiment starts by provisioning one backend VM which results in the provisioner to become READY. READY means that Bfire could successfully start the first VM on Snooze. When it becomes ready we start the actual benchmark which soon saturates the VM capacity. Bfire reacts by SCALING up the number of VMs to four. It takes approximately five minutes to provision the VMs. This is reflected in the subsequent SCALED event which signals the VM provisioning success. The same process happens until the end of the benchmark execution. In total four SCALING (resp. SCALED) are triggered which result in 17 VMs to be provisioned by the end of the Apache benchmark. Note that the *experiment with energy savings enabled lasts a bit more (1.2% of time) than without energy savings* because of the need to power on nodes and lightly increased response time (see the following paragraphs).

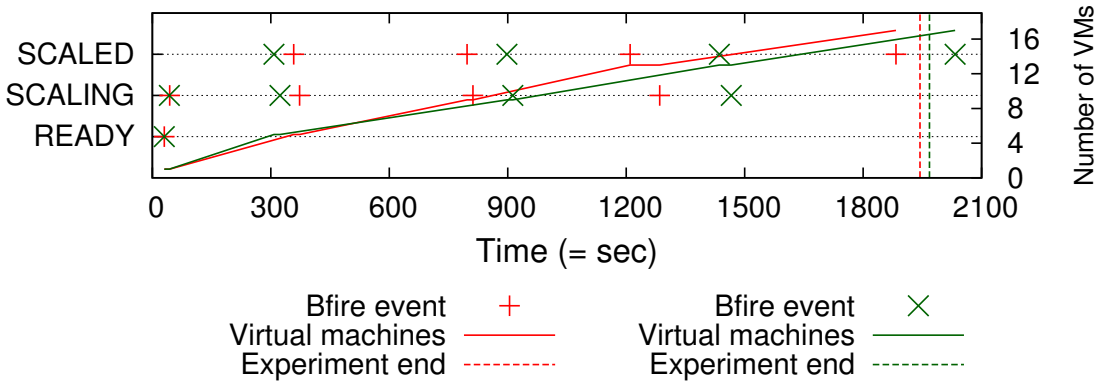


Figure 3.14: Energy management: elastic VM provisioner events

3.6.2.4 Apache Benchmark Performance

The Apache benchmark results (i.e. response time for each request) are depicted in Figure 3.15. As it can be observed, response time increases with the number of requests in both cases (i.e. without and with energy savings). This is not surprising as the VMs get overloaded with increasing number of requests. Note, that each request arrival results in the stress benchmark to be executed on the VMs which saturates the available CPU and memory resources. However, more interestingly is the fact that response time is *not significantly impacted when energy savings are enabled*. Particularly, in both scenarios a response time

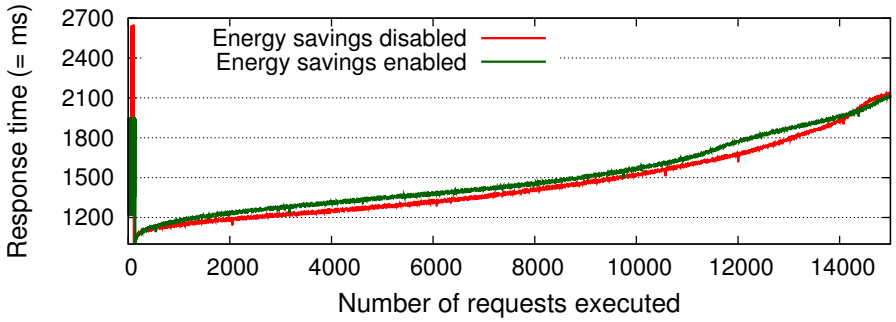


Figure 3.15: Energy management: Apache benchmark response time

peek exists at the beginning of the experiment. Indeed, one backend VM is quickly saturated. However, when times passes only minor performance degradation can be observed. The main reason for the minor performance degradation lies in the fact that once energy savings are enabled, *servers are powered down*, thus increasing the time requests remain in the HAProxy queue until they can be served by one of the backends. Moreover, Bfire dynamically increases the number of VMs with growing load. Increasing the number of VMs involves scheduling, powering on LCs as well as a software provisioning phase in which tools are installed on the scheduled VMs in order to register with HAProxy. This requires time and thus impacts application performance (i.e. requests are queued). Performance could be further improved by taking proactive scaling up decisions. Finally, underload relocation and consolidation are performed which involve VM migration which contributes to the performance degradation.

3.6.2.5 System Power Consumption and Events

The system power consumption without and with energy savings is depicted in Figure 3.16. Without energy savings our experimental data center first consumes approximately *5.7 kW of idle power*. With the start of the benchmark the load increases to *6.1 kW* and falls back with the end of the evaluation. Note that our experiments did not fully stress all the 744 compute cores which would have resulted in even higher power consumption ($\sim$ 7.1 kW) but would also have made harder to conduct the experiment due to the increased execution time.

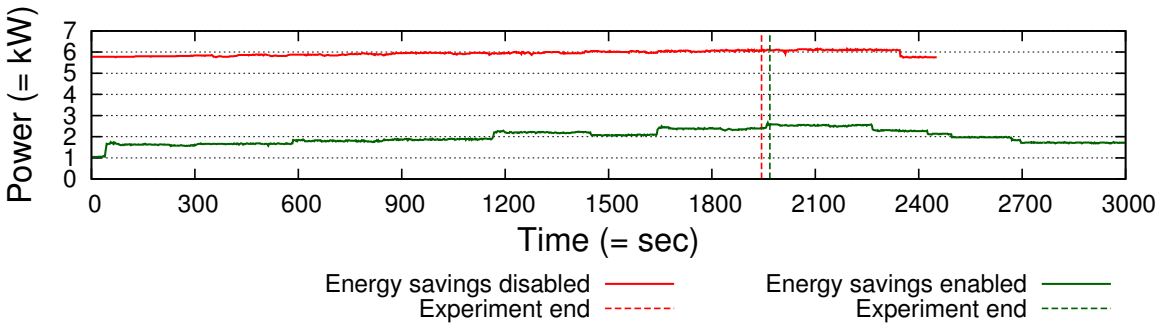


Figure 3.16: Energy management: power consumption

Snooze overcommits nodes by allowing them to host more VMs than physical capacity allows it. This leads to overloaded situations requiring VMs to be live migrated. In this context we distinguish between two types of events: overload relocation (OR) and migration plan enforced (MPE). The former is triggered in case of overload situation and results in a migration plan which needs to be enforced. MPE events signal the end of the enforcement procedure. Figure 3.17 shows the event profile including the number of migrations.

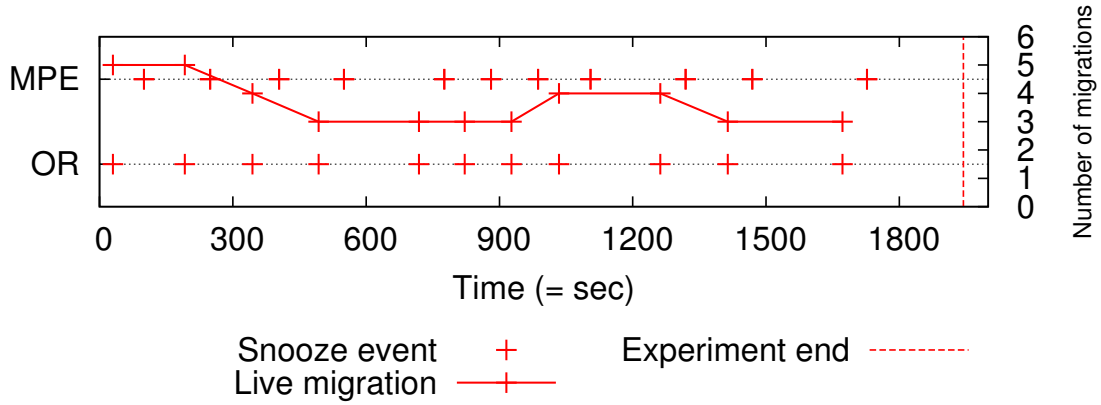


Figure 3.17: Energy management: Snooze system events without energy savings

As it can be observed the first two OR events trigger five migrations. This is due to the fact that the First-Fit placement is performed upon VM submission. This leads to an overload situation on the LCs which needs to be resolved. However, as time progresses the number of migrations decreases as VMs are placed on more lightly loaded LCs.

With energy savings enabled, when the experiment starts the system is idle, and thus the nodes have been powered down by Snooze, reducing the power consumption to approximately one kW (see Figure 3.16). When the benchmark is started the system reacts by taking actions required to provision just as many nodes as needed to host the VMs. This results in the power consumption following the system utilization (i.e. increasing number of VMs). Note that the power consumption never drops to the initial value (i.e. one kW) as the Snooze management nodes (i.e. EP, GL, GM) as well as VMs are kept in the system in order to illustrate the framework mechanisms. Consequently, once idle they still consume additional power. In a production environment VMs would be shutdown by the customers thus resulting in additional power savings.

Particularly, the following actions presented in Figure 3.18 are performed: (1) detect LC underload and overload; (2) trigger underload and overload relocation (UR resp. OR) algorithms; (3) enforce migration plans (MPE); (4) perform periodic consolidation (C); (5) take power saving actions such as power up and down (PUP resp. PDOWN) depending on the current load conditions. In order to get an insight in the system behaviour we have captured all these events.

During the benchmark execution the first OR event appears as the system becomes overloaded. The overload situation is resolved by powering up one LC and migrating five VMs. Then consolidation is started which migrates two VMs. The system continues to react to OR/UR events and adapt the data center size according to the current load (i.e. PUP and PDOWN events follow) until the end of the benchmark. Note that the number of migrations

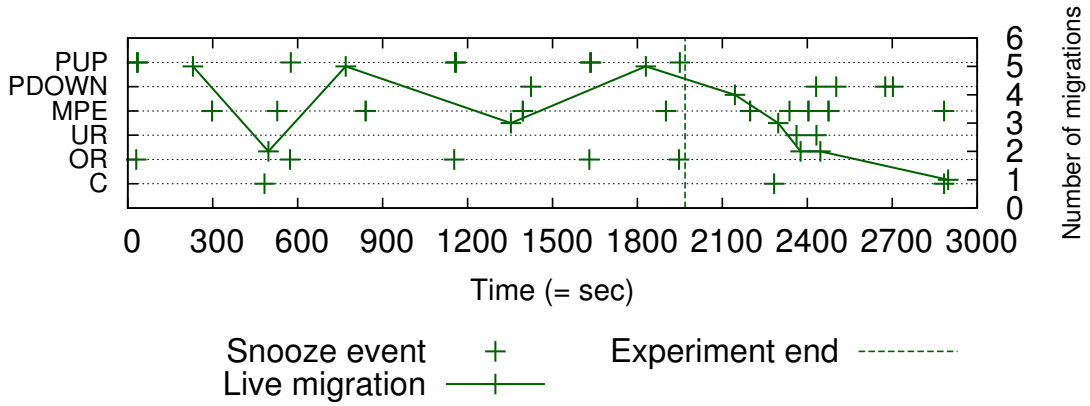


Figure 3.18: Energy management: Snooze system events with energy savings enabled

decreases with the benchmark execution time as the HAProxy load decreases with increasing number of backend VMs thus resulting in less OR events. Towards the end of the benchmark UR happens and results in a series of PDOWN events. Finally, consolidation is started and improves the VM placement by migrating one VM. *This shows that relocation and consolidation are complementary*.

Putting all the results together, data center energy consumption measured during the benchmark execution without and with power management enabled amounted to 3.19 kWh (34 nodes), respectively 1.05 kWh (up to 11 nodes), resulting in *67% of energy being saved*. We estimated that for the same workload with a smaller data center size of 17 nodes, the energy gains would have been approximately 34%.

3.7 Summary

In this chapter we have presented the design, implementation, and evaluation of a novel autonomic and energy-efficient IaaS cloud management system for large-scale virtualized data centers called *Snooze*. In contrast to existing works, Snooze employs a *self-configuring and healing hierarchical architecture* in which the VM management tasks are distributed across multiple autonomic managers, the so-called GMs. Each GM has only a *partial view of the system* and thus manages only a *subset of the data center compute nodes*. A fault-tolerant GL exists which accepts client VM submission requests, handle the VM networking, and dispatches the submitted VMs among the GMs. This allows Snooze to remain scalable as the GL does not require global system knowledge. Indeed, the GL dispatching decisions are solely based on the aggregated resource utilization data as received from the GMs. Finally, Snooze provides a unique holistic energy-efficient VM management solution via integrated advanced VM management mechanisms such as underload/overload mitigation, VM consolidation, and power management. This mechanisms allow Snooze to scale the data center power consumption proportionally to its load.

Snooze was implemented from scratch in Java and currently comprises approximately 15 000 lines of highly modular code. It has been extensively evaluated on the Grid'5000 experimentation testbed using realistic scientific and web applications and shown to be scalable,

autonomic, and energy-efficient. Particularly, our experimental results have shown that: (1) submission time is not impacted by performing distributed VM management; (2) system autonomy properties do not impact application performance; (3) the system scales well with increasing number of resources thus making it suitable for managing large-scale virtualized data centers; (4) the advanced VM management mechanisms allow Snooze to scale the data center proportionally to its load thus achieving substantial energy savings for the evaluated web application. Thanks to its flexible implementation, Snooze can by used either as a research testbed to experiment with novel VM management algorithms in a realistic environment or serve as an cloud management system in virtualized data centers. Since May 2012 Snooze is distributed in open-source under the GPL v2 license at http://snooze.inria.fr. It is known to be used by researchers (e.g. IRIT, LIFL) to experiment with advanced VM management algorithms and has been successfully validated on experimentation testbeds at EDF R&D and Medio Seattle.

Chapter 4

VM Management via Ant Colony Optimization

Contents

Two key algorithms to favour the creation of server idle-times in IaaS cloud management systems are VM placement and consolidation. VM placement and consolidation are NP-hard [284, 266] combinatorial optimization problems and thus are expensive to compute in time and space. Consequently, a number of VM placement and VM consolidation algorithms have been proposed over the past years aiming at computing approximate solutions in polynomial time. However, many of the proposed algorithms consider only a single resource dimension (i.e. CPU) and rely on centralized greedy heuristics which are known to be hard to distribute/parallelize [76]. This chapter investigates the use of the Ant Colony Optimization (ACO) meta-heuristic to compute solutions for the aforementioned problems and proposes novel ACO-based VM placement and consolidation algorithms. ACO is especially attractive for VM placement and consolidation due to its polynomial time worst-case complexity, ease of parallelization, and near optimal solutions. Indeed, by nature ants have the property to work independently.

This chapter is organized as follows. Section 4.1 gives a brief introduction to Ant Colony Optimization. The ACO-based VM placement algorithm and its evaluation are presented in Section 4.2. In Section 4.3 we describe our approach for VM consolidation and its evaluation. Note that these sections rely on the notations previously introduced in Section 2.4.4.2. Finally, Section 4.4 summarizes this chapter.

4.1 Ant Colony Optimization

Ant Colony Optimization is a meta-heuristic, which was initially introduced as Ant Systems (AS) in 1992 within the PhD thesis of the Italian researcher Marco Dorigo [125]. Initially, it was developed to solve the Travelling Salesman Problem. However, since then it has been successfully adapted to solve many other complex combinatorial optimization problems (e.g. vehicle routing, graph coloring, and bin packing). The main inspiration to develop this system was the natural food-discovery behaviour of real ants. Because of the limited abilities of the ants to see and hear their environment they have developed a form of indirect communications (also called Stigmergy) by use of a chemical substance referred as *pheromone*. This substance is deposited by each ant on the path it traverses and evaporates after a certain period of time. Other ants can smell the concentration of this substance and tend to favour paths probabilistically according to the amount of pheromone deposited on them. Surprisingly, after some time the entire ant colony converges towards the shortest path to the food source. This behaviour was studied by biologists in numerous controlled experiments [119] and can be explained as follows. At the beginning, when starting from the nest the ants choose a random path to follow. However, on the shortest path to the food source the ants will return faster. Consequently, this path will have a stronger pheromone concentration thus being more attractive for subsequent ants to follow it. When time passes, pheromone concentration on the shortest paths will continue to increase, while on the longer ones it will keep falling, making them less and less attractive.

4.2 VM Placement Algorithm

This section presents the design of our ACO-based VM placement algorithm. First, the design principles of the algorithm are discussed. Then, algorithm design principles, assumptions, components, and the pseudocode are introduced. Finally, the evaluation results are discussed.

4.2.1 Design Principles

The proposed algorithm is based on the ACO principles in which multiple agents (i.e. artificial ants) compute solutions probabilistically and simultaneously within multiple cycles. Thereby, they communicate indirectly by depositing a chemical substance called *pheromone* on paths they traverse. However, as the VM placement does not incorporate the notation of a path, in our algorithm the ants deposit pheromone on each VM-PM pair within a *pheromone matrix*. In each cycle the ants receive VMs, and start constructing local solutions (i.e. VM to PM assignments) by the use of a probabilistic decision rule which describes the desirability for an ant to choose a particular VM as the next one to pack in its current PM. This

rule is based on the current pheromone concentration information on the VM-PM pair in the pheromone matrix and a heuristic information which guides the ants towards choosing VMs leading to better overall PM utilization. Hence, the higher the amount of pheromone and heuristic information is associated with an VM-PM pair, the higher the probability that it will be chosen. Figure 4.1 visualizes the solution construction process of a single ant. The ant starts with four VMs, opens a PM, computes the probabilities for each of the VMs using the probabilistic decision rule, and starts assigning the VMs to the newly opened PM according to the computed probabilities. Once the PM is full it opens a new PM, recomputes the probabilities for the remaining VMs, and continues the same assignment procedure until all the VMs are assigned. At the end of each cycle, local solutions are compared and the one requiring the least number of PMs is saved as the new globally optimal solution. Afterwards, the pheromone matrix is updated to simulate pheromone evaporation and reinforce VM-PM pairs which belonged to the so-far best solution. This is achieved by the use of the so-called pheromone update rule. The stochastic nature of the algorithm allows it to explore a large number of potential solutions. Moreover, the algorithm is well suited for parallelization.

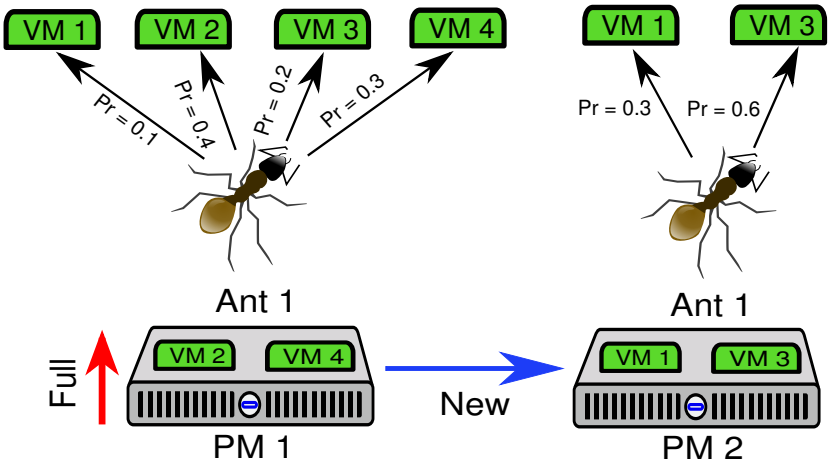


Figure 4.1: ACO-based VM placement: example

4.2.2 Assumptions

We consider the offline version of the VM placement problem. In other words, the proposed algorithm is assumed to be triggered during the VM submission on a *set of VMs* and outputs a solution consisting of VM to PM assignments such that the number of PMs is minimized. PMs can be either pre-filled or empty. In this work we assume empty PMs (i.e. PMs which do not host any VMs). Note, that any VM placement algorithm can be generalized to support pre-filled PMs by simply applying it on pre-filled PMs. The algorithm is evaluated using homogeneous PMs and heterogeneous VMs. However, nothing prevents it to be also used with heterogeneous PMs. Finally, we assume independent VMs thus no placement such as collocation and anti-collocation are considered.

4.2.3 Algorithm Description

We now present the VM placement algorithm. First, the definition of the probabilistic decision rule is given. Then, the pheromone trail update mechanisms is described. Finally, the algorithm pseudocode code is presented.

4.2.3.1 Probabilistic Decision Rule

We define the probability for an ant to choose an VM v as the next one to pack in its current PM p as follows.

$$Pr_p^v := \frac{[\tau_{v,p}]^\alpha \times [\eta_{v,p}]^\beta}{\sum_{u \in N_p} [\tau_{u,p}]^\alpha \times [\eta_{u,p}]^\beta}, \quad \forall v \in N_p \tag{4.1}$$

whereby, $\tau_{v,p}$ denotes the pheromone based desirability of packing VM v into PM p and $\eta_{v,p}$ the VMs heuristic information. Moreover, two parameters $\alpha, \beta \geq 0$ are used in order to either emphasize more the pheromone or the heuristic information. Finally, N_p defines the set of all VMs which qualify for inclusion into the current PM p (see Eq. 4.2). These are all VMs which have not been assigned to any PM yet and do not violate the PM capacity constraints in all dimensions. $\mathbf{l_p}$ represents the total used PM capacity. It is computed as the sum of all VM requested capacity vectors: $\mathbf{l_p} := \sum_{\forall v \in V} \mathbf{RC_v}$.

$$N_p := \{v \mid \sum_{p=0}^{n-1} x_{v,p} = 0 \wedge \mathbf{l_p} + \mathbf{RC_v} \leq \mathbf{TC_p}\} \tag{4.2}$$

As our objective is to minimize the number of PM (i.e. maximize the resource utilization), we define the heuristic information to favour VMs which utilize the PMs better. This is achieved by defining $\eta_{v,p}$ as the inverse of the scalar valued difference between the static capacity of PM v and the load of PM after packing the VM $v \in N_p$.

$$\eta_{v,p} := \frac{1}{|\mathbf{TC_p} - (\mathbf{l_p} + \mathbf{RC_v})|_1} \tag{4.3}$$

In order to compute the ratio defined by equation 4.3 the resulting d-dimensional resource demand vector needs to be mapped to a scalar value. In this work the L1-norm [294] is used. However, alternative methods such as taking the arithmetic mean are possible.

4.2.3.2 Pheromone Trail Update

After all ants have finished building a solution, pheromone trails on all VM-PM pairs need to be updated in order to help guiding the algorithm towards the optimal solution. A pheromone trail update rule $\tau_{v,p}$ exists and is used in order to simulate pheromone evaporation and reinforce VM-PM pairs which belonged to the so far best solution. In this work we follow the MAX-MIN Ant System (MMAS) [273] approach in which only the *iteration's-best* ant (i.e. ant whose solution's objective function value is minimal) is allowed to deposit pheromone. The pheromone update rule is defined in Eq. 4.4.

$$\tau_{v,p} := (1 - \rho) \times \tau_{v,p} + \triangle \tau_{v,p}^{best}, \quad \forall (v, p) \in V \times P \tag{4.4}$$

whereby, the constant ρ, $0 \leq \rho \leq 1$ is used to simulate pheromone evaporation. Hence, higher values for ρ lead to increased evaporation rate. Moreover, some VM-PM pairs need to be reinforced. Thereby, $\triangle\tau_{v,p}^{best}$ is defined as the *iteration's-best* VM-PM pheromone amount. Hence, if some VM belongs to a PM of the so far best solution S_{best}, its pheromone amount is reinforced. Consequently, only VM-PM pairs which are part of S_{best} will be reinforced and thus become more attractive. Others, which are not part of S_{best} will continue loosing pheromone according to the pheromone evaporation rate ρ. A solution $S := [x_{v,p}]_{|V|\times|P|}$ is defined as a binary matrix whose elements represent the mapping of VMs to PMs.

The ultimate goal of our ACO-based algorithm is to minimize the amount of PMs, thus increasing the average utilization of each PM. Hence, we target to favour solutions which utilize the least number of PMs. Therefore, we define the amount of pheromone iteration's *best* ant deposits on the VM-PM pair to be inverse proportional to the value of the objective function f applied on the iteration's best-solution S_{best}. Thereby, only VM-PM pairs which are marked as allocated in S_{best} will be reinforced.

$$\triangle\tau_{v,p}^{best} := \begin{cases} \frac{1}{f(S_{best})} & \text{if } x_{v,p} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

Finally, because only the *iteration's-best* ant is allowed to deposit pheromone, early stagnation of the search is most likely to happen, thus leading to a situation in which all ants always choose the same VMs. This reduces the ability of the algorithm to explore alternative solutions. In order to limit this effect, MMAS introduces lower and upper bounds for the pheromone values $\tau_{v,p}$. Hence, $\tau_{v,p}$ is restricted to the range $[\tau_{min}, \tau_{max}]$. Analogously, we define τ_{max} as $\tau_{max} := \frac{1}{f(S_{best})\times(1-\rho)}$ and τ_{min} as $\tau_{min} := \frac{\tau_{max}}{g}$, respectively with factor $g > 1$ [273] .

4.2.3.3 Pseudocode

The pseudocode of the ACO-based algorithm is depicted in Figure 4. The algorithm takes as input the set of VMs and PMs, including their static requested (resp. total) capacity vectors $\mathbf{RC_v}$ and $\mathbf{TC_p}$, respectively. Moreover, a set of parameters (i.e. α, β, ρ, g, τ_{max}, $nCycles$, $nAnts$) is required for initialization. First, the parameters are initialized and the pheromone trails of all the VM-PM pairs are set to τ_{max} (line 4). The algorithm then iterates until the specified number of cycles $nCycles$ (lines 5 to 35). In each iteration an ant a opens a PM p and starts building a solution S_a (lines 6 to 20). This is achieved by first initializing the set of VMs IS, the elements of the binary solution matrix S_a and the *PM-index* variable p. The algorithm then enters a loop and starts assigning the VMs to the PMs (lines 9 to 19). The current PM p is being filled until its resources are saturated. This is achieved by initializing the set N_v with all VMs which are not yet assigned to any PM and do not violate the capacity constraints of the current PM (line 10). If this set is not empty, the probabilistic decision rule Pr_p^v is used to select one VM v out of the set to be packed in the current PM p, stochastically (line 12). The VM is then marked as allocated in the solution matrix by setting the appropriate value in the matrix S_a to 1, removed from the set of VMs IS and the PM utilization is updated (lines 13 to 15). This process is performed as long as there are still VMs left to be assigned and enough capacity available in the current PM (line 9 and 10). Afterwards, when the PM capacity is saturated (i.e. N_v becomes empty) the PM-index variable is incremented and the packing process is continued until all VMs are placed

Algorithm 4 ACO-based VM placement

1: Input: Set of VMs V and set of PMs P with their associated resource demand vectors $\mathbf{RC_v}$ and $\mathbf{TC_p}$ respectively, Set of parameters
2: Output: Global best solution $\mathbf{S_{best}}$
3:
4: Initialize parameters, Set pheromone value on all VM-PM pairs to τ_{max}
5: **for all** $q \in \{0 \dots nCycles - 1\}$ **do**
6: **for all** $a \in \{0 \dots nAnts - 1\}$ **do**
7: $IS := V; p := 0$
8: $S_a := [x_{v,p} := 0], \forall v \in \{0, \dots, m-1\},\ \forall p \in \{0, \dots, n-1\}$
9: **while** $IS \neq \emptyset$ **do**
10: $N_p := \{v \mid \sum_{p=0}^{n-1} x_{v,p} = 0 \wedge \mathbf{l}_p + \mathbf{RC}_v \leq \mathbf{T}C_p\}$
11: **if** $N_p \neq \emptyset$ **then**
12: Choose VM $v \in N_v$ stochastically according to probability $Pr_p^v := \frac{[\tau_{v,p}]^\alpha \times [\eta_{v,p}]^\beta}{\sum_{u \in N_p} [\tau_{u,p}]^\alpha \times [\eta_{u,p}]^\beta}$
13: $x_{v,p} := 1$
14: $IS := IS - \{v\}$
15: $\mathbf{l_p} := \mathbf{l_p} + \mathbf{RC_v}$
16: **else**
17: $p := p + 1$
18: **end if**
19: **end while**
20: **end for**
21: Compare ants solutions S_a according to the objective function $f \rightarrow$ Save cycle best solution as S_{cycle}
22: **if** $q = 0 \vee IsGlobalBest(S_{cycle})$ **then**
23: Save cycle best solution as new global best S_{best}
24: **end if**
25: Compute τ_{min} and τ_{max}
26: **for all** $(v, p) \in V \times P$ **do**
27: $\tau_{v,p} := (1 - \rho) \times \tau_{v,v} + \triangle \tau_{v,p}^{best}$
28: **if** $\tau_{v,p} > \tau_{max}$ **then**
29: $\tau_{v,p} := \tau_{max}$
30: **end if**
31: **if** $\tau_{v,p} < \tau_{min}$ **then**
32: $\tau_{v,p} := \tau_{min}$
33: **end if**
34: **end for**
35: **end for**
36: **return** Global best solution $\mathbf{S_{best}}$

(lines 9 to 19). After all ants have constructed their solutions S_a, a comparison is performed and the *cycle's best* solution is saved (line 21) as S_{cycle}. Two criteria: *amount of utilized PMs* and *amount of failed VM allocations* are used in order to judge about the cycle best solution. While the first one seems natural, the second one is a result of two solutions which equal

in terms of utilizing all available PMs but differ in the utilization efficiently of the PMs. For instance, two solutions would use the same number of PMs, but the first one would fail allocating resources for 10% of the requests while the second one would satisfy all requests. Finally, if this is the first cycle, the cycle best solution becomes the global best one. Otherwise, a comparison is done with the current global best solution. If the cycle best solution yields to an improvement it becomes the new global best one (lines 22 to 24). Afterwards, the values for τ_{min} and τ_{max} are computed (line 25) and the pheromone trails on all VM-PM pairs (v, p) are updated using the pheromone update rule $\tau_{v,p}$ (lines 26 to 34). In order to respect the specified lower and upper bounds for $\tau_{v,p}$ two conditions exist. First condition guarantees that the upper bound is respected. Hence, if some VM-PM pair received a higher pheromone amount than τ_{max}, it is reinitialized to τ_{max} (lines 28 to 30). Similarly, when the pheromone amount of some VMs falls below the predefined lower bound τ_{min} it is updated accordingly (lines 31 to 33). The algorithm terminates after *nCycles* and returns the so far global best solution S_{best} (line 36).

4.2.4 Evaluation

This section present the performance evaluation of the proposed ACO-based VM placement algorithm. In order to gain a first insight into the performance of the algorithm at large scale before implementing it in a real environment, we have decided to conduct simulation-based experiments. An VM placement simulator was developed and used to compare our ACO-based VM placement algorithm with the frequently applied FFD heuristic. In order to improve the performance a multithreaded version of the algorithm was developed. Furthermore, the FFD heuristic was modified to to capture the multidimensional nature of the problem. Particularly, the VM requested capacity vectors were sorted in decreasing order according to the L1-norm [294].

4.2.4.1 System Setup

We simulated a cluster composed of homogeneous PMs with each having a static resource capacity of 10000 MIPS, 24 cores, 50 GB of RAM, 1 TB storage and 10 GBit/sec network connection. The amount of PMs was set to the amount of VMs in order to support the worst packing scenario, in which only one VM is assigned per PM. In total, up to 600 VMs were simulated with each requiring either 1000, 2000, 3000 or 5000 of MIPS, 2 cores, 4 GB of RAM, 200 GB of storage and 1 GBit/sec of network bandwidth.

4.2.4.2 Power Consumption Model

In order to estimate the energy consumed by a placement, we approximate the power of a PM as a linear function $P(u)$ [130] in its current utilization $u \in [0, 1]$ (see Eq. 4.6).

$$P(u) = (P_{max} - P_{idle}) \times u + P_{idle} \tag{4.6}$$

with P_{idle} and P_{max} being the average power values when the system is idle and fully utilized, respectively. Both values have been fixed to *171* and *218 Watt*, for all simulations according to the measurements performed on our own testbed. The testbed we use is equipped with one Dell PowerEdge 1950 server plugged into a Sentry POPS (Per Outlet Power Sensing)

switched Cabinet Distribution Unit (CDU). The server comprises 4 GB RAM and two Intel Xeon 5148 2.33 GHz CPUs, each with two cores. Idle power was derived by measuring the power drawn by the server when it only hosts the OS and the least amount of required system services (e.g. udev, sshd). Average peak power consumption was measured by running the *stress* benchmark application, with parameters set to stress all the system components.

4.2.4.3 Energy Consumption Estimation

For estimating the energy consumed by a placement a time period t was defined and set to 24 hours. Consequently, the energy values represent the power drawn by the cluster at the utilization given by the placement over the period of 24 hours. It was assumed that empty PMs are turned off after the VM placement. Hence, their idle power is not part of the total placement energy consumption. In particular, energy consumed by a placement was computed according to Eq. 4.7.

$$E(P) := \begin{cases} t \times \sum_{p=0}^{n-1} P(\frac{|\mathbf{l}_p|_1}{d}) & \text{if } |\mathbf{l}_p|_1 \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

Note, that because of the non-proportional power usage (i.e. high idle power) of traditional servers, no matter which energy model is used, turning off/suspending PMs always yield energy savings assuming that the algorithm is triggered during appropriate time periods (e.g. low utilization). Moreover, since the packing is based on VMs requested capacity, we assume that placed VMs will not suffer from significant performance degradation. Given such assumptions, consolidating the VM of two PMs at 0.3 and 0.7 utilization, respectively, onto one server running at peak utilization (i.e. 1) is advantageous.

4.2.4.4 ACO Parameters

The parameters of the ACO-based algorithm were derived empirically through numerous simulations and finally set as depicted in Table 4.1. Note, that the amount of cycles and ants were initialized to 2 and 5, respectively. According to our experiments, no improvements in the solutions quality (i.e. number of provisioned PMs) could be observed with greater number of ants and cycles. Nevertheless, as our results will show the resulting solutions are still close to the global optimum.

Table 4.1: ACO-based VM placement: algorithm parameters

α	β	ρ	g	τ_{max}	$nCycles$	$nAnts$
1	2	0.7	2	3	2	5

4.2.4.5 Experiment Results

We run the simulation for up to 600 VMs and measured the number of provisioned PMs, energy consumption of the placement and the average execution times for both algorithms (i.e. FFD and ACO-based). In addition, in order to judge the quality of the solutions, optimal

solutions were computed by integrating the previously introduced BIP model into the high-performance Mixed-integer linear programming (MILP) solver IBM ILOG CPLEX v12.2 [33]. The solver was set to emphasis optimality and run in parallel mode with 4 threads.

In order, to derive the actual energy savings, the amount of energy spent for computing the placement was estimated by multiplying the execution time of the algorithm with average power drawn (i.e. 198 Watt) of the system during the simulation. The resulting amount of energy spent for the simulation was included into the final energy consumption of the placement and accounted not more than 400 Wh. Therefore, it did not impact the total energy results of the algorithms which were in the order of *kWh*. The final numerical simulation results are depicted in Table 4.2.4.5.

VMs	Algorithm	Provisioned PMs	Execution time	Energy consumption (= kWh)	Gain (= %)
100	FFD	30	0.39 sec	139.62	
	ACO	28	37.46 sec	131.41	5.88
	CPLEX	28	0.451 sec	131.41	5.88
200	FFD	59	0.58 sec	275.13	
	ACO	56	4.51 min	262.83	4.47
	CPLEX	55	1.27 sec	258.71	5.96
300	FFD	88	0.77 sec	410.65	
	ACO	84	15.04 min	394.28	3.98
	CPLEX	83	2.86 sec	390.12	4.99
400	FFD	117	1.03 sec	546.16	
	ACO	112	34.23 min	525.75	3.73
	CPLEX	110	5.07 sec	517.43	5.26
500	FFD	146	1.39 sec	681.67	
	ACO	139	1.17 h	653.17	4.18
	CPLEX	138	9.41 sec	648.84	4.81
600	FFD	175	1.75 sec	817.19	
	ACO	167	2.01 h	784.75	3.96
	CPLEX	165	12.95 sec	776.14	5.02

Table 4.2: ACO-based VM placement: numerical simulation results

Figure 4.2 visualizes the results. As it can be observed, the computation time required to derive the placement and thus the energy spent in computation are higher using the ACO-based approach. This is because of our implementation which is *far from being optimal* while the used LP-solver (i.e. CPLEX) is *highly optimized*. In particular, 1.75 sec were required to compute the placement for the highest number of VMs (i.e. 600) by the FFD and 2.01 hours by the ACO-based algorithm, resulting in 0.09 Wh and 397.98 Wh of energy spent in computation. Nevertheless, the solutions of the ACO-based approach utilize *significantly lower number of PMs* and thus yield to *superior average PM utilizations and energy gains*. Particularly, on average 4.7% of PMs and 4.1% of energy were saved by applying the ACO approach. Moreover, the solutions computed by the ACO-based approach are nearly optimal (i.e. small deviation of 1.1%). In addition, complexity of both evaluated algorithms is quadratic in the number of VMs, while CPLEX despite being highly efficient is exponential in the worst-case.

Finally, it is worth mentioning that under a constrained number of PMs such as it is the case in a real system, FFD would need a longer time to place the VM as it requires higher number of PMs to place the same number of VMs. Consequently, the number of VMs which are required to reside in queues (i.e. *non-allocatable*) is higher when the FFD approach is applied.

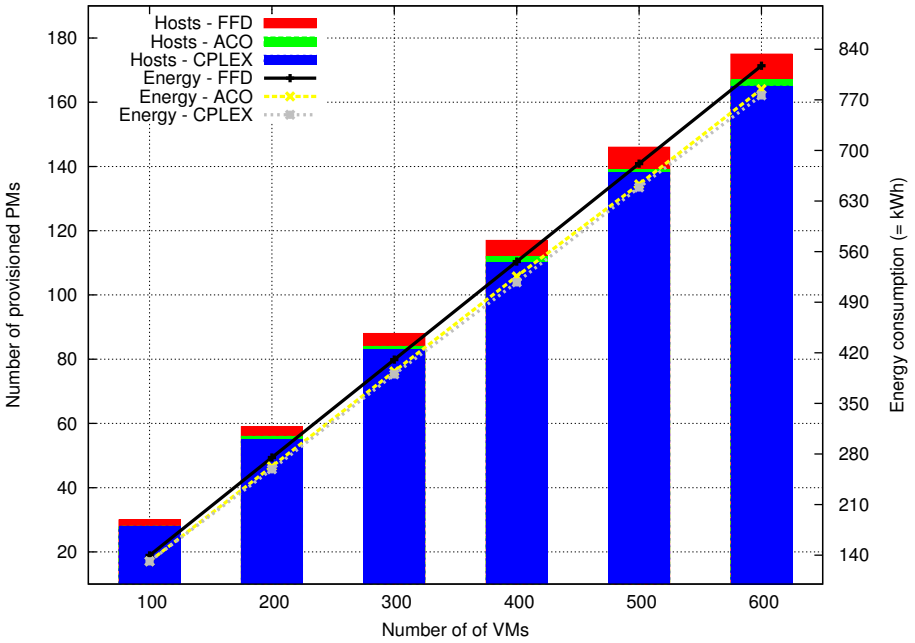


Figure 4.2: ACO-based VM placement: number of utilized PMs and energy consumption

4.3 VM Consolidation Algorithm and System

In the previous section we have introduced a novel ACO-based VM placement algorithm. Our results have shown that while the ACO-based algorithm computes near optimal solutions, its scalability is limited to a small number of PMs and VMs. Moreover, while providing a solution to the VM placement problem is important, once VMs are allocated to PMs, VM consolidation should be performed in order to: (1) remove resource fragmentation after a number of VMs have been added and removed to (resp. from) from the system; (2) facilitate the creation of idle times in order to create opportunities to transition servers into a low-power state. In order to enable VM consolidation as well as to improve the ACO scalability this section makes the following two contributions. First, we adapt our previously proposed ACO-based VM placement algorithm to the VM consolidation problem. Particularly, the new algorithm takes the current VM placement into account and attempts to minimize the number of migrations in order to arrive to the consolidated state. Second, we propose a novel fully decentralized VM consolidation system based on an unstructured P2P network of PMs. Both contributions are presented in the following sections. We start our discussion with the introduction of the assumptions. Then, the ACO-based VM consolidation algorithm is presented. Afterwards, the fully decentralized VM consolidation system is detailed. Finally, the evaluation results of both contributions are discussed.

4.3.1 Assumptions

This work considers the *VM consolidation problem*. Particularly, VMs are assumed to be already placed on the PMs and VM consolidation is triggered periodically according to the system administrator specified interval to *repack the VMs* on the least number of PMs while minimizing the number of migrations. Our VM consolidation decisions are based on the *requested* VM capacity as specified during VM deployment. Note, that given the appropriate VM resource demand estimation mechanisms can be provided the proposed system could be adapted to consider the estimated VM resource demands. This could further improve the data center resource utilization as it would enable more dense VM packing. Similarly to the previously introduced VM placement algorithm, PMs are assumed to be homogeneous while VMs are heterogeneous. VM live migration is assumed to be available.

4.3.2 Algorithm Description

This section presents the adaptation of the ACO-based VM placement algorithm to enable VM consolidation while minimizing the number of migrations. Particularly, we detail the modifications done to the objective function, heuristic information, pheromone evaporation rule, and the algorithm pseudo-code.

4.3.2.1 Objective Function

The Objective Function (OF) we attempt to maximize is defined by Eq. 4.8. It takes as input the set of PMs and a migration plan MP. Migration plan denotes the ordered set of new VM to PM assignments.

$$\textbf{max } \mathbf{f}(\mathbf{P}, \mathbf{MP}) := (nReleasedPMs)^e \times \text{Var}((|\mathbf{l_p}|_1)_{p \in P})^g \times (\frac{1}{|MP|})^m \tag{4.8}$$

Contrary, to the OF of the VM placement algorithm which solely focused on minimizing the number of PMs, the new OF is designed to favour the number of released PMs, the variance [154] of the scalar valued PM used capacity vectors $\mathbf{l_p}$, and smaller migration plans. In other words, the higher the number of released PMs and the variance between the PMs used capacity vectors, the better it is. Indeed, one of our objectives is to release as many PMs as possible. Releasing PMs also helps to increase the variance which is an important indicator for increased resource utilization. Particularly, a high variance shows that some PMs are more utilized than others. We use the L1 norm [294] to compute the scalar values. The second objective is to minimize the number of migrations. Consequently, we favour migration plans with the least number of migrations. This is reflected by defining the OF to be inverse proportional to the migration plan size. Migration plans with high number of migrations (i.e. VM-PM pairs) will lower its value, while smaller migration plans will increase it.

Three parameters, $e, g, m > 0$, are used to either give more weight to the number of released PMs, the PM load variance or the migration plan size.

4.3.2.2 Probabilistic Decision Rule

We now present the definition of the probabilistic decision rule which gives the probability for an ant to choose a VM v to be migrated to PM p. Note, that we rely on the same probabilistic decision rule as used by the previously introduced VM placement algorithm. For the sake of the ease of readability and completeness we present it again in Eq. 4.9.

$$Pr_p^v := \frac{[\tau_{v,p}]^\alpha \times [\eta_{v,p}]^\beta}{\sum_{v \in V_p} [\tau_{v,p}]^\alpha \times [\eta_{v,p}]^\beta}, \quad \forall v \in V, \forall p \in P \tag{4.9}$$

where $\tau_{v,p}$ represents the amount of pheromone associated with a particular VM-PM pair. The probabilistic decision rule relies on a set V_p which represents the the set of VMs hosted by a PM p. $\eta_{v,p}$ denotes the heuristic information. Its definition is shown in Eq. 4.10.

$$\eta_{v,p,} := \begin{cases} \frac{\kappa_{v,p}}{|MP|} & \text{if } \mathbf{l_p} + \mathbf{RC_v} \leq \mathbf{TC_p} \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

The key idea of the heuristic information to emphasize VM to PM migrations which yield in: (1) high PM used capacity and (2) are part of a small migration plan. Consequently, $\eta_{v,p}$ is defined as the ratio between $\kappa_{v,p}$ and the migration plan size in case the VM fits into the PM, and 0 otherwise. We use the constraint $\mathbf{l_p} + \mathbf{RC_v} \leq \mathbf{TC_p}$ to prevent new VMs from exceeding the total PM capacity.

To reward VMs which fill the PMs better $\kappa_{v,p}$ is defined as the inverse of the scalar valued difference between the static PM capacity and the utilization of the PM after placing VM v. Consequently, VMs which yield to better PM used capacity result in higher $\kappa_{v,p}$ value. Its definition is shown in Eq. 4.11. Note, that $\kappa_{v,p}$ is the equivalent of the $\eta_{v,p}$ in the previously introduced VM placement algorithm.

$$\kappa_{v,p} := \frac{1}{|\mathbf{TC_p} - (\mathbf{l_p} + \mathbf{RC_v})|_1} \tag{4.11}$$

Finally, two parameters, α, $\beta \geq 0$ are used to either emphasize the pheromone or heuristic information.

4.3.2.3 Pheromone Trail Update

After all ants have computed a migration plan, the pheromone trail update rule is used to reward VM-PM pairs which belong to the smallest migration plan (MP_{gBest}) as well as to simulate pheromone evaporation on the remaining VM-PM pairs. The pheromone trail update rule $\tau_{v,p}$ is defined in Eq. 4.12.

$$\tau_{v,p} := (1 - \rho) \times \tau_{v,p} + \Delta_{\tau_{v,p}}^{best}, \quad \forall (v,p) \in V \times P \tag{4.12}$$

where ρ, $0 \leq \rho \leq 1$ is used to control the evaporation rate. Consequently, higher values for ρ yield to faster pheromone evaporation. In order to reward VM-PM pairs which belong to the best migration plan, $\Delta_{\tau_{v,p}}^{best}$ is defined as the cycle-best VM-PM pheromone amount. Particularly, VM-PM pairs which belong to the best migration plan receive an increasing pheromone amount and thus become more attractive during subsequent cycles. Other pairs,

which are not part of the best migration plan continue loosing pheromone and thus become less attractive.

The goal of the algorithm is to release as many PMs as possible using the least number of migrations. Consequently, it attempts to maximize the OF f. Therefore, $\Delta^{best}_{\tau_{v,p}}$ is defined to give $f(P, MP_{gBest})$ pheromone amount to VM-PM pairs (v, p) which belong to MP_{gBest}, and 0 otherwise.

$$\Delta^{best}_{\tau_{v,p}} := \begin{cases} \mathbf{f(P, MP_{gBest})} & \text{if } (v,p) \in MP_{gBest} \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

Finally, to bound the pheromone amount on the VM-PM pairs, $\tau_{v,p}$ is restricted to $[\tau_{min}, \tau_{max}]$.

4.3.2.4 Pseudocode

The pseudo-code is shown in Algorithm 5. It takes as input the set of PMs P including their associated VMs and a set of parameters (e.g. τ_{max}, α, β, nCycles, nAnts) (line 1). The algorithm then sets the pheromone value on all the VM-PM pairs to τ_{max} and iterates over a number of nCycles (lines 5 to 35). In each cycle multiple (nAnts) ants compute migration plans concurrently in parallel (lines 7 to 21). The migration plans accommodate at most $|VM|$ migrations (line 10). Particularly, first the ants compute a probability $p_{v,p}$ for migrating a VM v to PM p for all VMs and PMs (line 11). Based on the computed probability they choose a VM-PM pair (v, p) stochastically and add it to the migration plan (lines 12 to 13). The source and destination PM capacity is then updated by removing the selected VM from the source and adding it to the destination PM (line 14). Afterwards, a score is computed by applying the OF (see Eq. 4.8) on the set of PMs P and the migration plan (MP_{tmp}) (line 15). Finally, if the newly computed score is greater than the local best score the local best score is updated and the VM-PM pair (v, p) is added to the local migration plan (lines 16 to 19). Note, that a VM is allowed to appear multiple times in the migration plan as long as it yields to a better score. After all ants have finished computing the migration plans, they are compared according to the OF f. The cycle best migration plan is selected and saved as MP_{cBest} (line 22). If the cycle best migration plan is also the global best one, it becomes the new global best one (lines 23 to 25). Finally, the pheromone values on all VM-PM pairs are updated by applying the pheromone trail update rule (see Eq. 4.12) and enforcing the τ_{min}, τ_{max} bounds (lines 26 to 34). The algorithm terminates after nCycles and returns the global best migration plan MP_{gBest} (line 36).

4.3.3 System Description

In order to improve the scalability of VM consolidation algorithms such as the one presented in the previously section, we have designed a novel fully decentralized VM consolidation system based on an unstructured P2P network of PMs. This section is devoted to the description of this system. First, the design principles are discussed. Afterwards, the neighbourhood topology construction mechanism is detailed. Finally, the VM consolidation process is presented.

Algorithm 5 ACO-based VM consolidation

Input: Set of PMs P with their associated VMs, Set of parameters
Output: Global best migration plan $\mathbf{MP_{gBest}}$

$MP_{gBest} := \emptyset$
Set pheromone value on all VM-PM pairs to τ_{max}
for all $q \in \{0 \dots nCycles - 1\}$ **do**
  for all $a \in \{0 \dots nAnts - 1\}$ **do**
    $Score_{lBest} := 0$
    $MP_{tmp}, MP_a := \emptyset$
    while $|MP_{tmp}| < |VMs|$ **do**
      Compute Pr_p^v, $\forall v \in V, \forall p \in P$
      Choose (v, p) randomly according to the probability Pr_p^v
      Add (v, p) to the migration plan MP_{tmp}
      Update PMs used capacities
      $Score_{tmp} := f(P, MP_{tmp})$
      if $Score_{tmp} > Score_{lBest}$ **then**
        $Score_{lBest} := Score_{tmp}$
        $MP_a := MP_a \cup \{(v, p)\}$
      end if
    end while
  end for
  Compare ants migration plans and choose the best one according to the objective function $f(P, MP_a) \rightarrow$ Save cycle best migration plan as MP_{cBest}
  if $f(P, MP_{cBest}) > f(P, MP_{gBest})$ **then**
    $MP_{gBest} := MP_{cBest}$
  end if
  for all $(v, p) \in V \times P$ **do**
    $\tau_{v,p} := (1 - \rho) \times \tau_{v,p} + \triangle\tau_{v,p}^{best}$
    if $\tau_{v,p} > \tau_{max}$ **then**
      $\tau_{v,p} := \tau_{max}$
    end if
    if $\tau_{v,p} < \tau_{min}$ **then**
      $\tau_{v,p} := \tau_{min}$
    end if
  end for
end for
return Global best migration plan $\mathbf{MP_{gBest}}$

4.3.3.1 Design Principles

The key idea of our system to achieve scalability as well as high packing efficiency is to apply VM consolidation only in the scope of small, randomly formed neighbourhoods of PMs. Limiting VM consolidation to a small set of nodes greatly decreases its computation time, while the randomized neighbourhoods allow the system to achieve a high packing efficiency by *periodically applying the VM consolidation algorithms in the scope of the neighbourhoods*.

As the neighbourhoods are modified periodically and randomly the entire system tends to converge towards a high packing efficiency by solely making local VM consolidation decisions within neighbourhoods without the need of a central server.

In order to enable the construction of random neighbourhoods we rely on the Cyclon protocol [291]. Cyclon is an epidemic membership protocol which allows to periodically construct randomized P2P overlays in which each PM has only a partial system view, the so-called neighbourhood. This property allows the system to scale with increasing number of PMs as it does not rely on a central server.

4.3.3.2 Neighbourhood Construction Mechanism

We now briefly discuss how the neighbourhoods are constructed. Our neighbourhood construction mechanism is based on the Cyclon membership protocol. Cyclon has been designed for fast information dissemination while dealing with a high number of PMs that can join and leave the system. It is based on a periodic and random exchange of neighbourhood information among the peers, the so called *shuffling operation*. Each peer maintains a local list of neighbours, called *cache entries*. A cache entry contains the address (IP/port) and the *age* value of a neighbour. The role of the *age* field is to bound the time a neighbour is chosen for shuffling thus facilitating the early elimination of dead peers. The shuffling operation is repeated periodically according to a parameter $\lambda t > 0$ on each PM. Each time a shuffling operation is performed the PM obtains a new partial view of the system, the so-called neighbourhood. The resulting system topology can be viewed as a directed graph where vertices represent PMs and edges the relations. For example, $X \longrightarrow Y$ means *Y is a neighbour of X*. Note, that the relations are asymmetric (i.e. Y is a neighbour of X does not imply that X is a neighbour of Y). More details on the system topology construction can be found in [291].

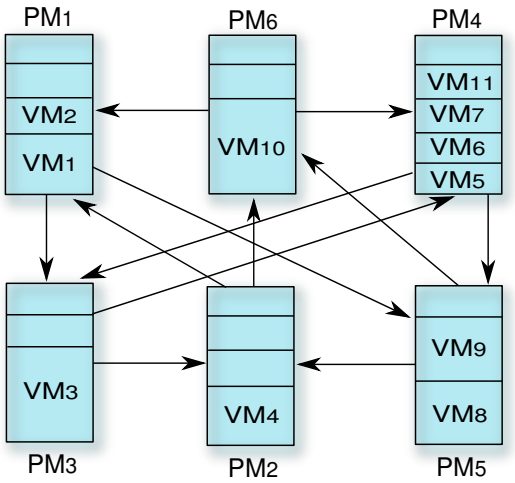
4.3.3.3 VM Consolidation Process

Each PM periodically triggers a VM consolidation process within its neighbourhood in order to optimize the VM placement. Locks are associated with all PMs in order to avoid concurrent access to PM resources in case of multiple ongoing consolidations. The VM consolidation process is composed of the following six steps:

1. First, PM p which initiates the consolidation checks whether it is not involved in an on-going consolidation. If not it attempts to acquire a lock for each member (including itself) of its neighbourhood. Otherwise, the consolidation is aborted. Acquiring a lock is a *non-blocking operation*. If it does not succeed, the member will not participate in the consolidation process.
2. For each successful lock acquisition, PM p requests from the corresponding neighbour PM its total capacity, currently packed VMs and their requested capacity vectors.
3. The VM consolidation algorithm is started once all the resource information is received from the locked members. It outputs a migration plan which corresponds to the ordered set of the new VM-PM assignments. Any VM consolidation algorithm can be used in this operation.
4. An actuation module on the PM enforces the migration plan by sending migration operations to the PMs hypervisors.

5. After the actuation all locks are released.
6. Each PM which does not accommodate VMs anymore power-cycles itself in order to save energy.

We now illustrate how the VM consolidation process works. Figure 4.3 depicts one example system topology which is constructed using the Cyclon protocol. It is composed of six PMs and eleven VMs that are distributed among the PMs. The neighbourhood size is two. For the sake of simplicity a single resource "number of cores" is considered in this example: physical (PCORES) and virtual (VCORES). PMs are homogeneous and have five PCORES. VMs can request one, two or three VCORES. The total capacity of PMs in the system is 30 PCORES. 19 PCORES are currently utilized, which corresponds to an utilization of approximately 63%.



PM	Neighbours
1	{3,5}
2	{1,6}
3	{2,4}
4	{3,5}
5	{2,6}
6	{1,4}

Figure 4.3: Fully decentralized VM consolidation: example system topology

Starting from the initial state as shown in Figure 4.3, PM_4 initiates the first consolidation with its neighbours PM_3 and PM_5. The result of this consolidation is shown in Figure 4.4 (1). VM_{11} has been migrated from PM_4 to PM_5 and VM_6 & VM_7 have been migrated from PM_4 to PM_3. PM_5 and PM_3 resources are now better utilized than in the previous configuration. In Figure 4.4 (2), PM_2 triggers a consolidation with its neighbours PM_1 and PM_6. VM_4 has been migrated from PM_2 to PM_1. PM_1 is now fully utilized and PM_2 has become idle. Finally, in Figure 4.4 (3) PM_6 starts another consolidation with PM_1 and PM_4. VM_5 has been moved from PM_4 to PM_6. The node PM_4 has now become idle and PM_6 better utilized. The final system state with two released PMs (PM_2 and PM_4) is shown in Figure 4.4 (4). It results in a new, almost global optimal data center utilization of approximately 95%.

4.3.4 Evaluation

This section presents the evaluation of the proposed VM consolidation algorithm and the fully decentralized VM consolidation system. First, the prototype implementation principles are introduced. Afterwards, the system setup is detailed and the results are discussed.

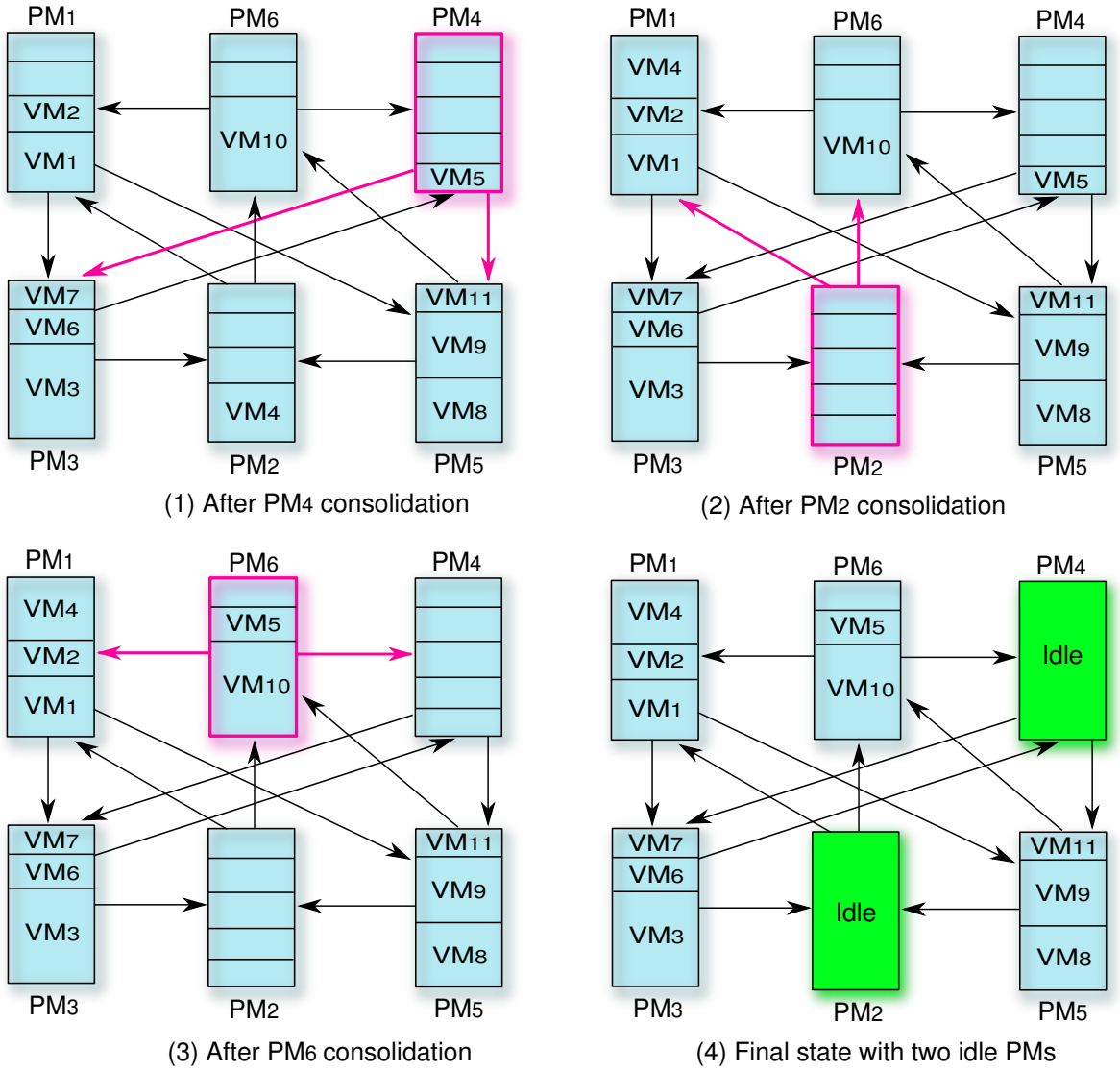


Figure 4.4: Fully decentralized VM consolidation: process example

4.3.4.1 Prototype Implementation

To validate our work we have implemented a distributed Python-based VM consolidation system emulator which integrates the Cyclon membership protocol [291]. Each PM is emulated by a daemon which listens for TCP connections to communicate with other PMs. One node serves as the introducer to bootstrap the system. To prevent concurrent access to PMs the prototype implementation integrates a distributed locking mechanism. PMs shutdown themselves when they do not host any VMs. VMs are represented by their requested capacity vectors. Each PM writes events (e.g. migration, consolidation, shutdown) in a local SQLite database during the experiment execution. Once the experiment is finished all databases are collected and merged into a single one for post-analysis. Emulator modules such as the introducer mechanism, consolidation algorithms, scheduler for shuffling and consolidation are defined in a configuration file for the ease of replacement. The current

implementation integrates four VM consolidation algorithms: the FFD [304], Sercon [224], V-MAN [216] state of the art algorithms and the introduced migration-cost aware ACO-based algorithm. FFD is an VM placement algorithm which is often applied in the context of VM consolidation. Consequently, it serves as the baseline for comparison in our work. In contrast, Sercon, V-MAN and our algorithm are VM consolidation algorithms which were specifically designed to reduce the number of migrations.

4.3.4.2 System Setup

We have deployed the emulator on 42 servers of the Grid'5000 testbed in France. All servers are equipped with two CPUs each having 12 cores (in total 1008 cores). This allowed us to emulate one PM per core. In other words, throughout all the experiments each server hosts 24 emulator instances (one per core) which represent the emulated PMs. The emulator considers three types of resources: CPU, memory, and network. It supports six kinds of VM instances: nano, micro, small, medium, large and xlarge, which are represented by their corresponding requested capacity vectors: (0.2, 0.5, 0.1), (1, 1, 1), (2, 1, 1), (4, 2, 2), (8, 4, 4) and (16, 8, 4) respectively. PMs have a total capacity of (48, 26, 20). They host 6 VMs, one of each type at the beginning of the experiment. Consequently, in total 6048 VMs are emulated. The experiment runs for six minutes. Consolidation is triggered by the PMs concurrently and independently every 30 seconds. The neighbourhood size is set to 16 PMs and the shuffling operation is triggered every 10 seconds by the PMs. Table 4.3 provides a summary of the introduced system parameters and their corresponding values. The ACO parameters shown in table were derived empirically through numerous experiments. We run the emulator once for each of the evaluated algorithms: FFD, Sercon, V-MAN and the proposed ACO-based algorithm. The evaluation is focused on: (1) analysis of the number of active PMs (packing efficiency) and migrations; (2) scalability of the system; and (3) comparison of the packing efficiency with the centralized topology for all the VM consolidation algorithms.

Table 4.3: Fully decentralized VM consolidation: system parameters

Parameter	**Value**
Number of PMs and VMs	1008 (resp. 6048)
Experiment duration	360s
Consolidation interval	30s
Shuffling interval	10s
Neighbourhood size	16 PMs
Considered resources	CPU, memory and network
PM total capacity vector	(48, 26, 20)
VM requested capacity vectors	(0.2, 0.5, 0.1), (1, 1, 1), (2, 1, 1), (4, 2, 2), (8, 4, 4), (16, 8, 4)
ACO parameters: α, β, ρ, τ_{min}, τ_{max}, e, g, m, $nCycles$, $nAnts$	0.1, 0.9, 0.1, 0.2, 1, 5, 3, 1, 2, 2

4.3.4.3 Number of Active PMs and Migrations

In this section we analyze the number of active PMs and migrations resulting from the evaluated algorithms. First, the number of active PMs is analyzed. The results of this evaluation are shown in Figure 4.5. As it can be observed the consolidation phase starts at the 30th

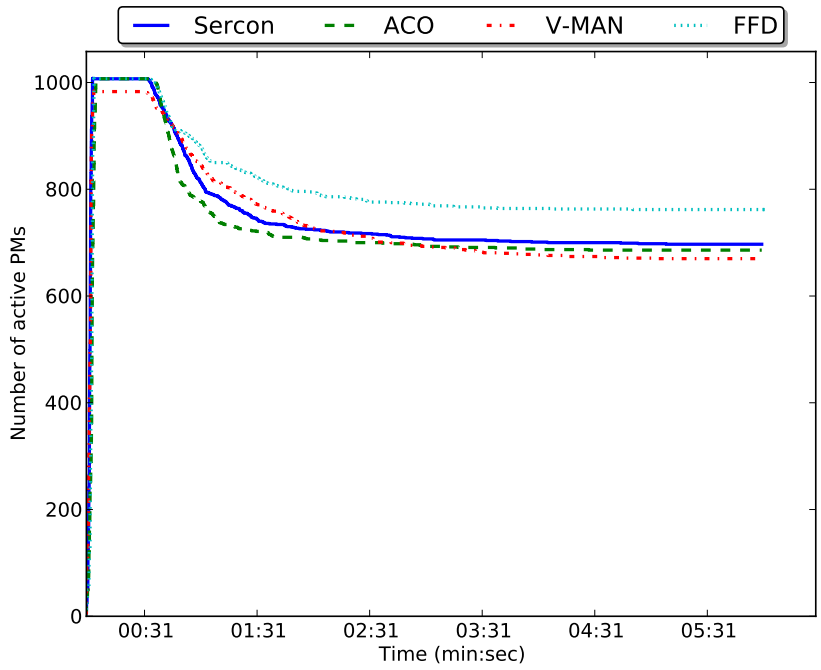


Figure 4.5: Fully decentralized VM consolidation: number of active PMs

second. FFD performs the worst as it only manages to release 246 nodes. V-MAN achieves the best result with 323 released PMs which is closely followed by the ACO-based algorithm with 322 released PMs. Note, that Sercon performs worse than V-MAN and ACO.

Figure 4.6 depicts the number of migrations with the progress of the experiment. As it can be observed the number of migrations quickly converges towards zero with Sercon, V-MAN and the ACO algorithm thus demonstrating the good reactivity of our system. Note, that the ACO algorithm requires more migrations than Sercon. Indeed, it trades the number of migrations for the amount of released PMs (see Figure 4.5). Finally, V-MAN performs the worst among the three VM consolidation algorithms. FFD yields in a tremendous amount of migrations (in total 96494). We explain this with the fact the algorithm is *not designed to take into account the current VM-PM assignment*. Particularly, due to its static nature it assumes that the PMs do not host any VMs prior computing the new VM-PM assignment resulting in a permanent movement of most of the VMs in each VM consolidation iteration.

4.3.4.4 Scalability

To evaluate the scalability of our system we have varied the number of PMs from 120 to 1008 and analyzed the obtained packing efficiency. The results are summarized in Table 4.4.

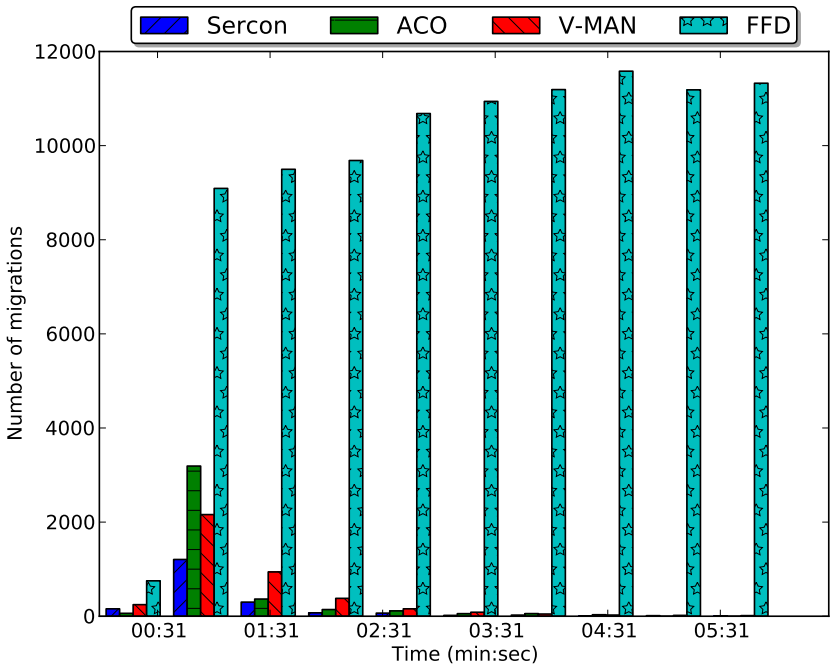


Figure 4.6: Fully decentralized VM consolidation: number of migrations

As it can be observed, except the outlier with V-MAN at 504 PMs, the packing efficiency does not change significantly with increasing numbers of PMs and VMs, thus demonstrating the good scalability of our system.

Table 4.4: Fully decentralized VM consolidation: scalability

Algorithm	**PMs**	**VMs**	**Released PMs**	**Migrations per VM**	**Packing efficiency (%)**
FFD	120	720	29	26	24.1
	240	1440	58	26	24.1
	504	3024	124	27	24.6
	1008	6048	246	26	24.4
ACO	120	720	36	5	30.0
	240	1440	77	7	32.0
	504	3024	161	8	31.9
	1008	6048	322	9	31.9
V-MAN	120	720	39	3	32.5
	240	1440	79	5	32.9
	504	3024	122	4	**24.2**
	1008	6048	323	4	32.0
Sercon	120	720	37	1	30.8
	240	1440	74	1	30.8
	504	3024	155	1	30.7
	1008	6048	311	1	30.8

Another important metric to evaluate is the *maximum number of migrations per VM* during the whole duration of the experiment. In other words, due to the fully decentralized nature of the system and the random neighbourhood construction, VMs could traverse multiple PMs during subsequent consolidation rounds. As our results show, the maximum number of migrations per VM highly depends on the current VM-PM assignment and the VM consolidation algorithm, less on the number of PMs and VMs. Particularly, in the current setup, Sercon requires at most one migration per VM. On the other hand, V-MAN results in at most 5 and ACO needs at most 9 migrations. Finally, FFD as it does not consider the current VM-PM assignment yields to the largest number of migrations.

4.3.4.5 Comparison with a Centralized System Topology

Table 4.5 depicts the results from the comparison of the number of migrations and packing efficiency of our approach with the centralized topology for 1008 PMs and 6048 VMs. To simulate a centralized topology we have run the VM consolidation algorithms (FFD, Sercon, ACO) on a single PM. Note that, V-MAN is a decentralized algorithm thus its evaluation is not part of the centralized topology evaluation. As it can be observed the ACO-based

Table 4.5: Fully decentralized VM consolidation: centralized vs. unstructured P2P

Topology	Algorithm	Migrations	Released PMs	Packing efficiency (%)
Centralized	FFD	6040	249	24.7
	Sercon	1920	320	31.7
	ACO	-	-	-
P2P	FFD	96494	246	24.4
	V-MAN	4189	323	32.0
	ACO	4015	322	31.9
	Sercon	1872	311	30.8

VM consolidation algorithm is unable to compute a solution in a reasonable amount of time when used in the centralized topology for this kind of scale. Sercon on the other hand outperforms FFD in both the number of migrations (1920 vs. 6040) and released PMs (320 vs. 249). This is not further surprising as in contrast to FFD, Sercon is designed to minimize the number of migrations. More interestingly, our fully decentralized VM consolidation system achieves almost equivalent packing efficiency for the evaluated algorithms when compared to the centralized topology. When considering the number of migrations, FFD achieves the worst result with 96494 migrations. We explain this with the fact that the algorithm by nature does not take into account the current VM-PM assignments. Consequently, its solutions result in a permanent reassignment of VMs within neighbourhoods during subsequent consolidation rounds.

Our ACO-based algorithm outperforms FFD as well as Sercon in the number of released PMs and performs equal with V-MAN. However, the gains in the number of released PMs come at the cost of an increased number of migrations. For example, when compared to Sercon twice as many migrations are required. On the other hand, when considering V-MAN more than 150 migrations are saved by the ACO algorithm. This demonstrates that the ACO algorithm can serve as a competitive alternative to the other evaluated algorithms in the fully decentralized VM consolidation system.

4.4 Summary

In this chapter we have presented three novel contributions. The first contribution is a nature-inspired VM placement based on the ACO. To the best of our knowledge this is the first work to apply ACO on the Multi-dimensional Bin Packing Problem. Moreover, it is the first work to evaluate the benefits of ACO in the context of VM placement. We have compared the proposed algorithm with the traditional FFD greedy algorithm. Both algorithms have been implemented and validated by means of simulations. The simulation results demonstrate that the ACO-based approach provides superior energy gains than the FFD algorithm and computes near optimal (1.1% deviation) solutions as computed using the IBM ILOG CPLEX optimizer. Particularly, on average 4.7% of PMs and 4.1% of energy were saved by applying the ACO-based algorithm compared to FFD. Nevertheless, despite its polynomial-time worst-case complexity our current implementation is limited to smaller number of PMs and VMs.

Motivated by the fact that our VM placement algorithm did not scale do a higher number of PMs and VMs, as well as the need to consider the VM consolidation problem in order to further consolidate already placed VMs this chapter has made two further contributions: (1) we have adapted our VM placement algorithm to the VM consolidation problem; (2) we have proposed a fully decentralized VM consolidation system based on an unstructured P2P network of PMs. The key idea of the fully decentralized VM consolidation system is to periodically and randomly form neighbourhoods of PMs. VM consolidation is applied periodically only within the scope of the neighbourhoods thus allowing the system to scale with increasing number of PMs and VMs as no global system knowledge is required. Moreover, the randomized neighbourhood construction property facilitates the VM consolidation convergence towards a global packing efficiency very similar to a centralized system by *leveraging existing centralized VM consolidation algorithms*.

A distributed Python-based VM consolidation-enabled Cyclon P2P system emulator was implemented and used it to evaluate two state of the art VM consolidation algorithms, namely Sercon and V-MAN along with our ACO-based VM consolidation algorithm. The evaluation was conducted on the Grid'5000 testbed which allowed to emulate up to 1008 PMs and 6048 VMs. The results show that the proposed fully decentralized VM consolidation system achieves a global packing efficiency very close to a centralized topology for all the evaluated algorithms. Moreover, the system remains scalable with increasing numbers of PMs and VMs. Finally, the proposed ACO-based VM consolidation algorithm outperforms FFD and Sercon in the number of released PMs and requires less migrations than FFD and V-MAN when used in our fully decentralized VM consolidation system. Table 4.6) summarizes the evaluation results.

Table 4.6: Fully decentralized VM consolidation: evaluation summary

Criteria	Best algorithm	2nd	3rd	4th
#Migrations	Sercon	ACO	V-MAN	FFD
Packing efficiency	V-MAN	ACO	Sercon	FFD

Chapter 5

Conclusion

Contents

THIS chapter wraps up the contributions of this thesis and presents future research directions.

5.1 Contributions

Cloud computing has recently emerged as a new computing paradigm which allows customers to lease services based on the pay-as-you-go model. Customers are charged for only what they use. To support the customers growing service demands cloud providers are now building an increasing number of large-scale data centers. Managing such data centers is a challenging task as it involves the design of novel cloud management frameworks and algorithms which are not only able to operate at scale but also lower the data center energy consumption during periods of low resource utilization. This thesis has focused on the IaaS cloud service model whose goal is to offer compute infrastructure by provisioning VMs on-demand. Particularly, in this thesis we have investigated the challenge of designing, implementing, and evaluating an autonomic and energy-efficient IaaS cloud management system for private clouds. In order to achieve this goal, Chapter 2, has first introduced the context of this work, namely server virtualization, autonomic computing, cloud computing, and energy management in computing clusters. Then, it has reviewed the related work on the design and implementation of autonomic, scalable, and energy-efficient IaaS cloud management systems and highlighted their limitations. Based on the lessons learned this thesis has proposed the following three novel contributions:

Snooze: A Scalable, Autonomic, and Energy-Efficient IaaS Cloud Manager. In order to address the scalability, autonomy, and energy efficiency limitations of existing IaaS cloud

management systems, this thesis has proposed Snooze, a novel scalable, autonomic, and energy efficient IaaS cloud management system for private clouds. In contrast to existing IaaS cloud management systems which are mostly based on centralized architectures and lack of autonomy and energy efficiency mechanisms, for scalability Snooze employs a *self-configuring and healing hierarchical architecture* for the VM management system and performs distributed VM management. To conserve energy Snooze provides a *unique holistic energy-efficient VM management solution*. Particularly, Snooze provides VM placement, ships with integrated VM resource (CPU, memory, network Rx, network Tx) utilization monitoring and estimation mechanisms, performs event-based underload and overload detection via data aggregation and smoothing while considering all resource dimensions, implements polynomial times greedy algorithms to resolve underload and overload situations, incorporates a modified version of the Sercon [224] algorithm for periodic VM consolidation which is the first real implementation of the algorithm, and finally performs power management by automatically detecting and power-cycling idle PMs. Idle PMs are woken up either in case not enough online PMs are available during VM placement or overload situations.

Snooze was extensively evaluated on more than 140 nodes of the Grid'5000 experimentation testbed using realistic applications. It was shown to be scalable, autonomic, and energy efficient. Thanks to its flexible design, Snooze allows researchers to plugin and experiment with novel VM management algorithms in a realistic environment. Moreover, the VM resource utilization data exported by Snooze can be exploited by elastic cloud services able to scale up and down VMs on-demand depending on the services QoS requirements (e.g. response time, deadlines) and current VM resource utilization. The software prototype has been distributed in open-source under the GPL v2 license at http://snooze.inria.fr since May 2012. It is known to be used by researchers (e.g. at IRIT Toulouse, LIFL) to experiment with VM management algorithms and has been successfully validated on experimentation testbeds at EDF R&D and Medio Seattle. Recently, an engineer was hired by Inria as part of the Snooze technological development action to support the development of the system.

VM Placement via Ant Colony Optimization. One issue which arises during the VM submission in IaaS cloud management systems is to place the VMs on PMs such that the number of PMs is minimized. Minimizing the number of PMs during VM submission is an NP-hard combinatorial optimization problem and thus is expensive (in time and space) to compute with increasing numbers of PMs and VMs. Many of the existing VM placement algorithms are limited to a single resource (e.g. CPU) and rely on centralized greedy algorithms such as First-Fit Decreasing (FFD) which are known to be hard to distribute/parallelize [76]. To address those limitations we have investigated the use of Ant Colony Optimization (ACO) for VM placement and proposed a novel ACO-based VM placement algorithm. ACO is especially attractive for VM placement due to its polynomial time worst-case complexity, close to optimal solutions, and the ease of parallelization. The proposed algorithm was compared with the FFD algorithm by means of simulations. Moreover, the optimal solution was computed using the IBM ILOG CPLEX optimizer. The results have shown that the proposed algorithm outperforms FFD in the number of released PMs and computes close to optimal (i.e. 1.1% deviation) solutions at the cost of increased execution time.

VM Consolidation via Ant Colony Optimization. The previous contribution has shown that ACO is able to compute close to optimal solutions. However, the scalability of the

proposed algorithm was limited to a small number of VMs and PMs. Moreover, while considering the VM placement problem is important, once VMs are placed, VM consolidation should be performed to remove resource fragmentation and thus improve the overall data center resource utilization. This can be achieved by continuously repacking already placed VMs on the least number of PMs. To tackle both issues this thesis has made two novel contributions. The first contribution has adapted the previously proposed ACO-based VM placement algorithm to enable *VM consolidation while minimizing the number of migrations*. In order to improve the scalability of the algorithm, the second contribution has proposed a novel *fully decentralized VM consolidation system based on an unstructured P2P network of PMs*. Considering the complexity of the VM consolidation problem, the key idea of the system is to apply VM consolidation only within small sets of PMs, the so-called neighbourhoods. This allows the system to scale with increasing numbers of PMs and VMs as no global system knowledge is required. Finally, in order to facilitate the VM consolidation convergence towards a global packing efficiency very similar to a centralized system, neighbourhoods are randomly modified via exchange of contact information between PMs. To evaluate both contributions we have developed a distributed emulator of the proposed fully decentralized VM consolidation system and deployed it on the Grid'5000 experimentation testbed. The emulator integrates two state of the art VM consolidation algorithms (i.e. Sercon and V-MAN) as well as the proposed ACO-based VM consolidation algorithm. It was used to emulate up to 1008 PMs and 6048 VMs. Our results have shown that the proposed system achieves good scalability and a packing efficiency very close to the one achieved with a centralized system.

5.2 Perspectives

As part of our research we have identified a number of future research directions. They can be divided in to five categories: (1) Improving the Snooze software; (2) Further evaluation of Snooze; (3) Autonomy; (4) Improved VM management mechanisms; (5) Energy and thermal aware data center management.

Improve the Snooze Software. We have identified two improvement directions for the Snooze software. First, the Snooze usability could be improved in order to make it more user friendly. Second, some of the its implementation limitations should be resolved in order to reach a broader community. Regarding the usability, in order to make Snooze interoperable with a wide range of tools developed for open-source and proprietary cloud management systems over the past years, well-known cloud management interfaces such as EC2 [9] and OCCI [43]) should be implemented on top of the Snooze own RESTful interface. In addition, recently a unified IaaS cloud management interface called Apache Libcloud [143] was introduced. Providing a Snooze driver for Libcloud could make Snooze more attractive for users which have already developed tools leveraging this interface. Last but not least, a graphical user interface could further increase the system usability.

We now present a number of improvements which could be done to the current prototype implementation. The first improvement involves the VM resource utilization data, storage, and networking management. VM resource utilization data is used to enable the VM management mechanisms (e.g. VM consolidation) decisions. It is currently stored inmemory. Consequently, the amount of data which can be preserved is limited. This lim-

itations can be resolved by integrating more scalable storage solutions such as MySQL [2] or Apache Cassandra [198]. Besides VM resource utilization data management, VM storage management is mandatory to automate the VM disk image propagation to PMs during the VM submission. For the time beeing, Snooze does not integrate any VM storage management mechanisms and assumes that the VM disk images are either hosted on a shared storage such as Network-File-System (NFS) or copied manually by the user to the compute nodes prior initiating VM submission. Finally, VM networking management is mandatory in order for the VMs to become reachable to the outside workload after they have booted. In order to achieve this, VMs need to get an IP address assigned. This can be achieved using different techniques (e.g. DHCP, manually). Currently, the Snooze VM networking manager assigns IP addresses manually to the VMs. This requires VMs to integrate a special contextualisation script which will perform the network configuration. It would be interesting to integrate alternative, more transparent IP address assignment approaches (e.g. DHCP).

The second improvement involves the *VM live migration* mechanisms. VM live migration allows Snooze to seamlessly move VMs between the PMs. In order to enable VM live migration, the NFS storage is required as most of the today's open-source hypervisors (e.g. KVM) do not support VM storage live migration. However, NFS results in a bottleneck which becomes critical especially when data intensive VMs are deployed. It is therefore interesting to investigate VM live migration with alternative, more distributed file systems such as GlusterFS, XtreemFS [175], or BlobSeer [229]. For instance, BlobSeer is known to have support for transparent VM storage live migration [230]. Finally, when VM live migrations are triggered concurrently, they all compete for the networking bandwidth. Given that each VM live migration has its own networking bandwidth demand, starvation can happen thus preventing certain VMs from beeing migrated. In order mitigate VM live migration starvation, live migration bandwidth capping features of the hypervisors should be used. Moreover, servers hosting the GM services could be placed in separate Virtual Local Area Networks (VLANs) thus providing networking bandwidth isolation.

The third improvement aims at enabling Snooze *multi-site deployment*. More precisely, today's cloud providers operate multiple clusters, possibly distributed in geographically different locations. For the time beeing, Snooze was validated on a single cluster. Moreover, it required IP multicast to be enabled on the networking infrastructure. In this context it is interesting to investigate how Snooze should be modified in order to remove this limitation. For instance, multicast could be implemented over messaging systems such as RabbitMQ [49] or ActiveMQ [142].

The fourth improvement related to the *security* of the system. For the time beeing Snooze does not integrate any security mechanisms such as user or system services authentication. This makes it vulnerable to attackers which could compromise a deployment by injecting malicious system services. For example, a deployment could be taken over by injecting fake LC or GM services. Security mechanisms should be integrated to mitigate such situations. Finally, a number of IaaS cloud management systems such as CloudStack [275], Eucalyptus [19], Nimbus [190], OpenNebula [223], and OpenStack [279] have been developed over the past years. In order for Snooze to reach a broader community it would be beneficial to integrate it with such systems. This would allow to take advantage of their tools ecosystem, mechanisms (e.g. storage management), as well as complement them with autonomy and energy efficiency mechanisms.

In order to improve Snooze and develop its open-source community, recently an engineer

was hired in the framework of the Snooze technological development action.

Further Evaluation of Snooze. We have identified a number of future evaluation opportunities. The first additional evaluation concerns the scalability of the system. We would like to deploy and evaluate Snooze on a larger number of PMs and VMs. In addition, it would be interesting to study the system stabilization time after a GL failure. While from our experience Snooze required only a few seconds to stabilize, we would like to evaluate and quantify the overheads more precisely. Ultimately, a direct scalability comparison with existing open-source IaaS cloud management systems (e.g. OpenStack) could bring interesting results. To the best of our knowledge no such evaluation has been performed yet. Indeed, not even among the existing IaaS cloud stacks (e.g. CloudStack, OpenNebula, Nimbus). The second evaluation targets the energy management mechanisms which were evaluated using a web application. It could be interesting to investigate how the energy management mechanisms would impact the energy and performance of scientific and data analysis applications such as MPI (resp. MapReduce). We have already started to work on scientific data analysis applications as part of a summer research internship at the Lawrence Berkeley National Laboratory in 2012 and plan to continue this activities in the future.

Moreover, the energy and performance overheads of different VM live migration techniques should be investigated. Particularly, in our experiments the pre-copy VM live migration technique was used. However, alternative live migration approaches (e.g. post-copy, hybrid) could potentially yield better performance and more energy savings. Finally, due to hardware restrictions on our testbed we could only use one power management technique (i.e. shutdown). Alternative power management methods such as suspend or hibernate could be evaluated in order to improve the wake up times. This could help to further lower the energy consumption and improve the performance. Finally, Snooze has been tested in a homogeneous environment. Given that nothing prevents it to be deployed in a heterogeneous environment it would be interesting to evaluate the effects of such a deployment on its VM management mechanisms.

Autonomy. Snooze implements a number of self-management properties (e.g. self-configuration and healing) which can be further improved. The first improvement could be done to *support VM failure recovery*. Particularly, when a LC fails all VMs which it hosts are also terminated. It would be interesting to extend the LCs such that they can periodically take VM snapshots on stable storage. This would allow the GMs to automatically restart VMs of the failed LCs on the remaining active LCs.

Another improvement involves the hierarchy management. Particularly, after some time the Snooze *hierarchy can become unbalanced* if Snooze system services (e.g. GMs, LCs) are shutdown either for maintenance reasons or due to failures. For instance, once a GM is shutdown all its managed LCs will join another GM. However, once a new GM is booted no new LCs will be assigned to it, unless new PMs are added to the cluster which are configured as LCs. After some time this behaviour will result in some GMs being heavily loaded while others remaining underutilized. Consequently, mechanisms and algorithms able to rebalance the hierarchy must be investigated. For example, a control-loop could be integrated on the GL which would instruct the GMs to perform hierarchical rebalancing actions based on the observed aggregated GM resource utilization. Ultimately, Snooze could be made even

more autonomic by *removing the distinction between GMs and LCs*. Consequently, the decisions when a PM should play the role of GM or LC in the hierarchy would be taken by the system instead of the system administrator upon configuration. Finally, in the current implementation a failing GM is never replaced by another one. Consequently, after a failure of the last GM the system will becomes unreachable. In this context it would be interesting to investigate automatic promoting of LCs to GMs or booting additional GMs. This will allow the system to operate even in such catastrophic scenarios. Last but not least, for energy saving reasons the GL could be enabled to automatically disable and enable parts of the hierarchy via shutdown (resp. wake up) of idle GMs.

Improved VM Management Mechanisms. A number of improvements can be done in the context of VM management mechanisms. The first improvement is related to the aggregated resource utilization. Particularly, the GL VM dispatching algorithm considers the available *aggregated resource utilization* of the GMs to determine GMs with enough resources to accommodate the VMs. However, aggregating resource utilization is not sufficient to take exact VM dispatching decisions. For example, when a GM reports to have 4 GB of available memory and a user submits a VM requesting 2 GB of memory, it does not necessary mean that the GM will finally be able to accommodate the VM. Indeed, the 4 GB could be the result of four LCs each running on PMs with 1 GB of memory. Currently, the GL VM dispatching algorithm returns a list of candidate GMs which are contacted by the GL until one with enough resources is found. Obviously this is not the most scalable approach. It would be interesting to investigate how this behaviour could be improved. For example, by studying metrics which could better capture the aggregated utilization or provide more detailed information thus allowing for more intelligent GL decisions.

The second improvement concerns the *handling of dependent VMs*. Particularly, many of the traditional VM management algorithms (e.g. FFD) as well as the ones proposed in Chapter 4 assume independent VMs. However, in a real environment VMs are inherently dependent. For example, in a web hosting environment, multiple web server VMs could have a dependency on a database VM. Similarly, VMs hosting scientific applications (e.g. MPI) can be subject to dependencies due to communicating applications. Ignoring such dependencies during VM management decisions could seriously degrade the VM performance and availability. Just consider the backend web server VMs and the database VM running on the same PM. In case of a PM failure the entire web hosting environment would become unreachable. In order to consider VM dependencies, VM management algorithms must be extended with support for collocation and anti-collocation constraints. This is an interesting area of research which has not received enough attention in the area of greedy algorithms and meta-heuristics yet. One possible direction to explore is graph theory. Particularly, algorithms which are capable of finding cliques in graphs could be the right direction to investigate.

The third improvement aims at mitigating the *performance interference of collocated VMs*. Despite the resource isolation properties server virtualization technologies, collocation of VM with similar characteristics (e.g. memory intensive) on the same PM can lead to performance degradation as they typically share the same cache [226]. In this context it is interesting to investigate the complementaries between VM resource demands in order to support more accurate VM to PM assignment decisions (e.g. collocate CPU and data intensive VMs). To achieve this machine learning techniques could be leveraged to cluster VMs with comple-

mentary resource demands. Once such cluster are determined VM management algorithms which support collocation and anti-collocation placement constraints could be used to perform the VM to PM assignments.

The fourth improvement targets the *overload management*. Particularly, one interesting issue which arises in the context of overload management is when PMs get concurrently overload along *multiple resource (e.g. CPU and memory)*. In such scenarios a decision must be taken by the VM management algorithms on *how to sort VMs* from the overload node in order to determine potential VMs which must be moved in order to avoid the overload situation. Sorting VMs along one dimension would certainly neglect other dimensions. On the other, giving equal weight to the dimensions (as done in our work) could yield sub-optimal results as well. Better metrics are required in order to take more intelligent decisions.

The fifth improvement focuses on *VM consolidation*. VM consolidation is often used to create idle times by migrating existing VMs on the least number of PMs. Thereby, VM consolidation algorithms are triggered periodically according to a predefined consolidation interval. However, static consolidation intervals can yield significant performance degradation when VM consolidation is triggered during periods of high utilization [262]. It is therefore important to investigating approaches for accurate time interval estimations. Not much work has been done in this area yet.

The sixth improvement concerns the *data center network topology*. Particularly, VM management algorithms could be extended to take the data center network-topology into account while computing the VM to PM assignments. This would allow to reduce the VM live migration time as well as energy consumption by selecting network links with the best performance vs. energy trade-offs.

The seventh improvement involves the *resource reservation*. Most of the modern hypervisors (e.g. KVM, Xen) allow VMs to specific a *min* and *max* amount of allocated resources. Once specified the hypervisor will guarantee that VMs always receive the minimum amount of resources. Moreover, VMs are allowed to dynamically increase their resource allocation until the maximum specified capacity. This mechanisms could be leveraged by the VM management algorithms (e.g. VM consolidation) to assign VMs to PMs such that the minimum resource allocations are guaranteed even during periods of high resource contention.

Finally, in Chapter 4 we have introduced a novel ACO-based VM placement algorithm. Our simulation results have shown that the proposed algorithms outperform the commonly used FFD algorithms in the number of released PMs and computes close to optimal solutions. However, despite its polynomial time worst-case complexity, the algorithm scalability was still limited only to a small number of PMs and VMs. It would be interesting it improve the algorithm prototype implementation and integrate it into Snooze. Moreover, given the scalability limitations of the ACO-based VM placement algorithm and the need to support VM consolidation in order to further consolidate already placed VMs this thesis has been two novel contributions: (1) ACO-based VM consolidation algorithm; (2) full decentralized VM consolidation system based on an unstructured P2P network of PMs. It could be interesting to investigate, how the neighbourhood construction schema of the fully decentralized VM consolidation system can be enhanced in order to create neighbourhoods taking into account the physical distance between the PMs. Ultimately, both works could be integrated into the Snooze cloud management system.

Energy and Thermal Aware Data Center Management. A number of improvements can be done in order to enable energy and thermal aware data center management with Snooze. The first improvement involves the power usage monitoring. Particularly, for the time being the Snooze VM management algorithms decisions are solely based on the VM CPU, memory, and networking utilization data. In other words, they do not take into account the PM/VM power consumption. The reason for this is two fold: (1) no PM/VM power consumption data is fed into the system; (2) no algorithms are implemented able to exploit such data. Providing the PM/VM power consumption data to the system and considering it in the VM management algorithms could further improve the Snooze energy efficiency. The power measurement techniques discussed in Chapter 2 can be leveraged in order to obtain both, the PM and VM power usage data.

The second improvement aims at leveraging fine-grained power management mechanisms. A number of such mechanisms have been proposed over the past years. Examples of such techniques include DVFS, Intel Turbo Boost, Core Off/On, and more recently Intel Running Average Power Limit (RAPL). Integrating such mechanisms in Snooze could further complement its course-grained power management mechanisms (e.g. suspend, shutdown). For example, the GM could be extended to enforce CPU and memory power consumption capping's via DVFS (resp. RAPL). Especially Core Off/On and RAPL have not been explored enough yet in the context of virtualized environments.

The third improvement aims at leveraging *green energy*. Particularly, data centers are now starting to investigate alternative power sources such as green energy (i.e. solar, wind) to complement the traditional brown energy sources (i.e. nuclear and coal plants). One challenge which arises in this context is how to perform VM management such that the amount of green energy is maximized while still beeing able to provide performance guarantees.

Finally, hot spot avoidance is an important issue in today's data centers. Hot spots are regions in a data center which experience an unusual increase in temperature (e.g. due to cooling infrastructure issues). The Snooze VM monitoring and management features could serve as a building block for researchers to experiment with *thermal management* algorithms. Particularly, Snooze is flexible enough to be enhanced to monitor PMs temperature and automatically migrate VMs away from PMs experiencing sudden temperature increases.

Bibliography

[1] lp_solve, a Mixed Integer Linear Programming (MILP) solver. http://lpsolve.sourceforge.net. 50

[2] MySQL Database. http://www.mysql.com/. 82, 122

[3] MySQL Replication. http://dev.mysql.com/doc/refman/5.0/en/replication.html. 27

[4] GLPK (GNU linear programming kit). http://www.gnu.org/software/glpk, 2006. 50

[5] Ganglia Monitoring System. http://ganglia.sourceforge.net/, 2011. 80

[6] Pressflow: Distribution of Drupal with integrated performance, scalability, availability, and testing enhancements. http://pressflow.org/, 2011. 87

[7] 80 PLUS Certified Power Supplies. http://www.plugloadsolutions.com/80PlusPowerSupplies.aspx, 2012. 36

[8] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.0/programs/ab.html, 2012. 88, 90

[9] Amazon Elastic Compute Cloud API Reference. http://docs.amazonwebservices.com/AWSEC2/latest/APIReference/, 2012. 26, 121

[10] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/, 2012. 24

[11] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/, 2012. 23, 35

[12] AMD Phenom II Key Architectural Features. http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-key-architectural-features.aspx, 2012. 38

[13] AMD-V. http://www.amd.com/virtualization, 2012. 11

[14] Bfire - A powerful DSL to launch experiments on BonFIRE. https://github.com/crohr/bfire, 2012. 90

[15] BonFIRE - Testbeds for Internet of Services Experimentation. http://www.bonfire-project.eu/, 2012. 90

[16] CHOCO. http://www.emn.fr/z-info/choco-solver/, 2012. 50

[17] Cloud Foundry. http://www.cloudfoundry.com/, 2012. 24

[18] Cloudify - The Open PaaS Stack. http://www.cloudfoundry.com/, 2012. 24

[19] Eucalyptus. http://www.eucalyptus.com/, 2012. 23, 122

[20] Google App Engine. https://cloud.google.com/products/, 2012. 23

[21] Google Apps. http://www.google.com/enterprise/apps/business/, 2012. 24

[22] Google Cloud Platform. http://cloud.google.com, 2012. 24

[23] Google Compute Engine. http://cloud.google.com/compute/, 2012. 23

[24] Google Data Centers. http://www.google.com/about/datacenters/efficiency/internal/, 2012. 32

[25] GREEN-NET Project : Power aware software frameworks for high performance data transport and computing in large scale distributed systems. http://www.ens-lyon.fr/LIP/RESO/Projects/GREEN-NET/, 2012. 41

[26] GRENKE. http://www.grenkeleasing.co.uk/en.html, 2012. 23

[27] Gurobi Optimizer. http://www.gurobi.com/, 2012. 50

[28] HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. http://haproxy.1wt.eu/, 2012. 90

[29] Heartbeat. http://linux-ha.org/, 2012. 27

[30] Hetzner Online AG. http://www.hetzner.de/en/, 2012. 23

[31] IBM ILOG CPLEX Optimizer for Constraint Programming. http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/about/, 2012. 50

[32] iCloud. https://www.icloud.com/, 2012. 24

[33] ILOG CPLEX: High-performance software for mathematical programming and optimization. http://www.ibm.com/software/integration/optimization/cplex-optimizer/, 2012. 4, 50, 105, 157

[34] Intel Energy Checker SDK. http://software.intel.com/en-us/articles/intel-energy-checker-sdk/, 2012. 34

[35] Intelligent Platform Management Interface. http://www.intel.com/design/servers/ipmi/ipmi.htm, 2012. 76

[36] LeaseWeb. http://www.leaseweb.com/en, 2012. 23

[37] lxc Linux Containers. http://lxc.sourceforge.net/, 2012. 14

[38] Microsoft Hyper-V. http://www.microsoft.com/hyper-v, 2012. 13

[39] Microsoft Virtual PC. http://www.microsoft.com/windows/virtual-pc/, 2012. 12

[40] Moab Energy-Aware Resource Management. http://www.adaptivecomputing.com/docs/460, 2012. 42

[41] Munin Network Monitoring Application. http://www.munin-monitoring.org/, 2012. 80

[42] NERSC: National Energy Research Scientific Computing Center. http://www.nersc.gov/, 2012. 23

[43] Open Cloud Computing Interface - OCCI. http://occi-wg.org/, 2012. 26, 121

[44] OpenVZ. http://www.openvz.org/, 2012. 14

[45] OVH. http://www.ovh.co.uk/, 2012. 23

[46] Pacemaker. http://www.clusterlabs.org/, 2012. 26

[47] Parallels Workstation / Desktop. http://www.parallels.com/products/workstation/, 2012. 12

[48] PowerTop. https://01.org/powertop/, 2012. 34

[49] RabbitMQ - Messaging that just works. http://www.rabbitmq.com/, 2012. 122

[50] Rackspace: The Open Cloud Company. http://www.rackspace.com, 2012. 23, 24

[51] RedHat OpenShift. http://openshift.redhat.com/, 2012. 23

[52] Salesforce. http://www.salesforce.com/, 2012. 24

[53] scp-tsunami. http://code.google.com/p/scp-tsunami/, 2012. 26

[54] SLURM Power Saving Guide. http://computing.llnl.gov/linux/slurm/power_save.html, 2012. 41

[55] Stress tool. http://weather.ou.edu/~apw/projects/stress/, 2012. 90

[56] The Corosync Cluster Engine. http://corosync.org/, 2012. 27

[57] The Green 500 List - June 2012. http://green500.org/lists/green201206, 2012. 29

[58] Virtualization History, Virtual Machine, Server Consolidation. http://www.vmware.com/virtualization/history.html, 2012. 9

[59] VMware Fusion. http://www.vmware.com/products/fusion/overview.html, 2012. 12

[60] VMware Player. http://www.vmware.com/products/player/, 2012. 12

[61] VMware vCloud Suite. http://www.vmware.com/products/datacenter-virtualization/vcloud-suite/, 2012. 24

[62] VMware Workstation. http://www.vmware.com/products/workstation/, 2012. 12

[63] Watts up? PRO. http://www.wattsupmeters.com/, 2012. 32

[64] Windows Azure. http://www.windowsazure.com/en-us/, 2012. 23, 24

[65] wol - Wake On LAN client. http://linux.die.net/man/1/wol, 2012. 76

[66] ZooKeeper recipes and solutions. http://zookeeper.apache.org/doc/trunk/recipes.html/, 2012. 69

[67] Inc. 10gen. MongoDB: A Scalable, High-Performance, Open Source NoSQL Database. http://www.mongodb.org/, 2012. 82, 87

[68] Sumalatha Adabala, Vineet Chadha, Puneet Chawla, Renato Figueiredo, José Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu, and Xiaomin Zhu. From virtualized resources to virtual computing grids: the in-vigo system. *Future Gener. Comput. Syst.*, 21(6):896–909, June 2005. 20

[69] Yuvraj Agarwal, Steve Hodges, Ranveer Chandra, James Scott, Paramvir Bahl, and Rajesh Gupta. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 365–380, 2009. 40

[70] Baris Aksanli, Jagannathan Venkatesh, Liuyi Zhang, and Tajana Rosing. Utilizing green energy prediction to schedule mixed batch and service jobs in data centers. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11, pages 5:1–5:5, 2011. 44

[71] Amazon. Elastic Compute Cloud. http://aws.amazon.com/ec2/, 2012. 23

[72] Amazon. Web Services. http://aws.amazon.com, 2012. 24

[73] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R. Ganger, Michael A. Kozuch, and Karsten Schwan. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 217–228, 2010. 42

[74] Hrishikesh Amur and Karsten Schwan. Achieving power-efficiency in clusters without distributed file system complexity. In *Proceedings of the 2010 international conference on Computer Architecture*, ISCA'10, pages 222–232, Berlin, Heidelberg, 2012. Springer-Verlag. 42

[75] Vlasia Anagnostopoulou, Susmit Biswas, Alan Savage, Ricardo Bianchini, Tao Yang, and Frederic T. Chong. Energy conservation in datacenters through cluster memory management and barely-alive memory servers. In *Proceedings of the Workshop on Energy-Efficient Design (WEED)*, 2009. 40

[76] R. J. Anderson, E. W. Mayr, and M. K. Warmuth. Parallel approximation algorithms for bin packing. *Inf. Comput.*, 82(3):262–277, September 1989. 4, 62, 97, 120, 156

[77] De Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Blackboard Books, 2012. 8

[78] Lee Badger, Tim Grance, Robert Patt-Corner, and Jeff Voas. DRAFT Cloud Computing Synopsis and Recommendations. Technical report, U.S. Department of Commerce Gary Locke, Secretary National Institute of Standards and Technology Patrick D. Gallagher, Director, May 2011. 22, 24

[79] David Bailey, John Barton, and Horst Simon. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991. 87

[80] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003. 13, 65

[81] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007. 3, 149

[82] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, 2005. 10

[83] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Gener. Comput. Syst.*, 28(5):755–768, May 2012. 55, 56

[84] Anton Beloglazov and Rajkumar Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 99:1, 2012. 55

[85] Anton Beloglazov and Rajkumar Buyya. OpenStack Neat: A Framework for Dynamic Consolidation of Virtual Machines in OpenStack Clouds - A Blueprint. https://blueprints.launchpad.net/nova/+spec/dynamic-consolidation-of-virtual-machines, 2012. 55

[86] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420, 2012. 55, 56

[87] Martin Bichler, Thomas Setzer, and Benjamin Speitkamp. Capacity Planning for Virtualized Servers. In *Workshop on Information Technologies and Systems (WITS)*, Milwaukee, Wisconsin, USA, 2006. 50, 51

[88] Amazon Web Services Blog. Amazon S3 Growth for 2011 - Now 762 Billion Objects. http://aws.typepad.com/aws/2012/01/amazon-s3-growth-for-2011-now-762-billion-objects.html, 2012. 35

[89] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management*, pages 119–128. IEEE, 2007. 56, 59

[90] Paolo Bonzini and Thomas Treutner. KVM convergence enforcement. http://lists.gnu.org/archive/html/qemu-devel/2011-09/msg01912.html, 2012. 16

[91] Damien Borgetto, Henri Casanova, Georges Da Costa, and Jean-Marc Pierson. Energy-aware service allocation. *Future Generation Computer Systems*, 28(5):769 – 779, 2012. Special Section: Energy efficiency in large-scale distributed systems. 49, 51, 55

[92] Damien Borgetto, Michael Maurer, Georges Da-Costa, Jean-Marc Pierson, and Ivona Brandic. Energy-efficient and sla-aware management of iaas clouds. In *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*, e-Energy '12, pages 25:1–25:10, 2012. 55, 56

[93] Sara Bouchenak, Fabienne Boyer, Benoit Claudel, Noel De Palma, Olivier Gruber, and Sylvain Sicard. From autonomic to self-self behaviors: The jade experience. *ACM Trans. Auton. Adapt. Syst.*, 6(4):28:1–28:22, October 2011. 21

[94] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 169–179, 2007. 15

[95] Boris Brugger, Karl F. Doerner, Richard F. Hartl, and Marc Reimann. Antpacking – an ant colony optimization approach for the one-dimensional bin packing problem. In Jens Gottlieb and Günther R. Raidl, editors, *Evolutionary Computation in Combinatorial Optimization*, volume 3004 of *Lecture Notes in Computer Science*, pages 41–50. Springer Berlin / Heidelberg, 2004. 49, 51

[96] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 776–783, 2005. 41

[97] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, 2005. 23

[98] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106, 2005. 41, 83

[99] Michael Cardosa, Madhukar R. Korupolu, and Aameek Singh. Shares and utilities based power consolidation in virtualized server environments. In *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, IM'09, pages 327–334, 2009. 48, 51

[100] Michael Cardosa, Aameek Singh, Himabindu Pucha, and Abhishek Chandra. Exploiting spatio-temporal tradeoffs for energy-aware mapreduce in the cloud. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 251–258, 2011. 43

[101] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 86–97, New York, NY, USA, 2003. ACM. 39

[102] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP), 1990. 32

[103] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev.*, 35(5):103–116, October 2001. 41

[104] Wissam Chedid and Chansu Yu. Survey on power management techniques for energy efficient computer systems. http://academic.csuohio.edu/yuc/mcrl/survey-power.pdf, 2002. 29

[105] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The ibm blue gene/q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10, 2011. 29

[106] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 43–56, 2012. 44

[107] Yanpei Chen, Laura Keys, and Randy H. Katz. Towards energy efficient mapreduce. Technical Report UCB/EECS-2009-109, EECS Department, University of California, Berkeley, Aug 2009. 43

[108] Yuan Chen, Daniel Gmach, Chris Hyser, Zhikui Wang, Cullen Bash, Christopher Hoover, and Sharad Singhal. Integrated management of application performance, power and cooling in data centers. In *NOMS*, pages 615–622. IEEE, 2010. 60, 61

[109] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin S. Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, Geoffrey Coulson, Dimiter R. Avresky, Michel Diaz, Arndt Bode, Bruno Ciciani, and Eliezer Dekel, editors, *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, chapter 4, pages 57–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. 24

[110] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, 2005. 8, 15, 16

[111] Benoit Claudel, Guillaume Huard, and Olivier Richard. Taktuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 91–100, 2009. 26

[112] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation algorithms for bin packing: a survey*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997. 48, 51

[113] Georges Da Costa, Marcos Dias de Assunção, Jean-Patrick Gelas, Yiannis Georgiou, Laurent Lefèvre, Anne-Cécile Orgerie, Jean-Marc Pierson, Olivier Richard, and Amal Sayah. Multi-facet approach to reduce energy consumption in clouds and grids: the green-net framework. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 95–104, 2010. 41

[114] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '10, pages 189–194, 2010. 33, 39

[115] J. M. V. de Carvalho. Exact Solution of Cutting Stock Problems Using Column Generation and Branch-and-Bound. *Int. Trans. Opl. Res.*, 5(1):35–44, 1998. 50

[116] Frederico Alvares de Oliveira, Jr. and Thomas Ledoux. Self-management of cloud applications and infrastructure for energy optimization. *SIGOPS Oper. Syst. Rev.*, 46(2):10–18, July 2012. 60, 61

[117] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. 43

[118] Victor Delaluz, Mahmut Kandemir, N. Vijaykrishnan, Anand Sivasubramaniam, and Mary Jane Irwin. Hardware and software techniques for controlling dram power modes. *IEEE Trans. Comput.*, 50(11):1154–1173, November 2001. 38

[119] Deneubourg, S. Aron, S. Goss, and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3(2):159–168, March 1990. 98

[120] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. Memscale: active low-power modes for main memory. *SIGPLAN Not.*, 46(3):225–238, March 2011. 39

[121] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 135–146, 2011. 15

[122] Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing. vgreen: a system for energy efficient computing in virtualized environments. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '09, pages 243–248, New York, NY, USA, 2009. ACM. 27, 29, 58, 59

[123] Bruno Diniz, Dorgival Guedes, Wagner Meira, Jr., and Ricardo Bianchini. Limiting the power consumption of main memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 290–301, 2007. 38

[124] Thanh Do, Suhib Rawshdeh, and Weisong Shi. pTop: A Process-level Power Profiling Tool. In *HotPower '09: Proceedings of the Workshop on Power Aware Computing and Systems*, October 2009. 34

[125] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. Ant algorithms for discrete optimization. *Artif. Life*, 5:137–172, April 1999. 49, 98

[126] Ellen Kotzbauer, BEP and Dennis Bouley. Guide for Reducing Data Center Physical Infrastructure Energy Consumption in Federal Data Centers. Technical report, Schneider Electric, 2007. xi, 31

[127] E.N. (Mootaz) Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *In Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, pages 179–196, 2002. 41

[128] EPA. EPA Report to Congress on Server and Data Center Energy Efficiency. Technical report, U.S. Environmental Protection Agency, 2007. 31

[129] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Memory controller policies for dram power management. In *Proceedings of the 2001 international symposium on Low power electronics and design*, ISLPED '01, pages 129–134, 2001. 38

[130] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 13–23, 2007. 33, 60, 103

[131] Eugen Feller and Christine Morin. Autonomous and Energy-Aware Management of Large-Scale Cloud Infrastructures. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2542–2545, 2012. 3, 156

[132] Eugen Feller, Christine Morin, and Armel Esnault. A Case for Fully Decentralized Dynamic VM Consolidation in Clouds. In *4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Taipei, Taiwan, Province Of China, December 2012. 5, 158

[133] Eugen Feller, Louis Rilling, and Christine Morin. Energy-Aware Ant Colony Based Workload Placement in Clouds. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, GRID '11, pages 26–33, 2011. 4, 157

[134] Eugen Feller, Louis Rilling, and Christine Morin. 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 '12, pages 482–489, 2012. 4, 156

[135] Eugen Feller, Louis Rilling, and Christine Morin. Towards Energy-Efficient, Scalable and Resilient IaaS Clouds. In Massimo Villari, Ivona Brandic, and Francesco Tusa, editors, *Achieving Federated and Self-Manageable Cloud Infrastructures: Theory and Practice*. IGI Global, May 2012. 3, 156

[136] Eugen Feller, Louis Rilling, Christine Morin, Renaud Lottiaux, and Daniel Leprince. Snooze: A Scalable, Fault-Tolerant and Distributed Consolidation Manager for Large-Scale Clusters. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 125–132, 2010. 3, 156

[137] Eugen Feller, Cyril Rohr, David Margery, and Christine Morin. Energy Management in IaaS Clouds: A Holistic Approach. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 204–212, 2012. 4, 156

[138] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, December 2007. 33

[139] J. Figueira, S. Greco, and M. Ehrgott. *Multiple criteria decision analysis: state of the art surveys*, volume 78. Springer Verlag, 2005. 54

[140] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, MobiCom '01, pages 260–271, 2001. 37

[141] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *2nd IEEE Workshop on Mobile Computing Systems and Applications*, February 1999. 34

[142] The Apache Foundation. ActiveMQ. http://activemq.apache.org/, 2012. 122

[143] The Apache Foundation. Libcloud: a unified interface to the cloud. http://libcloud.apache.org/, 2012. 121

[144] Vincent W. Freeh and David K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 164–173, 2005. 42

[145] Vincent W. Freeh, Feng Pan, Nandini Kappiah, David K. Lowenthal, and Rob Springer. Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 4.1, 2005. 42

[146] R Ge, X Feng, S Song, H Chang, D Li, and K Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1 –1, 2010. 33

[147] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, Guillaume Belrose, Tom Turicchi, and Alfons Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *DSN*, pages 326–335, 2008. 53, 56

[148] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Resource pool management: Reactive versus proactive or let's be friends. *Comput. Netw.*, 53(17):2905–2922, December 2009. 53, 56

[149] I. Goiri, Kien Le, M.E. Haque, R. Beauchea, T.D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. Greenslot: Scheduling energy consumption in green datacenters. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1 –11, nov. 2011. 41

[150] íñigo Goiri, Josep Ll. Berral, J. Oriol Fitó, Ferran Julií, Ramon Nou, Jordi Guitart, Ricard Gavaldí, and Jordi Torres. Energy-efficient and multifaceted resource management for profit-driven virtualized data centers. *Future Gener. Comput. Syst.*, 28(5):718–731, May 2012. 49, 51

[151] Íñigo Goiri, Kien Le, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. GreenHadoop: leveraging green energy in data-processing frameworks. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 57–70, 2012. 43

[152] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1 edition, January 1989. 49

[153] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, pages 34–45, June 1974. 8

[154] Leo A. Goodman. The Variance of the Product of K Random Variables. *Journal of the American Statistical Association*, 57(297), 1962. 52, 107

[155] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, MobiCom '95, pages 13–25, 1995. 37

[156] The G. Grid. The Green Grid Data Center Power Efficiency Metrics: PUE and DCiE. http://www.thegreengrid.org/Global/Content/white-papers/The-Green-Grid-Data-Center-Power-Efficiency-Metrics-PUE-and-DCiE, 2007. 31

[157] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11, pages 3–3, 2011. 27

[158] C. Gunaratne, K. Christensen, and B. Nordman. Managing energy consumption costs in desktop PCS and LAN switches with proxying, split TCP connections, and scaling of link speed. *Int. J. Netw. Manag.*, 15:297–310, 2005. 39

[159] Chamara Gunaratne, Kenneth Christensen, Bruce Nordman, and Stephen Suen. Reducing the energy consumption of ethernet with adaptive link rate (alr). *IEEE Trans. Comput.*, 57(4):448–461, April 2008. 39

[160] M. Gupta and S. Singh. Dynamic ethernet link shutdown for energy conservation on ethernet links. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 6156–6161, June 2007. 39

[161] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. Drpm: dynamic speed control for power management in server class disks. *SIGARCH Comput. Archit. News*, 31(2):169–181, May 2003. 39

[162] Taliver Heath, Bruno Diniz, Enrique V. Carrera, Wagner Meira, Jr., and Ricardo Bianchini. Energy conservation in heterogeneous server clusters. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 186–195, 2005. 41

[163] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd annual international conference on Mobile computing and networking*, MobiCom '96, pages 130–142, 1996. 39

[164] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000. 37

[165] Fabien Hermenier, Sophie Demassey, and Xavier Lorca. Bin repacking scheduling in virtualized datacenters. In *Proceedings of the 17th international conference on Principles and practice of constraint programming*, CP'11, pages 27–41, Berlin, Heidelberg, 2011. Springer-Verlag. 58, 59

[166] Fabien Hermenier, Julia Lawall, Jean-Marc Menaud, and Gilles Muller. Dynamic Consolidation of Highly Available Web Applications. Rapport de recherche RR-7545, INRIA, February 2011. 58, 59

[167] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 41–50, 2009. 27, 29, 51, 57, 59

[168] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, 2009. 15, 16, 17

[169] Takahiro Hirofuchi, Hidemoto Nakada, Hirotaka Ogawa, Satoshi Itoh, and Satoshi Sekiguchi. A live storage migration mechanism over wan and its performance evaluation. In *Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, VTDC '09, pages 67–74, 2009. 15, 16

[170] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009. xi, 35

[171] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, 2001. 17

[172] Chung hsing Hsu and Wu chun Feng. Effective dynamic voltage scaling through cpu-boundedness detection. In *In Workshop on Power Aware Computing Systems*, pages 135–149, 2004. 43

[173] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, August 2008. 19

[174] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, 2010. 69, 153

[175] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The xtreemfs architecture—a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, 2008. 122

[176] Urs Hölzle and Bill Weihl. High-efficiency power supplies for home computers and servers. Technical report, Google, 2006. Presented at the Intel Developer Forum, September, 2006. 35

[177] IBM. An Architectural Blueprint for Autonomic Computing. http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf, June 2005. 17, 18, 19

[178] Intel. Turbo boost technology in intel core microarchitecture (nehalem) based processors. download.intel.com/design/processor/applnots/320354.pdf, November 2008. 38

[179] Intel. Intelligent Platform Management Interface. http://www.intel.com/design/servers/ipmi/, 2012. 32

[180] Greenpeace International. Make IT Green: Cloud Computing and its Contribution to Climate Change. http://www.greenpeace.org/usa/en/media-center/reports/make-it-green-cloud-computing/, 2010. 2, 148

[181] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202, 1998. 36

[182] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the maui scheduler. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP '01, pages 87–102, London, UK, UK, 2001. Springer-Verlag. 42

[183] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), August 2007. 28

[184] JUNG. The Java Universal Network/Graph Framework. http://jung.sourceforge.net/, 2012. 80

[185] Gueyoung Jung, Matti A. Hiltunen, Kaustubh R. Joshi, Richard D. Schlichting, and Calton Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 62–73, 2010. 28, 29, 59, 61

[186] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. Generating adaptation policies for multi-tier applications in consolidated server environments. In *Proceedings of the 2008 International Conference on Autonomic Computing*, ICAC '08, pages 23–32, 2008. 60

[187] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 39–50, 2010. 34

[188] Nandini Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 33–, 2005. 42

[189] Rini T. Kaushik and Milind Bhandarkar. GreenHDFS: towards an energy-conserving, storage-efficient, hybrid Hadoop compute cluster. In *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, pages 1–9, 2010. 35, 42, 44

[190] Kate Keahey, Tim Freeman, Jerome Lauret, and Doug Olson. Virtual workspaces for scientific applications. *Journal of Physics: Conference Series*, 78(1):012038, 2007. 23, 26, 29, 122

[191] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. 17

[192] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application Performance Management in Virtualized Server Environments. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 373–381, 2006. 52, 56

[193] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007. 13, 65

[194] Younggyun Koh, R. Knauerhase, P. Brett, M. Bowman, Zhihua Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 200 –209, april 2007. 59

[195] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. http://www.analyticspress.com/datacenters.html, August 2011. 2, 148

[196] Ryogo Kubo, Jun-Ichi Kani, Yukihiro Fujimoto, Naoto Yoshimoto, and Kiyomi Kumozaki. Sleep and adaptive link rate control for power saving in 10g-epon systems. In *Proceedings of the 28th IEEE conference on Global telecommunications*, GLOBECOM'09, pages 1573–1578, Piscataway, NJ, USA, 2009. IEEE Press. 39

[197] Sanjay Kumar, Vanish Talwar, Vibhore Kumar, Parthasarathy Ranganathan, and Karsten Schwan. vmanage: loosely coupled platform and virtualization management in data centers. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 127–136, 2009. 53, 56

[198] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a P2P network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009. 82, 87, 122

[199] Willis Lang and Jignesh M. Patel. Energy management for mapreduce clusters. *Proc. VLDB Endow.*, 3(1-2):129–139, September 2010. 43, 44

[200] Klaus-Dieter Lange. Identifying shades of green: The specpower benchmarks. *Computer*, 42:95–97, 2009. 34

[201] Duncan Laurie. IPMItool. http://ipmitool.sourceforge.net/, 2012. 79

[202] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996(29es), September 1996. 10

[203] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: the laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, pages 1–8, 2010. 37

[204] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. *SIGOPS Oper. Syst. Rev.*, 34(5):105–116, November 2000. 38

[205] William Leinberger, George Karypis, and Vipin Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 404–, 1999. 48, 51

[206] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1):61–65, March 2010. 42, 43, 44

[207] Jacob Leverich, Matteo Monchiero, Vanish Talwar, Parthasarathy Ranganathan, and Christos Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Comput. Archit. Lett.*, 8(2):48–51, July 2009. 37

[208] John Levine and Frederick Ducatelle. Ant colony optimisation and local search for bin packing and cutting stock problems. *Journal of the Operational Research Society*, 93:2003, 2003. 49, 51

[209] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. Enacloud: An energy-saving application live placement approach for cloud computing environments. In *CLOUD '09: Proceedings of the 2009 IEEE International Conference on Cloud Computing*, pages 17–24, 2009. 48, 51

[210] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 22–22, 1994. 39

[211] Min Yeol Lim, Freeman Rawson, Tyler Bletsch, and Vincent W. Freeh. Padd: Power aware domain distribution. *International Conference on Distributed Computing Systems (ICDCS)*, 0:239–247, 2009. 53, 56

[212] Ching-Chi Lin, Pangfeng Liu, and Jan-Jan Wu. Energy-efficient virtual machine provision algorithms for cloud systems. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 81–88, 2011. 49, 51

[213] Jacob R. Lorch and Alan Jay Smith. Operating system modifications for task-based speed and voltage. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, pages 215–229, 2003. 37

[214] Yung-Hsiang Lu and Giovanni de Micheli. Adaptive hard disk power management on personal computers. In *Proceedings of the Ninth Great Lakes Symposium on VLSI*, GLS '99, pages 50–, 1999. 39

[215] Dan Marinescu and Reinhold Kroeger. State of the art in autonomic computing and virtualization. Technical report, Distributed Systems Lab, Wiesbaden University of Applied Sciences, September 2007. 11

[216] Moreno Marzolla, Ozalp Babaoglu, and Fabio Panzieri. Server consolidation in Clouds through gossiping. In *Proceedings of the 2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, WOWMOM '11, 2011. 4, 28, 29, 58, 59, 114, 157

[217] Carlo Mastroianni, Michela Meo, and Giuseppe Papuzzo. Self-economy in cloud data centers: statistical assignment and migration of virtual machines. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 407–418, Berlin, Heidelberg, 2011. Springer-Verlag. 49, 51

[218] Jon McGary and Weimin Pan. Exploring the DRAC 5. http://www.dell.com/downloads/global/power/ps3q06-20060118-McGary.pdf/, 2012. 33

[219] S. Mehta and A. Neogi. Recon: A tool to recommend dynamic server consolidation in multi-cluster data centers. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 363 –370, april 2008. 50, 51

[220] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 205–216, 2009. 40

[221] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, July 2009. 21, 22

[222] Jean-Marc Menaud and Rémy Pottier. btrScript : a safe management system for virtualized data center. *ICAS 2012*, March 2012. 27

[223] Dejan Milojicic, Ignacio M. Llorente, and Ruben S. Montero. OpenNebula: A cloud management tool. *IEEE Internet Computing*, 15:11–14, March 2011. 23, 26, 29, 122

[224] Aziz Murtazaev and Sangyoon Oh. Sercon: Server Consolidation Algorithm using Live Migration of Virtual Machines for Green Computing. *IETE Technical Review*, 28 (3), 2011. 4, 58, 59, 77, 114, 120, 157

[225] Susanta Nanda and Tzi-cker" Chiueh. A survey of virtualization technologies. Technical report, 2005. 11

[226] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 237–250, 2010. 59, 61, 124

[227] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 323–336, 2008. 39

[228] Neil Rasmussen. Electrical Efficiency Measurement for Data Centers. Technical report, Schneider Electric, 2012. 32

[229] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.*, 71:169–184, February 2011. 122

[230] Bogdan Nicolae and Franck Cappello. A hybrid local storage transfer scheme for live migration of i/o intensive workloads. In *HPDC '12: 21th International ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 85–96, Delft, The Netherlands, 2012. 15, 122

[231] Mathias Noack. Comparative evaluation of process migration algorithms. Diploma thesis, Dresden University of Technology, Operating Systems Group, 2003. 16

[232] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. Runtime monitoring of software energy hotspots. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 160–169, 2012. 34

[233] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009. 27, 29

[234] Oracle. VirtualBox. https://www.virtualbox.org/, 2012. 12

[235] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle - do nothing, efficiently... In *Linux Symposium*, pages 119–126, 2007. 38

[236] Jinha Park, Sungjoo Yoo, Sunggu Lee, and Chanik Park. Power modeling of solid state disk for dynamic power management policy design in embedded systems. In *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS '09, pages 24–35, Berlin, Heidelberg, 2009. Springer-Verlag. 39

[237] Marcus Peinado, Yuqun Chen, Paul Engl, and John Manferdelli. Ngscb: A trusted open system. In *In Proceedings of 9th Australasian Conference on Information Security and Privacy ACISP*, pages 86–97. Springer, 2004. 11

[238] Srinath Perera and Dennis Gannon. Enforcing user-defined management logic in large scale systems. In *IEEE Congress on Services*, pages 243–250, 2009. 20, 64, 149

[239] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 international symposium on Low power electronics and design*, ISLPED '98, pages 76–81, 1998. 36

[240] Karen E. Petrie, Barbara Smith, and Neil Yorke-smith. Dynamic symmetry breaking in constraint programming and linear programming hybrids. In *In European Starting AI Researcher Symp*, 2004. 51

[241] Guillaume Pierre, Ismail El Helw, Corina Stratan, Ana Oprescu, Thilo Kielmann, Thorsten Schütt, Jan Stender, Matej Artač, and Aleš Černivec. Conpaas: an integrated runtime environment for elastic cloud applications. In *Proceedings of the Workshop on Posters and Demos Track*, PDT '11, pages 5:1–5:2, 2011. 24

[242] Jean-Marc Pierson and Henri Casanova. On the utility of dvfs for power-aware job placement in clusters. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 255–266, Berlin, Heidelberg, 2011. Springer-Verlag. 49, 51

[243] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 89–102, 2001. 37

[244] Eduardo Pinheiro and Ricardo Bianchini. Nomad: A scalable operating system for clusters of uni and multiprocessors. In *Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing*, IWCC '99, pages 247–, 1999. 41

[245] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 68–78, 2004. 42

[246] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. 2001. 41

[247] Flavien Quesnel, Adrien Lèbre, and Mario Südholt. Cooperative and reactive scheduling in large-scale virtualized platforms with dvms. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2012. 28, 29, 54, 56

[248] Rackspace. Hosting Reports Third Quarter. http://ir.rackspace.com/phoenix.zhtml?c=221673&p=irol-newsArticle&ID=1627224&highlight=, 2011. 2, 148

[249] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: coordinated multi-level power management for the data center. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 48–59, 2008. 57, 59

[250] Red Hat. libvirt: The virtualization API. http://libvirt.org/, 2012. 80

[251] Pierre Riteau, Christine Morin, and Thierry Priol. Shrinker: Improving live migration of virtual clusters over wans with distributed data deduplication and content-based addressing. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 431–442. Springer Berlin Heidelberg, 2011. 15

[252] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 365–376, 2007. 34

[253] Jerry Rolia, Artur Andrzejak, and Martin Arlitt. Automating Enterprise Application Placement in Resource Utilities. In *14th˜IFIP/IEEE˜Workshop on Distributed Systems:˜Operations and Management (DSOM˜2003), (Workshop Theme: "Self-Managing˜Systems")*, Heidelberg, October 2003. 50, 51

[254] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, May 2005. 8

[255] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006. 50

[256] Barry Rountree, David K. Lowenthal, Shelby Funk, Vincent W. Freeh, Bronis R. de Supinski, and Martin Schulz. Bounding energy consumption in large-scale mpi programs. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 49:1–49:9, 2007. 43

[257] Jonathan Rouzaud Cornabas. A distributed and collaborative dynamic load balancer for virtual machine. In *Proceedings of the 5th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '10) Euro-Par 2010*, Ischia, Naples Italy, 2010. 29

[258] Jonathan Rouzaud-Cornabas. A distributed and collaborative dynamic load balancer for virtual machine. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 641–648, Berlin, Heidelberg, 2011. Springer-Verlag. 28

[259] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, 2002. 42

[260] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., 1986. 50

[261] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, March 1972. 10

[262] T. Setzer and A. Stage. Decision support for virtual machine reassignments in enterprise data centers. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, pages 88 –94, april 2010. 125

[263] Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, 2009. 33

[264] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, February 1993. 11

[265] Christine Solnon and Derek Bridge. An Ant Colony Optimization Meta-Heuristic for Subset Selection Problems. Technical Report RR-LIRIS-2005-017, LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/École Centrale de Lyon, December 2005. Chapitre d'un livre édité par Nadia Nedjah et Luiza Mourelle. 49, 51

[266] Benjamin Speitkamp and Martin Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE Trans. Serv. Comput.*, 3(4):266–278, October 2010. 46, 50, 51, 77, 97

[267] Robert Springer, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing execution time in mpi programs on an energy-constrained, power-scalable cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 230–238, 2006. 42

[268] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pages 10–10, 2008. 46

[269] Vaidyanathan Srinivasan, Gautham R. Shenoy, Srivatsa Vaddagiri, Dipankar Sarma, and Venkatesh Pallipadi. Energy-aware task and interrupt management in linux. In *Proceedings of the Linux Symposium*, July 2008. 38

[270] M. Stillwell, F. Vivien, and H. Casanova. Virtual machine resource allocation for service hosting on heterogeneous distributed platforms. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 786 –797, may 2012. 49, 51

[271] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *J. Parallel Distrib. Comput.*, 70:962–974, September 2010. 49, 50, 51

[272] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003. 28

[273] Thomas Stutzle and Holger Hoos. Improvements on ant-system: Introducing max-min ant system, 1996. 100, 101

[274] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A multi-agent systems approach to autonomic computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, pages 464–471, 2004. 20

[275] The Apache Software Foundation. CloudStack: Open Source Cloud Computing. http://www.cloudstack.org/, 2012. 23, 26, 29, 122

[276] The Apache Software Foundation. Hadoop Distributed File System (HDFS). http://hadoop.apache.org/hdfs/, 2012. 42

[277] The Apache Software Foundation. Hadoop MapReduce. http://hadoop.apache.org/mapreduce/, 2012. 87

[278] The Apache Software Foundation. Hadoop project. http://hadoop.apache.org, 2012. 43

[279] The OpenStack Project. OpenStack: The Open Source Cloud Operating System. http://www.openstack.org/software/, 2012. 23, 27, 29, 122

[280] My Ton and Brian Fortenbury. High performance buildings: Data centers server power supplies. http://hightech.lbl.gov/documents/ps/Final_PS_Report.pdf, 2005. 35, 36

[281] E. Tsang. *Foundations of constraint satisfaction*, volume 289. Academic press London, 1993. 51

[282] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005. 11

[283] O. S. Unsal and I. Koren. System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, 91(7):1055–1069, 2003. 30

[284] Bhuvan Urgaonkar, Arnold Rosenberg, and Prashant Shenoy. Application placement on a cluster of servers. Oct 2007. 46, 48, 51, 77, 97

[285] Hien Nguyen Van, F.D. Tran, and J.-M. Menaud. Performance and power management for cloud infrastructures. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 329 –336, july 2010. 60, 61

[286] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008. 21

[287] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: power and migration cost aware application placement in virtualized systems. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 243–264. Springer-Verlag New York, Inc., 2008. 27, 29, 56, 59

[288] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 28–28, 2009. 48, 51

[289] VMware. Distributed Resource Management: Design, Implementation and Lessons Learned. http://labs.vmware.com/publications/gulati-vmtj-spring2012, 2012. 27, 29, 55, 56

[290] VMware. Understanding full virtualization, paravirtualization, and hardware assist. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf, 2012. 11

[291] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13, 2005. 111, 113, 157

[292] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002. 13

[293] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, 1994. 37

[294] Eric W Weisstein. L1-norm. http://mathworld.wolfram.com/L1-Norm.html, 2012. 48, 73, 100, 103, 107

[295] Joshua White and Adam Pilbeam. A survey of virtualization technologies with performance testing. *CoRR*, abs/1010.3233, 2010. 11

[296] Thomas Wirtz and Rong Ge. Improving mapreduce energy efficiency for computation intensive workloads. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, 2011. 43

[297] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Comput. Netw.*, 53(17):2923–2938, December 2009. 52, 56

[298] Jing Xu and José Fortes. A multi-objective approach to virtual machine management in datacenters. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 225–234, 2011. 54, 56

[299] Jing Xu, Ming Zhao, and José A. Fortes. Cooperative autonomic management in dynamic distributed systems. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '09, pages 756–770, Berlin, Heidelberg, 2009. Springer-Verlag. 20

[300] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 374–, 1995. 37

[301] Yagiz Onat Yazir, Chris Matthews, Roozbeh Farahbod, Stephen Neville, Adel Guitouni, Sudhakar Ganti, and Yvonne Coady. Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 91–98, 2010. 54, 56

[302] AndyB. Yoo, MorrisA. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer Berlin Heidelberg, 2003. 41

[303] Doe Hyun Yoon, Jichuan Chang, Naveen Muralimanohar, and Parthasarathy Ranganathan. Boom: enabling mobile memory based low-power server dimms. *SIGARCH Comput. Archit. News*, 40(3):25–36, June 2012. 35

[304] Minyi Yue. A simple proof of the inequality ffd (l) $\leq$ 11/9 opt (l) + 1, $\forall$l for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7:321–331, 1991. 4, 47, 48, 114, 156

[305] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010. 21

[306] Zeqiang Zhang, Wenming Cheng, Liansheng Tang, and Yue Cheng. Improved ant colony optimization for one-dimensional bin packing problem with precedence constraints. In *Proceedings of the Third International Conference on Natural Computation - Volume 04*, ICNC '07, pages 539–543, 2007. 49, 51

[307] Xiaoyun Zhu, Donald Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret Mckee, Chris Hyser, Daniel Gmach, Robert Gardner, Tom Christian, and Ludmila Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12(1):45–57, March 2009. 53, 56

Appendix A

Résumé en français

A.1 Motivation

Le cloud computing est récemment apparu comme un nouveau paradigme de l'informatique selon lequel les services sont proposés suivant le modèle de paiement à l'utilisation. Les clients qui utilisent ces services sont facturés uniquement sur ce qu'ils ont consommé. Un modèle particulier de service de cloud très prisé ces dernières années est qualifié d'*Infrastructureas-a-Service* (IaaS). Dans les clouds IaaS, les ressources de calcul et de stockage sont fournies à la demande par les prestataires de cloud. Ainsi, la capacité de calcul est généralement fournie sous la forme de machines virtuelles (VM). Les VMs se présentent aux clients comme si elles étaient de véritables machines physiques (PM). L'utilisation de VMs a été rendue possible par les progrès des technologies de virtualisation de serveurs qui permettent de multiplexer efficacement les ressources des PMs (*e.g.* processeurs, mémoire, dispositifs d'entrée/sortie).

Depuis l'apparition du cloud computing, de nombreux prestataires de cloud (*e.g.* Amazon, Google, Rackspace) sont apparus et offrent actuellement une multitude de services comme la capacité de calcul et le stockage des données à la demande. Pour répondre à la demande grandissante de services de la part des clients, les prestataires de cloud ont récemment commencé à déployer un nombre croissant de centres de données de grande ampleur. Leur gestion exige des prestataires de cloud qu'ils relèvent plusieurs défis. Notamment, les fournisseurs de cloud doivent aujourd'hui concevoir et mettre en œuvre des systèmes de gestion de clouds IaaS innovants capables de fonctionner à grande échelle. Les systèmes de *cloud computing* doivent en particulier passer à l'échelle pour supporter le nombre croissant de clients et de ressources (c'est-à-dire les VMs et les PMs). En outre, le risque de défaillances matérielles et logicielles augmente avec la taille des centres de données. Par conséquent, les systèmes de gestion du cloud doivent être conçus pour un fonctionnement autonome, ce qui leur permettra de détecter automatiquement les défaillances et de lancer une restauration. En outre, l'administration système de clouds de grande taille requiert plusieurs experts informaticiens hautement qualifiés. Pour automatiser les tâches de configuration, les systèmes de gestion de clouds IaaS doivent intégrer des mécanismes d'auto-configuration afin de per-

mettre une configuration du système nécessitant une intervention humaine minimale. Enfin, les centres de données hébergent actuellement de l'équipement (*e.g.* serveurs de stockage et de calcul, système de climatisation) qui consomme une quantité d'énergie très importante. Par exemple, Google, moteur de recherche sur Internet dominant et prestataire de services de cloud, héberge à lui seul plus de 900 000 serveurs qui ont consommé environ 2 milliards de kWh d'électricité en 2010 [195]. Alors que les besoins énergétiques des centres de données de Google sont encore inférieurs à 1% des besoins énergétiques de l'ensemble des centres de données du monde, réduire la consommation énergétique durant les périodes de faible utilisation des centres de données est capital pour réduire le coût total de propriété (TCO) des centres de données et leur empreinte carbone à une époque où la plupart des centres de données sont toujours alimentés par des centrales au charbon ou nucléaires [180]. Compte tenu de l'importance des économies d'énergie, il faut concevoir des systèmes de gestion de clouds IaaS efficaces en énergie.

Ces dernières années, plusieurs tentatives ont été faites pour concevoir et mettre en œuvre des systèmes de gestion de clouds IaaS visant à faciliter la création de clouds IaaS privés. Compte tenu de l'ampleur croissante des centres de données, ces systèmes font face à des défis en termes de passage à l'échelle, d'autonomie et d'efficacité énergétique. Cependant, de nombreuses tentatives faites pour concevoir et mettre en œuvre des systèmes de gestion de clouds IaaS reposent toujours sur des architectures centralisées, ont une autonomie limitée et n'intègrent pas de mécanismes d'économie d'énergie. Par conséquent, ils constituent un point unique de défaillance, ne passent pas à l'échelle et ont une faible efficacité énergétique.

A.2 Objectifs

L'objectif de cette thèse est de concevoir, de mettre en œuvre et d'évaluer un système de gestion de clouds IaaS pour les centres de données de grande envergure. Pour atteindre son objectif principal, cette thèse explore les quatre objectifs secondaires suivants:

- **Passage à l'échelle:** les centres de données hébergent désormais plusieurs milliers de serveurs. Par exemple, Rackspace, prestataire de service de cloud IaaS renommé hébergeait environ 78 000 serveurs en 2011 [248]. La gestion d'un tel nombre de serveurs exige des systèmes de gestion de clouds IaaS passant à l'échelle. Par conséquent, notre objectif consiste à concevoir un système en adéquation avec le nombre croissant de serveurs (PM et VM).
- **Haute disponibilité:** avec un nombre croissant de serveurs, le risque de défaillance des composants du système (matériels et logiciels) augmente. Afin d'assurer un fonctionnement du système sans interruption de service, le système de gestion de clouds IaaS devra être hautement disponible. Par conséquent, il faut mettre en œuvre des mécanismes de haute disponibilité. Notre objectif est de concevoir un système intégrant ces mécanismes.
- **Facilité d'administration:** gérer des centres de données de grande envergure peut s'avérer être une lourde tâche nécessitant plusieurs experts informaticiens très expérimentés. Fournir un système facile à configurer peut réduire considérablement les coûts et faciliter la gestion du système. L'un de nos objectifs est de concevoir un système

dont la configuration nécessite une intervention humaine minimale. En outre, une fois le système déployé et configuré, il faut effectuer des mises à jour et/ou ajouter de nouveaux serveurs. Dans ce genre de scénarios, des serveurs devront être déconnectés et réintégrés ultérieurement. Notre objectif est de concevoir un système suffisamment flexible pour supporter l'ajout et la suppression dynamiques de serveurs. Enfin, comme les composants du système peuvent défaillir à tout moment, il est souhaitable que le système puisse être réparé sans intervention humaine en cas de défaillance. Par conséquent, nous cherchons à concevoir un système utilisant des mécanismes d'auto-réparation pour permettre la haute disponibilité du service cloud IaaS.

- **Efficacité énergétique:** ces dernières années, l'augmentation des factures énergétiques a fait de l'efficacité énergétique une contrainte de conception majeure pour les centres de données. Comme les centres de données sont rarement utilisés au maximum de leur capacité, il est possible de réaliser des économies d'énergie significatives durant les périodes de faible utilisation en faisant passer les serveurs inactifs dans un mode d'économie d'énergie. Cependant, comme les serveurs sont rarement totalement inutilisés, il faut d'abord créer des périodes d'inactivité [81]. Notre objectif est de concevoir un système de gestion de clouds IaaS et des algorithmes de gestion de VMs capables de créer des périodes d'inactivité, de faire passer automatiquement les serveurs inactifs en mode d'économie d'énergie et de les réveiller lorsque c'est nécessaire (*e.g.* lorsque la charge augmente). Ceci permettra d'adapter la consommation énergétique du centre de données à sa charge.

A.3 Contributions

Pour atteindre les objectifs présentés, cette thèse apporte les trois contributions exposées dans ce paragraphe.

A.3.1 Snooze: un gestionnaire de clouds IaaS passant à l'échelle, autonome et économique en énergie.

L'objectif principal de cette thèse est de concevoir et de mettre en œuvre un système de gestion de clouds IaaS passant à l'échelle, autonome et économique en énergie. Pour obtenir le passage à l'échelle et l'autonomie, nous avons fait le choix de conception décisif de construire un système de gestion de clouds IaaS reposant sur une architecture hiérarchique auto-(re)configurable et auto-réparante. Nos choix de conception pour le passage à l'échelle et l'autonomie sont motivés par des travaux antérieurs qui ont montré que les architectures hiérarchiques peuvent améliorer considérablement le passage à l'échelle des systèmes. L'architecture de Snooze s'inspire en partie du système autonome Hasthi [238] qui a montré, à l'aide de simulations, sa capacité à gérer jusqu'à 100 000 ressources. Cependant, contrairement à Hasthi dont la conception est présentée de façon générique, indépendamment de tout système particulier, et qui utilise une table de hachage distribuée (DHT) reposant sur un réseau pair-à-pair (P2P), Snooze a une conception plus simple et ne nécessite pas l'utilisation de la technologie P2P. En outre, il cible des systèmes virtualisés, sa conception et sa mise en œuvre sont donc conditionnées par les objectifs et les problèmes spécifiques à ce type de systèmes.

Organiser le système de manière hiérarchique favorise son passage à l'échelle car les composants aux niveaux les plus élevés de la hiérarchie n'ont pas besoin d'une connaissance globale du système. L'idée au cœur de notre système consiste à répartir les tâches de gestion des VMs entre plusieurs gestionnaires autonomes indépendants ayant chacun uniquement une vision partielle du centre de données. Notamment, chaque gestionnaire a seulement à gérer un sous-ensemble des nœuds de calcul du centre de données et des VMs. Un coordinateur est choisi automatiquement parmi les gestionnaires durant la configuration automatique de la hiérarchie et en cas de défaillance du coordinateur. Le coordinateur supervise les gestionnaires et il est contacté par les clients qui soumettent leurs VMs.

Pour économiser l'énergie, il existe des mécanismes de gestion de l'énergie sur chaque gestionnaire autonome qui détectent les nœuds de calcul inactifs, les fait passer en mode d'économie d'énergie et les réveille en cas de besoin (*e.g.* s'il n'y a pas assez de nœuds actifs lors du placement de VMs). Pour favoriser les périodes d'inactivité, des gestionnaires autonomes mettent en œuvre des mécanismes avancés de gestion de VMs comme la résolution des situations de sous-utilisation des ressources et le regroupement de VMs.

Dans les paragraphes suivants, nous présentons l'architecture hiéarchique du système Snooze et les mécanismes de gestion dynamique de la hiérarchie. Nous décrivons également les algorithmes et mécanismes de gestion de VMs que nous avons conçus pour améliorer l'efficacité énergétique du centre de données.

A.3.1.1 Architecture du système

L'architecture du système Snooze est décomposée en trois couches: calcul, administration et client. Dans la couche de calcul, les nœuds hébergeant les VMs sont organisés en une grappe. Chaque nœud de calcul est contrôlé par un service système appelé contrôleur local (LC). La couche d'administration permet le passage à l'échelle du système. Elle est composée de nœuds hébergeant des services système tolérants aux fautes: le coordinateur des gestionnaires de groupe (GL) et un ou plusieurs gestionnaires autonomes, à savoir les gestionnaires de groupe (GM). Les services du système sont organisés de manière hiérarchique. Le GL surveille les GMs. Il est choisi parmi les GMs lors de l'auto-configuration de la hiérarchie et suite à la défaillance du GL. Chaque GM gère un sous-ensemble des LCs et des VMs. Le GL reçoit les requêtes de soumission de VMs émanant des clients et les répartit entre les GMs. Une fois les VMs soumises, les clients interagissent directement avec les GMs concernés pour contrôler leurs VMs (*e.g.* arrêt, ré-initialisation). Comme le GL peut changer au cours du temps, une méthode est nécessaire pour que les clients puissent découvrir automatiquement le GL actif. Cette fonctionnalité est fournie par la couche client. Cette dernière est composée d'un nombre prédéfini de services appelés points d'entrée (EP) qui connaissent à tout moment l'identité du GL courant. Tous les services du système sont accessibles via une interface RESTful. Par conséquent, n'importe quel logiciel client (*e.g.* interface en ligne de commande (CLI), web, bibliothèque de cloud) peut être implémenté pour interagir avec les EPs, le GL et les GMs.

Pour des questions de performance et de passage à l'échelle tous les composants du système ont des rôles spécifiques (*e.g.* le GL et les GMs n'hébergent pas de VMs). Pour un ensemble de nœuds physiques donné, c'est l'administrateur du système qui décide du nombre de LCs et de GMs pour le déploiement de Snooze. Par exemple, dans le scénario de déploiement le plus élémentaire, on a besoin de deux GMs et d'un LC. L'un des GMs sera

promu au rang de GL par l'exécution d'un algorithme d'élection. Les services du système sont suffisamment flexibles pour coexister sur le même nœud. Par conséquent, il est possible de déployer toute la hiérarchie sur un seul nœud physique.

Contrôleurs locaux. Chaque LC gère le cycle de vie des VMs qu'il héberge et exécute les commandes de gestion des nœuds que lui envoie le GM qui lui a été assigné. Des exemples de ces commandes comprennent le démarrage et la migration à chaud des VMs ainsi que la régulation de l'état des nœuds (*e.g.* mise en mode d'économie d'énergie). Le LC surveille également les VMs qu'il héberge, détecte les périodes de surcharge et de sous-utilisation et envoie régulièrement des données sur l'utilisation des ressources par les VMs qu'il exécute au GM désigné. Des indicateurs de surcharge/de sous-utilisation sont envoyés en même temps que ces données. Chaque LC conserve dans un espace de stockage local les informations relatives aux VMs qui s'exécutent sur le nœud qu'il gère.

Gestionnaires de groupe. Chaque GM est chargé de la gestion d'un sous-ensemble des LCs. Il reçoit les données sur les ressources utilisées par les VMs en provenance de ces LCs et les stocke localement. À partir de ces données, le GM estime l'utilisation des ressources par les VMs et prend des décisions relatives à la gestion de VMs qui implique trois types d'actions : placement des VMs, résolution des situations de surcharge et de sous-utilisation d'un LC et regroupement de VMs.

Les mécanismes de placement des VMs sont déclenchés lors du traitement des requêtes de soumission des VMs en provenance du GL. Les mécanismes de résolution des situations de surcharge et de sous-utilisation des LC sont déclenchés lorsque des indicateurs de surcharge (ou de sous-utilisation) arrivent des LCs et ont pour objectif de déplacer les VMs des LCs fortement (ou faiblement) chargés. Le regroupement des VMs est effectué périodiquement selon l'intervalle de temps spécifié par l'administrateur du système. Cet algorithme peut être utilisé par exemple pour optimiser l'utilisation des LCs modérément chargés sur une base hebdomadaire en regroupant les VMs existantes sur le moins de LCs possible. Les deux politiques de résolution des situations de surcharge/sous-utilisation et de regroupement des VMs génèrent un plan de migration qui spécifie les nouvelles affectations des VMs sur les LCs. Un GM met en œuvre le plan de migration en indiquant aux LCs qu'il gère d'effectuer la migration à chaud des VMs concernées.

La gestion de l'énergie est intégrée dans chaque GM pour la mise en veille des LCs inactifs et les réveiller. Des LCs sont réveillés lorsqu'un GM n'a pas assez de LCs actifs pour traiter une requête de placement de VMs ou pour résoudre une situation de surcharge.

Chaque GM envoie périodiquement un résumé des informations qu'il possède au GL courant pour lui permettre de décider de l'allocation des VMs nouvellement créées aux GMs. Le résumé des informations d'un GM comprend l'agrégation des informations d'utilisation des ressources pour l'ensemble des LCs qu'il gère. Enfin, les clients contactent les GMs en charge de la gestion de leurs VMs pour les contrôler (*e.g.* arrêter) et récupérer les informations les concernant (*e.g.* utilisation des ressources, état).

Coordinateur des gestionnaires de groupe. Le GL gère les GMs. Il a la charge d'affecter les LCs aux GMs au démarrage, d'accepter les requêtes de soumission de VMs en provenance des clients, d'administrer le réseau permettant aux VMs de communiquer entre

elles ou avec l'extérieur, et de répartir les VMs soumises sur les GMs. De plus, il reçoit les résumés des informations en provenance des GMs et les stocke localement. Nous allons à présent détailler davantage ces tâches.

Lorsque les LCs démarrent, ils doivent être affectés à un GM. Les décisions d'affectation d'un LC à un GM sont guidées par une politique d'affectation de LCs. Par exemple, un LC peut être affecté à un GM selon une politique *round-robin* ou en fonction de l'utilisation courante des GMs. Une fois les LCs affectés aux GMs, des VMs peuvent être soumises par les clients au GL. Si des requêtes de soumission de VMs par des clients arrivent avant l'affectation des LCs aux GMs, un message d'erreur est retourné. Il faut noter que des LCs supplémentaires peuvent intégrer le système à tout moment sans perturber son fonctionnement normal.

Lorsqu'un groupe de VMs (contenant une ou plusieurs VMs) est soumis au GL, les décisions de répartition des VMs sur GMs sont prises par le GL. L'affectation des VMs aux GMs est effectuée selon une politique de répartition des VMs. Le calcul de cette affectation utilise les informations envoyées par les GMs. Suivant cette affectation, le GL répartit les VMs sur les GMs. Cependant, avant de pouvoir affecter les VMs aux GMs, il faut gérer leur connexion réseau pour que le monde extérieur puisse accéder aux VMs après leur démarrage. Cette procédure comporte deux étapes: (1) obtenir l'affectation d'une adresse IP à la VM; (2) configurer l'interface réseau selon l'adresse IP affectée. Le GL se charge de la première étape. Aussi, il gère un sous-réseau que l'administrateur système peut configurer, à partir duquel il est autorisé à attribuer des adresses IP. Lorsque les VMs sont soumises au GL, chacune d'elles reçoit automatiquement une adresse IP attribuée dans ce sous-réseau. L'adresse IP attribuée est intégrée dans l'adresse MAC des VMs. Lorsque la VM démarre, elle décode son adresse IP à partir de son adresse MAC et procède à la configuration du réseau. Le GL ne conserve pas de vision globale des VMs dans le système. Après avoir réparti les VMs, les informations relatives aux GMs sur lesquels les VMs ont été placées sont stockées par le client, ce qui permet aux clients d'interagir directement avec les GMs concernés pour les requêtes de gestion de VMs ultérieures. Même si les décisions de répartition des VMs ont un faible coût et que le GL ne possède pas de vision globale de l'ensemble des VMs, il est toutefois possible d'améliorer son passage à l'échelle en le dupliquant et en appliquant une politique d'équilibrage de la charge.

Points d'entrée. Dans Snooze, le GL est automatiquement choisi parmi les GMs au démarrage du système et un nouveau GL est élu en cas de défaillance du GL courant. Aussi, un GL peut changer au cours du temps. Pour que les clients puissent ordonner à un GL de démarrer des VMs, un moyen de déterminer le GL courant est nécessaire. Pour ce faire, nous introduisons un nombre prédéfini d'EPs. Les EPs sont des services système qui résident généralement sur les nœuds dans le même réseau que les GMs et sont capables de connaître l'identité du GL courant (voir le paragraphe suivant pour de plus amples détails). Les clients qui veulent soumettre des VMs contactent un des EPs pour déterminer l'identité du GL courant.

A.3.1.2 Gestion de la hiérarchie

Ce paragraphe décrit comment la hiérarchie du système Snooze est construite et préservée pendant le fonctionnement du système. Tout d'abord, nous présentons les mé-

canismes d'envoi de messages de bon fonctionnement (*heartbeats*). Ensuite, nous exposons les mécanismes d'auto-configuration et d'auto-réparation. L'auto-configuration est la capacité du système à construire de manière dynamique la hiérarchie lors du démarrage du système. L'auto-réparation permet de reconstruire automatiquement la hiérarchie en cas de défaillance des services système ou des nœuds.

Messages de bon fonctionnement. Pour supporter l'auto-configuration et l'auto-réparation, Snooze intègre des protocoles d'envoi de messages de bon fonctionnement bi-directionnels à tous les niveaux de la hiérarchie. Le GL envoie régulièrement son identité à un groupe de diffusion spécifique contenant tous les EPs et les GMs. Les messages du GL permettent aux EPs de tenir à jour l'identité du GL courant. Les messages du GL sont également nécessaires aux LCs et aux GMs pour déterminer le GL courant au démarrage et en cas de défaillance du GM (ou du GL). En effet, le GL courant doit être connu des LCs lors de leur initialisation pour être affectés à un GM. Les GMs doivent également informer le GL courant de leur présence pour permettre la distribution des VMs aux GMs par le GL. En cas de défaillance d'un GM, les LCs qui y sont rattachés doivent contacter le GL pour obtenir une nouvelle affectation de GM. Enfin, en cas de défaillance du GL, les GMs doivent informer le GL nouvellement élu de leur présence. Il existe un groupe de diffusion des messages de bon fonctionnement par GM sur lequel il annonce sa présence aux LCs qui lui sont affectés. Il est utilisé par les LCs pour détecter la défaillance du GM auquel ils sont rattachés. Enfin, pour détecter la défaillance d'un GM (respectivement d'un LC), le GL (respectivement les GMs) s'appuient sur des messages de bon fonctionnement qui leur sont envoyés directement par les GMs (respectivement les LCs). Les messages de bon fonctionnement sont transmis en même temps que les informations de surveillance des VMs envoyés périodiquement par les GMs (respectivement les LCs) au GL (respectivement au GM qui leur est assigné).

Auto-configuration. Lorsqu'un service système démarre sur un nœud, il est configuré de manière statique pour devenir soit un LC soit un GM. Lorsque les services démarrent, la première étape dans la construction de la hiérarchie implique l'élection d'un GL parmi les GMs. Après le choix du GL, les autres GMs doivent s'enregistrer auprès de lui. Pour qu'un LC intègre la hiérarchie, il doit d'abord déterminer quel est le GL courant et le contacter pour se voir affecter un GM. Une fois affecté à un GM, il peut s'enregistrer auprès de celui-ci. Nous allons à présent décrire toutes ces étapes de façon plus détaillée.

L'auto-configuration de la hiérarchie Snooze fonctionne de la manière suivante. Lorsqu'un GM tente pour la première fois d'intégrer le système, un algorithme d'élection est déclenché pour choisir le GL parmi les GMs. Actuellement, l'algorithme d'élection du GL repose sur le système de coordination à haute disponibilité ZooKeeper [174] d'Apache. S'il existe un GL, le GM s'enregistre auprès de lui et commence à envoyer ses messages de bon fonctionnement. Sinon, il devient le nouveau GL et commence à envoyer ses messages de bon fonctionnement en tant que GL. Lorsqu'un LC démarre, il doit intégrer la hiérarchie. Aussi, les informations relatives au GL courant ainsi qu'au GM auquel il va être rattaché sont requises. Pour obtenir les informations du GL, il se met à l'écoute des messages de bon fonctionnement émis par le GL sur un groupe de diffusion spécifique. Lorsqu'il reçoit un tel message, il contacte le GL pour se voir attribuer un GM. Différentes politiques d'attribution des LCs aux

GMs peuvent être appliquées par le GL. Par exemple, les LCs peuvent être affectés aux GMs selon une politique *round robin* ou selon la charge courante des GMs (*e.g.* affectation aux GMs les moins chargés). Enfin, le LC commence à interagir avec le GM qui lui est attribué. Il se met à l'écoute des messages de bon fonctionnement du GM considéré et commence à lui envoyer les siens.

Auto-réparation. L'auto-réparation s'effectue à tous les niveaux de la hiérarchie. Elle implique la détection et la réparation automatique des défaillances des LCs, des GMs et du GL.

Les défaillances d'un LC sont détectées par le GM qui lui est affecté lorsqu'il ne reçoit plus les messages de bon fonctionnement que le LC lui envoie périodiquement. Une fois la défaillance d'un LC détectée, le GM enlève le LC fautif de sa liste de LCs pour qu'il ne soit pas pris en considération dans les prochaines tâches de gestion de VMs. Il faut noter qu'en cas de défaillance d'un LC, les VMs qu'il héberge sont arrêtées brutalement. Actuellement, Snooze ne traite pas la restauration des VMs dont l'exécution est interrompue, mais les LCs peuvent utiliser les fonctionnalités de sauvegarde d'état offertes par les hyperviseurs pour sauvegarder régulièrement des points de reprise des VMs. Ainsi, le GM pourra redémarrer les VMs victimes de la défaillance sur les LCs actifs restants.

Les défaillances du GM sont détectées par le GL et les LCs lorsqu'ils ne reçoivent pas les messages de bon fonctionnement qui leur sont destinés directement ou qui sont envoyés à un groupe de diffusion auquel ils appartiennent. Lorsqu'un GM est défaillant, il est supprimé de la liste des GMs gérés par le GL afin que ce dernier ne lui transmette plus de requêtes de soumission de VMs. Les LC gérés par le GM fautif lancent une procédure de réintégration. Comme la procédure d'intégration, une réintégration implique l'affectation d'un GM au LC. Cependant, pendant sa réintégration, un LC doit en plus transmettre au GM qui lui est nouvellement affecté les informations qu'il possède sur les VMs en cours d'exécution sur le nœud qu'il gère pour que ce GM puisse mettre à jour les informations qu'il détient.

La défaillance du GL est détectée suite à l'absence dans un délai prévu des messages de bon fonctionnement normalement transmis au groupe de diffusion dont font partie les GMs. Lorsque le GL est défaillant, il faut d'abord élir un nouveau GL parmi les GMs. Le GL nouvellement élu doit ensuite être découvert par les GM restants qui doivent s'enregistrer auprès de lui. Enfin, les LCs qui étaient précédemment affectés au GM devenu le nouveau GL doivent amorcer la procédure de réintégration dans la hiérarchie pour se rattacher à un autre GM. En cas de défaillance d'un GL, toutes ses informations sur les GMs existants sont perdues. C'est pourquoi il faut reconstruire ces informations pour que le système reste cohérent. Nous utilisons le service Apache ZooKeeper pour élir un nouveau GL parmi les GMs. Lorsqu'un GM existant devient le nouveau coordinateur des gestionnaires de groupe, il passe en mode GL et commence à envoyer ses messages de bon fonctionnement en tant que GL. Pour détecter le nouveau GL, les GMs sont constamment à l'écoute des messages de bon fonctionnement du GL. Sur réception de l'identité du nouveau GL, chacun d'eux déclenche la procédure de réintégration de GM. Contrairement à la procédure d'intégration de GM présentée précédemment, une réintégration de GM nécessite l'envoi de données supplémentaires. En outre, les GMs renvoient régulièrement le résumé des informations qu'ils détiennent au nouveau GL, ce qui lui permet de reconstruire ses informations

relatives à l'utilisation des ressources.

A.3.1.3 Efficacité énergétique dans la gestion de machines virtuelles

Les décisions pour la gestion des VMs sont prises à deux niveaux: celui du GL et celui des GMs. Au niveau du GL, l'affectation des VMs aux GMs s'effectue en s'appuyant sur le résumé des informations sur l'utilisation des ressources transmis par les GMs au GL. Par exemple, les VMs peuvent être réparties sur les GMs selon une politique *round-robin* ou *first-fit* prenant en compte la capacité des GMs. Dans nos travaux, nous utilisons une politique *round-robin*. Ainsi, le GL favorise les GMs disposant d'une capacité active (correspondant aux nœuds qui exécutent des VMs ou qui sont en attente de VMs à exécuter) suffisante et prend en considération la capacité passive (correspondant aux nœuds placés en mode d'économie d'énergie) uniquement lorsqu'il n'y a pas suffisamment de capacité active disponible. Le résumé des informations fourni par les GMs au GL ne suffit pas à prendre des décisions de répartition exactes. Par exemple, lorsqu'un client soumet une VM demandant 2 GB de mémoire et qu'un GM indique que 4 GB sont disponibles, cela ne signifie pas nécessairement que la VM sera placée sur ce GM car sa mémoire disponible peut être distribuée entre plusieurs LCs (*e.g.* 4 LCs avec chacun 1 GB de RAM). Aussi, une liste de GMs potentiels est retournée par la politique de répartition. Suivant cette liste, une recherche linéaire est effectuée pour la transmission des requêtes de placement de VMs aux GMs.

C'est le GM qui prend les décisions pour le placement effectif des VMs en suivant quatre types de politiques: placement, déplacement en cas de surcharge d'un nœud, déplacement en cas de sous-utilisation d'un nœud, et enfin regroupement. Les politiques de placement (*e.g. round-robin* ou *first-fit*) sont déclenchées par événement pour placer les VMs entrantes sur les LCs. De même, les politiques de déplacement sont mises en œuvre lorsque des situations de surcharge (ou de sous-utilisation) sont signalées par des LCs et elles ont pour objectif de déplacer les VMs des nœuds très (peu) chargés vers d'autres nœuds pouvant les accueillir. Par exemple, en cas de surcharge, les VMs doivent être transférées vers un nœud moins chargé pour pallier une baisse de performance. Au contraire, en cas de sous-utilisation, pour économiser de l'énergie, il est souhaitable de déplacer les VMs vers des LCs modérément chargés afin de créer une période d'inactivité suffisante pour pouvoir faire passer les LCs inactifs d'un centre de données modérément chargé en mode économie d'énergie (*e.g.* arrêt).

Pour compléter les politiques de placement et de déplacement déclenchées par événement, il est possible de spécifier des politiques de regroupement de VMs qui seront appelées périodiquement suivant un intervalle de temps spécifié par l'administrateur système afin d'optimiser encore davantage le placement des VMs sur des nœuds modérément chargés. Par exemple, un algorithme de regroupement des VMs peut être activé sur un rythme hebdomadaire pour optimiser le placement des VMs en regroupant les VMs sur le moins de nœuds possible.

Enfin, pour économiser de l'énergie, les nœuds inactifs doivent être placés dans un mode d'économie d'énergie. Aussi, Snooze intègre sur chaque GM un module de gestion du mode de fonctionnement des serveurs, qui une fois activé par l'administrateur système, permet de surveiller périodiquement l'activité des serveurs gérés par les LCs et de déclencher des actions visant à économiser l'énergie (*e.g.* arrêt d'un serveur) lorsque des serveurs sont inactifs.

A.3.1.4 Évaluation

Pour évaluer Snooze, un prototype a été mis en œuvre et évalué de manière approfondie sur la plate-forme d'expérimentation Grid'5000 à l'aide d'applications réalistes exécutées sur une grappe comprenant plus de 140 nœuds. Les résultats expérimentaux ont montré que notre système passe à l'échelle, est autonome et permet d'économiser de l'énergie. Les principes de conception du système Snooze ont été publiés dans [131, 135, 136]. L'évaluation du passage à l'échelle et de l'autonomie a été publiée dans [134]. Enfin, la description et l'évaluation des mécanismes de gestion de l'énergie ont été publiées dans [137].

A.3.2 Placement des VMs via l'optimisation par colonie de fourmis

Une approche traditionnelle pour économiser de l'énergie consiste à favoriser les périodes d'inactivité des nœuds dans un centre de données modérément chargé. Lors de la soumission de VMs dans les systèmes de gestion de clouds IaaS, il s'agit d'attribuer un ensemble de VMs à des PMs de sorte à minimiser le nombre de PMs nécessaires à l'hébergement des VMs. C'est possible en mettant en œuvre ce que l'on appelle des algorithmes de placement de VMs. Cependant, bon nombre des algorithmes de placement de VMs classiques ne prennent en considération qu'une seule ressource (*e.g.* le processeur) pour évaluer la charge des PMs et les demandes de ressources des VMs. En outre, ils reposent sur des algorithmes centralisés comme le *First-Fit Decreasing* (FFD) [304] connus pour être difficiles à distribuer/paralléliser [76]. Pour pallier ces limitations, nous avons étudié l'utilisation de l'optimisation par colonie de fourmis (ACO) pour résoudre le problème du placement de VMs et proposé un algorithme de placement de VMs reposant sur l'ACO. L'ACO est particulièrement intéressante pour le problème du placement de VMs en raison de sa complexité dans le pire cas polynomiale et de sa facilité de parallélisation.

L'algorithme proposé repose sur les principes de l'ACO où plusieurs agents (c'est-à-dire les fourmis artificielles) calculent en parallèle des solutions probabilistes en plusieurs cycles. Ainsi, ils communiquent de manière indirecte en déposant une substance chimique appelée phéromone sur chaque paire VM-PM dans une matrice de phéromone. Lors de chaque cycle, les fourmis reçoivent les VMs et commencent à construire des solutions locales (c'est-à-dire des affectations d'une VM à une PM) en utilisant une règle de décision probabiliste qui représente le désir qu'a une fourmi de choisir une VM donnée comme la prochaine à placer sur son PM courant. Cette règle repose sur les informations courantes sur la concentration de phéromone sur la paire VM-PM dans la matrice de phéromone et sur une information d'heuristique qui guide les fourmis dans leurs choix de VM pour parvenir à une meilleure utilisation générale des PMs. Ainsi, plus la quantité de phéromone et l'information d'heuristique associées à une paire VM-PM sont élevées, plus la probabilité que cette paire soit choisie est forte. À la fin de chaque cycle, les solutions locales sont comparées et celle qui nécessite le nombre le plus faible de PMs est conservée comme la nouvelle solution globalement optimale. Ensuite, la matrice de phéromone est mise à jour pour simuler l'évaporation de la phéromone et renforcer les paires VM-PM qui appartiennent à la meilleure solution calculée jusqu'à présent. La nature stochastique de l'algorithme lui permet d'explorer un grand nombre de solutions potentielles. En outre, l'algorithme est aisément parallélisable.

Nous avons évalué l'approche reposant sur l'ACO en la comparant à l'algorithme FFD et

à la solution optimale calculée à l'aide de la solution d'optimisation IBM ILOG CPLEX [33]. Les résultats de simulation démontrent que l'ACO surpasse l'algorithme FFD car il permet d'obtenir des économies d'énergie plus importantes grâce à une meilleure utilisation des PMs et nécessite moins de PMs. En outre, il calcule des solutions proches de l'optimal. Ces travaux sont publiés dans [133].

A.3.3 Regroupement de VMs via l'optimisation par colonie de fourmis

La contribution précédente a montré que même si l'ACO calcule des solutions proches de la solution optimale, l'algorithme conçu ne passe pas bien à l'échelle en termes de temps de calcul avec un nombre élevé de PMs et de VMs. En outre, alors que la résolution du problème de placement de VMs est importante pour favoriser la création de périodes d'inactivité lors de la soumission de VMs, des algorithmes de regroupement de VMs sont nécessaires pour permettre en continu le regroupement sur le plus petit nombre possible de PMs des VMs en cours d'exécution. C'est particulièrement important pour éviter la fragmentation des ressources et augmenter encore davantage l'utilisation des ressources du centre de données. Pour traiter ces deux aspects, cette thèse apporte les deux contributions suivantes: (1) nous avons adapté notre algorithme de placement de VMs fondé sur l'ACO proposé précédemment afin de permettre le regroupement de VMs en continu ; (2) pour résoudre les problèmes de passage à l'échelle, nous avons proposé un système de regroupement de VMs totalement décentralisé reposant sur un réseau pair-à-pair non-structuré de PMs.

L'idée essentielle au cœur du système proposé pour permettre à la fois le passage à l'échelle et une utilisation élevée des ressources dans les centres de données consiste à appliquer le regroupement de VMs seulement dans des groupes de PMs constitués aléatoirement. Etant donnée la complexité des algorithmes de regroupement dynamique de VMs, limiter son application à des petits groupes de PMs améliore considérablement le passage à l'échelle du système. En outre, le caractère aléatoire de la formation des groupes de PMs facilite la convergence du système vers un compactage global très proche de celui qui serait obtenu à l'aide d'un système centralisé exploitant des algorithmes de regroupement de VMs centralisés traditionnels. L'efficacité du compactage est définie comme le rapport entre le nombre de PMs libérés et le nombre total de PMs. Pour permettre la construction de groupes de PMs aléatoirement, nous nous appuyons sur le protocole Cyclon [291]. Cyclon est un protocole épidémique qui permet de construire périodiquement des réseaux logiques P2P aléatoires où chaque PM n'a qu'une vue partielle du système, appelée voisinage. Cette propriété permet au système de passer à l'échelle avec le nombre de PMs car il ne repose pas sur un serveur central.

Nous avons mis en œuvre un émulateur de notre système distribué et nous l'avons validé à l'aide de deux algorithmes de regroupement de VMs bien connus, Sercon [224] et V-MAN [216], et l'algorithme de regroupement de VMs reposant sur l'ACO. Les nombreuses expérimentations menées sur la plate-forme d'expérimentation Grid'5000 montrent qu'une fois intégrés dans notre système de regroupement de VMs totalement décentralisé, les algorithmes traditionnels de regroupement de VMs parviennent à une efficacité de compactage global très proche d'un système centralisé. De plus, le système passe à l'échelle avec le nombre de PMs et de VMs. Enfin, l'algorithme de regroupement de VMs fondé sur l'ACO se comporte mieux que l'algorithme Sercon pour ce qui concerne le nombre de PMs libérés et exige moins de migrations de VMs que l'algorithme V-MAN. Ces résultats ont été publiés

dans [132].

A.4 Aperçu de la thèse

Cette thèse est organisée de la manière suivante:

- Le chapitre 2 présente l'état de l'art. Notamment, il présente tout d'abord le contexte de cette thèse en introduisant brièvement la virtualisation des serveurs, l'informatique autonome et le *cloud computing*. Ensuite, les approches existantes de gestion de l'énergie dans les grappes de calculateurs sont étudiées. Comprendre les approches proposées pour économiser l'énergie est indispensable pour les comparer à nos contributions sur la gestion de l'énergie.

- Le chapitre 3 décrit notre première contribution : Snooze, un système de gestion de clouds IaaS autonome et économique en terme de consommation d'énergie, reposant sur une architecture hiérarchique auto-configurable et auto-réparante. Nous donnons tout d'abord un aperçu de l'architecture du système. Ensuite, nous expliquons de manière détaillée d'une part la gestion de la hiérarchie des services système et d'autre part la gestion des VMs pour économiser l'énergie dans le centre de données. Ceci implique la description des mécanismes d'auto-configuration et d'auto-réparation ainsi que des algorithmes de gestion de VMs pour l'efficacité énergétique. Enfin, les aspects importants de la mise en œuvre sont présentés et les résultats de l'évaluation expérimentale sont analysés.

- Le chapitre 4 est consacré à la gestion des VMs par une approche fondée sur l'optimisation par colonie de fourmis. Tout d'abord nous présentons une introduction à l'ACO. Ensuite, nous décrivons l'algorithme de placement de VMs fondé sur l'ACO et présentons les résultats d'évaluation. Ensuite, l'algorithme de placement de VMs est adapté pour permettre le regroupement de VMs. En outre, pour améliorer le passage à l'échelle, nous proposons un système de regroupement de VMs totalement décentralisé reposant sur un réseau P2P non-structuré de PMs. Enfin, les résultats d'évaluation de l'algorithme de regroupement de VMs et du système de regroupement de VMs totalement décentralisé sont présentés.

- Le chapitre 5 conclut ce manuscrit en résumant nos contributions et en présentant quelques directions de recherche.