October 2014

Advances in Dynamic Virtualized Cloud Management

Michael Tighe
The University of Western Ontario

Supervisor
Dr. Michael Bauer
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

ADVANCES IN DYNAMIC VIRTUALIZED CLOUD MANAGEMENT

(Thesis format: Integrated Article)

by

Michael Tighe

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

Cloud computing continues to gain in popularity, with more and more applications being deployed into public and private clouds. Deploying an application in the cloud allows application owners to provision computing resources on-demand, and scale quickly to meet demand. An Infrastructure as a Service (IaaS) cloud provides low-level resources, in the form of virtual machines (VMs), to clients on a pay-per-use basis. The cloud provider (owner) can reduce costs by lowering power consumption. As a typical server can consume 50% or more of its peak power consumption when idle, this can be accomplished by consolidating client VMs onto as few hosts (servers) as possible. This, however, can lead to resource contention, and degraded VM performance. As such, VM placements must be dynamically adapted to meet changing workload demands. We refer to this process as *dynamic management*. Clients should also take advantage of the cloud environment by scaling their applications up and down (adding and removing VMs) to match current workload demands.

This thesis proposes a number of contributions to the field of dynamic cloud management. First, we propose a method of dynamically switching between management strategies at run-time in order to achieve more than one management goal. In order to increase the scalability of dynamic management algorithms, we introduce a distributed version of our management algorithm. We then consider deploying applications which consist of multiple VMs, and automatically scale their deployment to match their workload. We present an integrated management algorithm which handles both dynamic management and application scaling. When dealing with multi-VM applications, the placement of communicating VMs within the data centre topology should be taken into account. To address this consideration, we propose a topology-aware version of our dynamic management algorithm. Finally, we describe a simulation tool, DCSim, which we have developed to help evaluate dynamic management algorithms and techniques.

Keywords: data centre management, virtualization, cloud management, energy management, SLA management, application scaling.

Co-Authorship Statement

This thesis is composed of several published papers, which were co-authored with other authors. This section describes the contributions of the authors of each paper. Dr. Michael Bauer is a co-author on all papers, and served the role of supervisor, offering advice and guidance on both the research and writing of each paper. In some papers, Dr. Hanan Lutfiyya also appears as a co-author, in a similar supervisory role.

Dynamic Virtual Machine Management

This chapter is presented as necessary background work for the subsequent chapters, and is based on a paper primarily authored by Gaston Keller, a PhD Candidate at the University of Western Ontario. The thesis author was a co-author on this publication, providing some feedback and input on the work. The version presented in this chapter has been heavily modified to match the style of the thesis. Furthermore, additional content has been added to define the problem and general approach taken by the work, including some material from a journal paper currently under submission by the thesis author.

Switching Data Centre Management Strategies at Run-time

This chapter represents the work of three primary authors, including the thesis author, Gaston Keller, and Graham Foster, a former Masters student at the University of Western Ontario. An equal contribution to the work was made by all three primary authors. More specifically, the main contribution of the thesis author were the *SP-DSS* algorithm and its evaluation, the *Optimal Power Efficiency* metric, and the method of scoring and comparing the performance of strategies. A portion of the work was used as the subject of the Masters Thesis of Graham Foster. Writing of the paper was also completed by all three primary authors. Some text in the paper has been adapted for this thesis.

A Distributed Approach to Dynamic VM Management

This chapter contains a full length version of a published short paper, with some adaptation for this thesis. The thesis author is primarily responsible for the work and writing, with co-author Gaston Keller providing the *Periodic* and *Reactive* centralized management algorithms against which the novel distributed algorithm is compared.

Integrated Cloud Application Autoscaling with Dynamic VM Management

This chapter presents work from a conference publication, adapted for this thesis. It also includes some content from an extended version, currently under submission to a journal. The thesis author is the primary author of this work.

Topology-aware Dynamic VM Management

This chapter presents a portion of work from a journal paper, currently under submission. The paper is an extension of a conference paper, and this chapter presents the additional content. The thesis author is the primary author of this work.

DCSim

This chapter details a simulation tool used to evaluate algorithms and techniques throughout the thesis work. The tool was primarily developed by the thesis author, with contributions from Gaston Keller, a PhD Candidate at the University of Western Ontario. The content of this chapter is based on two workshop publications, as well as additional content added to reflect the latest additions to the tool. Gaston Keller served as co-author on both publications, providing contributions and design input for the tool itself, as well as feedback and the writing of some sections of the publications. More specifically, Gaston Keller is responsible for the implementation of *Rack & Cluster* features of DCSim, the initial management policy implementations and contributed bug fixes. The thesis author developed the core functionality of the tool. Jamil Shamy, a former student at the University of Western Ontario, was a co-author on the second publication. He contributed a visualization tool for DCSim, which was described in the paper.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rise of cloud computing has been swift, since it first started appearing almost a decade ago. While some of its fame can be attributed to pure marketing, it represents a clear shift in computing paradigm. An increasing number of applications are being deployed in the cloud, whether they are migrated to it or developed for it explicitly. Cloud-based applications are a diverse group, ranging from personal web sites, to e-commerce sites, to major online applications, such as Netflix and Dropbox. The promise of cloud computing is simple: computing resources, on-demand, billed as a utility. Driven by economies of scale and the maturation of virtualization technologies, the cloud is delivering on this promise.

As cloud computing is an overly broad term, it can be further sub-divided into a number of more specific categories. The first classification is whether the cloud is *public* or *private*. A private cloud is owned and utilized by the same organization, while a public cloud offers its services to all. A public cloud is always multi-tenant, although a private cloud may be as well (different company divisions, locations, etc.). In this work, we focus on a public cloud offering, although the algorithms are applicable to any multi-tenant cloud, public or private. Clouds can be again classified into three additional categories: *Software as a Service* (SaaS), *Platform as a Service* (PaaS), and *Infrastructure as a Service* (IaaS). The SaaS model provides software online and on-demand, either free or by subscription. Examples include Gmail [4] and SalesForce [12]. A PaaS cloud provides a specific development platform, or stack, for developing cloud-based applications. The PaaS cloud handles all deployment details, allowing the developer to simply build the application and let the PaaS provider (such as Google App Engine [5]) handle the rest. Finally, an IaaS cloud provides resources with low-level access, allowing the client to provision complete servers and to control everything from the operating system level above. The pioneer IaaS cloud was Amazon Web Services [1], released in 2006. Since then, a multitude of IaaS offerings have been deployed, from Microsoft Azure [9], to Google Compute Engine [6], to Rackspace Public Cloud [11]. In this work, we focus on managing an IaaS cloud.

Clouds consist of physical machines combined with a layer of software which provides access to these resources to users, or clients. The physical machines, or servers, are deployed within a data centre, which is a facility housing a very large number of servers. In addition to servers, the data centre must also provide power, cooling, networking, and other infrastructure requirements. An IaaS cloud is typically virtualized, providing clients with virtual machines (VMs) rather than full, "bare metal" servers. Virtualization is a technology that allows multiple virtual machines to run on a single server (called a *host*), providing the illusion that each has control over a physical machine. The VMs on a host are managed by the *hypervisor*, and each VM contains its own complete operating system. As such, clients can configure and deploy VMs exactly to their requirements, regardless of the underlying cloud environment. Furthermore, multiple VMs can be co-located on a single host, with each VM using a portion of the total physical resources. This helps enable higher server resource utilization within the data centre.

There are three primary actors in this environment: the *provider*, the *client*, and the *user*. The cloud provider is the owner of the cloud infrastructure, which is rented to clients. Clients provision VMs from the provider in order to deploy applications in the cloud, which are accessed by their users. The provider offers services to the clients, and the clients provide services to their users.

One of the primary goals of the cloud provider is to minimize costs while continuing to provide clients with the expected level of service. Towards this goal, cloud providers should take measures to ensure that their resources are highly utilized. A typical server (host) can consume 50% (or more) of its peak power consumption when idle [2]. As such, an underutilized server represents a significant expense in terms of wasted power. Furthermore, by achieving high host utilization, fewer hosts are required to handle a given workload, and more clients can be served on the same infrastructure. Therefore, proper placement of VMs and allocation of resources is extremely important in minimizing the costs and maximizing the revenue of an IaaS cloud.

The simplest approach to VM placement is to statically place each VM and allocate it enough resources to meet the peak demand of the application(s) it is running. The workload of an application can be highly variable [3], however, and can on average be as low as 30% of peak load. This leads to a significant underutilization of resources, even with multiple VMs co-located on a single host. Host utilization could be increased by allocating VMs only enough resource to meet average demand, but this comes at the cost of application performance degradation should demand rise above average, due to competition between VMs. If the demands of each VM are well known, or do not vary, then a static placement could be effective. The placement of VMs is similar to a bin-packing problem [13, 14], and as such, finding an optimal placement is NP-Hard [7]. Some work uses a linear programming approach to computing a static VM placement [13]. In this case, however, any change in VM resource demand or the set of VMs present in the data centre would require the placement to be recomputed from scratch. Clearly, this is not feasible. Given the high variability of many application workloads [8], a dynamic approach to VM management is required. For dynamic placement of VMs, linear programming techniques are too slow to provide the necessary responsiveness and suffer from limited scalability [10]. For large dynamic systems, first-fit heuristics for the bin packing problem can be a more effective choice [14].

In order to contend with constantly changing VM resource demands, the VM placement must be managed and adapted dynamically. By continuously adapting the VM placement to meet current demands, the cloud provider can achieve high resource utilization without sacrificing VM performance. The CPU of each cost can be *oversubscribed*, meaning that the VMs co-located on a host are promised more CPU than is actually available. In the event that VM demands begin to stress the host resources, a VM can be moved to another host with spare capacity. This is accomplished using a *VM live migration* operation. Live migration allows a running VM to be moved from one host to another (from a *source* to a *target*) without incurring significant downtime or performance degradation. However, live migrations are not entirely without cost (e.g. bandwidth and CPU overhead), and should be minimized as much as possible. Dynamic VM management is a difficult challenge, as it must compute and adapt VM placement under highly dynamic resource demands. Conflicting goals, such as power conservation versus performance, further complicate the situation.

The cloud client has similar goals: minimize infrastructure costs while providing expected service to their clients. This is accomplished by leveraging the on-demand nature of the cloud, which provides a seemingly infinite pool of resources which can be provisioned at a moments notice. Clients can use only the resources they require, when they require them. That is, applications can scale up and down to meet current demands by dynamically adding and removing resources, in order to pay only for the resources they require to achieve their performance goals. This can be done automatically, by monitoring the performance of the deployed application and scaling the number of VMs performing tasks within it. We refer to this process as *autoscaling*. As this ability is a key advantage to deploying applications in the cloud, cloud providers should offer autoscaling as a service, or provide the necessary infrastructure to support it. Ideally, autoscaling could be integrated into dynamic VM management, in order to ensure that the goals of both the cloud provider and client are met.

Evaluating new algorithms for dynamic management can be a challenge, due to the scale and complexity of the infrastructure on which they are intended to run. Even if a reasonably sized infrastructure was available for experimentation (which in itself is unlikely to be feasible), the implementation and execution of experiments would be difficult and time consuming. In order to facilitate both rapid development and evaluation of new algorithms for dynamic management in the cloud, simulation is typically employed.

There remain a number of open challenges in dynamic management in the cloud. While basic concepts and algorithms have been explored, significant complexity remains in removing assumptions and adding in the pieces required to build a truly complete and viable solution. In this thesis, we tackle several of these challenges, advancing the state-of-the-art in dynamic virtualized cloud management and constructing an ever more thorough solution.

1.1 Contributions

This thesis consists of a set of contributions to several different aspects of dynamic virtualized cloud (data centre) management. Each chapter deals with a particular sub-problem in the area, all of which contribute towards the advancement of dynamic virtualized cloud management. Below is a brief description of the contributions of each chapter.

Chapter 3 - Switching Data Centre Management Strategies at Run-time

A cloud provider (data centre operator/owner) may have several goals they which to achieve, such as minimizing their *power consumption* and minimizing the *Service Level Agreement (SLA) violations* they incur. An SLA violation occurs when a client VM does not receive the resources it has requested. Most work focuses on achieving a single primary goal, with other goals being either ignored or considered secondary. In addition, goals often may be in conflict with one another. In this chapter, we present a method of pursuing multiple goals simultaneously.

Chapter 4 - A Distributed Approach to Dynamic VM Management

Most work on dynamic management makes use of a centralized architecture. In this approach, a single manager must maintain global knowledge of the data centre state, and perform all management computations. Due to the scale and highly dynamic nature of the problem, a centralized solution is unlikely to scale well enough. As such, a distributed solution may be more appropriate. In this chapter, we present a distributed version of a dynamic management algorithm, to address this issue.

Chapter 5 - Integrating Cloud Application Autoscaling with Dynamic VM Management

The cloud client (application/VM owner), has its own set of goals, similar to those of the cloud provider. The client should attempt to minimize their infrastructure costs, while providing good service to their users. Since a cloud client pays for resource on a per-use basis, this can be accomplished by automatically scaling an application up and down (by adding and removing VMs) to match the current workload demand. In this chapter, we examine how running both autoscaling and dynamic VM management together affects performance, and propose integrating the two into a single algorithm. The integrated algorithm is capable of leveraging some control over autoscaling in order to assist dynamic VM management, resulting in a significant reduction in required migrations. Reducing the number of migrations performed during dynamic management is important, as migrations incur both bandwidth and performance overheads.

Chapter 6 - Topology-aware Dynamic VM Management

When an application consisting of multiple VMs is deployed in the cloud, their location within the network topology can have an effect on application performance. Most work treats the data centre as a flat, structureless collection of servers. In this chapter, we introduce a topology-aware dynamic management approach, which attempts to place application VMs near each other (in terms of network links) to reduce both communication latency, and utilization of higher level network elements.

Chapter 7 - DCSim: A Data Centre Simulation Tool

Algorithms for dynamic VM management in the cloud are targeted at a very large scale deployment. This poses a challenge to researchers, as experimentation on a representative scale is simply not feasible. As such, most work turns to simulation in order to evaluate new algorithms and techniques. There is a lack of open-source simulation tools targeted at simulating a multi-tenant, infrastructure as a service cloud. In this chapter, we present DCSim (Data Centre Simulator), an extensible simulation tool designed specifically to address these requirements.

1.2 Organization

This thesis is presented in an *integrated article* format, meaning that each chapter presents a single piece of work in publication format. Each source paper has been modified for consistency, readability, and to avoid repetition between chapters, where appropriate. Furthermore, the *related work* section of each chapter has been modified to present only relevant *additional* literature which was not presented in Chapter 1 or Chapter 2. Each chapter lists references cited within it in its own bibliography section.

The remainder of the thesis is organized as follows: Chapter 2 presents background work on dynamic VM management, which provides a framework and base for the subsequent work. Chapters 3 to 7 present the main contributions of the thesis. Finally, Chapter 8 concludes.

Bibliography

[1] Amazon. Amazon web services. `http://aws.amazon.com/`, July 2014.

[2] L.A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec 2007.

[3] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *IM Proceedings, 2007 IEEE/IFIP Int. Symp. on*, pages 119–128, 2007.

[4] Google. Gmail. `http://mail.google.com`, July 2014.

[5] Google. Google app engine. `http://appengine.google.com/`, July 2014.

[6] Google. Google compute engine. `http://cloud.google.com/`, July 2014.

[7] Chris Hyser, Bret Mckee, Rob Gardner, and Brian J. Watson. Autonomic virtual machine placement in the data center. Technical Report HPL-2007-189, HP Laboratories, December 2007.

[8] A. Kochut and K. Beaty. On Strategies for Dynamic Resource Management in Virtualized Server Environments. In *MASCOTS Proceedings, 2007 15th Int. Symp. on*, pages 193–200, 2007.

[9] Microsoft. Microsoft azure. `http://azure.microsoft.com/`, July 2014.

[10] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. Technical report, Microsoft Research, 2011.

[11] Inc. RackSpace. Rackspace. `http://www.rackspace.com/`, July 2014.

[12] Inc. salesforce.com. salesforce. `http://www.salesforce.com/`, July 2014.

[13] B. Speitkamp and M. Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE TSC*, 3(4):266 –278, 2010.

[14] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *J. Parallel Distrib. Comput.*, 70(9):962–974, September 2010.

Chapter 2

Dynamic Virtual Machine Management

2.1 Introduction

Dynamic Virtual Machine (VM) Management is the process of placing a set of VMs into a set of hosts, and dynamically adapting this placement as VM workloads and resource demands change over time. These hosts are located in a data centre, which is a large collection of interconnected physical servers (hosts), contained within a single building (e.g. a warehouse). The data centre also provides all of the necessary additional infrastructure, such as cooling, power and networking. As described in Chapter 1, the data centre is owned and operated by the *Cloud Provider*, whose goals are to minimize costs while providing the expected service to its clients. As idle hosts still consume a large portion of their peak power usage, it is necessary to ensure that hosts are either highly utilized or powered off (or suspended) in order to reduce power costs. On the other hand, VMs must be given enough resources to meet their demand. VM performance is encapsulated in a *Service Level Agreement* (SLA), which in a basic form defines some thresholds on performance metrics, such as response time. Specifically, we define two primary objectives:

- *Minimize SLA violation*
- *Minimize power consumption*

In order to accomplish both goals, VMs must be placed into as few hosts as possible, and their placement must be dynamically adapted as their resource demands change.

The problem of dynamic virtual machine management is similar to the bin-packing problem, with the set of VMs considered as items and hosts as bins. The goal is then to pack VMs into hosts such that we use the fewest number of hosts possible. Greedy heuristics for the bin-packing problem can be effective, such as *first-fit decreasing* [6, 14]. The first-fit decreasing algorithm first sorts items in decreasing order by size, and then places them sequentially in the first available bin with enough spare capacity. There are other factors, however, which further complicate the situation. As VM resource demands are highly variable and change over time, the initial placement must be constantly adapted. The entire placement cannot simply be recalculated from scratch, as VMs are now placed and running in hosts, and cannot all be removed to perform a complete new placement. Rather, a few individual VMs must be moved via VM live migration in order to ensure that all VMs have enough resources to meet their demand. Live migration, however, incurs some performance and networking costs, and as such the number of migrations performed should be minimized. We include secondary objectives related to minimizing the impact of dynamic management:

- *Minimize the number of migrations*
- *Minimize management bandwidth usage*
- *Ensure management is responsive and scalable*

The goals of minimizing SLA violations and reducing power consumption are often in conflict. Power consumption is best minimized with a tight packing of VMs into hosts, but this comes at the cost of an increased risk of resource contention and SLA violations. On the other hand, if additional host resources are left unused to accommodate increased demand from VMs, more hosts are required and power consumption is increased. The order in which VMs and hosts are considered in a bin-packing heuristic can have an effect on which migrations are performed, and whether the heuristic algorithm favours the achievement of one goal or another. In this chapter, we define the general problem of dynamic VM management, present a division of the problem into three distinct operations, and examine a set of variations on a first-fit algorithm presented by Keller et al. [9], which provide a base for the algorithms presented in subsequent chapters.

The remainder of this section is organized as follows: Section 2.2 presents some related work, and Section 2.3 formally introduces the problem. Section 2.4 introduces a set of management operations for dynamic VM management, Section 2.5 proposes algorithms to perform these operations, and Section 2.6 evaluates the algorithms. Finally, we conclude in Section 2.7.

2.2 Related Work

Initial work in the literature examined computing static VM placements, often based on resource demand predictions, which may be periodically recomputed to adjust for changing demands. Bobroff et al. [4] propose a first-fit descreasing heuristic based on the forecasted demand of each VM. The placement is periodically re-computed to adapt to changing demands.

Best-fit approaches have also been investigated [5]. Speitkamp and Bichler [11] compute a VM placement using a linear programming based heuristic. Finally, vector bin-packing has also proven effective, as demonstrated by Stillwell et al. [12]. Since these approaches calculate a VM placement from scratch, they are not suitable for managing VMs with highly dynamic demands.

Best-fit and first-fit heuristics have also been adapted to perform dynamic management. Wood et al. [13] propose Sandpiper, which uses a first-fit heuristic to dynamically adapt VM placement. Beloglazov and Buyya [3] propose a set of best-fit heuristic algorithms for dynamic VM management. Khanna et al. [10] propose an optimization model to solve the problem, as well as a heuristic solution. Their solution sorts VMs by CPU utilization and memory consumption, and hosts by available capacity, in an attempt to minimize migration costs and maximize host resource utilization. Such heuristics offer the benefit of quick execution and scalability while still calculating a good placement. Other techniques have been employed as well, such as fuzzy logic-based controllers [7].

2.3 Problem Definition

In this section, we present various aspects of the dynamic VM management problem. In its simplest form, the problem of dynamic VM management is to place a set of VMs, V, into a set of hosts, H, and to dynamically adapt this placement as the resource demand of each VM changes. Hosts are located within *racks*, which are sets of hosts directly connected via a single switch. Racks are connected to each other through additional layers of networking, the details of which are not considered for the purpose of this work. We denote a rack as $r \in R$, and the set of hosts in r as $H_r \subseteq H$.

We consider two characteristics of a VM ($v \in V$):

- Its *CPU usage*: denoted $\omega_v(s)$. It also has a fixed upper limit on CPU usage, ω_v^{max}. The CPU utilization of a VM is therefore $\omega_v(s)/\omega_v^{max}$. CPU usage is quantified in terms of *CPU units*, in which a single unit is equivalent to 1MHz clock speed. For example, a 2.5GHz processor core would have 2500 CPU units.

- Its *memory*: denoted μ_v. Memory size is static.

The *placement* (also called *allocation*) of a VM, is defined by the following function:

$$\alpha_{h,v}(s) = \begin{cases} 1 & \text{if } v \text{ is placed on } h \\ 0 & \text{otherwise} \end{cases}$$

A VM can only be placed on a single host at a time. Therefore, if there exists a host h_1 such that $\alpha_{h_1,v}(s) = 1$, then $\forall h \in H \setminus h_1, \;\; \alpha_{h,v}(s) = 0$.

Each host, $h \in H$, has a total memory capacity M_h. The placement must be such that the sum of the memory allocations for each VM on a host does not exceed the memory capacity of the host.

$$\sum_{v \in V} \mu_v * \alpha_{h,v}(s) \leqslant M_h, \quad \forall h \in H \;\; \forall s \in \mathbb{R}^+$$

Each host, h, has a total CPU capacity Ω_h. The CPU usage of a host, $\Omega'_h(s)$, is equal to the sum of the CPU usage of all VMs placed on that host, and cannot exceed the total CPU capacity of the host. Consequently, for $s \in \mathbb{R}^+$, we have:

$$\Omega'_h(s) = \sum_{v \in V} \omega_V(s) * \alpha_{h,v}(s) \leqslant \Omega_h$$

Note that this is not a rule that must be enforced, but rather is a physical property of the host.

For simplification, we denote the set of VMs on a host h as V_h, Consequently, we have: $v \in V_h \iff \alpha_{h,v} = 1$.

It is important to avoid overloading the CPU of a host, in order to avoid compromising the performance of VMs placed on it. To this end, we define an upper threshold on CPU utilization, τ (e.g. 0.9, or 90%). We calculate the amount by which each host, h, exceeds τ. We denote this as $\overline{\tau}_h(s)$, and calculate is as follows:

$$\overline{\tau}_h(s) = \begin{cases} \dfrac{\Omega'_h(s)}{\Omega_h} - \tau & \text{if } \dfrac{\Omega'_h(s)}{\Omega_h} - \tau > 0 \\ 0 & \text{otherwise} \end{cases}$$

Then, our goal is to minimize this value for all hosts, which gives use the following objective:

$$\forall s \in \mathbb{R}^+, \;\; \min \sum_{h \in H} \overline{\tau}_h(s)$$

A second goal is to reduce power consumption, which, in this work, is accomplished by minimizing the number of *active* hosts. An active host contains at least one VM. The number of VMs on a host is denoted $|h|$. Formally, a host is active at time $s \in \mathbb{R}^+$ if the following function is equal to 1:

$$\mathit{active}(h, s) = \begin{cases} 1 & \text{if } |h|(s) > 0 \\ 0 & \text{if } |h|(s) = 0 \end{cases}$$

By minimizing the number of active hosts, we can ensure that hosts are not underutilized. Since an idle server can still use 50% of its maximum power consumption [2], higher host utilization results in better power efficiency in terms of CPU processing per watt. Therefore, the total number of active hosts should be minimized, giving us the following objective:

$$\forall s \in \mathbb{R}^{+}, \quad \min \sum_{h \in H} active(h, s)$$

2.4 Management Operations

We define three primary operations that are involved in dynamic VM management, as well as four distinct host states used to classify a host. Hosts are classified based on their CPU utilization as follows:

- *stressed* hosts (denoted $H^{!}$) have a high CPU utilization, and are therefore considered to be at risk of overloading. In this case, the performance of VMs running on the host would degrade, violating SLA. It is therefore undesirable for a host to be in the stressed state. Stressed hosts have a CPU utilization higher than a specified upper threshold, denoted $\overline{\Omega}^{\tau}$;
- *partially utilized* hosts (denoted H^{+}) have a *normal* CPU utilization level, representing a healthy host. This is the ideal state for a host, as it is highly utilized (and therefore energy efficient), but not stressed. Partially utilized hosts have a CPU utilization below $\overline{\Omega}^{\tau}$ and above a lower threshold, denoted $\underline{\Omega}^{\tau}$;
- *underutilized* hosts (denoted H^{-}), have a low CPU utilization. Since servers are at their least power efficient under light loads, hosts in this state represent a waste of power consumption and should be consolidated onto fewer, partially utilized hosts or shut down entirely. Hosts in the underutilized state have a CPU utilization below $\underline{\Omega}^{\tau}$;
- *empty* hosts (denoted $H^{\emptyset}$) are not hosting any VMs, and may be in a lower power state such as *suspended* or *off* to conserve power.

Our approach to dynamic VM management consists of three operations, described in the remainder of this section.

VM Placement

The VM Placement operation is responsible for the initial placement of a new VM in the data centre. When a client requests the instantiation of a new VM, the VM Placement operation is invoked, a target host is selected, and the new VM is instantiated on that host.

VM Relocation

The VM Relocation operation is responsible for relieving *stress situations*. A stress situation refers to one or more hosts being in the *stressed* state. The VM Relocation operation is responsible for alleviating the stressed hosts by selecting one or more VMs on the stressed host to migrate away, as well as selecting target hosts on which to place the migrating VMs. The target host must have enough spare capacity to host the incoming VMs without becoming stressed itself. Once migrations have been found to eliminate the stress situation, they are executed. The VM Relocation operation can either be triggered on a periodic interval, or can be triggered in response to a host becoming stressed.

VM Consolidation

The VM Consolidation operation is responsible for minimizing the number of hosts required to contain the current set of VMs. It does so by identifying underutilized hosts, and migrating all VMs on the host away in order to be able to shut down the underutilized host. Similar to the VM Relocation operation, it searches for migration target hosts for each VM on an underutilized host, and if successful, performs the migrations and shuts down (or suspends) the underutilized host. Note that the operation may be unable to migrate all VMs on an underutilized host. In this case, it may migrate only a subset of the VMs, in the hope that placements can be found for the remaining VMs in subsequent invocations of the operation. The VM Consolidation operation is usually triggered on a periodic interval, less frequently than VM Relocation.

2.5 Algorithms

We now present algorithms to perform each of the three management operations. We sometimes refer to a specific implemention of a management operation as a *policy* (e.g. a VM Reloction policy). Each of the algorithms is similar, based on a first-fit heuristic. This heuristic algorithm first sorts the VMs present on a host, then sorts the possible target hosts for migration, and finally iterates through each VM, choosing the first available target host from the list with enough capacity to host the VM. Once this process is complete, the migrations are performed. The order in which VMs and target hosts are sorted has an impact on the performance and characteristics of the algorithm.

2.5.1 VM Relocation

As mentioned above, a host is considered stressed when its CPU utilization exceeds the upper threshold, $\overline{\Omega}^{\tau}$. The VM Relocation operation is then tasked with relieving the stress situation through VM migration. The VM Relocation operation is triggered on a regular interval (e.g. every 10 minutes). Algorithm 1 describes the VM Relocation algorithm. It takes as input the set of hosts in the data centre (H), and first classifies them into *stressed* ($H^!$), *partially utilized* (H^+), *underutilized* (H^-), and *empty* ($H^\emptyset$) sets (line 1). We use the set of *stressed* hosts ($H^!$) as the sources for migration, as we want to move VMs off of these hosts to solve the stress situations. This set is sorted in line 2. The remaining categories are sorted and then combined into a single list in line 3. As mentioned previously, the sorting order has an impact on the algorithm, and will be discussed further in Section 2.5.4. The VMs of each source host (V_h) are sorted (line 6) and iterated through, searching for a target host (h_t) for each $v \in V_h$ (lines 7-13). Once a suitable VM and target has been found, the algorithm performs the migration (line 10) and moves on to the next source host.

Algorithm 1: VM Relocation

```
   Data: $H$
1  $H^!, H^+, H^-, H^\emptyset \leftarrow$ classify($H$)
2  $sources \leftarrow$ sort($H^!$)
3  $targets \leftarrow$ combine(sort($H^+$), sort($H^-$), sort($H^\emptyset$))
4  for $h \in sources$ do
5      $success \leftarrow$ FALSE
6      $vmCandidates \leftarrow$ sort($V_h$)
7      for $v \in vmCandidates$ do
8          for $h_t \in targets$ do
9              if $hasCapacity(h_t, v)$ then
10                 migrate($h$, $v$, $h_t$)
11                 $success \leftarrow$ TRUE
12                 break
13         if $success$ then break
```

2.5.2 VM Consolidation

The VM Consolidation operation is similar in operation to VM Relocation. It is responsible for migrating VMs off of underutilized hosts, with CPU utilization below the lower threshold, $\underline{\Omega}^{\tau}$. Algorithm 2 describes the VM Consolidation algorithm. Again, it takes as input the set of hosts in the data centre (H). Hosts are classified (line 1), and the *underutilized* (H^-) set is sorted used as the source set (line 2). The target hosts for migration come from the *partially utilized* (H^+) and *underutilized* (H^-) hosts (line 3). We initialize the $usedTargets$ and $usedSources$ sets to keep track of hosts that have been used as a target or source for migration, in order to avoid using the same host as both a source and a target for migration (lines 4-5). This is necessary as *underutilized* hosts are in both the soure and target sets. Each host in the source set is then iterated through (line 6), skipping any host that has been used as a target (line 7). All VMs in the source host are then iterated through (line 8), searching for target hosts to which to migrate them. If a host is found that has enough spare capacity, is not the same host as the source, and has not been used previously as a source, then we proceed with a migration (line 10). First, we update the $usedTargets$ and $usedSources$ sets (lines 11-12), and finally perform the migration (line 13).

Algorithm 2: VM Consolidation

```
   Data: $H$
1  $H^!, H^+, H^-, H^\emptyset \leftarrow$ classify($H$)
2  $sources \leftarrow$ sort($H^-$)
3  $targets \leftarrow$ combine(sort($H^+$), sort($H^-$))
4  $usedTargets \leftarrow \emptyset$
5  $usedSources \leftarrow \emptyset$
6  for $h \in sources$ do
7      if $h \in usedTargets$ then continue
8      for $v \in V_h$ do
9          for $h_t \in targets$ do
10             if $hasCapacity(h_t, v) \wedge h_t \neq h \wedge h_t \notin usedSources$ then
11                 $usedTargets \leftarrow usedTargets \cup h_t$
12                 $usedSources \leftarrow usedSources \cup h$
13                 migrate($h$, $v$, $h_t$)
14                 break
```

2.5.3 VM Placement

The VM Placement operation is, again, similar to VM Relocation. When a client requests the creation of a new VM in the data centre, the VM Placement operation is executed to place the VM on a host. Algorithm 3 describes the VM Placement algorithm. It takes the set of hosts in the data centre (H), and the new VM to be placed (v) as input. Hosts are classified (line 1), and the target list is built from the *partially utilized* (H^+), *underutilized* (H^-), and *empty* ($H^\emptyset$) sets of hosts (line 2). The list of targets is iterated through (line 3), and if the potential target host has enough capacity (line 4), then the new VM is placed on that host (line 5) and the algorithm terminates. Note that it is possible that no suitable target host is found, in which case the placement fails.

Algorithm 3: VM Placement

```
  Data: $H, v$
1 $H^!, H^+, H^-, H^\emptyset \leftarrow$ classify($H$)
2 $targets \leftarrow$ combine(sort($H^+$), sort($H^-$), sort($H^\emptyset$))
3 for $h_t \in targets$ do
4     if hasCapacity($h_t$, $v$) then
5         place($v$, $h_t$)
6         break
```

2.5.4 Sort Ordering

The algorithms for each of the three management operations include steps to sort both the list of target hosts, and the list of potential VMs to migrate. For example, in the VM Relocation algorithm (Algorithm 1), potential target host sets are sorted and then combined in line 3, and candidate VMs are sorted in line 6. Keller et al. [9] propose varying the order of both target hosts and candidate VMs to affect the performance and characteristics of the algorithms. Three options for target host sorting are proposed:

- **Increasing**: sort *partially utilized* (H^+) and *underutilized* (H^-) hosts in increasing order by CPU utilization, and combine the sets as $H^- \cdot H^+ \cdot H^\emptyset$.
- **Decreasing**: sort *partially utilized* (H^+) and *underutilized* (H^-) hosts in decreasing order by CPU utilization, and combine the sets as $H^+ \cdot H^- \cdot H^\emptyset$.
- **Mixed**: sort *partially utilized* (H^+) in increasing order and *underutilized* (H^-) hosts in decreasing order by CPU utilization, and combine the sets as $H^+ \cdot H^- \cdot H^\emptyset$.

These three sorting orders represent three different approaches to host target selection. The *increasing* order attempts to place the VM on the most lightly loaded host available which is not empty (i.e. *off* or *suspended*). This approach gives the VM the best chance of having enough resources to fulfil its current and future needs, but may result in load being spread across more hosts than necessary. The *decreasing* sorting order attempts to place VMs on the target host with the highest utilization which has enough spare capacity to fit the VM. This keeps VMs tightly packed on fewer hosts, but comes at the risk of causing a new stress situation on the target host in the near future. Finally, the *mixed* sorting order attempts to find a balance between

Policy	VM sorting	Target sorting
FFDI	Decreasing	Increasing
FFDD	Decreasing	Decreasing
FFDM	Decreasing	Mixed
FFII	Increasing	Increasing
FFID	Increasing	Decreasing
FFIM	Increasing	Mixed

Table 2.1: Dynamic Management Policies

the two by starting its search at the boundary between *partially utilized* and *underutilized*, and prefering *partially utilized*.

Next, the candidate VMs for migration must be sorted, in one of two ways:

- **Increasing**: sort VMs in increasing order by CPU utilization (prefer smaller VMs).

- **Decreasing**: sort VMs in decreasing order by CPU utilization (prefer largers VMs).

By combining the three sorting orders for target hosts with both sorting orders for candidate VMs, we can construct six different *policies* for dynamic VM management. Table 2.1 lists all six policies (sort orders).

2.6 Evaluation

Evaluation of the algorithms for VM Placement, VM Relocation and VM Consolidation was conducted by Keller et al. [9] through simulation, using the DCSim open-source simulation tool (see Chapter 7). Each of the policies defined in Section 2.5.4 were evaluated, as well as a *random* policy which selected candidate VMs and target hosts at random. Individual *VM Relocation*, *VM Consolidation* and *VM Placement* policies were created for each overarching policy (i.e. FFDI, FFDD, etc.). Each policy and configuration was simulated five times, each with a different randomized workload, and the average results were taken. The simulated data centre contained 100 hosts, and 400 VMs. Each VM is driven by one of 5 traces: the *ClarkNet*, *EPA*, and *SDSC* traces [1], and two different job types from the *Google Cluster Data* trace [8]. The normalized rate of incoming requests, in 100 second intervals, is calculated for each trace. The request rates are used to define the current workload of each VM, with the CPU resource requirements of the VM calculated as a linear function of the current rate. Each VM starts its trace at a randomly selected offset time.

2.6.1 Metrics

The simulations were evaluated based on the following reported metrics:

Migrations

The total number of migrations performed during the experiment. Since migrations incur both performance and networking overhead, fewer migrations are preferred.

Active Hosts

The average number of *active* hosts. An *active* host is defined as a host in the *on* state, containing at least one VM. Fewer *active* hosts typically implies reduced power consumption, and demonstrates the ability to host a set of VMs with a smaller infrastructure.

Host Utilization

The average CPU utilization of *active* hosts. Higher host utilization values implies that the management system is making better use of available resources, which should result in decreased power consumption. It may also, however, come at the cost of VM performance.

Power Consumption

The total power consumed during the simulation, in kWh. Lower values are preferred, as this translates directly into energy cost savings.

SLA Violation

Generally speaking, we consider that an SLA violation occurs when a VM is not given the resources it requires to perform its task. In this evaluation, SLA violations are recorded as the percentage of CPU which the VM requires but does not receive due to contention with other VMs. Note that it is impossible to obtain this value in a real-world environment. It is assumed that a VM which does not receive the CPU time it requires suffers a degradation in performance, and therefore is in violation of its SLA. This metric is further explained as *CPU Underprovisioned* in Chapter 7.

2.6.2 Results

Table 2.2 shows the results of the experiment. We summarize the important insights gained from the experiments below. For additional results and details, refer to Keller et al. [9].

	FFDI	FFDD	FFDM	FFII	FFID	FFIM	Random
Migrations	478	737	585	814	2315	826	1015
Active Hosts	88.6	84.7	86.4	82.9	78.0	84.7	93.0
Host Utilization	65.3%	67.5%	66.4%	68.8%	72.5%	67.8%	59.2%
Power Consumption	7879.6	7696.4	7781.8	7610.8	7300.6	7684.2	8397.6
SLA Violation	0.6%	0.8%	0.7%	0.7%	1.7%	0.7%	0.7%

Table 2.2: First-fit Algorithm Comparison

FFDI

The FFDI (*decreasing* VM sorting, *increasing* target host sorting) achieved the best SLA performance, while consuming the most power and using the fewest migrations. This policy attempts to take the largest VM from a stressed host and place it on the least loaded non-empty host available. Thus, it creates a situation in which the stressed host utilization is significantly reduced, and the migrated VM is on a machine with a large spare capacity. This reduces the risk of subsequent stress situations, and therefore improves SLA performance and decreases the probability that an additional migration will be required. By the same token, VMs become spread across a larger number of *partially utilized* or *underutilized* hosts, thus increasing power consumption.

FFID

The FFID (*increasing* VM sorting, *decreasing* target host sorting) achieved the opposite effect, providing the best power consumption at the expense of the worst SLA performance and most triggered migrations. This policy attempts to remove the smallest possible VM to relieve the stress situation, and place it on the highest utilized host that still has enough spare capacity to contain it. Thus, while it keeps hosts highly utilized and thus maintains low power consumption, it increases the risk that a new stress situation will be triggered at the target host, and may even leave the source host in danger of again becoming stressed. This inevitably triggers additional migrations to deal with the subsequent stress situations, and incurs SLA violation.

Remaining Policies

The remaining variations all fall somewhere in between the two extremes presented by FFDI and FFID. They do not perform the best at any particular metric, nor do they perform the worst.

2.7 Conclusions

We have introduced the problem of *Dynamic VM Management*, and taken a look at some of the challenges that it brings. Furthermore, we have presented the general approach taken to address dynamic management throughout the remainder of the work. That is, dynamic VM management is divided into three primary operations: *VM Placement*, *VM Relocation*, and *VM Consolidation*. Variations on a first-fit heuristic, proposed by Keller et al. [9], are employed to carry out these operations. The behaviour of these algorithms can be influenced through the use of different sorting and concatenation orders for target hosts and candidate VMs for migration, allowing the algorithms to be tuned towards one goal over another (i.e. SLA versus power consumption, or vise versa). In fact, there is no single approach that satisfies both primary goals, but rather there is a spectrum of options to select from in order to match algorithm behaviour with business goals and priorities.

This basic structure and set of algorithms is used and built upon throughout the remaining chapters, in order to address ever more complex situations and goals.

Bibliography

[1] The Internet Traffic Archive. The internet traffic archive. `http://ita.ee.lbl.gov/`, July 2014.

[2] L.A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec 2007.

[3] 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 Computat.: Pract. Exper.*, pages 1–24, 2011.

[4] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *IM Proceedings, 2007 IEEE/IFIP Int. Symp. on*, pages 119–128, 2007.

[5] Michael Cardosa, Madhukar R. Korupolu, and Aameek Singh. Shares and utilities based power consolidation in virtualized server environments. In *IM Proceedings, 2009 IEEE/IFIP Int. Symp. on*, 2009.

[6] Gyrgy Dsa. The tight bound of first fit decreasing bin-packing algorithm is FFD (I)≤11/9 OPT (I)+6/9. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74450-4_1.

[7] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *38th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, June 2008.

[8] Google Inc. Google cluster data. `http://code.google.com/p/googleclusterdata/`, July 2014.

[9] Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. An analysis of first fit heuristics for the virtual machine relocation problem. In *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, October 2012.

[10] Gunjan Khanna, Kirk Beaty, Gautam Kar, and Andrzej Kochut. Application performance management in virtualized server environments. In *NOMS Proceedings, 2006 IEEE/IFIP*, 2006.

[11] B. Speitkamp and M. Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE TSC*, 3(4):266 –278, 2010.

[12] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *J. Parallel Distrib. Comput.*, 70(9):962–974, September 2010.

[13] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI Proceedings, 4th Symp. on*, pages 229–242, Cambridge, MA, USA, April 2007.

[14] 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 (English Series)*, 7:321–331, 1991. 10.1007/BF02009683.

Chapter 3

Switching Data Centre Management Strategies at Run-time

3.1 Introduction

In Chapter 2, we introduced two primary goals for dynamic VM management:

- *Minimize SLA violation*
- *Minimize power consumption*

The operator of the data centre (the *cloud provider*), may want to conserve power to reduce costs. This is accomplished by consolidating load onto as few hosts (physical servers) as possible, and switching remaining hosts into a power saving state (*off* or *suspended*). At the same time, they must provide the level of service that clients expect, in terms of resources available to VMs. As such, either additional capacity needs to be left in reserve for VMs when required, or VM placement needs to be dynamically adapted to meet changing resource demands, or both. If VMs are tightly packed onto hosts to conserve power, they are likely to contend for resources with co-located VMs and violate their SLA. On the other hand, if VMs are less tightly packed, they will have fewer SLA violations but consume more power. As such, the two primary goals are in conflict with one another.

We define a *dynamic management strategy* to consist of a set of policies, such that there is a policy that governs each of the defined management operations (i.e. a *VM Placement* policy, a *VM Relocation* policy, and a *VM Consolidation* policy). The set of policies which form a management strategy are designed in such a way as they work together as a cohesive unit, following a common approach to dynamic management. The design of management strategies

This chapter is based on work published in [6] often focuses on achieving a single goal, or on prioritizing goals such that a single goal is considered the primary goal and others are considered secondary, e.g., [4, 11, 13, 14]. Designing a management strategy to achieve multiple goals is difficult, as improving performance towards one goal frequently results in degradation of performance towards another. Such is the case with our primary goals of *minimize SLA violation* and *minimize power consumption*.

In order to address the problem of achieving multiple goals simultaneously, we propose *dynamic strategy switching*. Dynamic strategy switching involves changing between a set of strategies at run-time, each of which focuses on a single goal. In this case, one strategy is designed to minimize SLA violations, and another to minimize power consumption. We hypothesize that depending on the current state of the data centre, one strategy will be more effective/necessary than the other. By recognizing these situations and switching to the appropriate strategy for the given situation, we aim to achieve better performance in attaining both goals. The main contributions of this work are three novel methods of dynamically switching between single-goal management strategies, and a method of comparing the performance of strategies that aim to achieve more than one goal.

The remainder of this chapter is organized as follows: Section 3.2 reviews related work in this area, Sections 3.3 and 3.4 describe the management strategies and strategy switching approaches we explored, respectively. Section 3.5 presents experiments and evaluation, and finally, Section 3.6 concludes and discusses future work.

3.2 Related Work

Research in dynamic management of virtualized data centres focuses on one or more of the three primary management operations (i.e. *VM Placement*, *VM Relocation* or *VM Consolidation*). Most solutions focus on pursuing a single goal, or in the case of multiple goals, prioritizing one over all others. Wood et al. [14] addressed the VM Relocation problem using a first-fit decreasing heuristic, with the goal of reducing SLA violations. In the process of attaining this goal, it spreads VMs out across hosts, and thus sacrifices power consumption. As described in Chapter 2, Keller et al. [10] studied variants of a first-fit heuristic to address the VM Relocation problem, showing that the order in which VMs and hosts are considered for migration impacts the behaviour of the algorithms. In particular, the algorithm can be tuned to favour SLA or power related goals, but always at the expense of the other goal.

Some work addresses VM Relocation and VM Consolidation together, in most cases, focusing primarily on minimizing power consumption. Minimizing SLA violations becomes a secondary goal. Khanna et al. [11] implemented a first-fit heuristic that migrated the least loaded VMs (in terms of CPU and memory usage) into the highest loaded hosts. Verma et al. [13] relied on a first-fit decreasing heuristic which placed VMs in the most power efficient servers first. Beloglazov and Buyya [3] proposed a best-fit decreasing heuristic which selected for migration the VMs with the smallest memory footprint and placed them in the host that provided the least increase in power consumption. Much of the existing work on dynamic management uses some form of first-fit heuristics, though occasionally alternatives are proposed, such as fuzzy logic-based controllers [7].

Unlike the existing work, we focus on achieving multiple goals simultaneously. Neither goal is considered secondary, and although they may be opposing goals (e.g. minimizing SLA violations and minimizing power consumption), we attempt to achieve both without prioritizing one goal over another.

3.3 Management Strategies

As mentioned in Section 3.1, minimizing SLA violations and minimizing power consumption are commonly studied goals in data centre management. In this section, we will discuss the design of management strategies to pursue these goals, which are representative of those found in the literature. We design two *single-goal* management strategies, *Power* and *SLA*, to work towards achieving a single goal each. Then, we present one *Hybrid* strategy, which makes a best effort to achieve both goals simultaneously, still within a single strategy. The strategies presented assume frequent monitoring. Calculations are performed on monitored values over a sliding window of time, referred to as the *monitoring window*.

3.3.1 Terminology

This section presents the terms and metrics used in the description of management strategies.

SLA Violation

An SLA violation, denoted S^v, occurs when resources required by a VM are not available to it, as this situation leads to a degradation in performance. The percentage of required CPU not available is the SLA violation, as described in Chapter 2.6.

Data Centre Utilization

The overall utilization of the data centre is calculated as the percentage of total CPU capacity in the data centre that is currently in use.

Power Efficiency

For a host, h, the power efficiency, ψ_h, is the amount of processing being performed per watt of power. This is measured in CPU-shares-per-watt (cpu/watt). Let Ω'_h be the CPU utilization of host h, and Ψ_h be its power consumption. Power efficiency (ψ_h) of a single host is then calculated as follows:

$$\psi_h = \frac{\Omega'_h}{\Psi_h}$$

As an active host machine consumes a significant amount of power even when under little or no CPU load (i.e. very low power efficiency) increased host utilization corresponds with increased power efficiency for that host. This metric is used to calculate the power efficiency for the entire data centre, ψ_{dc}, calculated as

$$\psi_{dc} = \frac{\sum\limits_{h \in H} \Omega'_h}{\sum\limits_{h \in H} \Psi_h}$$

Maximum Power Efficiency

This metric represents the best power efficiency a host can achieve, calculated as the power efficiency of the host at maximum CPU utilization, and is denoted ψ_h^{max}.

Optimal Power Efficiency

Optimal Power Efficiency, ψ_{dc}^{opt}, represents the best possible power efficiency achievable at the data centre level, given the current workload and set of host machines available. The best power efficiency would be achieved by placing VMs in such a way that each host is 100% utilized, with the most power efficient hosts being filled first. We first calculate the total CPU-in-use across the data centre. We order the available hosts by maximum power efficiency, and allocate the CPU-in-use to hosts such that each host is allocated 100% of its CPU capacity. We calculate ψ_{dc}^{opt} to be the power efficiency of the data centre given this allocation.

3.3.2 Power and SLA Strategies

The *Power* and *SLA* strategies are *single-goal* strategies, which means that all management decisions are geared towards achieving a single, primary goal. Single-goal strategies may pursue secondary goals, but always give them lower priority than the primary goal. The presented strategies are based on the management operations and algorithms presented in Chapter 2.4.

These strategies have several opportunities to differentiate themselves in order to pursue their individual goals, and the choices are reflected in the individual policies of each strategy. Strategies can be tailored to a specific goal through the following methods:

- Setting appropriate threshold values for classifying hosts as *stressed* (the upper threshold, $\overline{\Omega}^{\tau}$) and *underutilized* (the lower threshold, $\underline{\Omega}^{\tau}$).
- Modifying the sorting order of target hosts in the three management operations.
- Setting the execution frequency of *VM Relocation* and *VM Consolidation* operations.

See Chapter 2.4 for details on host classification and management operations.

In the following sections, we describe the specific settings for the policies of each strategy, and how they work to achieve the goal of the strategy.

Host Classification

Each strategy uses a different value for the upper threshold, $\overline{\Omega}^{\tau}$, which controls the point at which a host is classified as *stressed*. The *power* strategy sets $\overline{\Omega}^{\tau} = 95\%$, allowing hosts to be very highly utilized before triggering a stress situation and subsequent VM migration. The *SLA* strategy set $\overline{\Omega}^{\tau} = 85\%$, which provides an extra safety margin to handle increases in VM CPU demand. Both strategies use a lower threshold value of $\underline{\Omega}^{\tau} = 60\%$, which defines the point at which a host is classified as *underutilized* and is a candidate for *VM Consolidation*.

VM Placement

The *VM Placement* operation runs each time a new VM creation request is received, and selects a host in which to place the new VM. See Algorithm 3 for the general placement algorithm. We modify the construction of the target host list (line 2) for each strategy. The *Power* strategy sorts the *partially utilized* (H^{+}) and *underutilized* (H^{-}) sets in decreasing order first by maximum power efficiency (ψ_h^{max}), and then by CPU utilization (Ω_h'). The *empty* ($H^{\emptyset}$) host set is sorted in decreasing order by power efficiency. Finally, the target list is built as $H^{+} \cdot H^{-} \cdot H^{\emptyset}$. This sorting method ensures that the placement focuses on power efficiency over any other considerations.

The *SLA* strategy constructs its target host list differently. The *partially utilized* hosts are sorted in increasing order first by CPU utilization and then by maximum power efficiency, and the *underutilized* hosts are sorted in decreasing order first by CPU utilization, and then by maximum power efficiency. The sorting order of the *empty* hosts, as well as the final construction of the target list, remains the same as in the *Power* strategy. This sorting method ensures that the placement focuses on spreading load across the set of hosts, leaving spare resources to handle spikes in resource demand, at the expense of other goals.

VM Relocation

The *VM Relocation* operation responds to *stressed* hosts by migrating a VM to free additional capacity and eliminate the stress situation. Both strategies execute the operation every 10 minutes. See Algorithm 1 for the general VM relocation algorithm. We modify the host classification (line 1) and target host list construction (line 3) for each strategy, and both strategies sort *stressed* hosts (line 2) in decreasing order by CPU utilization (Ω'_h). During host classification, the *Power* strategy considers a host to be *stressed* if its CPU utilization has remained above the $\overline{\Omega}^{\tau}$ threshold over a specified monitoring window. This helps ensure that a migration is not triggered due to a transient spike in demand, but also may result in a slow reaction to a sustained increase. It constructs the target host list in the same manner as described for the *VM Placement* operation.

The *SLA* strategy, on the other hand, considers a host to be *stressed* if its CPU utilization exceeds the $\overline{\Omega}^{\tau}$ threshold in the last recorded monitoring value, or on average over a specified monitoring window. It constructs the target host list in the same manner as described for the *VM Placement* operation.

VM Consolidation

The *VM Consolidation* operation consolidates VMs onto the fewest number of hosts possible by migrating VMs off of underutilized hosts and switching them into a power saving mode (i.e. *off* or *suspended*). This operation executes less frequently then *VM Relocation*. See Algorithm 2 for the general VM consolidation algorithm. Both strategies construct their target host lists (line 3) in the same manner as in their respective *VM Placement* and *VM Relocation* policies. Furthermore, the *Power* strategy executes *VM Consolidation* relatively frequently, on a 1 hour interval, in order to aggressively consolidate load. The *SLA* strategy executes *VM Consolidation* only every 4 hours, thereby decreasing the risk of overloading hosts by consolidation.

3.3.3 Hybrid Strategy

We designed a *dual-goal* strategy as a combination of the Power and SLA strategies; the *Hybrid* strategy consists of the *VM Placement* and *VM Relocation* policies of the *SLA* strategy and the *VM Consolidation* policy of the *Power* strategy. Furthermore, the stress check performed by the *VM Relocation* policy represents a compromise between the checks of *SLA* and *Power*: it determines that a host is stressed only if its average CPU utilization over the last monitoring window exceeds the $\overline{\Omega}^{\tau}$ threshold. The thresholds $\overline{\Omega}^{\tau}$ and $\underline{\Omega}^{\tau}$ were set to 90% and 60%, respectively, as a compromise between the values set for the *Power* and *SLA* strategies.

3.4 Dynamic Strategy Switching

Dynamic Strategy Switching (DSS) refers to changing between strategies at run-time in response to changing data centre state. DSS periodically performs an evaluation of data centre metrics monitored between executions to determine if the strategy currently in use (the *active strategy*) should be changed. In this section, we present three different DSS meta-strategies.

3.4.1 SP-DSS

The SLA-Power Thresholds (SP-DSS) meta-strategy uses the SLA violation (S^v) and power efficiency ratio (ψ') metrics to evaluate whether the active strategy should be switched. The power efficiency ratio is calculated as the ratio of optimal power efficiency (ψ_{dc}^{opt}) to current power efficiency (ψ_{dc}) over the last hour. A strategy switch is triggered when the metric related to the goal of the active strategy (i.e., S^v for the *SLA* strategy, ψ' for the *Power* strategy) is below a normal (i.e. acceptable) threshold (S^v_{norm} or ψ'_{norm}), while the metric related to the inactive strategy exceeds a high threshold (S^v_{high} or ψ'_{high}). See Algorithm 4 for details. The algorithm requires the current active strategy (*strategy*), as well as current values for ψ' and S^v. Switching strategies in this manner allows the data centre to respond to a situation in which performance in one metric has deteriorated, by activating the strategy that focuses on optimizing it.

Algorithm 4: SP-DSS Switching Conditions

```
  Data: $strategy, \psi', S^v$
1 if $strategy = Power$ then
2     if $\psi' < \psi'_{norm} \wedge S^v > S^v_{high}$ then
3         switchStrategy($SLA$)
4 else if $strategy = SLA$ then
5     if $S^v < S^v_{norm} \wedge \psi' > \psi'_{high}$ then
6         switchStrategy($Power$)
```

3.4.2 Goal-DSS

We define two goals, $S^v = 0\%$ and $\psi_{dc} = \psi_{dc}^{opt}$, to evaluate performance with respect to the S^v and ψ metrics. By calculating the distance to these goals, it is possible to determine towards which goal the system is performing worst and thus switch to the strategy that would improve achievement of that goal. The Distance to Goals (Goal-DSS) meta-strategy is based on this principle. Evaluating whether or not the active strategy should be switched requires the calculation of two metrics that represent the distance to those goals. Let S^v_{worst} be an operator defined parameter which indicates the worst acceptable SLA violation percentage. Then we calculate the SLA distance, denoted S^v_δ, as

$$S^v_\delta = \frac{S^v}{S^v_{worst}}$$

Similarily, let ψ_{worst} be the worst acceptable power efficiency, and ψ_Δ be an operator-defined parameter which indicates how large of a deviation from the optimal power efficiency is acceptable. Then, we calculate ψ_{worst} as

$$\psi_{worst} = \psi^{opt}_{dc} * \psi_\Delta$$

The power distance, Ψ_δ, is then calculated as

$$\Psi_\delta = 1 - \frac{\psi_{dc} - \psi_{worst}}{\psi^{opt}_{dc} - \psi_{worst}}$$

Calculating the distances in this manner is necessary in order to equate values of S^v with values of ψ, based on the parameters S^v_{worst} and ψ_Δ. These two values are considered equivalent in terms of distance to their respective goals. At each iteration of the strategy switching mechanism, the strategy for which the corresponding distance is greater is selected to become active. The switching algorithm, presented in Algorithm 5, is similar to that of SP-DSS.

Algorithm 5: Goal-DSS Switching Conditions

```
  Data: $strategy, S^v_\delta, \Psi_\delta$
1 if $strategy = Power$ then
2     if $S^v_\delta > \Psi_\delta$ then
3         switchStrategy($SLA$)
4 else if $strategy = SLA$ then
5     if $\Psi_\delta > S^v_\delta$ then
6         switchStrategy($Power$)
```

3.4.3 Util-DSS

Through experimentation, two key situations in which one strategy had an advantage over the other became apparent. When overall data centre utilization is growing, increasing the stress on host machines, the *SLA* strategy is more effective as it places greater emphasis on preventing SLA violations. Conversely, when utilization is decreasing or stable, thus increasing the likelihood of hosts becoming underutilized, the *Power* strategy is more effective as it can quickly make changes to conserve power. Data centre utilization is defined as the percentage of CPU shares in use across the entire data centre.

The Data Centre Utilization Trends (Util-DSS) meta-strategy is designed to exploit this pattern. It uses the rate of change of overall data centre utilization, m, to determine appropriate times to switch strategies. Measurements of the overall data centre utilization are taken at regular intervals. Linear regression over the last n data centre utilization measurements provides the rate of change, m, over a window of time. The value m_S defines a threshold for m over which a switch is made to the SLA strategy. Similarly, the value m_ψ defines a threshold for m under which the Power strategy is set to be active. The switching algorithm is presented in Algorithm 6.

Algorithm 6: Util-DSS Switching Conditions

```
   Data: $strategy, m, m_{S^v}, m_\psi$
1  if $strategy = Power$ then
2      if $m > m_S$ then
3          switchStrategy($SLA$)
4  else if $strategy = SLA$ then
5      if $m < m_\psi$ then
6          switchStrategy($Power$)
```

3.5 Experiments

This section presents our experimental approach and results.

3.5.1 Strategy Evaluation and Comparison

In order to evaluate the effectiveness of the strategies, two metrics are used: power efficiency (ψ) and SLA violation (S^v). Comparing strategies based only on the use of these two metrics is problematic. If one strategy were to perform well with respect to SLA violations at the expense of power, and another performed well with respect to power at the expense of SLA violations, it is difficult to conclude which strategy is preferable. The decision depends in part upon the relative change in each area as well as the importance placed on each metric by the data centre operators based on their business objectives, the relative costs of power and SLA violations and the potential for lost revenue due to poor application behaviour.

In order to determine whether DSS can offer improved results over a single strategy, we propose a method of evaluating the performance of a strategy based on experimental results. We use the *SLA* and *Power* strategies as benchmarks, with their SLA violation and power efficiency results serving as baseline measurements with which to evaluate other strategies. The *SLA* strategy provides the bounds for the best SLA violation value ($S^v_{best} = S_{sla}$) and the worst power efficiency ($\psi_{worst} = \psi_{sla}$), while the *Power* strategy provides the worst SLA violation ($S^v_{worst} = S^v_{power}$) and best power efficiency ($\psi_{best} = \psi_{power}$). Values from a candidate strategy, i, are then normalized using these bounds to produce the normalized vector, v_i, represented by $[S^v_{norm}, \psi_{norm}]$, where ψ_{norm} is the normalized power efficiency and S^v_{norm} is the normalized SLA violation. The values S^v_{norm} and ψ_{norm} are calculated as follows:

$$\begin{aligned} S^v_{norm} &= \frac{(S^v_i - S^v_{best})}{(S^v_{worst} - S^v_{best})} \\ \psi_{norm} &= \frac{(\psi_{best} - \psi_i)}{(\psi_{best} - \psi_{worst})} \\ v_i &= (S^v_{norm}, \psi_{norm}) \end{aligned}$$

Note that $\psi_{best} > \psi_{worst}$, but $S^v_{best} < S^v_{worst}$, so the normalization equations differ to reflect this. Once we have the normalized vector, v_i, we calculate its L^2-norm, $|v_i|$, and use this as an overall score ($score_i$) for the candidate strategy.

$$score_i = |v_i| = \sqrt{(S^v_{norm})^2 + (\psi_{norm})^2}$$

where a smaller score is considered better, as it represents a smaller distance to the *best* bounds of each metric (defined by S^v_{best} and ψ_{best}). The *SLA* and *Power* strategies always achieve a score of 1 by definition, as they achieve the *best* score in one metric and the *worst* in the other. Scores less than 1 indicate that overall performance of the candidate strategy has improved relative to the baseline strategies.

Note that this score is only valid for a single experiment in which all factors except for the active management strategy remain constant. In our work, we vary the workload pattern experienced by the data centre. As such, the baselines and score must be calculated separately for each workload pattern. The average final score across all experiments can then be used to evaluate the strategy. We use this method to evaluate and compare competing management strategies.

3.5.2 Experimental Setup

We conduct our experimentation by simulation using DCSim [12]. Our simulated data centre consists of 200 host machines, of which there are an equal number of two types: *small* and *large*. The *small* host is modelled after the HP ProLiant DL380G5, with 2 dual-core 3GHz CPUs and 8 GB of memory. The *large* host is modelled after the HP ProLiant DL160G5, with 2 quad-core 2.5GHz CPUs and 16GB of memory. Cores in the *large* host have 2500 CPU shares, and cores in the *small* host have 3000 CPU shares. The power consumption of both hosts is calculated using results from the SPECPower benchmark [5]. The maximum power efficiency of the *large* host (85.84 cpu/watt) is roughly double that of the *small* host (46.51 cpu/watt).

Three VM sizes are created: *small* requires 1 virtual core with at least 1500 CPU shares and 512MB of memory, *medium* requires 1 virtual core with at least 2500 CPU shares and 512MB of memory, and *large* requires 2 virtual cores with at least 2500 CPU shares each and 1GB of memory.

Hosts are modelled to use a work-conserving CPU scheduler, as available in major virtualization technologies. That is, any CPU shares not used by a VM can be used by another. No maximum cap on CPU is set for VMs. In the case of CPU contention, VMs are assigned shares in a round-robin fashion until all shares have been allocated. No dynamic voltage and frequency scaling (DVFS) is considered. Memory is statically allocated and not overcommitted.

During a VM migration, an SLA violation of 10% of CPU utilization is added to migrating VMs, and an additional CPU overhead of 10% of the migrating VMs CPU utilization is added to both the source and target host [3].

Measurements of metrics used by management policies, such as host CPU utilization and SLA violation, are drawn from each host every 2 minutes and evaluated by the policy over a sliding window of 5 measurements.

3.5.3 Workload

A data centre experiences a highly dynamic workload, driven by VM arrivals and departures, as well as dynamic workloads and resource requirements of VMs. We generate random *workload patterns* to evaluate our strategies, where a workload pattern consists of a set of VMs with specific start and stop times, each with dynamic trace-driven resource requirements. Each VM is driven by one of 5 individual traces: the *ClarkNet*, *EPA*, and *SDSC* traces [1], and two different job types from the *Google Cluster Data* trace [9]. The normalized rate of incoming requests, in 100 second intervals, is calculated for each trace. The request rates are used to define the current workload of each VM, with the CPU resource requirements of the VM calculated as a linear function of the current rate. Each VM starts its trace at a randomly selected offset time.

The number of VMs within the data centre is also varied dynamically to simulate the arrival and departure of VMs. A base of 600 VMs is created within the first 40 hours and remain running throughout the entire experiment, to maintain a reasonable minimum level of load. After 2 simulated days, new VMs begin to arrive at a changing rate, and terminate after about 1 day. The arrival rates are generated such that on a fixed interval of once per day, the total number of VMs in the data centre is equal to a randomly generated number uniformly distributed between 600 and 1600. The maximum number of VMs, 1600, was chosen because beyond that point, the SLA strategy is forced to deny admission of some incoming VMs due to insufficient available resources. This continues for 10 simulated days at which point the experiment terminates. Data from the first 2 days of simulation are discarded to allow the simulation to stabilize before recording results.

3.5.4 Strategy Switching Tuning Parameters

Each DSS strategy has some tuning parameters that must be configured to provide the best possible results, as described in Section 3.4. The first of these is the frequency with which the strategy switching algorithm is run. We evaluated the meta-strategies over multiple frequency values and found 1 hour to be an appropriate frequency for evaluating a strategy switch. Each DSS strategy looks at a set of data centre metrics sampled by a monitor over a certain window size: SP-DSS and Goal-DSS sample every 5 minutes and use a window size of 6 samples; Util-DSS samples every 20 minutes and uses a window size of 6. Util-DSS uses a longer monitoring frequency and window size in order to ignore minor fluctuations in data centre utilization and focus on longer term trends. This helps identify periods of real change in overall utilization, and avoid thrashing between strategies. For the remaining DSS tuning parameters, each combination of values was evaluated over a set of 5 randomly generated workload patterns, and the values that resulted in the best *score* were chosen. Table 3.1 contains the values of the best performing of all parameters defined in Section 3.4.

3.5.5 Results

The results of the experiments are presented in Table 3.2. Each management strategy was evaluated with the same set of 100 randomly generated workload patterns. Each experiment was repeated only once per workload pattern, as the simulation is deterministic. Results were averaged across all workload patterns. We report the following metrics: Average Active Host Utilization (*Host Util*) is the average CPU utilization of powered on hosts; # of Migrations (*# of Migs*) is the number of VM migrations triggered by the management strategies; Power consumed (Ψ) is the total power consumed by all hosts in kWh; Power Efficiency is ψ_{dc} over the entire simulation; SLA Violation is S^v over the entire simulation; and # of Strategy Switches

Strategy	Param.	Value
SP-DSS	ψ'_{norm}	0.004
SP-DSS	ψ'_{high}	0.006
SP-DSS	S^v_{norm}	1.15
SP-DSS	S^v_{high}	1.3
Goal-DSS	S^v_{worst}	0.01
Goal-DSS	ψ_Δ	0.83
Util-DSS	m_S	0.00255
Util-DSS	m_ψ	0.00255

Table 3.1: DSS Tuning Parameters

	SLA	Power	Hybrid	SP-DSS	Goal-DSS	Util-DSS
Host Util	75%	88%	81%	80%	81%	82%
# of Migs	15818	24378	14643	18608	19448	19580
Ψ (kWh)	5488	4384	5049	4840	4821	4778
ψ_{dc}	60.6	75.2	65.9	69.7	69.0	69.8
S^v	0.033%	0.474%	0.092%	0.198%	0.222%	0.220%
Switches	N/A	N/A	N/A	20	56	30
S^v_{norm}	0.0	1.0	0.135	0.360	0.430	0.425
ψ_{norm}	1.0	0.0	0.636	0.452	0.425	0.373
Score	1.0	1.0	0.651	0.588	0.607	0.576

Table 3.2: Strategy Results

(*Switches*) is the number of times that the active strategy was changed. We also report the normalized SLA and power values for each strategy, as well as the *score*. Figure 3.1 presents a graphical representation of the scores. The benchmark strategies (*SLA* and *Power*) both achieve a score of 1, by the definition of the score in Section 3.5.1. The angle of the line from the origin to each point gives an indication of how fairly the strategy behaved towards each goal, with a 45 degree angle representing a perfect balance between SLA and power.

Analysis of Variance was performed on the score results, as well as paired t-tests for each pair of management strategies. The resulting scores for each management strategy were found to be significantly different from each other.

3.5.6 Discussion

All three DSS meta-strategies, as well as Hybrid, achieved better scores than the single-goal *SLA* and *Power* strategies. Util-DSS achieved the lowest score, followed by SP-DSS, then Goal-DSS, and finally *Hybrid*. The meta-strategies improved the score by about 40% when

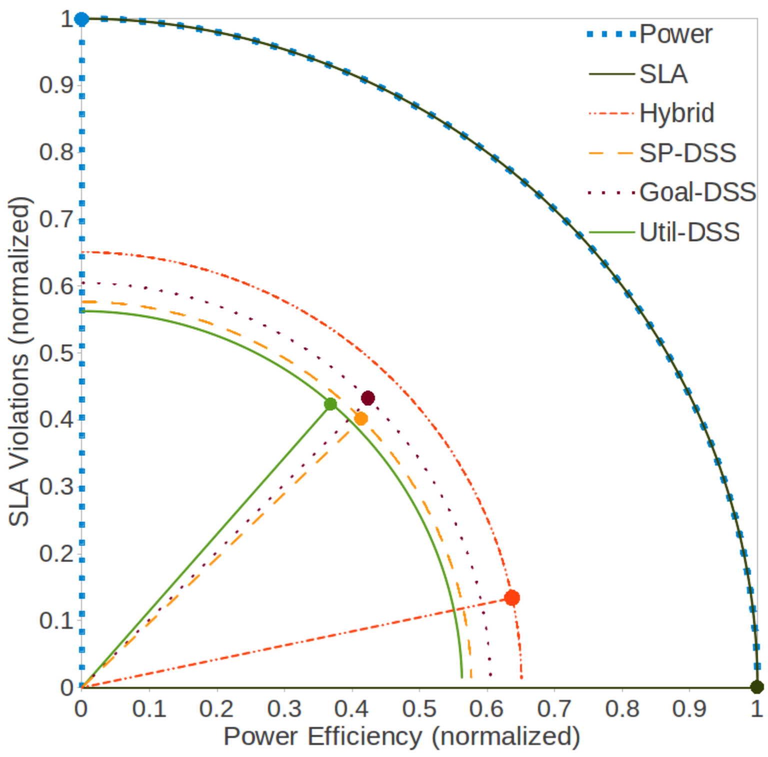

Figure 3.1: Strategy Scores

compared to *Power* and *SLA*, and by about 7-12% when compared to *Hybrid*. Util-DSS and SP-DSS each slightly favoured one of the goals, with Util-DSS favouring power and SP-DSS favouring SLA. Goal-DSS behaved fairly towards both goals. *Hybrid*, on the other hand, was considerably more skewed towards SLA than power, potentially limiting its usefulness in a practical application. The improved overall performance, as well as the balanced treatment of each goal, may therefore favour the selection of DSS over *Hybrid*. Among the meta-strategies, Util-DSS showed to be the most effective, though Goal-DSS was the most balanced.

All meta-strategies triggered 31 to 33% more migrations than the *Hybrid* strategy. While migration overhead was taken into consideration and reflected in the SLA violation and host utilization metrics, further work investigating the effect of migrations on networking should be conducted to determine if this migration count is acceptable. The increase in migration count from Hybrid to DSS is likely a side-effect of switching between strategies with different *stress* ($\overline{\Omega}^{\tau}$) thresholds. The *Power* strategy efficiently pushes the utilization of a large number of hosts to a high value, just below its $\overline{\Omega}^{\tau}$ threshold. A switch to the *SLA* strategy at this point causes a large number of hosts to be considered stressed, as its $\overline{\Omega}^{\tau}$ threshold is below the current utilization achieved by the *Power* strategy. Thus, a spike in migrations is triggered. This also causes a spike in SLA violations due to migration overhead. It may be possible to introduce a mechanism to mitigate this effect and thus lower the DSS meta-strategy migration count. Such a mechanism may also result in an overall better *score* for the meta-strategies.

SP-DSS switched strategies the least number of times, followed by Util-DSS and Goal-DSS. This may be an indication that Util-DSS and Goal-DSS performed some strategy switches that did not contribute to improving performance towards the intended goals (possibly exhibiting a thrashing behaviour), and should be investigated.

3.6 Conclusions and Future Work

The development of data centre management strategies that can simultaneously pursue opposing goals, such as maximizing power efficiency and minimizing SLA violations, is a difficult task. In this work, we proposed dynamically switching between two strategies, each designed to achieve a single goal, to better adapt to changing data centre conditions. We developed three *meta-strategies* to perform dynamic strategy switching, and evaluated them through simulation. The meta-strategies improve overall performance by about 40% when compared to either of the single-goal strategies, and by 7-12% when compared to a hybrid strategy designed to pursue both goals simultaneously.

There are several directions for future work. Regarding DSS, the meta-strategy behaviour when switching between strategies could be improved, so as to avoid spikes in migrations. DSS could also be applied separately to subsets of hosts, such as individual racks or clusters. Finally, threshold values and tuning parameters could be learned rather than fixed.

Networking overhead was not considered in this work. In the future, we intend to develop networking metrics to be used in our evaluations of management policies and strategies. Another topic of interest is the inclusion of constraints and affinity rules to help in determining the placement of VMs on hosts, as discussed by Gulati et al. [8].

Currently, our work relies only on CPU measurements to determine the level of load of a host or VM, with other resources used only as a constraint on placement. In the future, load calculation should take into consideration memory and bandwidth, in addition to CPU (e.g.,[2]).

Future work could incorporate forecasting of VM resource demands, such as done by Bobroff et al. [4]. This would have an effect in the detection of stress situations, and in VM and host selection for migration.

Finally, this work essentially assumes a central manager for all decision making. Other architectural models could be explored, such as that presented by Zhu et al. [15].

Bibliography

[1] The Internet Traffic Archive. The internet traffic archive. `http://ita.ee.lbl.gov/`, July 2014.

[2] Emmanuel Arzuaga and David R. Kaeli. Quantifying load imbalance on virtualized enterprise servers. In *Perf. Eng. Proceedings, 1st WOSP/SIPEW Int. Conf. on*, 2010.

[3] 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 Computat.: Pract. Exper.*, pages 1–24, 2011.

[4] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *IM Proceedings, 2007 IEEE/IFIP Int. Symp. on*, pages 119–128, 2007.

[5] Standard Performance Evaluation Corporation. Specpower_ssj2008 benchmark. `http://www.spec.org/power\_ssj2008/`, July 2014.

[6] Graham Foster, Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. The Right Tool for the Job: Switching data centre management strategies at runtime. In *Integrated Network Management (IM), 2013 IFIP/IEEE International Symposium on*, May 2013.

[7] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *38th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, June 2008.

[8] A. Gulati, G. Shanmuganathan, A. Holler, C. Waldspurger, M. Ji, and X. Zhu. Vmware distributed resource management: design, implementation, and lessons learned. *VMware Technical Journal*, 1(1), 2012.

[9] Google Inc. Google cluster data. `http://code.google.com/p/googleclusterdata/`, July 2014.

[10] Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. An analysis of first fit heuristics for the virtual machine relocation problem. In *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, October 2012.

[11] Gunjan Khanna, Kirk Beaty, Gautam Kar, and Andrzej Kochut. Application performance management in virtualized server environments. In *NOMS Proceedings, 2006 IEEE/IFIP*, 2006.

[12] Michael Tighe, Gastón Keller, Michael Bauer, and Hanan Lutfiyya. DCSim: A data centre simulation tool for evaluating dynamic virtualized resource management. In *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, October 2012.

[13] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX Int. Conf. on Middleware*, 2008.

[14] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI Proceedings, 4th Symp. on*, pages 229–242, Cambridge, MA, USA, April 2007.

[15] Xiaoyun Zhu, Don Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach, Rob Gardner, Tom Christian, and Lucy Cherkasova. 1000 islands: Integrated capacity and workload management for the next generation data center. In *Proceedings of the 2008 International Conference on Autonomic Computing (ICAC'08)*, pages 172–181, Chicago, IL, USA, June 2008.

Chapter 4

A Distributed Approach to Dynamic VM Management

4.1 Introduction

Most work thus far on dynamic management employs a centralized architecture, computing the VM placement for the entire data centre within a single algorithm [2, 8, 10]. In a centralized approach, a single manager must collect regular monitoring information from a large set of hosts in order to maintain global knowledge of the data centre state. Furthermore, it must then compute all management decisions. Given the large scale and highly dynamic nature of the dynamic VM management problem and its intended environment, a centralized solution is unlikely to scale to meet realistic demands [4]. A central manager also presents a single point of failure, and in the event of a network partition, could leave a set of hosts without management.

In this situation, a distributed approach may be better suited to solve the problem. We propose a distributed adaptation of a centralized method, using a first-fit heuristic algorithm [5, 8]. The goals of this approach are to achieve similar performance compared to the centralized approach in terms of SLA and power consumption, while spreading management computation across all hosts, and reducing bandwidth usage for management. We evaluate our approach using the DCSim [11] simulation tool.

The remainder of this chapter is organized as follows: Section 4.2 presents an overview of related work. Section 4.3 introduces Dynamic VM Management, and presents a centralized algorithm which we use as the starting point for our work. Section 4.4 presents our Distributed Dynamic VM Management algorithm. The algorithm is evaluated in Section 4.5, and we con-

This chapter is based on work published in [12] clude and discuss future work in Section 4.6.

4.2 Related Work

The majority of work in the area focuses on a centralized approach to dynamic VM management, especially work targeted specifically at a data centre providing an IaaS cloud. A single, central manager makes all decisions using global knowledge of the state of every component of the data centre [2, 8, 6]. Evaluation in this work frequently assumes the presence of an "oracle", with perfect knowledge of data centre state. Our proposed distributed approach, as well as the centralized approaches which we implement and compare against, eliminate this assumption, and eliminate the need for a single entity to have full knowledge of the data centre state.

Several distributed approaches to dynamic resource management in the cloud have been recently proposed. Each of the approaches considers a host to be an autonomous entity, capable of making management decisions on its own. Yazir et al. [15] propose distributing migration decisions to each host. If a host detects an over-utilization or under-utilization situation, it selects a VM to migrate and a target host, and performs the migration. This work assumes, as with several of the centralized approaches, that each host has complete global knowledge of the state of every other host, and uses an "oracle" in their evaluation. As mentioned above, we eliminate this assumption. Furthermore, the work does not consider SLA performance, and requires a performance model of applications running in the cloud. We treat SLA performance, as well as power consumption, as a primary goal, and we do not require a performance model of applications, as this may not be readily available.

Wuhib, Stadler and Spreitzer [13] propose a novel approach to distributed load-balancing in the cloud using a gossip protocol. Hosts periodically communicate with a random neighbour and attempt to balance workloads between them. It does so by adjusting load-balancing settings to adjust the amount of incoming requests being sent to each server, as well as by starting and stopping module (application) instances. The work was extended by Yanggratoke, Wuhib and Stader [14] who not only perform load balancing, but also to attempt to consolidate workload for the purposes of reducing power consumption. The proposed solutions make use of a demand profiler to estimate resource requirements of modules, as well as control over load-balancing and the starting/stopping of module instances. The target environment is a Platform as a Service (PaaS) cloud, although the authors claim that the approach could be adapted to manage an IaaS cloud, provided that similar control over load-balancing and module instances are available. In contrast, our approach directly targets an IaaS environment, and does not require a demand profiler or control over load-balancing and module instances. Rather, we treat

VMs as a black-box, and assume no control over the operation of client applications.

Feller, Morn and Esnault [4] propose a method of adapting centralized algorithms to run in a decentralized manner. The set of hosts is divided into a constantly changing set neighbourhoods, with each neighbourhood being a subset of hosts of a certain size. Neighbourhoods are dynamic, unstructured peer-to-peer networks. Hosts periodically trigger a centralized management algorithm to consolidate VMs within their neighbourhood. Unlike our work, however, they do not consider dynamic VM resource requirements, instead simply allocating the full amount of resources requested by each VM. This leads to resource under-utilization and excess power consumption.

Finally, Quesnel, Lebre and Sudhold [9] present a decentralized VM scheduler which attempts to build small subsets of hosts to react to management events such as overload or underload. Hosts are arranged in a ring topology overlay network. When an over or underload situation is detected, the host starts a partition of hosts and adds its next neighbour to this partition. It then attempts to solve the resource allocation problem using only the hosts in the partition. If it fails, a new host is added and the process is repeated. The work does not consider SLA violations, especially given that a host is not considered overloaded until it reaches 100% utilization. Moreover, their approach is to essentially execute a centralized algorithm on a smaller subset of hosts, which in their evaluation, leads to performance problems and the disabling of consolidation with larger host sets.

Our work differs from the current literature in that we propose a distributed, decentralized approach to dynamic VM management in an IaaS environment. Several works propose distributed approaches to management in other cloud environments, which do not directly translate to IaaS. Other work does not consider dynamic VM resource requirements, minimizing SLA violations or minimizing power consumption. Our approach considers all of these aspects to the problem.

4.3 Dynamic VM Management

As described in Chapter 2, we define *Dynamic Virtual Machine Management* as the dynamic allocation and re-allocation of VMs within a data centre in response to highly variable workloads and VM resource requirements. The primary goals and motivation of this form of management is to consolidate the set of VMs onto as few physical hosts as possible, while still providing the resources each VM requires to perform up to client expectations. These goals can be expressed as:

- Minimize SLA Violation
- Minimize power consumption

We define *SLA Violation* as the percentage of CPU which a VM requires but does not receive due to contention with other VMs, as described in Chapter 2.6. We assume applications to be running interactive, request-response workloads. When a VM does not have enough resources to meet the current rate of incoming requests to the application running within it, then performance degrades and we consider that the VMs SLA has been violated. This metric is further explained as *CPU Underprovisioned* in Chapter 7.

It is also important to minimize the impact of management on the operation of the data centre. Towards this objective, we include secondary goals:

- Minimize the number of migrations
- Minimize management bandwidth usage

The minimization of migrations is important to reduce bandwidth consumed by migrations and the performance impact of migration on VMs. We use this set of goals to evaluate the performance of dynamic VM management methods.

4.3.1 A Centralized Approach

As a representative centralized approach to dynamic VM management, we use the proposed solution in Chapter 2. More specifically, we make use of the *Hybrid* management strategy presented in Chapter 3, which aims to achieve a balance between SLA and power performance, and implements the three primary management operations, *VM Placement*, *VM Relocation*, and *VM Consolidation*. We use this as a base for developing and evaluating the distributed management solution.

Host Monitoring

Each host monitors its own resource utilization, and sends status data to the central manager at a specified time interval. We use 5 minutes, but this value can be tuned as desired. The status data consists of resource utilization for each VM on the host, as well as for the host as a whole.

4.3.2 Periodic versus Reactive

The *VM Relocation* and *VM Consolidation* policies are typically triggered on a regular periodic interval. We refer to this method of triggering relocation and consolidation as *Periodic VM Management*. Varying the length of these intervals can affect the performance of the algorithm. For example, triggering *VM Relocation* less frequently results in more SLA violations but a lower number of migrations, while triggering *VM Consolidation* less frequently results in fewer SLA violations at the expense of power consumption. In this work, we trigger *VM Relocation* every 10 minutes, and *VM Consolidation* every hour [5].

The *VM Relocation* operation can also be triggered in a reactive fashion rather than periodic. We implement *Reactive VM Management* by checking a host for stress each time host state messages are received, and by triggering *VM Relocation* immediately upon stress detection. In this way, we can both respond more quickly to stress situations, and also avoid running *VM Relocation* when no hosts are stressed.

4.4 Distributed VM Management

We now present a distributed adaptation of the centralized dynamic VM management approach described in Section 4.3, and Chapter 2. The goal of developing a distributed approach is to eliminate the requirement of a single, central manager, and to reduce the network bandwidth required for management messaging. Furthermore, VM management should be done continuously, rather than on scheduled intervals, to spread migration overhead over time rather than trigger large bursts of migrations. Decision making is moved into individual hosts, which communicate with each other asynchronously with small messages. Management operations are initiated with a *broadcast* message, and each host makes a decision as to whether or not it should participate in the action. The action is then completed with only the set of participating hosts.

4.4.1 Architecture

Each host within the system runs an Autonomic Manager, which handles all of the VM management operations for the host. We will simply use the term *host* to refer to both the physical host and its autonomic manager. It performs monitoring, checks for stress or under-utilization situations, and triggers *VM Relocation* or *VM Consolidation* operations as required. New VMs can be placed within the data centre by triggering the *VM Placement* operation on any (powered on) host. Each host is either in the `active` state, in which it is actively hosting VMs and participating in management operations, or it is in the `inactive` state, in which it is in a power-saving mode such as `suspended` or `off`.

4.4.2 Management Operations

The distributed VM management system consists of the following operations:

Monitoring

Hosts monitor their resource utilization on a periodic interval, every 5 minutes. Unlike the centralized algorithm, however, they do not send this data. Rather, the host manager itself performs a check for stress or under-utilization based on the threshold values $\overline{\Omega}^{\tau}$ and $\underline{\Omega}^{\tau}$, introduced in Chapter 2. The average CPU utilization over the last 5 monitoring intervals is compared against the threshold values, to avoid thrashing. If the host is found to be *stressed*, then the monitoring algorithm will trigger the *VM Relocation* operation to relocate one of its VMs to a non-stressed host. If it is *underutilized*, it will trigger the *VM Consolidation* operation to attempt to migrate its VMs to other hosts and shut down. Note that this is only performed if the host is *not* currently involved in any other operations, and is in the `active` state.

VM Relocation

When a host is *stressed*, it must relocate one of its VMs to another host to relieve the situation. We refer to this process as *eviction*. Each host performs VM relocation itself, first by locating potential target hosts for VM migration, and then by selecting a VM to migrate and a specific target. A host can be in one of three states in relation to *VM Relocation*: `normal`, `offering`, and `evicting`. The `normal` state indicates that the host is not involved in an eviction. The `evicting` state indicates that the host is currently attempting to evict a VM. The `offering` state indicates that the host is offering to receive a VM from an `evicting` host. A host must be in the `normal` state to perform any other operation.

The `evicting` host determines the minimum amount of CPU required to be available on another host to evict one of its VMs and relieve the stress situation. CPU is measured in *CPU units*, as described in Chapter 2. It then broadcasts a *Resource Request* message to all hosts, containing this value. To conserve bandwidth when broadcasting *Resource Request* messages, we choose to send the minimum required available CPU *only* as it is a good indication of overall load, and is highly contentious. Each host determines if it has enough CPU remaining to accommodate the minimum request, as well as ensuring that its memory is not full. If it passes these checks, it responds to the `evicting` host with a *Resource Offer*. The *Resource Offer* message contains the total amount of resource available, for *all* resources (not just CPU), which may be higher than the original request. The `evicting` host waits a specified time for responses, and then selects a VM and a target host using a first-fit heuristic algorithm. The algorithm is similar to Algorithm 1, except that each *stressed* host performs the algorithm itself, and the target list contains only those `offering` hosts who responded to the *Resource Request*. That is, *sources* (line 2) contains only the `evicting` host, and the set of hosts *H* classified in line 1, which are used for target hosts, contains only `offering` hosts. Once a VM and target have been chosen, notification of the decision is sent to all offering hosts (so they may return to the `normal` state), and the migration is performed. The sorting of VMs and targets is done as specified for the *Hybrid* strategy in Chapter 3.3.3.

If no hosts respond to the *Resource Request*, then the `evicting` host must boot an `inactive` host (`suspended` or `off`), if available. If no such host exists, the eviction fails. Inactive hosts are sorted by power efficiency in order to prefer the selection of more power efficient hosts. Once the host has powered on, a VM is selected and migrated. If, once a VM has been evicted, the host remains *stressed*, the process repeats to evict another VM.

In order to reduce thrashing between highly utilized hosts, we implement an *relocation freeze*, preventing a host from offering resources for a specified amount of time after the same host evicts a VM. Similarly, if a host offers resources and is chosen as the target, we again apply a *relocation freeze*, this time preventing it from evicting a VM for a specified time period. This mechanism helps to reduce unnecessary migrations, and tuning the *relocation freeze* time parameter enables trading a lower migration count for increased SLA violations. This trade-off is explored in Section 4.5.4.

VM Consolidation

When a host is *underutilized*, it is desirable to migrate its VMs to other hosts and shut it down. However, there may be other hosts that are potentially more beneficial to shut down. For example, a greater reduction in power consumption is achieved by shutting down two hosts running at 10% utilization rather than one at 20% utilization. Preference should also be given to hosts with poorer power efficiency. Furthermore, selecting the host with the lowest utilization increases the probability that a host will successfully shut down. We therefore introduce a *Shutdown Selection* process to select the most appropriate host for shut down.

When a host detects that it is *underutilized*, and is *not* involved in a *VM Relocation* or *VM Consolidation* operation already, it triggers a *Shutdown Selection* operation. A host involved in *VM Consolidation* can be in one of three states: `coordinating`, `claiming`, or `shutting down`. The first host to detect *underutilization* and begin the process is the *coordinator*, and changes to the `coordinating` state. It broadcasts a *Shutdown Selection* message to all hosts, containing its current CPU utilization. When a host receives this message, it checks to see if its current CPU utilization is less or if its power efficiency is worse than the `coordinating` host. If so, it responds with a *Shutdown Claim* event, containing its own CPU utilization, and changes to the `claiming` state. The `coordinating` host waits a specified time for responses, and then selects a host first by power efficiency, and then by CPU utilization (favouring lower power efficiency and lower CPU utilization). This host is selected as the *winner*, and all hosts are notified of the outcome.

The *winner* host changes its state to `shutting down`, indicating that it is attempting to shut down. It then attempts to find migration targets for *all* of its hosted VMs by sending a *Resource Request* message and collecting *Resource Offers*, as in the *eviction* process. Once the offers have been received, it attempts to place VMs using an algorithm similar to that of the centralized version (Algorithm 2). The differences are as follows: *sources* (line 2) set contains only the *winner* host, which is running the algorithm; the host set (H) (line 1) contains only hosts responding to the *Resource Request*; line 2 is not performed; migrations (line 13) are recorded but not triggered immediately. The output of the algorithm is a set of migrations to perform. If this set contains migrations for *every* VM on the host, then the host performs the migrations and shuts down. Otherwise, the shut down fails and is cancelled, as migrating hosts without shutting down will only serve to increase the likelihood of target hosts becoming stressed without gaining any reduction in power consumption.

In order to control the frequency of host shut downs, we add a *shutdown freeze* time during which no host can attempt shut down after a Consolidation operation has taken place. This *shutdown freeze* is introduced even if the consolidation fails, as if the host with the lowest utilization failed to shut down then it is highly unlikely that another host will succeed. Controlling the *shutdown freeze* time has the effect of trading power consumption for SLA performance, which is explored in Section 4.5.4. In order to add some intelligence to the *shutdown freeze* duration, we allow the shutting down host to determine its length as follows: once a complete set of migrations has been found, the *winner* host calculates the amount of remaining resources in the *offering* hosts, after the migrations complete. If there is enough resource remaining to fit at least the total resources in use on the *winner* host, then it indicates that a short shut down *shutdown freeze* can be used, since a subsequent shut down has a good chance of success.

VM Placement

The VM Placement operation is triggered whenever a new VM arrives at the data centre and must be placed on a host. It is performed in the same manner as *VM Relocation*, except that the result is a new VM instantiation rather than a migration. Any host can perform the *VM Placement* operation; no central placement controller is required.

Host Power State Knowledge

Each host maintains a list of other hosts that are known to be `inactive` (`off` or `suspended`), for use in *VM Relocation* and *VM Placement*. When a host shuts down, it broadcasts a *Host Shutting Down* message indicating that it is doing so, and each host adds it to their individual list of `inactive` hosts. Similarly, when a host boots up, it sends a *Host Booting Up* message, instructing each host to remove it from their `inactive` list. When a host is `inactive`, however, it does not receive messages. As such, when it boots up, it will have an outdated copy of the `inactive` list. To overcome this issue, when a host instructs another to boot up, it also forwards its copy of the current `inactive` list to the newly started host.

4.4.3 Management Bandwidth

One of our goals in developing a distributed dynamic VM management system was to reduce the amount of bandwidth used for VM management. The centralized system transmits host state data from each host on a regular interval, regardless of whether or not that data is required at the time. Each message contains the resource utilization of each hosted VM, which consists of the resource vector $(cpu, memory, bandwidth, storage)$. During our experiments (see Section 4.5), we found that, on average, each message contains data on 9.2 VMs. We assume each individual resource value to be 4 bytes in size (the size of an *int* or *float* value in Java), combining for a total of 16 bytes. The distributed system attempts to send data only when required. Many messages contain no payload data, such as the *Host Booting Up* message. Others contain only a single value, such as the *Resource Request* and *Shutdown Selection* messages. Messages that contain full resource data contain only a *single* vector, thus resulting in reduced message sizes. In the case of *broadcast* messages, we total each time the message is received, rather than simply considering it as equivalent to a single message.

4.5 Experiments

In this section, we will evaluate our distributed approach through conducting a set of experiments using a simulation tool, DCSim [11]. We first evaluate a set of possible tuning parameter values for the distributed VM management system. We then compare the distributed approach with two forms of centralized management, namely, *periodic* and *reactive*.

4.5.1 Metrics

Average Active Host Utilization

The average CPU utilization of all hosts (*Host Util*) that are currently in the *On* state. The higher the value, the more efficiently resources are being used.

Number of Migrations

The number of migrations (*# Migs*) triggered during the simulation. Typically, a lower value is more desirable, as less bandwidth would be used for VM migrations.

Power Consumption

Power consumption (Ψ) is calculated for each host, and the total kilowatt-hours consumed during the simulation are reported. Power consumption is calculated using results from the SPECPower benchmark [3], and is based on CPU utilization.

SLA Violation

An SLA violation, denoted S^v, occurs when resources required by a VM are not available to it, as this situation leads to a degradation in performance. The percentage of required CPU not available is the SLA violation, as described in Chapter 2.6.

SLA Violation Duration

SLA Violation Duration (S^v Duration) is total amount of time that all VMs spent in a state of SLA violation. For example, if two VMs were each in SLA violation for 5 minutes, the *SLA Violation Duration* would be 10 minutes.

Management Bandwidth Usage

The total amount of bandwidth used for VM management messages. Message sizes are calculated as defined in Section 4.4.3.

4.5.2 Experimental Setup

Our simulated data centre consists of 200 host machines, with two different types of host: *small* and *large*. The *small* host is based on the HP ProLiant DL380G5, with 2 dual-core 3GHz CPUs and 8 GB of memory. The *large* host is based on the HP ProLiant DL160G5, with 2 quad-core 2.5GHz CPUs and 16GB of memory. Cores in the *large* host have 2500 CPU shares, and cores in the *small* host have 3000 CPU shares. The power efficiency of the *large* host (85.84 cpu/watt) is roughly double that of the *small* host (46.51 cpu/watt). An equal number of each host type is created within the data centre.

We define three different VM sizes: *small* requires 1 virtual core with a minimum of 1500 CPU shares and 512MB of memory, *medium* requires 1 virtual core with a minimum of 2500 CPU shares and 512MB of memory, and *large* requires 2 virtual cores with a minimum of 2500 CPU shares each and 1GB of memory. Note that these are requested, maximum resource requirements. We initially place VMs based on these values, and attempt to guarantee them when required, but once VMs are running they are placed and allocated based on their resource *usage*, not request. We create an equal number of all three VM types.

Hosts are modelled to use a work-conserving CPU scheduler, as available in major virtualization technologies. As such, CPU shares that are not used by one VM can be used by another. No maximum cap on CPU is set for VMs. In the event that the CPU is at maximum capacity and VMs must compete for resources, VMs are assigned CPU shares in a fair-share manner. Memory is statically allocated and is *not* overcommitted.

4.5.3 Workload

We model a set of interactive applications running within the data centre, with each VM running a single application. VMs arrive and depart the system throughout the experiment, and exhibit dynamic resource requirements driven by real workload traces. Each VM uses a trace built from one of 5 sources: the *ClarkNet*, *EPA*, and *SDSC* traces [1], and two different job types from the *Google Cluster Data* trace [7]. We compute a normalized rate of requests, in 100 second intervals, for each trace. These rates are used to define the current workload of each VM, with the CPU resource requirements of the VM calculated as a linear function of the current rate. To ensure that VMs do not exhibit identical behaviour, each VM starts its trace at a randomly selected offset time.

A set of 600 VMs is created within the first 40 hours and remain running throughout the entire experiment, to maintain a minimum level of load. After 2 simulated days, new VMs begin to arrive at a changing rate, and terminate after about 1 day. The total number of VMs in the system varies daily, using randomly chosen values uniformly distributed between 600 and 1600. This continues until the conclusion of the experiment. A specific randomly generated instance of VM arrivals, departures, and trace offset times is referred to as a *workload pattern*, and is entirely repeatable through specifying the random seed used to generate it. We use a set of 10 different *workload patterns* to evaluate our work, and all presented results are averaged across experiments using these 10 *workload patterns*. We discard data from the first 2 days of simulation to allow the system to stabilize before recording results.

4.5.4 Tuning Parameters

The operation and performance of the distributed VM management system can be fine-tuned through the manipulation of the *relocation freeze* and *shutdown freeze* durations discussed in Section 4.4. During *VM Relocation*, once a host has evicted a VM, it must wait a *relocation freeze* duration before it can make a resource offer to accept a VM from another host. It may still evict VMs, it simply cannot accept them. Conversely, once a host has made a resource offer and has been selected as the target for a migration or VM placement, it must wait the *relocation freeze* duration before it can evict one of its own VMs. Adjusting this duration allows us to control how aggressively VMs are relocated in response to stress situations. We define two levels of relocation tuning: *Normal*, in which the *relocation freeze* is set to 30 minutes; and *Light*, in which the *relocation freeze* is set to 60 minutes. Note that a longer duration results in a *less* aggressive algorithm.

During *VM Consolidation*, once a *Shutdown Selection* has been performed, hosts must wait the *shutdown freeze* duration before triggering another. If, however, the shutting down host determines that it is likely that there are enough spare resources in the system to shut down another host (see Section 4.4.2), it can indicate that only the *short shutdown freeze* duration should be used. By adjusting these values, we control how aggressively VMs are consolidated and hosts are shut down to conserve power. We define three levels of consolidation tuning: *Heavy*, in which *shutdown freeze* is set to 15 minutes and *short shutdown freeze* to 5 minutes; *Normal*, in which *shutdown freeze* is set to 30 minutes and *short shutdown freeze* to 5 minutes; and *Light*, in which *shutdown freeze* is set to 30 minutes and *short shutdown freeze* is also set to 30 minutes.

We evaluated all six combinations of the relocation and consolidation levels. Table 4.1

Reloc	Consol	Host Util	# Migs	Ψ	S^v	S^v Duration	Man BW
Normal	Heavy	79%	12779	5113kWh	0.069%	17.05 days	12.57GB
Normal	Normal	77%	10818	5252kWh	0.056%	14.17 days	12.05GB
Normal	Light	76%	9091	5352kWh	0.043%	11.36 days	11.01GB
Light	Heavy	79%	11524	5072kWh	0.113%	25.87 days	10.77GB
Light	Normal	78%	9980	5123kWh	0.092%	21.72 days	10.17GB
Light	Light	77%	8367	5265kWh	0.069%	16.99 days	9.9GB

Table 4.1: Distributed Tuning Parameters

presents the average results over 10 simulations using 10 different *workload patterns*.

We can see that moving from *Normal* to *Light* relocation results in a reduction of the number of migrations and power consumption, at the expense of increased SLA violation. This is to be expected, as less frequent *VM Relocation* means triggering migrations to relieve stress situations less often, and booting new hosts to accommodate evicted VMs less often as well. The effect of changing from *Heavy*, to *Normal*, to *Light* consolidation also shows expected results. As consolidation is performed less aggressively, SLA violations and migrations decrease at the expense of power consumption.

4.5.5 Distributed versus Centralized

We now compare the performance of distributed VM management with that of the centralized approach. For the sake of comparison, we have chosen two configurations of the distributed system, although any results from Table 4.1 are directly comparable to the centralized algorithm results. We compare the centralized algorithms against a distributed configuration with very good SLA performance (*Dist. SLA*) and one with very good power performance (*Dist. Power*). The *Distributed SLA* algorithm uses *Normal* relocation and *Light* consolidation, while the *Distributed Power* algorithm uses *Light* relocation and *Heavy* consolidation.

We compare against the *Periodic* and *Reactive* versions of the centralized system, as defined in Section 4.3.2. Table 4.2 presents the average results over 10 simulations using 10 different *workload patterns*.

The *Reactive* centralized management system provides improved SLA performance when compared to *Periodic*, at the expense of power and migrations. This is due to the fact that it responds immediately to stress situations, triggering *VM Relocation* as soon as one is detected. *Distributed Power* achieves similar power consumption and SLA violation performance to *Periodic*, at the cost of a slight increase in migrations. It is expected that there should be a trade-off involved in using a distributed version, as it uses only partial knowledge to perform

	Host Util	# Migs	Ψ	S^v	S^v Duration	Man BW
Dist. Power	79%	11524	5072kWh	0.113%	25.87 days	10.77GB
Dist. SLA	76%	9091	5352kWh	0.043%	11.36 days	11.01GB
Periodic	80%	10261	5056kWh	0.109%	26.58 days	38.04GB
Reactive	79%	12508	5121kWh	0.059%	15.42 days	38.08GB

Table 4.2: Distributed vs Centralized Results

management operations. *Distributed SLA*, on the other hand, achieves reduced SLA violations when compared to all other systems, as well as a greatly reduced number of migrations. It does so at the expense of an increase in power consumption though. Both distributed versions offer about a 71% reduction in management bandwidth usage, falling in line with our goals for distributed VM management.

4.6 Conclusions and Future Work

Most existing work makes use of a centralized architecture, where a central manager handles management for the entire data centre. However, given the highly dynamic resource needs of VMs and the scale at which management must take place, a centralized approach may not be a realistic solution. Furthermore, centralized management represents a potential single point of failure for the system.

We present a distributed VM management system, adapted from an existing centralized system. The goals of the approach were to replicate the SLA and power performance of the centralized system, while eliminating the central manager and reducing the bandwidth consumed by management. Through evaluation with a simulation tool, we have shown that the distributed approach can in fact achieve these goals, although there is always a trade-off to be made between number of migrations, SLA violation and power. Furthermore, the distributed system can be tuned to favour SLA or power, to suit the requirements of the data centre operator. In all cases, the distributed system provided a reduction in management bandwidth usage.

There are a number of possible directions for future work. In terms of the distributed system, broadcast messaging could be replaced by a different communication method or overlay network to attempt to further reduce management bandwidth usage. Hosts could be split into smaller multicast groups, with a new mechanism introduced to communicate between them only when necessary. Also, we suspect that the effectiveness of the *freeze* values used for tuning the management system may be closely tied to data centre size and overall utilization.

This could provide an opportunity to dynamically and automatically adjust the values to match current data centre conditions, possibly with a machine learning approach.

Our work considers only variable CPU demand, and static memory requirements of VMs. This needs to be expanded to consider VM bandwidth and storage requirements. Using prediction methods to predict future resource demand and take proactive action could also be explored. We could also consider the topology of the data centre, which may drive the design of algorithms as well as impose constraints on VM placement. Finally, the thermal state of the data centre, and cooling costs, could also be considered in VM management and drive the selection of hosts to power on or off.

Bibliography

[1] The Internet Traffic Archive. The internet traffic archive. `http://ita.ee.lbl.gov/`, July 2014.

[2] 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), 2012.

[3] Standard Performance Evaluation Corporation. Specpower_ssj2008 benchmark. `http://www.spec.org/power\_ssj2008/`, July 2014.

[4] Eugen Feller, Christine Morin, and Armel Esnault. A case for fully decentralized dynamic vm consolidation in clouds. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 26–33. IEEE, 2012.

[5] Graham Foster, Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. The Right Tool for the Job: Switching data centre management strategies at runtime. In *Integrated Network Management (IM), 2013 IFIP/IEEE International Symposium on*, May 2013.

[6] A. Gulati, G. Shanmuganathan, A. Holler, C. Waldspurger, M. Ji, and X. Zhu. Vmware distributed resource management: design, implementation, and lessons learned. *VMware Technical Journal*, 1(1), 2012.

[7] Google Inc. Google cluster data. `http://code.google.com/p/googleclusterdata/`, July 2014.

[8] Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. An analysis of first fit heuristics for the virtual machine relocation problem. In *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, October 2012.

[9] Flavien Quesnel, Adrien Lèbre, and Mario Südholt. Cooperative and reactive scheduling in large-scale virtualized platforms with dvms. *Concurrency and Computation: Practice and Experience*, 2012.

[10] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *J. Parallel Distrib. Comput.*, 70(9):962–974, September 2010.

[11] Michael Tighe, Gastón Keller, Michael Bauer, and Hanan Lutfiyya. DCSim: A data centre simulation tool for evaluating dynamic virtualized resource management. In *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, October 2012.

[12] Michael Tighe, Gastón Keller, Hanan Lutfiyya, and Michael Bauer. A Distributed Approach to Dynamic VM Management. In *Network and Service Management (CNSM), 2013 9th International Conference on*. IEEE, 2013.

[13] Fetahi Wuhib, Rolf Stadler, and Mike Spreitzer. Gossip-based resource management for cloud environments. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 1–8. IEEE, 2010.

[14] Rerngvit Yanggratoke, Fetahi Wuhib, and Rolf Stadler. Gossip-based resource allocation for green computing in large clouds. In *Network and Service Management (CNSM), 2011 7th International Conference on*, pages 1–9. IEEE, 2011.

[15] 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 *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 91–98. Ieee, 2010.

Chapter 5

Integrating Cloud Application Autoscaling with Dynamic VM Management

5.1 Introduction

One of the key advantages to migrating an application into the cloud is the ability to dynamically scale the application to meet current demand. This essentially gives the application owner an elastic infrastructure, scaling up to meet increasing demand and scaling down when demand is low and resources are underutilized. For example, typical web applications experience highly variable loads, with average demands coming in far below peak. If enough resources are provisioned to meet the peak load (and likely more, since it may not be possible to accurately predict peak resource requirements), then a large amount of the provisioned resources will be significantly underutilized. This results in increased operating costs, as the client is charged based on resources provisioned, not by actual utilization. Conversely, if the application is provisioned for average load, then performance can suffer during peak periods, causing potential loss of customers and business. Therefore, in order to both reduce costs while still meeting performance requirements, applications must dynamically add and remove resources, in the form of virtual machines, to match ever-changing demands from users.

As detailed in Chapter 2, the data centre operator (IaaS cloud provider) faces a set of challenges in order to make the best use of their infrastructure. To accomplish the two goals of minimizing SLA violations and minimizing power consumption, the cloud provider performs *Dynamic VM Management*. Dynamic VM Management involves the placement of VMs within the data centre, as well as the adaptation of this placement via VM live migration to meet dynamic VM resource demands.

This chapter is based on work published in [15] as well as content from an extended journal version, currently under submission

Many clients do not simply deploy a single VM in the cloud, but rather a set of interacting VMs, each of which is a component of a larger application providing a service to their users. For the purposes of this work, we define an application as a set of interacting VMs which provide a user-facing service. The cloud provider, upon request from a client, places client VMs within the physical data centre. As was previously discussed, applications should dynamically scale to match current demands. This process, known as *autoscaling*, can be accomplished by adding or removing VMs to and from the application. For example, a web application may contain a tier of web servers, with incoming requests load balanced between them. When load increases, an additional instance of the server (VM) could be added to the tier to serve the additional requests. Conversely, when load decreases and VMs are underutilized, a server instance (VM) can be removed. Commercial implementations of autoscaling exist, such as Amazon Web Services' *Auto Scale* [1], which allow autoscaling rules to be defined by the client.

While there is significant existing work examining autoscaling and dynamic VM management, most work looks at one or the other, in isolation. The goal of this work is to develop a unified approach to support both operations, and to leverage basic control over autoscaling to assist in dynamic VM management and fulfil the goals of both the cloud provider and the cloud client. The goals of the cloud client are to:

- achieve service level agreements (SLA);
- provision the smallest amount of resources possible.

Similarily, the goals of the cloud provider are to:

- reduce infrastructure usage to conserve power;
- ensure client VMs have resources they require;
- avoid affecting application autoscaling decisions (which could unfairly increase client costs).

Considering this set of goals, we present a new approach to handling application autoscaling and dynamic VM management. Our approach is designed in such as way as to allow autoscaling decisions to remain under the control of the client, potentially implementing alternative rules or autoscaling algorithms customized for their specific application, without modification to the primary algorithm. Autoscaling decisions are made considering only the best interests of the application, with the execution of these decisions being handled in a manner that helps achieve dynamic VM management goals.

The remainder of this chapter is organized as follows: Related work is introduced in Section 5.2. Section 5.3 defines the problem area. We present a method for automatically scaling applications in Section 5.4. Section 5.5 presents a new algorithm integrating autoscaling and dynamic VM allocation, we evaluate our work in Section 5.6, and conclude in Section 5.7.

5.2 Related Work

Some of the literature focuses on scaling resource allocations of a single VM to meet its current demands, but do not scale applications out to multiple VMs [14]. Ghanbari et al. [9] examine alternative autoscaling approaches in the literature, classifying them into two categories: control theory approaches, and rule-based heuristics. They propose a control theory approach to scaling [8] an application running in a public IaaS cloud, which takes into consideration costs, constraints, and the characteristics of the IaaS environment. Their work is from the perspective of the client running applications within the cloud, with autoscaling logic running outside of the cloud. Our approach runs autoscaling algorithms within the cloud infrastructure itself. Chieu et al. [3] propose that the automatic scaling of application resources is a critical capability for clients deploying applications in the cloud. They present an autoscaling algorithm that falls into the rule-based category, scaling applications up or down based on the number of active user sessions in deployed web applications. Ferretti et al. [6] autoscale applications based on SLA achievement. They specify a threshold on response time as their SLA objective, and attempt to proactively scale the application based on continuous monitoring. Finally, Amazon Web Services provides an autoscale feature (AWS Auto Scale) [1], allowing clients to specify conditions under which additional VM instances should be added or removed. These conditions are rules based on monitored metrics, again falling into the rule-based heuristic category.

There are a small number of works that examine something similar to combining autoscaling with dynamic VM management. Wuhib, Stadler and Spreitzer [18] propose an approach to distributed load-balancing in the cloud using a *gossip* protocol. Hosts balance workloads between them by adjusting load-balancing settings as well as by starting and stopping module (application) instances. The work was extended [19] to also consolidate workload for the purposes of reducing power consumption. The proposed solutions make use of a demand profiler to estimate resource requirements of modules and targets a Platform as a Service (PaaS) cloud, although the authors claim that the approach could be adapted to manage an IaaS cloud. Our approach does not require a demand profiler. Furthermore, their application scaling is performed with the goal of load balancing, rather than reducing the costs of clients running applications in the cloud. We focus on letting applications decide how they want to scale, and providing them with the resources they require. Jung et al. [10] perform application scaling and dynamic VM management using a layered queueing model of applications to determine the utility of configurations. They also require offline experimentation to determine the effects of configuration adaptation methods, which are later used to estimate effects online. Our approach does not require detailed models nor prior offline processing. Petrucci et al. [13] use a mixed integer programming model to optimize application placements. Unlike our work, they do not allow the client to determine auto-scaling conditions and the scalability of their approach is not fully evaluated.

Li et al. present CloudOpt [12], a system for deploying applications consisting of multiple components in a cloud environment. CloudOpt overcomes scalability issues with mixed integer programming by combining it with a bin-packing heuristic. It makes use of detailed performance models of applications, using a layered queueing model, and optimizes for a number of goals, including Quality of Service, memory, energy, and license costs. The system performs a one-time placement, although the authors state that dynamic management could be handled by repeated execution of the algorithm. Our work differs from this in that we do not assume that models of deployed applications are available, nor do we take control over replication decisions. Our focus is more on allowing application owners to decide what is best for the application, while making use of these decisions to assist dynamic management. We also explicitly consider dynamic management.

We focus on combining autoscaling and dynamic VM management into a single algorithm, capable of scaling applications while also consolidating load for power conservation. Other work considers only one of these goals, targets a different environment, or makes use of detailed application models.

5.3 Problem Definition

In this section, we present the various aspects of the problem we are addressing. We build upon the description of *Dynamic VM Management* presented in Chapter 2.3, adding a description of the model we are using for applications running in the cloud, as well as management operations. Figure 5.1 provides a general overview of the structure of hosts, VMs, and applications, as described throughout the remainder of this section.

5.3.1 Applications

Every VM that within the data centre is running a component of an *application*. In this work, we consider *interactive* applications, such as a multi-tiered web application, in which a number of clients interact with the system, waiting for responses before issuing a new request. There is a single class of requests, entering and exiting the application through a single point.

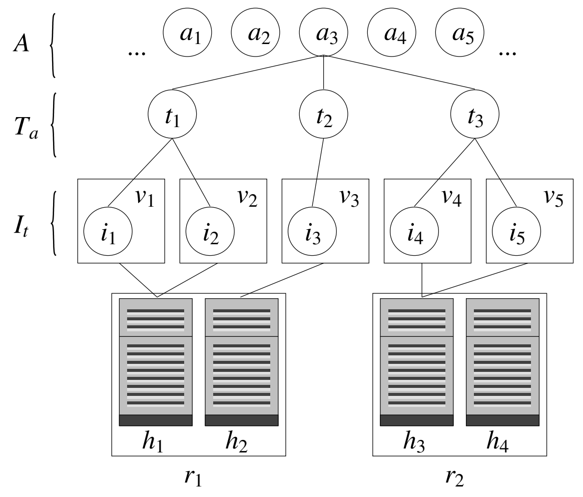

Figure 5.1: Data Centre Model

An application, $a \in A$, contains a set of *tasks*, T_a, where a single task is denoted $t \in T_a$. Tasks represent different components of the application, such as a web server(s), application server(s), or database. Tasks are performed by one or more identical *task instances*, $i \in I_t$, where I_t is the set of task instances in task t. Task instances are the actual deployed software components that perform the task. For example, a web server task of an application may consist of a deployed set of replica web servers, with incoming requests load balanced between them. A VM, v_i, is assigned to run a single task instance i. Furthermore, $\forall i \in I_t \ \exists! \ v_i$. That is, every task instance must be running within a single VM.

For any application, at least one instance of each task in the application must exist. That is, $|I_t| > 0, \quad \forall t \in T_a$. The number of instances in each task can change over time, as instances are added or removed dynamically to match current workload demands. Task instances share the total workload (i.e. requests) of their tasks, and in this work, it is assumed that the workload is distributed equally.

We consider the Service Level Agreement (SLA) for an application to be defined as an upper threshold on application response time. Note that this calculation of SLA is different than that used in previous chapters, and provides a more realistic and useful measure of SLA performance. The response time of an application $a \in A$, at time s, is denoted $\rho_a(s)$. Similarly, its SLA threshold is denoted as ρ_a^τ. If the response time of an application falls below its threshold, then the SLA is said to be *achieved*. Otherwise, SLA is *violated*. The SLA achievement of an application is:

$$\forall s \in \mathbb{R}^+, \quad S_a(s) = \begin{cases} 1 & \text{if } \rho_a(s) \leqslant \rho_a^\tau \\ 0 & \text{otherwise} \end{cases}$$

Conversely, the SLA violation is defined by:

$$S_a^v(s) = 1 - S_a(s)$$

The percentage of time for which SLA is *achieved* over a time interval $[s_i..s_j]$ is defined as:

$$\overline{S_a}(s_i, s_j) = \frac{\sum_{s_i}^{s_j} S_a(s)}{s_j - s_i}$$

Similarly, the percentage of time for which SLA is *violated* over a time interval $[s_i..s_j]$ is defined as:

$$\overline{S_a^v}(s_i, s_j) = 1 - \overline{S_a}(s_i, s_j)$$

For simplification, we refer to the overall SLA achievement of an application over the entire time interval under consideration (e.g. the duration of an experiment) simply as $\overline{S_a}$. Our third goal consists of minimizing SLA violation:

$$\min \sum_{a \in A} \overline{S_a^v}$$

5.3.2 VM Live Migration

In order to modify the placement of VMs within the data centre, a live migration operation is performed. This operation consists of transferring the state of a running VM to another host, and incurs both performance and network overhead. Therefore, an additional goal is to minimize the number of migrations required to perform dynamic VM management. The number of migrations that occur between time $s - 1$ and s is:

$$\alpha^{\Delta}(s) = \frac{\sum_{h \in H,\, v \in V} \alpha_{h,v}(s-1) \oplus \alpha_{h,v}(s)}{2}$$

Note that since every placement change for a VM results in the modification of $\alpha_{h,v}$ for both hosts involved in the move, we divide the sum of the placement changes by two.

Our goal is to minimize the number of migrations that are performed overall, rather than at a specific point in time. For example, if a larger set of migrations at one point results in a placement in which far fewer migrations are required in the future, than it may be desirable to perform these migrations instead of a smaller set which requires additional migrations later on. Therefore, we introduce the following objective:

$$\forall s \in \mathbb{R}^{+}, \quad \min \sum_{s_1}^{s_n} \alpha^{\Delta}(s)$$

5.4 Application Autoscaling

5.4.1 Application Model

Applications are typically deployed with enough spare resources to handle peak traffic and workload demands. As a result, they spend most of their time significantly overprovisioned, resulting in and underutilization of resources and increased costs [6]. By dynamically scaling applications to match changing workload demands, the client can use only the resources it requires, releasing unused resources to save costs. In this work, we examine autoscaling resources through the addition or removal of VMs (task instances) to and from the application. We do not look at scaling resource allocations on hosts to individual VMs. We present a rule-based, heuristic autoscaling algorithm for use in our implementation. It would be possible, however, to make use of a more complex autoscaling algorithm without modification to our final, application-aware dynamic allocation algorithm (Section 5.5).

As described in Section 5.3, we consider an *interactive* application in which a number of clients interact with the system, waiting for responses before issuing a new request. We implement a model of such an application in an open source simulator, DCSim [5, 16], for the purposes of evaluating our approach. Figure 5.2 shows the structure of a basic application that might be deployed in our evaluation, consisting of two tiers. Each tier has a load balancer component which evenly divides incoming requests between all task instances in the tier. Both tiers are able to dynamically add or remove instances (autoscale) to match current workload demands. The *size* of a task refers to the number of task instances it contains.

Within the simulation, we model the application as a closed queueing network, and solve it using Mean Value Analysis (MVA). We make use of some additional information about the applications and tasks. An application has a specified *think time* for clients, which defines the amount of time that clients wait in between receiving a reply from their previous request and sending another. It also contains a *workload* component, which defines the current number of clients using the application. The workload changes at discrete time points in the simulation,

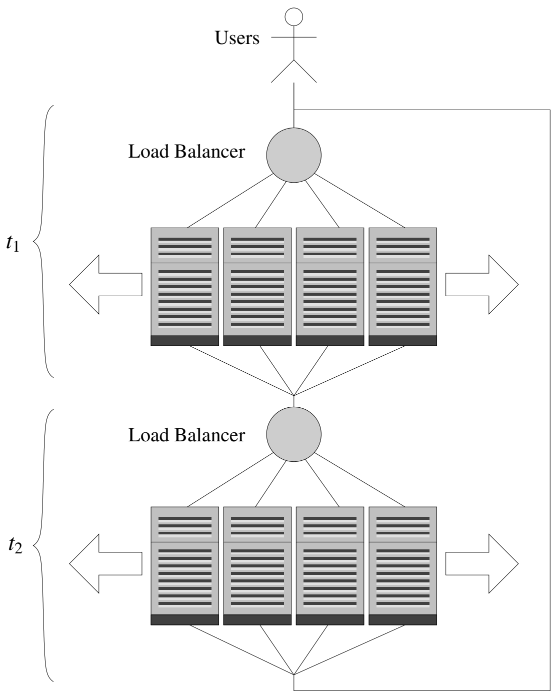

Figure 5.2: Application Example

based on a trace file. The description of a task is expanded to include:

- *default* and *maximum size*, which defines the default and maximum number of task instances;
- *service time*, which defines the time it takes to process a single request;
- *visit ratio*, which defines the average number of times each request must visit the task;
- and *resource size*, which defines the expected amount of CPU and memory allocated to each task instance of the task, as well as the expected CPU speed, in order to achieve the specified service time.

The actual service time for each task instance may be scaled based on a CPU speed factor, and in the case of CPU contention (when the CPU becomes overloaded), an additional delay may be added to it to account for processor queuing.

The performance of a deployed application is evaluated based on its SLA achievement (see Section 5.3). Note that the SLA is based on the response time seen by the application users, and defines an agreement between cloud clients and their users, rather than the cloud provider and client. A lower SLA achievement (higher application response time), indicates that the application is underprovisioned, either due to a failure of the autoscaling algorithm to scale up the application, or by contention with other VMs caused by poor VM placement and dynamic management. By comparing experimental results running only the autoscaling algorithm with those running both autoscaling and dynamic VM management, we can then determine how much SLA violation is attributable to the cloud provider.

5.4.2 Algorithm

We have implemented a basic autoscaling algorithm, which scales applications up and down based on current SLA achievement and CPU utilization. We make use of a rule-based, heuristic algorithm, as it is light-weight and does not require detailed knowledge or models of the applications being managed. Furthermore, with such an algorithm, it would be possible for application owners (cloud clients) to define their own threshold values and rules for autoscaling, customized to their own needs and goals.

Each application has a single manager, which monitors the application and executes the autoscaling algorithm (Algorithm 7) on a fixed interval (e.g. every 5 minutes). Task instances regularly send monitoring data to the manager. The algorithm requires the following as input:

- the application being scaled (a);
- the time (s);
- the window sizes for sliding average response time (W_ρ) and CPU utilization (W_ω);
- a *Response Time Threshold* (ρ^τ) defining the SLA of the application;
- an *SLA Warning* value (S^{warn}), specified as a percentage of the SLA response time threshold;
- and a *CPU Safe* value (ω^{safe}).

We calculate the average response time, ρ_a, over a window W_ρ using the function $\overline{\rho_a(W_\rho)}$ (line 1). Then, ρ_a is compared against the warning threshold (line 2). If it exceeds this value, a *scale up* operation is performed to add additional VMs to the application. The algorithm then iterates through the tasks that comprise the application (lines 4-7), and calculates the change in response time (ρ_a^Δ) since the last execution of the algorithm. We assume a function $\rho_t^{task}(s)$, which returns the response time of a task at a specified time. The algorithm chooses the task with the largest increase in response time, and whose size, $|I_t|$, has not reached its maximum size, $|I_t|^{max}$. A new instance is then added to this task (line 8).

If the response time is below the warning threshold, the algorithm looks for a task on which to perform a *scale down* operation. Scaling down is performed based on CPU utilization, rather than SLA. Function $\overline{\omega(i, window)}$ computes the average CPU utilization of a task instance over a sliding window. The total CPU utilization of all instances in the task ($task_\theta$), averaged over a sliding window of size W_ω (line 12), is computed and divided by task size minus one to estimate the utilization of remaining task instances, should one be removed (line 13). If this value falls below a specified *CPU Safe* value (ω^{safe}), then the task is a candidate for a *scale down* operation. The algorithm chooses the candidate task with the lowest utilization from which to remove an instance (lines 14-15), and removes an arbitrary instance from the task (line 16).

The threshold values for SLA Warning and CPU Safe, as well as the metric to evaluate (e.g. response time, throughput, CPU utilization) could easily be defined by the application owner, in a similar manner as the rules defined in Amazon Web Services Auto Scale [1]. This gives the client control over autoscaling decisions and trade-offs between SLA performance and cost.

Algorithm 7: Autoscaling Algorithm

```
   Data: $a$, $s$, $W_\rho$, $W_\omega$, $\rho^\tau$, $S^{warn}$, $\omega^{safe}$
1  $\rho_a \leftarrow \overline{\rho_a(W_\rho)}$ ; $target \leftarrow$ NIL
2  if $\rho_a > \rho^\tau \times S^{warn}$ then
3      $target_\Delta \leftarrow 0$
4      for $t \in T_a$ do
5          $\rho_a^\Delta \leftarrow \rho_t^{task}(s) - \rho_t^{task}(s-1)$
6          if $\rho_a^\Delta > target_\Delta$ & $|I_t| < |I_t|^{max}$ then
7              $target_\Delta \leftarrow \rho_a^\Delta$ ; $target \leftarrow t$
8      if $target \neq NIL$ then addInstance($target$)
9  else
10     $target_\theta \leftarrow \infty$
11     for $t \in T_a$ do
12         $t_\theta \leftarrow \sum_{i \in I_t} \overline{\omega(i, W_\omega)}$
13         if $|I_t| > 1$ & $\frac{t_\theta}{|I_t| - 1} \leqslant \omega^{safe}$ then
14             if $t_\theta < target_\theta$ then
15                 $target_\theta \leftarrow t_\theta$ ; $target \leftarrow t$
16     if $target \neq NIL$ then removeInstance($target$)
```

5.5 Integrated Application-aware Algorithm

Dynamic VM management requires the use of VM live migration operations in order to adapt the allocation to changing workload conditions. This operation, however, is not free. Rather, it degrades VM performance during migration, and consumes network bandwidth in order to transfer the running state of the VM. We propose a single algorithm, integrating autoscaling (see Section 5.4) with dynamic VM management (see Chapter 2), in order to leverage autoscaling operations to assist dynamic VM management and reduce the number of migrations required. In this approach, each application retains control over deciding to perform *scale up* and *scale down* operations. The choice of which task instances to shut down (in the case of a scale down operation), and where to place new instances (in the case of a scale up operation), however, are performed in such a way as to aid *VM Relocation* and *VM Consolidation* and reduce the need for live migrations.

The integrated, application-aware algorithm combines *autoscaling* operations with *VM Relocation* and *VM Consolidation* into a single algorithm, referred to as the *Dynamic Allocation* operation, which is run on a regular time interval. The *VM Placement* operation is handled separately, triggered by the arrival of new *applications* to the data centre. Algorithm 8 presents a segment of the integrated algorithm, which deals with relieving stressed hosts (i.e. *VM Relocation*). This is the first piece of the algorithm, the remainder of which continues on to perform remaining scaling and consolidation operations. The inputs of the algorithm are the set of hosts in the data centre (H), the set of running applications (A), and a timeout limit on host stress (Θ^{τ}). The algorithm attempts to resolve stress situations through careful execution of autoscaling operations. Once a host has been stressed for the specified number of algorithm executions, however, the algorithm triggers a relocation via migration rather than continuing to wait for autoscaling operations to clear the stress situation.

First, hosts are classified as described in Section 2.4. The number of consecutive times that the host has been *stressed* is then recorded in lines 2-3. In line 4, the *evaluateScaling()* method executes a modified version of the autoscaling algorithm (Algorithm 7) for each application. It varies from Algorithm 7 in that rather than performing the autoscaling operations directly, it returns a list of *tasks* to be scaled up ($T^{\uparrow}$) and scaled down ($T^{\downarrow}$). Note that this algorithm could easily be exchanged for a more complex autoscaling algorithm, or even a different algorithm for each application, as required by the application owner. It is simply required to return the list of tasks to scale. For our current implementation, however, the algorithm presented in Section 5.4 is used.

The algorithm then attempts to relieve the stress situation of each *stressed* host, iterating through them in line 5. Line 7 calculates the amount by which CPU utilization on the host exceeds the stress threshold (Ω^{Δ}). The first step in relieving this stress is to determine whether or not the effect of pending *scale up* operations will relieve the stress without further action required. Since task instances of each task share incoming work, a *scale up* operation will result in a portion of the work from each existing task instance being taken by the new instance. We calculate the total estimated reduction in CPU utilization for the host (Ω^{Γ}), and determine if it will be enough to return the host to a non-*stressed* state (lines 8-12). If the pending *scale* *up* operation(s) will sufficiently reduce the CPU utilization, no further action is required.

Next, the algorithm looks for pending *scale down* operations in which the task being scaled down contains an instance which is running on the *stressed* host (lines 13-21). If a task is found, we select for shut down the task instance running on the *stressed* host (line 18). The task is then removed from the set of tasks pending *scale down* operations (line 19). No further action is taken. If, however, neither of the previous attempts were successful, and the host has remained *stressed* for the specified timeout period (Θ^τ), we perform a relocation via live migration. Based on an evaluation of multiple possible values, we set the timeout value to 2 algorithm executions. Relocation is then performed in a similar manner to Algorithm 1.

Algorithm 8: Integrated Algorithm - Stress Handling

```
   Data: $H, A, \Theta^\tau$
1  $H^!, H^+, H^-, H^\emptyset \leftarrow$ classify($H$)
2  for $h \in H^!$ do $\theta_h \leftarrow \theta_h + 1$
3  for $h \in H^+ \cdot H^- \cdot H^\emptyset$ do $\theta_h \leftarrow 0$
4  $T^\uparrow, T^\downarrow \leftarrow$ evaluateScaling($A$)
5  for $h \in H^!$ do
6      done $\leftarrow$ FALSE
7      $\Omega^\Delta \leftarrow \Omega'_h - \overline{\Omega}^\tau$
8      $\Omega^\Gamma \leftarrow 0$
9      for $t \in T^\uparrow$ do
10         for $i \in I_t$ do
11             if $v_i \in V_h$ then
12                 $\Omega^\Gamma \leftarrow \Omega^\Gamma$ + estimateReduction($i$)
13     if $\Omega^\Gamma \geqslant \Omega^\Delta$ then done $\leftarrow$ TRUE
14     if done = FALSE then
15         for $t \in T^\downarrow$ do
16             for $i \in I_t$ do
17                 if $v_i \in V_h$ then
18                     removeInstance($t$, $i$)
19                     $T^\downarrow \leftarrow T^\downarrow \setminus \{t\}$
20                     done $\leftarrow$ TRUE
21                     break
22             if done then break
23     if done = FALSE & $\theta_h \geqslant \Theta^\tau$ then relocate($h$)
```

The remainder of the algorithm continues to make use of pending *scale up* and *scale down* operations to avoid migrations. First, we look for *scale down* operations which can help prevent hosts that are *close* to becoming stressed from reaching the *stressed* threshold. *Scale* *down* operations on tasks that have an instance located on a host with CPU utilization within a specified distance of the $\overline{\Omega}^{\tau}$ threshold are performed (e.g. 5%), choosing an instance from that host for removal. This lowers the probability that the host will become *stressed* in the near future. Next, we perform all pending *scale up* operations, placing new instances first on *partially-utilized* hosts, and then on *underutilized* hosts.

Next, we integrate the *VM Consolidation* operation into the algorithm in order to consolidate load onto fewer hosts. The algorithm proceeds similarly to how the operation is described in Chapter 2, except that we make use of remaining *scale down* operations for tasks that have instances on underutilized hosts. First, we calculate a *shutdown cost*, which is defined as the number of VMs that must be migrated in order to empty the host, assuming that any tasks with pending *scale down* operations and instances on the candidate host will choose those instances to remove. We then iterate through the remaining *scale down* operations, choosing instances on *underutilized* hosts with the lowest *shutdown cost* for removal. If no task instances are located on an *underutilized* host, then the task instance on the host with the lowest CPU utilization is chosen. By targeting hosts in this manner, we are intelligently selecting *scale down* operations such that we remove instances from only a small set of hosts, and increase the likelihood of successfully shutting down an *underutilized* host with fewer (or zero) migrations required. Finally, if a host has been *underutilized* for a specified timeout period, we attempt to remove VMs from the host via VM live migration and subsequently shut it down. Based on a evaluation of multiple possible values, we set the timeout to 12 algorithm executions.

5.6 Evaluation

In this section, we find suitable tuning parameter values, and evaluate our approach through conducting a set of experiments. Due to the scale and complexity of the problem domain, experimentation on a real system was not feasible. As such, we evaluate our work using a simulation tool, DCSim [16] [5]. We begin by evaluating our autoscaling algorithm, comparing it against a static allocation for peak application demand. We look at a number of possible tuning parameter values, and select a configuration for use in subsequent experiments. We do not compare the autoscaling algorithm against other existing methods for autoscaling, as the specific autoscaling algorithm is not the focus of this work. Next, we examine running autoscaling alongside an existing dynamic VM management algorithm from our previous work [11] [7] (see Section 2.5), over a number of parameter values. We then perform a similar evaluation of our new, integrated application-aware algorithm, and compare it to running separate autoscaling and dynamic allocation algorithms.

5.6.1 Metrics

The following metrics were reported for this evaluation:

*Active Hosts (***Hosts***)*: The average number of hosts in the *On* state. The higher the value, the more physical hosts are being used to run the workload. Note that the peak value may be considerably higher.

*Average Active Host Utilization (***Host Util.***)*: The average CPU utilization of all hosts in the *On* state. The higher the value, the more efficiently resources are being used.

Power Consumption (Ψ): Power consumption is calculated for each host, and the total kilowatt-hours consumed during the simulation are reported. Power consumption is calculated using results from the SPECPower benchmark [4], and is based on CPU utilization.

*Application Size (***Size***)*: Application Size is the number of VMs belonging to the application. We report the average size as a percentage of the maximum size defined for each application.

SLA Achievement ($\overline{\mathbf{S}}$): SLA Achievement is the percentage of time in which the SLA conditions are met. See Section 5.3 for details.

*Autoscaling Operations (***AS Ops***)*: The number of *scale up* and *scale down* operations executed.

*Number of Migrations (***Migrations***)*: The number of migrations triggered during the simulation. Typically, a lower value is more desirable, as less bandwidth would be used for VM migrations.

*Total VM-time (***VM-t***)*: The combined total amount of VM running time. That is, the sum of the running time of every VM that existed at some point in the data centre. In a public IaaS cloud, where clients are charged on a pay-per-use basis (i.e. x dollars per VM per hour), VM-time translates directly into cost. Reported in *days*.

*VM-time Reduction (***VM-t Red.***)*: The percentage by which Total VM-time was reduced versus a static allocation of peak resource requirements.

5.6.2 Experimental Setup

Our simulated data centre consists of 8 racks of 40 host machines (320 total) based on the HP ProLiant DL160G5, with 2 quad-core 2.5GHz CPUs and 16GB of memory. Hosts are modelled to use a work-conserving CPU scheduler, as available in major virtualization technologies. As such, CPU shares that are not used by one VM can be used by another. No maximum cap

	S^{warn}	ω^{safe}	Hosts	Ψ	$\overline{S}$	Size	AS Ops	VM-t	VM-t Red.
Static	N/A	N/A	157.47	4735kWh	100.0%	100%	N/A	5605 days	N/A
AS-1	0.9	0.5	73.5	2502kWh	87.6%	46%	5175	2523 days	55%
AS-2	0.9	0.3	85.6	2845kWh	89.5%	51%	3878	2881 days	49%
AS-3	0.8	0.3	87.1	2885kWh	90.8%	52%	4066	2931 days	48%
AS-4	0.6	0.3	90.0	2964kWh	92.8%	54%	4479	3040 days	46%
AS-5	0.3	0.5	85	2826kWh	94.7%	53%	7171	2918 days	48%
AS-6	**0.3**	**0.3**	**97.6**	**3168kWh**	**95.7%**	**59%**	**5495**	**3301 days**	**41%**

Table 5.1: Autoscaling Algorithm (AS)

App.	Task	Service Time (s)	Visit Ratio	Default Size
1	1	0.005	1	1
	2	0.02	1	4
	3	0.01	1	2
2	1	0.005	1	1
	2	0.02	1	4
3	1	0.005	1	1
	2	0.02	1	4
	3	0.01	1	2
	4	0.01	0.5	1
	5	0.02	0.5	2
4	1	0.01	1	1

Table 5.2: Applications

on CPU is set for VMs. When the CPU is at maximum capacity and VMs must compete for resources, VMs are assigned CPU shares in a fair-share manner. Memory is statically allocated and is *not* overcommitted.

We model a set of interactive applications running within the data centre, as described in Section 5.3. Task instances run on VMs with 1 virtual core and 1GB of RAM. We defined a set of 4 artificial applications of varying configurations. Parameter details for the applications are listed in Table 5.2. The maximum size for each application is calculated as its default size scaled up by a *scale factor* of 3-6x, chosen randomly. The user think time for each application is set at 4 seconds.

The number of active users for each application changes dynamically based on real workload traces. Each application uses a trace built from one of 3 sources: *ClarkNet*, *EPA*, and *SDSC* [2]. We compute a normalized workload level based on request rate, in 100 second intervals, for each trace. These levels are used to define the current number of users of each application. The normalized workloads are scaled such that the peak number of clients receive a response time of 0.9 seconds (just under the defined SLA threshold) when all application tasks have their maximum number of instances. To ensure that applications do not exhibit synchronized behaviour, each applications starts its trace at a randomly selected offset time.

For each experiment, a base load of 10 applications is generated in the first 20 hours (in simulation time) of the experiment, which remain in the data centre for the duration of the experiment. Beginning at the 24 hour mark, additional applications dynamically arrive and depart the system, varying the total number of applications randomly between 10 and 50 every 2 simulated days. These applications have a life span of approximately 2 days, after which they terminate and depart the data centre. Every application is generated with randomly chosen configurations and scale factors. We use 10 different randomly generated application sets to evaluate our work, and all presented results are averaged across experiments using these 10 sets. We run our experiments for 8 simulation days, and we discard data from the first day of simulation to allow the system to stabilize before recording results (resulting in 1 full week of recorded simulation time).

5.6.3 Autoscaling

We compared the autoscaling algorithm (Section 7) against a static allocation. The static allocation creates the maximum number of instances for each application task, places all task instances (VMs) at the start of the experiment, and does not modify the allocation or number of instances further. As such, each application is allocated enough resources to meet peak demand, at all times. In contrast, the autoscaling algorithm attempts to adapt the size of each application to match fluctuating workload demands. Various values for tuning parameters S^{warn} and ω^{safe} were also evaluated, with the different configurations labelled as AS-1 through AS-6. Other tuning parameters, such as W_ρ, were chosen based on a separate evaluation of potential values. The autoscaling algorithm is executed every 5 minutes, again based on a evaluation of a number of possible frequencies. New task instances from *scale up* operations are placed in the first available host. During a *scale down* operation, an arbitrary task instance is chosen, as the autoscaling algorithm is based on potentially client-defined rules and has no knowledge of host state.

Table 5.1 presents a portion of the results of the autoscaling parameter evaluation. Results for other tested parameter values were omitted as they did not add any significant additional insights. All tested tuning parameter configurations result in a reduction in VM-time and power consumption. As our goal is to preserve SLA performance while reducing costs, we consider **AS-6** to be the best configuration of the autoscaling algorithm. It provides a 41% reduction in total VM-time and a 33% reduction in power consumption, while maintaining 95.7% SLA

$\geqslant 99\%$	21.3
$\geqslant 95\%$	51.2
$\geqslant 90\%$	63.4
$< 90\%$	9.2
Mean	95.6%
StDev	4.9%
Max	99.97%
95th	99.9%
75th	99.1%
50th	97.6%
25th	93.7%
Min	77.8%

Table 5.3: Detailed SLA for AS-6

achievement. As such, we use this configuration in subsequent experiments. Table 5.3 provides a more detailed look at SLA achievement under algorithm AS-6. Clearly, a few applications suffer more SLA violations than the majority. This is likely due to the fact that we are applying the same tuning parameter values to a set of diverse, randomly configured applications, rather than finding ideal values for each application. In a real deployment, these values would be chosen on a per-application basis, or learned over time, to provide the required SLA performance.

5.6.4 Autoscaling with Dynamic VM Management

We now add an existing dynamic VM management approach, running alongside our autoscaling algorithm. They run independently, with no knowledge of the actions being performed by the other. The autoscaling algorithm has knowledge only of application metrics, such as response time and VM CPU utilization, and makes decisions accordingly. The dynamic VM management algorithm has knowledge of host states, but no knowledge or control over application metrics and operations. New task instances created by the autoscaling algorithm are, however, placed using the *VM Placement* operation (Chapter 2). We run a set of 12 different experiments with varying values of $\overline{\Omega}^{\tau}$ and $\underline{\Omega}^{\tau}$. Tuning parameters for the autoscaling algorithm are identical to the values used in algorithm AS-6 in Section 5.6.3.

Table 5.4 presents the results of our experiments. Compared to the autoscaling algorithm alone (Table 5.1), we see a 25-33% savings in power consumption. This, however, comes at the expense of SLA achievement. While lower $\overline{\Omega}^{\tau}$ values lead to better SLA achievement, they also lead to a significant increase in the number of migrations performed. In addition, there is an increase seen in the number of autoscaling operations, which indicates that consolidation is affecting the behaviour of the autoscaling algorithm. Decreasing $\underline{\Omega}^{\tau}$ trades increased

$\overline{\Omega}^{\tau}$	$\underline{\Omega}^{\tau}$	Hosts	Host Util.	Ψ	$\overline{S}$	Migrations	AS Ops	VM-t
90	60	57.7	78.6%	2116kWh	89.0%	9653	7144	3824 days
90	50	58.3	78.3%	2134kWh	89.0%	8750	7167	3841 days
90	40	58.3	77.9%	2132kWh	88.9%	8350	7129	3818 days
85	60	59.5	77.0%	2172kWh	90.2%	16481	7043	3772 days
85	50	59.6	76.8%	2174kWh	90.2%	15356	7037	3770 days
85	40	59.9	76.3%	2181kWh	90.2%	14688	7002	3758 dayss
80	60	61.6	74.8%	2233kWh	91.5%	23734	6886	3698 days
80	50	62.1	74.5%	2245kWh	91.4%	22238	6901	3703 days
80	**40**	**62.6**	**73.8%**	**2260kWh**	**91.6%**	**21337**	**6891.7**	**3703 days**
75	60	64.9	72.0%	2326kWh	92.7%	31297	6792	3657 days
75	50	65.4	71.5%	2342kWh	92.6%	29056	6816	3667 days
75	40	65.8	70.9%	2349kWh	92.6%	27886	6779	3648 days

Table 5.4: Separate Autoscaling and Allocation (AS+DVM)

power consumption for improved SLA and migration count. We argue that using $\overline{\Omega}^{\tau} = 80\%$ and $\underline{\Omega}^{\tau} = 40\%$ is the best choice of configuration for the separate dynamic VM management algorithm, as it provides significant power savings while maintaining good SLA achievement and requiring significantly fewer migrations than with a further reduced $\overline{\Omega}^{\tau}$. This, of course, is valid only for our particular experimental configuration, but serves to show both the importance of the parameter values as well as the control they provide over the algorithm. In the future, these values could be determined automatically via a machine learning algorithm. We use this configuration for further comparison in Section 5.6.6.

5.6.5 Integrated Algorithm

We now evaluate our newly developed integrated autoscaling and dynamic VM management algorithm. Once again, we run a set of 12 different experiments with varying values of $\overline{\Omega}^{\tau}$ and $\underline{\Omega}^{\tau}$. Autoscaling decisions are made in the *evaluateScaling()* method (see Algorithm 8) using the same autoscaling tuning parameter values as algorithm AS-6 in Section 5.6.3, to provide a fair comparison.

Table 5.5 presents the results of our experiments with the integrated algorithm. Compared with separate autoscaling and dynamic VM allocation algorithms, we can see a notable improvement in SLA achievement, reaching nearly that of autoscaling alone. Furthermore, we note a significant decrease in the number of migrations, falling in line with our goals for the development of the algorithm, as well as fewer autoscaling operations. Again, we argue that using $\overline{\Omega}^{\tau} = 80\%$ and $\underline{\Omega}^{\tau} = 40\%$ is the best choice of configuration for the integrated algorithm,

$\overline{\Omega}^{\tau}$	$\underline{\Omega}^{\tau}$	Hosts	Host Util.	Ψ	$\overline{S}$	Migrations	AS Ops	VM-t
90	60	57.8	78.6%	2122kWh	88.8%	6612	7241	3855 days
90	50	63.8	72.8%	2291kWh	92.4%	4464	6928	3697 days
90	40	70.2	66.9%	2458kWh	93.6%	3471	6646	3600 days
85	60	58.3	78.0%	2136kWh	89.6%	6652	7136	3805 days
85	50	64.2	72.3%	2301kWh	92.8%	4790	6866	3675 days
85	40	70.9	66.7%	2480kWh	93.9%	3924	6634	3604 days
80	60	59.5	77.1%	2172kWh	90.7%	6954	7093	3760 days
80	50	65.3	71.7%	2334kWh	93.1%	5256	6850	3673 days
80	**40**	**70.5**	**66.6%**	**2469kWh**	**94.1%**	**4346**	**6573**	**3573 days**
75	60	61.0	75.8%	2220kWh	91.7%	7304	7016	3734 days
75	50	66.0	70.9%	2355kWh	93.4%	5686	6808	3652 days
75	40	71.0	66.1%	2483kWh	94.2%	4815	6520	3563 days

Table 5.5: Integrated Algorithm (INT)

as it provides significant power savings, sacrifices very little in terms of SLA achievement, and exhibits one of the lowest migration counts. We use this configuration for further comparison in Section 5.6.6.

5.6.6 Discussion

Figure 5.3 presents a visual comparison between the evaluated algorithms over four different metrics: *STATIC* refers to a static allocation of peak resource requirements; *AS* is the autoscaling algorithm running alone; *AS+DVM* is the autoscaling and dynamic VM management algorithms running separately; and *INT* is the integrated application-aware algorithm. Note that we compare *SLA Violation*, which is the inverse of *SLA achievement*, and as such lower values are better.

The static allocation (*STATIC*) naturally involved no SLA violation or migrations, but compared to all other algorithms, has the highest VM-time and power consumption. This represents higher costs for both cloud provider (power costs) and cloud client (VM rental costs). The autoscaling algorithm (*AS*) adds some level of SLA violation, but significantly reduces power consumption and VM-time compared to static. Further improvements in power consumption can be made through the use of dynamic VM management in the remaining algorithms. The *AS+DVM* algorithm results in the most power savings, but at the expense of the worst SLA violation and a very large migration count. Finally, the *INT* algorithm corrects this problem, offering similar power savings (22% better than *AS*) while drastically reducing the required number of migrations. Total VM-time for all experiments featuring dynamic VM management was slightly higher than for *AS*, indicating that dynamic VM management caused additional

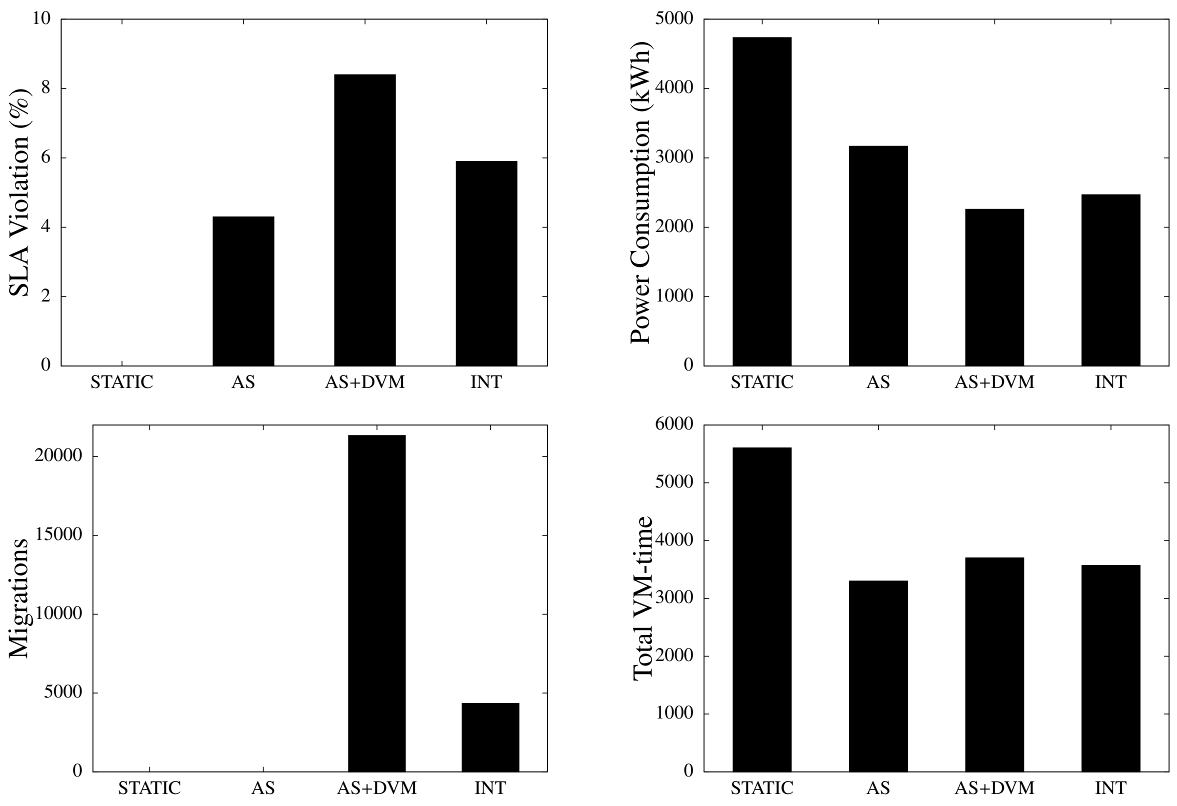

Figure 5.3: Algorithm Comparison

scale up operations to be triggered due to CPU contention.

5.7 Conclusions and Future Work

Both cloud providers and clients must strive to make efficient use of their resources in order to reduce their infrastructure requirements and costs. Cloud providers must consolidate load onto as few physical machines as possible in order to reduce power consumption and/or serve more clients with less hardware, while still providing the resources that their clients require. Cloud clients should provision only the resources they require to meet their current demand and objectives. To this end, application *autoscaling* dynamically adds and removes resources (VMs) to and from an application as demand changes. We introduced a rule-based heuristic autoscaling algorithm to address cloud client needs, and ran it alongside a dynamic VM allocation algorithm from our previous work [11] [7] to perform consolidation. While power consumption was reduced, it was at the cost of SLA achievement and a large number of VM live migrations. We developed a new, integrated application-aware algorithm which leverages some control over autoscaling operations to assist dynamic VM allocation. Through this approach, we were able to restore much of the lost SLA and greatly reduce the number of migrations required.

Future work includes investigating methods for reducing the effect of dynamic VM allocation on total VM-time. A more intelligent autoscaling algorithm could also be investigated, with the potential for each application to make use of a different, custom algorithm. A number of tuning parameters influence the performance of the autoscaling and dynamic VM allocation algorithms. Rather than determining values for these parameters experimentally, a machine learning approach could be developed to automatically discover the best values for each environment. Finally, we have previously developed a distributed algorithm for dynamic VM allocation [17], which could be extended to handle both autoscaling and dynamic VM management.

Bibliography

[1] Inc. Amazon Web Services. Amazon EC2 Auto Scale. `http://aws.amazon.com/autoscaling/`, July 2014.

[2] The Internet Traffic Archive. The internet traffic archive. `http://ita.ee.lbl.gov/`, July 2014.

[3] Trieu C Chieu, Ajay Mohindra, Alexei A Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *e-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*, pages 281–286. IEEE, 2009.

[4] Standard Performance Evaluation Corporation. Specpower_ssj2008 benchmark. `http://www.spec.org/power\_ssj2008/`, July 2014.

[5] Univeristy of Western Ontario DiGS Research Group. Dcsim on github. `http://github.com/digs-uwo/dcsim`, July 2014.

[6] Stefano Ferretti, Vittorio Ghini, Fabio Panzieri, Michele Pellegrini, and Elisa Turrini. Qos–aware clouds. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 321–328. IEEE, 2010.

[7] Graham Foster, Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. The Right Tool for the Job: Switching data centre management strategies at runtime. In *Integrated Network Management (IM), 2013 IFIP/IEEE International Symposium on*, May 2013.

[8] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal autoscaling in a iaas cloud. In *Proceedings of the 9th international conference on Autonomic computing*, pages 173–178. ACM, 2012.

[9] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723. IEEE, 2011.

[10] Gueyoung Jung, Kaustubh R Joshi, Matti A Hiltunen, Richard D Schlichting, and Calton Pu. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Middleware 2009*, pages 163–183. Springer, 2009.

[11] Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. An analysis of first fit heuristics for the virtual machine relocation problem. In *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, October 2012.

[12] Jim Zw Li, Murray Woodside, John Chinneck, and Marin Litoiu. Cloudopt: multi-goal optimization of application deployments across a cloud. In *Proceedings of the 7th International Conference on Network and Services Management*, pages 162–170. International Federation for Information Processing, 2011.

[13] Vinicius Petrucci, Enrique V Carrera, Orlando Loques, Julius CB Leite, and Daniel Mossé. Optimized management of power and performance for virtualized heterogeneous server clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 23–32. IEEE, 2011.

[14] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.

[15] Michael Tighe and Michael Bauer. Integrating Cloud Application Autoscaling with Dynamic VM Allocation. In *IEEE/IFIP Network Operations and Management Symposium (NOMS), 2014*. IEEE, 2014.

[16] Michael Tighe, Gastón Keller, Michael Bauer, and Hanan Lutfiyya. Towards an improved data centre simulation with DCSim. In *SVM Proceedings, 7th Int. DMTF Academic Alliance Workshop on*, October 2013.

[17] Michael Tighe, Gastón Keller, Hanan Lutfiyya, and Michael Bauer. A Distributed Approach to Dynamic VM Management. In *Network and Service Management (CNSM), 2013 9th International Conference on*. IEEE, 2013.

[18] Fetahi Wuhib, Rolf Stadler, and Mike Spreitzer. Gossip-based resource management for cloud environments. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 1–8. IEEE, 2010.

[19] Rerngvit Yanggratoke, Fetahi Wuhib, and Rolf Stadler. Gossip-based resource allocation for green computing in large clouds. In *Network and Service Management (CNSM), 2011 7th International Conference on*, pages 1–9. IEEE, 2011.

Chapter 6

Topology-aware Dynamic VM Management

6.1 Introduction

When placing VMs within the data centre, most work considers the data centre to be a simple pool of hosts, without structure. This approach is reasonable when placing individual VMs, but if we consider placing applications consisting of a set of communicating VMs, this simplification becomes problematic. A data centre is not simply a flat collection of servers, but rather has a hierarchical networking topology. Hosts are organized into *racks*, and *racks* into *clusters*, as defined in Chapter 2.3. Thus, the placement of communicating VMs, relative to each other, can have an effect on both application performance as well as network utilization. As such, VM placement should consider the data centre network topology when placing sets of communicating VMs. For example, VMs that communicate frequently with each other would benefit from being located near each other, ideally in the same rack, to reduce both network latency and load on higher level networking devices.

The data centre topology should also be considered when performing migrations for *dynamic VM management*. As much as possible, VM migrations should be performed across the fewest networking links possible (e.g. within a single rack), in order to reduce the utilization of higher level networking devices. In this work, we present a topology-aware approach to placing applications. We attempt, whenever possible, to place VMs belonging to an application within a single rack, and to maintain this constraint during dynamic VM management.

The remainder of the chapter is organized as follows: Section 6.2 presents related work on topology-aware VM placement. We define the problem in Section 6.3 and propose a topology-

This chapter is based on a journal paper, currently under submission aware dynamic VM management algorithm in Section 6.4. The algorithm is evaluated in Section 6.5, and we conclude in Section 6.6.

6.2 Related Work

Most work on dynamic VM management does not consider the data centre topology. Rather, the data centre is treated as a flat collection of hosts (servers), and VMs are placed without regard for their location within the network topology [1, 5, 6, 9].

Some work considers data centre topology when placing applications. SecondNet [4] handles placement of *virtual data centres* (VDCs). VDCs not only define a set of VMs, but also the communication links required between them. SecondNet is capable of providing bandwidth guarantees between VM pairs, and of modifying the VDC allocation in response to scaling or server failures. They also perform live migration in order to move VMs in a VDC *closer* together (fewer network hops), when possible. Zhani et al. [10] introduce VDCPlanner, which also focuses on VDC placement, as well as scaling and dynamic consolidation. Finally, Mann et al. [7] present Remedy, a VM management system which takes data centre networking into consideration. Remedy monitors bandwidth utilization, and makes use of VM migrations to shift load and relieve congested network links. Our work differs from existing topology-aware application placement in that we consider data centre topology while performing dynamic VM management with overcommitting of resources.

6.3 Problem Definition

In this section, we present the new aspects of the dynamic VM management problem that we are addressing in this work. We build upon the description of the core problem presented in Chapter 2.3, as well as the extension to include *applications* presented in Chapter 5.3. We present a further extension to consider the placement of applications in a topology-aware manner.

6.3.1 Application Placement

Tasks in an application must communicate with each other in order to process user requests. We assume that, for the purpose of reducing network latency, it is desirable to place all task instances of an application within a single rack, when possible. Furthermore, this reduces network usage of higher level switches (i.e. above rack-level). We define the *spread* of an application, $spread_a(s)$, as the number of racks which contain VMs running task instances of application $a \in A$.

$$\min \sum_{a \in A} spread_a(s)$$

There are situations in which it may be desirable to place application components in physically distinct locations, for a variety of purposes, including fault tolerance. This can mean placing replicated components in different clusters of a data centre, or more likely, in different data centre locations entirely. For the purposes of this work, we do not consider this situation. We believe that this simplification is reasonable, as including such placement considerations would represent an extension of this work, rather than an invalidation of it.

6.3.2 VM Live Migration

In order to reduce the impact of dynamic VM management on networking, it is desirable to contain migrations within a single rack as much as possible. That is, the *source* and *target* host in a VM migration should both be located in the same *rack*. Let λ be a migration operation, defined as $\lambda = (v, h_s, h_t)$, where h_s is the *source* host (the original location of the migrating VM), and h_t is the *target* host (the destination of the VM). Let the set of all migrations in a given time period be defined as Λ. We define a basic cost function for migrations as follows:

$$\text{cost}(\lambda(v, h_s, h_t)) = \begin{cases} 0 & \text{if } h_s \in r_1 \wedge h_t \in r_2 \wedge r_1 \neq r_2 \\ 1 & \text{otherwise} \end{cases}$$

Finally, we define an objective to minimize this cost.

$$\min \sum_{\lambda \in \Lambda} \text{cost}(\lambda)$$

6.4 Topology-aware Algorithm

When placing and dynamically managing *applications*, consideration should be taken as to where in the topology of the data centre individual components of the application (i.e. VMs) are placed. Individual tasks, and therefore task instances, must communicate with each other in order to complete user requests. We therefore attempt to place all task instances belonging to a single application within the same *rack*, as described in Section 6.3.1. The assumption is that such a placement will reduce network latency between communicating task instances, as well as reducing the usage of higher level networking elements. We incorporate the targeting of applications to racks into the *integrated* algorithm presented in Chapter 5.5, to define a new, topology and application-aware dynamic VM management algorithm. In order to enforce the placement of applications within a single rack, both the *VM Placement* operation and the unified *Dynamic Allocation* operations must be modified.

6.4.1 VM Placement

The *VM Placement* operation is modified to make a best effort at placing all VMs of an *application* within the same rack. It does so in a first-fit, greedy manner by first attempting to place the initial request of VMs into each rack in the data centre, executing the placement as soon as a suitable candidate rack is found. If none is found, it then attempts to place the application in two racks, then three, and so on, terminating either when the application is placed or has failed placement on the set of all available hosts. Note that we do not attempt to move existing applications to other racks in order to clear space for the newly arriving application, as the performance overhead of live migrating an entire application is assumed to be unacceptable. Nevertheless, the feasibility of such an approach should be investigated in future work.

The number of applications placed in each rack, and their size, can have an effect on the efficiency of dynamic VM management. Too few VMs within a rack can result in a reduction in the ability of the algorithm to effectively consolidate load, and too many results in task instances of an application being spread across multiple racks as applications scale and workload changes. Targeting a specific rack utilization (say, in terms of total CPU usage) is challenging, however, since applications scale up and down with changing workloads, and even individual VMs consume varying amounts of resource. As such, we simply use the number of *active* (powered on) hosts in a rack as a rough measure of rack utilization, and target a specified number of *active* hosts (Υ_h). Racks with fewer than Υ_h active hosts are sorted in descending order by number of active hosts for placement of a new application. If no placement is found, then placement is attempted using racks with more than Υ_h active hosts.

6.4.2 Dynamic Management

When performing migrations for *VM Relocation* (relieving a stressed host), *VM Consolidation*, or when instantiating a new task instance for a *scale up* operation, we sort potential target hosts and greedily choose the first successful match. Since hosts are simply sorted based on their utilization, as described in Chapter 2.5 and Algorithm 1, this may result in VMs being migrated into a different rack, thus breaking the single-rack application placement. In order to correct this situation, we modify the target host sorting. First, we define the *majority rack* of an application to be the rack which contains more task instances of the application than any other rack. We consider this to be a home rack for the application, and attempt to keep all task instances (VMs) within this rack.

Next, the target host list is divided into two new categories: those within the *majority rack* ($H^{\mathfrak{R}}$) and those outside of it ($H^{\not{\mathfrak{R}}}$). Each of these categories is then further subdivided into the utilization categories described in Chapter 2.4, namely, *stressed* ($H^{!}$), *partially utilized* (H^{+}), *underutilized* (H^{-}), and *empty* ($H^{\emptyset}$). Finally, we modify the target list construction in Algorithm 1 (line 3) as follows:

$$\begin{aligned} \textit{targets} \leftarrow\ & \mathrm{sort}(H^{+} \cap H^{\mathfrak{R}}) \\ & \cdot \mathrm{sort}(H^{-} \cap H^{\mathfrak{R}}) \\ & \cdot \mathrm{sort}(H^{\emptyset} \cap H^{\mathfrak{R}}) \\ & \cdot \mathrm{sort}(H^{+} \cap H^{\not{\mathfrak{R}}}) \\ & \cdot \mathrm{sort}(H^{-} \cap H^{\not{\mathfrak{R}}}) \\ & \cdot \mathrm{sort}(H^{\emptyset} \cap H^{\not{\mathfrak{R}}}) \end{aligned}$$

This target list ordering favours hosts within an applications *majority rack* first, and only resorts to outside hosts when none inside are available. This sorting is performed on a VM-by-VM basis, since each VM may belong to a different application and therefore have a different *majority rack*. This also leaves room for alternate sort orders to be implemented in order to enforce alternative placement rules (that is, other than single-rack placement).

Despite attempts to contain an application within a single rack, it may be the case that a migration or a *scale up* operation results in an application becoming spread across more than one rack. We add two measures to combat this: 1) carefully choosing task instances to remove in *scale down* operations, and 2) correcting placement via migration. Prior to performing *VM Consolidation*, in the integrated *dynamic allocation* algorithm described in Chapter 5.5, we look for pending *scale down* operations in which task instances belonging to the task are not located on the applications *majority rack*. If such operations are found, the instance located on a different rack is chosen for removal, thus completing the *scale down* operation and resolving the rack placement problem. If there is no such *scale down* operation pending, and a VM is located outside of its applications *majority rack*, then we perform a live migration in order to move the VM onto it, if such a target host is available. We refer to this as a *placement correction migration*.

6.5 Evaluation

In this section, we evaluate a range of values for the target number of active hosts per rack (Υ_h), and the compare the topology-aware algorithm against the integrated autoscaling and dynamic VM management algorithm proposed in Chapter 5. We conduct our evaluation using a simulation tool, DCSim [8] [3].

6.5.1 Metrics

The following metrics were reported for this evaluation:

Power Consumption (Ψ): Power consumption is calculated for each host, and the total kilowatt-hours consumed during the simulation are reported. Power consumption is calculated using results from the SPECPower benchmark [2], and is based on CPU utilization.

SLA Achievement ($\overline{\mathbf{S}}$*)*: SLA Achievement is the percentage of time in which the SLA conditions are met. See Section 5.3 for details.

*Number of Migrations (***Migrations***)*: The number of migrations triggered during the simulation. Typically, a lower value is more desirable, as less bandwidth would be used for VM migrations.

*Inter-rack Migrations (***Inter-rack Migs***)*: The number of migrations in which a VM was moved into a new rack. A lower value is more desirable, as fewer network elements would be involved in migrations.

*Placement Correction Migrations (***P.C. Migs***)* The number of migrations performed specifically to correct the rack placement of an application.

*Spread Penalty (***SP***)*: The Spread Penalty is a penalty applied when an application(s) in the data centre are placed across more than one rack, thus violating the single-rack application placement objective. It is calculated as the combined total amount of time all applications spent placed on more than one rack, in hours.

6.5.2 Experimental Setup

For the purposes of this evaluation, we use the same experimental setup as in Chapter 5.6.2.

Υ_h	Ψ	$\overline{S}$	Migrations	Inter-rack Migs	P.C. Migs	SP
0.1	2465kWh	92.0%	7628	3.2	3.2	0
0.2	2466kWh	92.3%	6620	9	8.7	1.3
0.3	2460kWh	92.6%	6310	17.4	17.4	0.1
0.4	2469kWh	92.7%	5741	60.8	59.4	5.9
0.5	**2458kWh**	**92.9%**	**5484**	**149.7**	**144.1**	**22.3**
0.6	2442kWh	92.8%	5386	286.9	278	33.7
0.7	2435kWh	92.7%	5392	572.3	549.1	79.6
0.8	2411kWh	92.4%	5515	1082.7	1000.6	182.6
0.9	2387kWh	92.3%	5523	1600.4	1422.6	280.5
1.0	2348kWh	91.8%	5714	2397.7	2032.4	486.2

Table 6.1: Topology-aware (INT+TOPO) Rack Target (Υ_h)

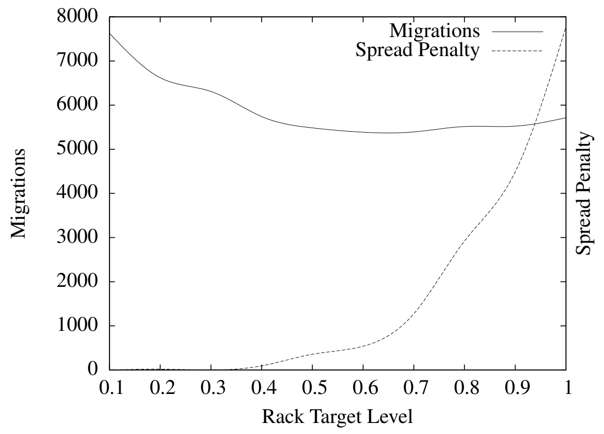

Figure 6.1: Rack Utilization Target

6.5.3 Topology-aware Algorithm

First, we evaluate a range of potential values for the rack utilization target parameter (Υ_h). Table 6.1 presents the results of these experiments. We can see that power consumption decreases slightly with higher rack utilization targets, but not by a large amount. SLA achievement and migration counts are best in the mid-range values, and inter-rack migrations, placement correction migrations, and spread penalty all increase with rack utilization target. Figure 6.1 plots migrations and spread penalty as the rack utilization target increases. We use $\Upsilon_h = 0.5$ for our final evaluation, as it provides the best SLA, close to the lowest migration count, and does not exhibit the increase in inter-rack migrations and spread penalty seen in higher values.

Table 6.2 compares the results of the original integrated autoscaling and dynamic VM management algorithm (*INT*) with the topology-aware version (*INT+TOPO*). The topology-aware algorithm sacrifices a small amount of SLA, and exhibits an increase in total migrations. On

Alg.	Ψ	$\overline{S}$	Migs	Inter-rack Migs	P.C. Migs	SP
INT	2469kWh	94.1%	4346	2601	N/A	3481.7
INT+TOPO	2458kWh	92.9%	5484	149.7	144	22.3

Table 6.2: Topology-aware vs Topology-unaware

the other hand, topology-aware significantly reduces the number of inter-rack migrations and the application spread penalty, thus demonstrating that it achieves the goal of containing each application within a single rack.

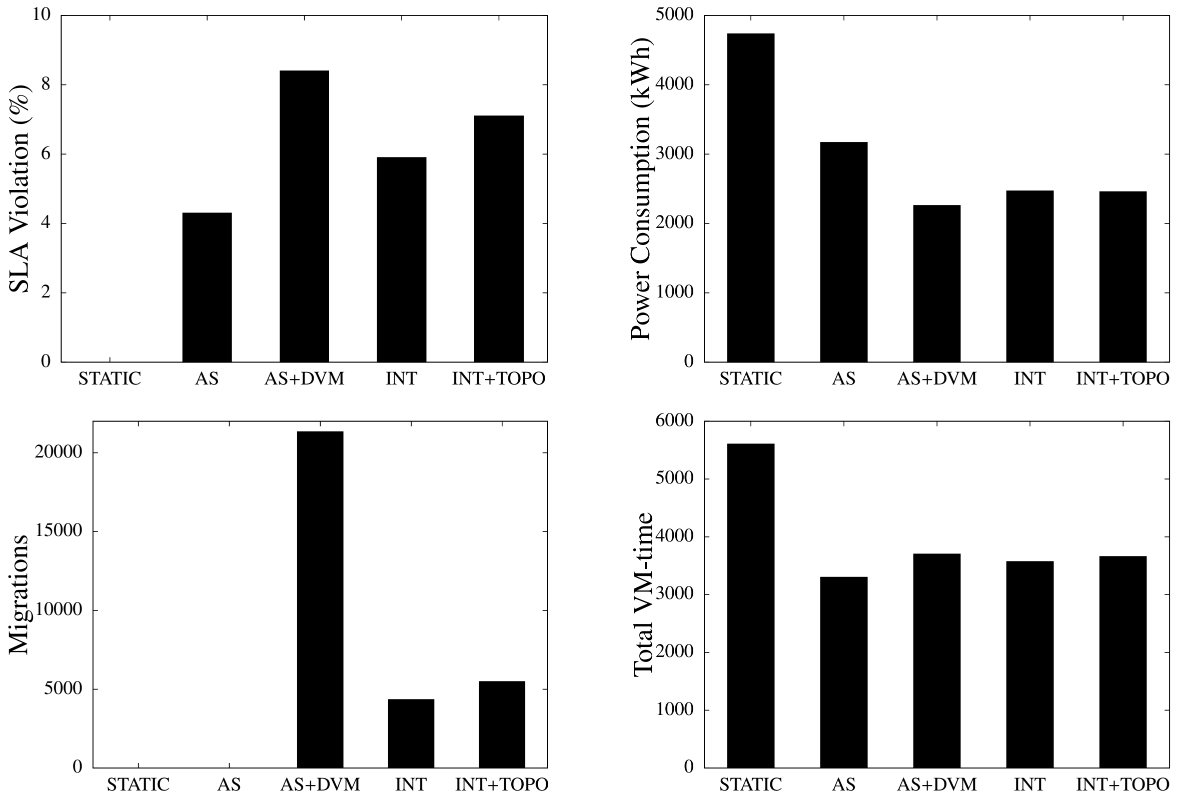

Figure 6.2: Algorithm Comparison

6.6 Conclusions and Future Work

Most work models the data centre as a flat pool of hosts (servers), with no structure. When placing applications consisting of multiple VMs, however, this simplistic view becomes inadequate. The network topology should be considered in this situation, as the placement of communicating VMs can have an effect on application performance (due to network latency) as well as overall network utilization in the data centre. Furthermore, migrations should be performed across as few network links as possible, in order to reduce the impact of migration bandwidth usage.

We present a topology-aware approach to application and VM placement, as well as dynamic VM management. In this work, we make a best effort attempt to place all VMs belonging to a single application within the same rack, to reduce network latency between VMs and to reduce traffic in higher levels of the network. Furthermore, we strive to maintain this placement during dynamic VM management, and keep as many migrations as possible within a single rack. In the event that an application becomes spread across more than one rack, we perform corrective actions to resolve the issue. Through evaluation by simulation, we showed that the topology-aware algorithm achieves single-rack placement well, and significantly reduces the number of migrations which take place across more than one rack.

In the future, more complex placement constraints for application tasks and task instances should be investigated. For example, different placement rules for fault-tolerance or availability could be incorporated into the algorithm. The possibility of migrating entire applications to a different rack in order to improve overall placement or make room for an incoming application could also be investigated.

Bibliography

[1] 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 Computat.: Pract. Exper.*, pages 1–24, 2011.

[2] Standard Performance Evaluation Corporation. Specpower_ssj2008 benchmark. `http://www.spec.org/power\_ssj2008/`, July 2014.

[3] Univeristy of Western Ontario DiGS Research Group. Dcsim on github. `http://github.com/digs-uwo/dcsim`, July 2014.

[4] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, page 15. ACM, 2010.

[5] Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. An analysis of first fit heuristics for the virtual machine relocation problem. In *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, October 2012.

[6] Gunjan Khanna, Kirk Beaty, Gautam Kar, and Andrzej Kochut. Application performance management in virtualized server environments. In *NOMS Proceedings, 2006 IEEE/IFIP*, 2006.

[7] Vijay Mann, Akanksha Gupta, Partha Dutta, Anilkumar Vishnoi, Parantapa Bhattacharya, Rishabh Poddar, and Aakash Iyer. Remedy: Network-aware steady state vm management for data centers. In *NETWORKING 2012*, pages 190–204. Springer, 2012.

[8] Michael Tighe, Gastón Keller, Michael Bauer, and Hanan Lutfiyya. Towards an improved data centre simulation with DCSim. In *SVM Proceedings, 7th Int. DMTF Academic Alliance Workshop on*, October 2013.

[9] Michael Tighe, Gastón Keller, Hanan Lutfiyya, and Michael Bauer. A Distributed Approach to Dynamic VM Management. In *Network and Service Management (CNSM), 2013 9th International Conference on*. IEEE, 2013.

[10] Mohamed Faten Zhani, Qi Zhang, Gwendal Simona, and Raouf Boutaba. Vdc planner: Dynamic migration-aware virtual data center embedding for clouds. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 18–25, 2013.

Chapter 7

DCSim

7.1 Introduction

Techniques and algorithms for dynamic resource management in the data centre are intended for use on a very large scale. Data centres providing cloud services continue to grow in size, with thousand to tens-of-thousands of servers to manage. This presents a unique challenge to researchers developing methods and algorithms for management, as the scale of the target environment precludes the use of a physical testbed. Consequently, simulation is commonly used for the evaluation of management techniques. Simulation also helps researchers quickly evaluate and fine-tune algorithms at a speed and scale not possible with a real implementation. Once a technique has been evaluated and fine-tuned using a simulation, further experimentation can be performed using a real infrastructure, albeit very likely on a much smaller scale.

There is therefore a need for an easily customizable and extensible simulation tool that model a virtualized, multi-tenant data centre. Furthermore, the tool must provide an application model to simulate the interactions and dependencies between VMs working together as a single service (e.g. a multi-tiered web application). Even if management algorithms only treat VMs as 'black boxes', with no knowledge of their applications, it is still important to model these applications within the simulation to drive VM behaviour and resource utilization. That being said, some IaaS providers (such as Amazon AWS [1]) offer advanced services, such as auto-scaling and dynamic load balancing, thus presenting an even stronger case for a detailed application model within the simulation tool.

Other features of virtualization, such as a *work conserving* CPU scheduler used in modern hypervisors, resource allocation and VM migration and replication must also be available. Host power states (*on*, *off*, *suspended*) must be modelled with appropriate transition times between

This chapter is based on work published in [13] and [14] states. The tool must also focus on usability, providing structures to allow for rapid prototyping and evaluation of management systems and algorithms. We present a new simulator, DCSim (Data Centre Simulator), designed specifically to address these requirements. DCSim is an extensible simulation framework designed to study VM management in a data centre providing an IaaS Cloud. DCSim is flexible, has features to aide in event and message sending, event callbacks and sequencing, and the creation of autonomic managers and communication between them. It also provides classes to help streamline the creation of new experiments, detailed output options and metrics, and a visualization tool to help provide a new perspective on the behaviour of data centre management methods and systems.

The remainder of this chapter is organized as follows: Section 7.2 presents related work in data centre and cloud simulation. Section 7.3 describes the architecture, core features and components. Section 7.4 gives some detail on how to configure and run experiments with DCSim. Section 7.5 provides an evaluation of the simulator through a demonstration of its use, and finally, Section 7.6 presents some conclusions and future work.

7.2 Related Work

There are a small set of existing simulation tools available, each with their own strengths, weaknesses, and target environments. GreenCloud [10] is designed to evaluate the energy costs of operating a data centre. It is a packet-level simulator built as an extension to Ns-2 [12], and provides a detailed model of communication hardware and power consumption of each element of the data centre. It does not, however, include modelling of virtualization. As such, it is not suitable for virtualized resource management research.

MDCSim [11] is a data centre simulation platform designed to simulate large scale, multi-tier data centres. It focuses on data centre architecture and cluster configuration, measuring both performance and power metrics. The simulator models a data centre running a three-tiered web application (web, application and database tiers), with the ability to modify and evaluate the configuration of each tier. Again, virtualization is not considered, nor are multiple tenants of the data centre. Also, it is built using a commercial product and is therefore not publicly available.

GDCSim (Green Data Centre Simulator) [8] aims to simulate both the management and physical design of a data centre, examining the interactions and relationships between the two. The goal is to fine-tune the interactions between management algorithms and the physical layout of the data centre, such as thermal and cooling interactions with workload placement. Resource management considers HPC (High Performance Computing) job placement and scheduling, power modes, and cooling settings. Transactional workloads (such as a web server) are modelled using a single workload, load-balanced across the data centre. Multiple tenants of the data centre and virtualization technology are not considered.

CloudSim [5] simulates a virtualized data centre, with multiple clients operating VMs. However, it implements an HPC-style workload, with *Cloudlets* (jobs) submitted by users to VMs for processing. It can be manipulated to simulate an interactive, continuous workload such as a web server [4], but it lacks a real model of such an application. An extension of CloudSim, NetworkCloudSim [7], considers communication costs between VMs performing parallel computations, but again focuses on HPC-style workloads rather than interactive workloads. Additionally, our work on DCSim adds data centre organization components such as racks and clusters not present in CloudSim.

SimWare [15] targets the modelling of data centre cooling and power costs, including the impact of server fan power consumption as related to the temperature of the data centre, and air travel time from CRACs to servers. Their simulated client workload is based on traces of HPC systems, rather than interactive applications.

DCSim differs from GreenCloud, MDCSim, and GDCSim in that it is focused on a virtualized data centre providing IaaS to any multiple tenants, similar to CloudSim. It differs from CloudSim in that it focuses on transactional, continuous workloads, and models such an application using a basic queueing model. As such, DCSim provides the additional capability of modelling replicated VMs sharing incoming workload as well as dependencies between VMs that are part of a multi-tiered application. SLA achievement can also be more directly and easily measured using response time values generated by the model. It also provides metrics to gauge power consumption, host utilization, and other performance metrics that serve to evaluate a data centre management approach or system. Furthermore, DCSim is designed to be easily extended, implementing new features and functionality.

7.3 DCSim Architecture & Components

DCSim (Data Centre Simulator) is an event-based, extensible data centre simulator implemented in Java, designed to provide an easy framework for developing and experimenting with data centre management techniques and algorithms. It models a data centre offering IaaS to multiple clients, focusing on transactional, continuous workloads (such as a web server), but can be extended to model other workloads as well. Figure 7.1 gives a high-level overview of the basic data centre model implemented by DCSim. The remainder of this section outlines the components and underlying mechanisms that drive DCSim, as well as other useful features and simulation output.

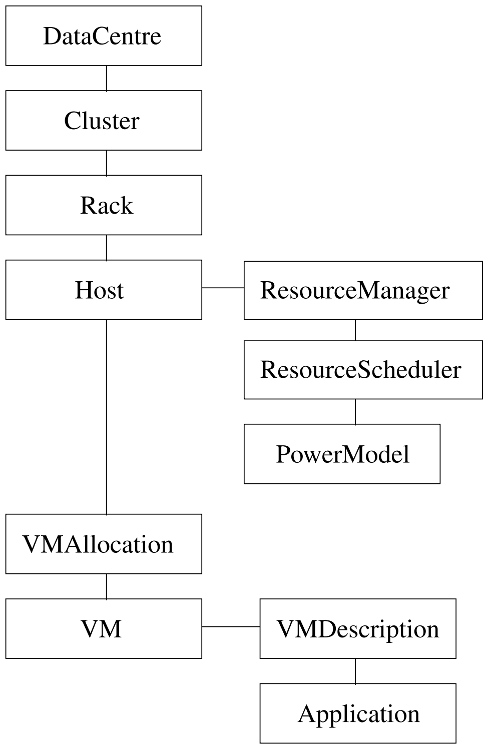

Figure 7.1: DCSim Overview

7.3.1 VMs, Hosts, Racks & Clusters

DCSim uses a series of abstractions to organize the architecture of a data centre. These abstractions are VM, Host, Rack, Cluster and DataCentre. In DCSim, a data centre consists of a collection of clusters, each cluster being a collection of racks, and each rack a collection of hosts. Both Cluster and Rack are designed to be homogeneous collections (in terms of their composing elements), but DataCentre may be an heterogeneous collection.

VM

A VM in DCSim represents a virtual machine running a single *application*. The properties and requirements of a VM are defined in its VMDescription, which is used to create an instance of the VM. The VMDescription defines the number of virtual cores and the amount of CPU, memory, bandwidth and storage resources requested. In its present state, DCSim allocates memory, bandwidth and storage statically to a VM, in the full requested amount (they are not oversubscribed). CPU resources, however, do not need to be fully allocated, allowing a host CPU to be oversubscribed. Once a VM is created and started on a host, its CPU requirements are driven dynamically by the needs of the application it is running (See Section 7.3.5 for details on applications in DCSim).

In order to perform dynamic management of VMs in a data centre, VMs must be moved from one host to another using *VM Live Migration*. Live migration allows a VM to be moved to another physical host with minimal downtime. DCSim supports simulating VM live migration, and calculates the time to migrate a VM based on available bandwidth and VM memory size. A CPU overhead is added to both the source and target host for the duration of the migration, which can be configured as desired. Additionally, an SLA penalty is attributed to the migrating VM.

Host

A host represents a physical machine in the data centre, capable of running VMs. Its physical properties are defined by the following set of attributes: the number of CPUs; the number of cores per CPU; core capacity; memory capacity; network capacity; storage capacity; and a power model. Core capacity is defined in terms of CPU Units, where one CPU unit is equivalent to 1MHz of processor speed (e.g., a 2.4GHz processor has 2400 CPU units). The power model defines how much power the host consumes at a given CPU utilization level, and is calculated using results from the SPECPower benchmark [6]. The benchmark provides power consumption levels of real servers in 10% CPU utilization intervals. We use these values and calculate intermediary values using linear interpolation. The resource utilization of the host at any given time is calculated as the sum of the resources in use by the set of VMs it is hosting (including its privileged domain).

A host can be in one of three primary states: `on`, `off`, or `suspended`, as well as transitional states between those three. VMs are only given resources to run their applications when the host is in the `on` state. The host consumes some small amount of power when in the `suspended` state, and no power when `off`. Transition times between states can be defined in the simulation configuration file.

Rack

A rack represents a collection of hosts in the data centre. This collection is homogeneous; that is, all hosts in the rack are of the same type. A rack has a given number of *slots* that can be filled with hosts, and this number may vary between racks. A rack counts also with two switches to which every host in the rack is connected.

Cluster

A cluster represents a collection of racks in the data centre. As with racks, this collection is also homogeneous. The number of racks per cluster is not fixed, so different clusters can have different numbers of composing elements. The cluster also contains two collections of switches, one for the data network and one for the management network (more information on networks in the next section).

7.3.2 Data Centre Network

In DCSim, a data centre has two different networks: a data network and a management network. The first is used to meet the communication needs of the hosted VMs, while the second is used for the internal management of the data centre. VM migrations make use of the management network as do status update messages or migration requests exchanged between management entities.

A network consists of nodes and edges, namely, *NetworkElement* objects and *Link* objects. A Link has a certain bandwidth capacity and it connects two NetworkElement objects. There are two types of NetworkElement: *NetworkCard* and *Switch*.

Every host has two network cards, one for each network. These network cards are connected through links to their corresponding switch in the rack (two switches per rack, one per network). At cluster-level, two network arrangements are possible: one, every rack in the cluster is connected to a single switch (per network), which is referred to as *main switch* and requires as many ports as there are racks in the cluster; and two, there is a two-level hierarchy of switches (per network), where racks are connected to low-level switches and low-level switches are connected to a single high-level switch (referred to as *main switch*). At data centre-level, there is a central switch (per network) to which each cluster's main switch is connected.

7.3.3 Simulation Engine

As DCSim is intended to be a simulation platform that can be extended to suit the needs of a particular area of research, it is useful to take a look at the mechanism by which DCSim advances through simulated time in order to help gauge the feasibility of possible extensions. Algorithm 9 outlines a simplified version of the main simulation loop.

The *eventQueue* contains all future events that must be executed, *simTime* records the current simulation time, and *duration* is the length of the simulation (in simulation time). The outer loop is responsible for advancing in simulation time to the next scheduled event(s). Changes to data centre state only occur via events; in-between events, the state of the data cen-

Algorithm 9: Main Simulation Loop

```
Data: $duration, eventQueue$
$simTime \leftarrow 0$
while $|eventQueue| > 0 \land simTime \leq duration$ do
    scheduleResources()
    postScheduling()
    $e \leftarrow$ peek($eventQueue$)
    $simTime \leftarrow e$.getTime()
    advanceSimulation($simTime$)
    updateMetrics()
    performLogging()
    while $|eventQueue| > 0 \land$ peek(eventQueue).getTime() $= simTime$ do
        $e \leftarrow$ pop($eventQueue$)
        handleEvent($e$)
```

tre is static. The first phase of the loop uses the current data centre state to *schedule resources* (line 3), in which an allocation of resources/second for each VM is calculated (see Section 7.3.6). CPU scheduling is based on current application demands, in a fair-share manner up to the maximum capacity of the host processor.

Next, the post-scheduling hook (line 4) allows data centre components to create new events or move existing events based on dynamic resource scheduling. This enables the simulation of operations and processes that exhibit variable runtime based on available resources. This feature is included primarily for the planned future development of variable VM migration times (due to changes in available network bandwidth), and batch/HPC type jobs whose runtime is based on dynamically scheduled CPU. We then check for the simulation time of the next event (which may have changed based on processing in postScheduling()), and *advance* the simulation to the time of the next event using the calculated resource scheduling (lines 6 & 7). Simulation metrics are then updated (see Section 7.4.2) and logging is performed. The inner loop (lines 10 to 12) executes all events that take place at the current simulation time. The process is then repeated to advance to the next set of events.

7.3.4 Events

As DCSim is an event-driven simulation, all actions, operations and state changes in the simulation are triggered by an *event*. Events are also used for communication between data centre elements and management components. The basic properties of an event are the simulator component(s) that will receive the event, and the time at which to execute it. Events are ordered such that, in the case of multiple events being executed at the same simulation time, they are executed in the order in which they were sent. Any component can send an event to another component, or to itself in order to trigger some functionality at a specific time in the future. The basic event class (*Event*) is abstract, with specific event types implementing any behaviour or storing any data they require.

There are a number of hooks and methods which can be used to add additional functionality to an event. Pre-execution and post-execution methods can be implemented to perform operations before and after the event is executed, such as logging event details. An *event callback* can also be registered with an event, allowing one or more objects to be notified once an event has been executed. In some cases, an event can cause several other events to be generated in order to complete an operation, which may require the post-execution and callback methods to be triggered only after the complete sequence of events has been executed. To accomplish this, events can be strung together in a *sequence*. For example, instructing a host (i.e., a server; see Section 7.3.1) to boot up involves one event sent to the host, and another event sent by the host to itself some time later indicating the completion of the operation (hosts take time to boot). These events are added in sequence together, allowing a management component to receive a callback only once the full boot up operation has completed.

A special event subclass, *MessageEvent*, can be used for communication between components by extending it with any additional functionality required. MessageEvent automatically keeps track of the number of messages of each specific subclass that are sent during the simulation. Finally, a special type of event, called the *RepeatingEvent*, can be used to trigger repeated executions of the event on a regular interval.

7.3.5 Application Model

The application model in DCSim makes use of a basic queueing network. The target application is an *interactive* application, such as a multi-tiered web application, in which a number of clients interact with the system, waiting for responses before issuing a new request. Requests enter and exit the application through a single point, with all components of the application working together to serve each request. Individual components (tasks) of an application each run within a VM. These VMs may be co-located on a physical machine with other VMs, and may compete for CPU resources. This results in a reduction of VM, and therefore application, performance. Furthermore, each physical machine in DCSim may have a different CPU, which can also affect application performance.

In the remainder of this section, we describe the implemented model, and discuss its implementation in DCSim.

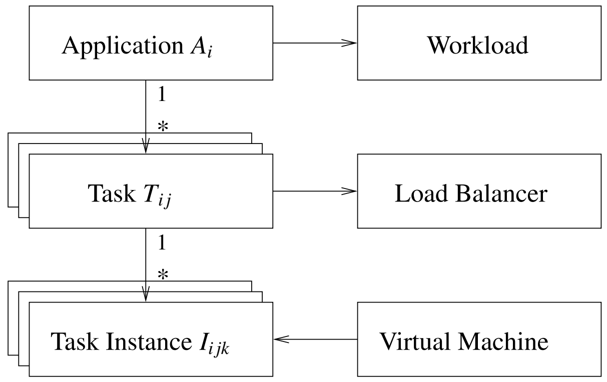

Figure 7.2: Application Model

Application Model

An *application* ($a \in A$) contains a set of tasks T_a, where a single task is denoted $t \in T_a$. Tasks are performed by one or more identical *task instances* ($i \in INSTSET$), which share incoming workload via a *load balancer* component. Each task instance runs within a single VM, and each VM runs a single task instance. Figure 7.2 provides an overview of the application model.

We model the application as a *closed* queueing network, and solve it using Mean Value Analysis (MVA) (or alternatively by Schweitzer's Approximation of MVA for improved simulator performance). In Addition to a set of tasks, an application has a specified client *think time* (Z_a), and a *workload* component. The workload defines the current number of clients (N_a) connected to the application, which can change at discrete points in the simulation based on a trace file or random number generation. Tasks are defined by the default and maximum number of instances in the task, the *service time* of a single request, its *visit ratio* (v_t; the number of times each request visits the task), and a *resource size*, which will be discussed later.

Load Balancing

Each task instance i is modelled by a single queue. We assume that load balancing is calculated in some manner between instances of a task, with ϕ_i, $\sum_{i \in I_t} \phi_i = 1$ denoting the percentage of requests sent to each instance i. Let v_t be the visit ratio for task t, and v_i be the visit ratio for instance i. Then $v_i = v_t * \phi_i$. For example, if every incoming request visits t once, and the load balancer distributes requests to each instance equally, then $\phi_i = 1/|I_t|$ for every instance. If there were 4 instances, then $v_i = 1 * 1/4 = 0.25$ for each instance. The visit ratio of the instances are changed, but the visit ratio of the task remains the same. Instance visit ratios will change dynamically as new instances are added or removed. Note that we do not presently model the load balancer as an actual "physical" component running on a server in the simulated data centre, but is simply a detached entity that calculates request shares for instances of a task.

Task Resource Requirements

The resource requirements of a task are defined as its *resource size*. The resource size defines the expected amount of CPU, memory, bandwidth and storage expected to be available to each instance of the task. We treat memory, bandwidth and storage as fixed requirements, which are identical and static for all instances of a task. CPU, on the other hand, does not have to be completely allocated, and the CPU demand of an instance varies over time based on application workload. As such, we will focus only on the CPU resource. We define the CPU size of t as ω_t^{size}.

Within DCSim, we quantify the CPU capacity of server as a single scalar value, called *CPU units*. A single CPU unit is equivalent to a 1MHz clock speed, so a 2.5GHz processor would have 2500 CPU units. A dual core 2.5GHZ processor would have 5000 CPU units. A VM (and therefore a task instance) cannot use more CPU units than the number of virtual cores it has been assigned multiplied by the CPU units per core on its host server. We refer to this maximum amount of CPU that a VM can use as its *CPU max*, denoted ω_i^{max}). The CPU load balancing features of modern hypervisors balance load between cores and CPUs sufficiently to justify treating the entire set of cores as a single value.

CPU size (ω_t^{size}) is the total amount of CPU expected by the task, as defined in the task resource size, and is identical for all instances of the same task. Since the actual CPU needs of an instance vary over time, CPU demand (ω_i) represents the amount of CPU required at a specific point in time in order to process the current workload. Note that $\omega_i <= \omega_t^{size}$. A task instance may not, however, receive its entire CPU demand, due to contention with other task instances (VMs) on the same server. Each server has a resource scheduler component, which fairly divides CPU between task instances (VMs) running on the server, given their CPU demand. The amount of CPU scheduled for a task instance is denoted ω_i'.

Task Instance Service Time

The service time for a single request to a task, σ_t, is defined assuming that the full CPU size, ω_t^{size}, is available to the task instance. This, however, may not always be the case. The service time σ_i for each task instance must be adjusted to account for both the case where the server CPU on which a instance is running has a different CPU core capacity than is specified in the task resource size, as well as when CPU demand cannot be met due to contention ($\omega_i' < \omega_i$).

First, we will look into accounting for differences in CPU core capacity. Let Ω_h^{core} be the CPU core capacity of the physical server running instance i, and $\omega_t^{coreSize}$ be the defined CPU core capacity in the resource size of t. Then we calculate $\sigma_i = \sigma_t \times (\omega_t^{coreSize} / \Omega_h^{core})$, resulting in a faster service time for instances running on faster cores, and a slower service time for slower cores.

Next, we calculate the effective service time, σ'_i for a instance i, given that ω'_i may not be equal to ω_i. If $\omega'_i = \omega_i$, then σ_i is unchanged. If, however, the $\omega'_i < \omega_i$, then we add an additional, surrogate delay to account for processor queueing. We therefore calculate σ'_i as

$$\sigma'_i = \begin{cases} \sigma_i & \text{if } \omega'_i \geqslant \omega_i \\ \sigma_i \times \frac{\omega_i}{\omega'_i} & \text{if } \omega'_i < \omega_i \end{cases}$$

Using our new service time calculations for task instances, their updated visit ratios based on load balancing, and the application properties N_a and Z_a, we can calculate mean application response time and throughput, using the MVA algorithm (or an approximation). This, of course, depends on the calculation of ω_i and ω'_i, which we discuss next.

CPU Scheduling

Each time there is a change in the data centre state, (e.g. a change in workload level, a new VM to place, a VM is migrated, a VM terminates, etc.), we must determine CPU demands for each task instance and schedule host CPU to each instance. CPU demand can be determined using utilization values, calculated with the MVA algorithm, for each task instance (denoted U_i). CPU demand is calculated as the maximum CPU that can be used by a task instance (ω_i^{max}) multiplied by U_i. Once CPU demand has been calculated for all task instances in all applications, the server resource schedulers can use these values to determine the amount of CPU to schedule to each instance. Since scheduling may have an effect on instance service time, we then must run the MVA algorithm once more to determine the new values. In a closed system, a change in effective service time of one task instance may affect the utilization and demand of another. Furthermore, since task instances of several applications may be co-located on a single server, they will interact with each other as well in the form of CPU contention. Therefore, we must recalculate CPU demand, and repeat the process. This repeats until there is no longer a change in utilization values, and we have arrived on the correct CPU demand and scheduled values for each application (and therefore the corresponding response time and throughput values). Unfortunately, there are some cases where the algorithm enters a cycle and does not terminate. To deal with this situation, we first use a small threshold value on the change in utilization (δ^τ), whereby if the change in utilization is sufficiently small, we stop the algorithm. Second, in the event that this does not work, we have a hard limit (ϱ^τ) on the number of loop iterations. DCSim reports the number of times that the limit is reached in the simulation results.

Algorithm 10: VM Scheduling

```
Data: $A, H, \delta^\tau, \varrho^\tau$
1  for $i \in A$ do $\omega_i \leftarrow \omega'_i \leftarrow \omega_i^{max}$
2  for $a \in A$ do executeMVA($a$)
3  for $i \in A$ do $\omega_i \leftarrow \omega_i^{max} \times U_i$
4  $\Delta^{max} \leftarrow \infty$
5  $c \leftarrow 0$
6  while $(\Delta^{max} > \delta^\tau) \wedge (c < \varrho^\tau)$ do
7      for $h \in H$ do scheduleResources($h$)
8      for $i \in A$ do $U_i^{prev} \leftarrow U_i$
9      for $a \in A$ do executeMVA($a$)
10     $\Delta^{max} \leftarrow 0$
11     for $i \in A$ do
12         $U_i \leftarrow X_i \times \sigma_i \times \nu_i$
13         if $|U_i^{prev} - U_i| > \Delta^{max}$ then $\Delta^{max} \leftarrow |U_i^{prev} - U_i|$
14     for $i \in A$ do $\omega_i \leftarrow \omega_i^{max} \times U_i \times \frac{\sigma'_i}{\sigma_i}$
15     $c \leftarrow c + 1$
```

Algorithm 10 shows the VM scheduling algorithm. We initialize CPU demand and CPU scheduled for all task instances (line 1) to be equal to the maximum amount of CPU that instance can use (based on the number of virtual cores it has been assigned). We then execute the MVA algorithm (line 2), and calculate initial CPU demand (line 3). Note that since CPU demand is equal to CPU scheduled, we don't need to take into account the effective service time, yet. This gives us the 'ideal' CPU demand, assuming no CPU contention. Line 4 initializes the maximum utilization change value (Δ^{max}), and line 5 initializes the iteration counter (c).

Using these initial values, we begin our loop (lines 6-15) to refine the values based on scheduling. The loop terminates when the maximum change in task instance utilization (Δ^{max}) is less than a threshold value (e.g. 0.02). First, we schedule CPU for all task instances on all servers, using the current CPU demand values (line 7). We then store the current utilization value for all task instances (line 8), and execute the MVA algorithm again (line 9). Using the MVA throughput results (X_i), we calculate utilization values for all instances, and record the largest change in utilization of any instance (lines 11-13). Note that we use the original instance service time to calculate utilization, not the effective service time, to prevent scheduling from having an effect on the CPU demand value. CPU demand is recalculated (line 14), this time considering the difference between effective service time and normal service time. Finally, we increment the iteration counter.

Again, this process occurs each time there is a change in the simulated data centre state.

Notation	Definition
$a \in A$	An application in the set of all applications
$t \in T_a$	Task t of application a
$i \in I_t$	Instance i of task t
N_a	Number of clients currently using a
Z_a	User think time for a
ν_t	Visit ratio of task t
ϕ_i	Percentage of requests for task t sent to instance $i \in I_t$
U_i	Utilization of i
X_i	Throughput of i
ω_t^{size}	CPU size of task t
ω_i	CPU demand of i
ω_i'	CPU scheduled for i
ω_i^{max}	Max CPU of i
σ_t	Service time for task t
σ_i	Service time for instance i
σ_i'	Effective service time for instance i
$h \in HOSTSET$	Host h in the set of all hosts, H
Ω_h^{core}	CPU core capacity (in CPU units) of physical server h
$\omega_t^{coreSize}$	CPU core capacity defined in resource size of task t
δ^τ	Threshold on instance utilization change
ϱ^τ	Threshold on the number of scheduling rounds

Table 7.1: Notation

We record the calculated mean response time and throughput for each time interval throughout the simulation.

Application Workload Configuration

DCSim workload traces consist of normalized workload level values in the range [0, 1], given on a fixed time interval (e.g. every 100 seconds). Current traces are based on the number of incoming requests to web servers from publicly available traces, in 100 second intervals. These values then must be scaled up to match the size the application, which varies depending on the application configuration (i.e. number of tasks, task service times, user think time, etc.). In order to determine the proper value to scale the workload by such that the application exhibits the desired range of utilization, DCSim includes a helper function which calculates the scaling factor required in order to achieve either a specified maximum task instance utilization, or maximum application response time.

Service Level Agreements

The final piece of the puzzle is the inclusion of a basic *Service Level Agreement* (SLA) which can be attached to an application in DCSim. An SLA consists simply of an upper threshold on either response time or throughput (or both). DCSim records the percentage of time for which SLA was achieved over the duration of the simulation, for each application. Statistics over the set of application SLA achievement values are available. It is also possible to define a penalty value, in terms of penalty-per-second applied during SLA violation. This can be defined per application, so some applications can have higher penalty values than others.

7.3.6 Resource Managers & Scheduling

Host resources in DCSim are managed by a *Resource Manager* component on each host. The resource manager is responsible for allocating and deallocating resources for VMs, keeping track of the total amount of resource allocated, and deciding whether or not the host is capable of running a given VM. The resource manager is an abstract class and must be extended to provide the desired functionality. The default resource manager allocates memory, bandwidth and storage statically, with no oversubscription. CPU is oversubscribed, allocating to VMs as much CPU as they request, although they may not actually receive it if the host CPU becomes overloaded.

While the resource manager handles allocations, the *Resource Scheduler* handles the scheduling of dynamic resources, such as CPU, based on current demand. At present, the resource scheduler schedules only CPU, although it could be extended to dynamically calculate usage of other resources as well, such as bandwidth. Other resources are simply given their full allocation, as determined by the resource manager. During the *schedule resources* phase of the simulation loop (see Section 7.3.3), the resource scheduler for each host calculates the amount of resources/second that each VM is given. In the case of CPU, this would be the number of CPU units given to each VM. It does so in an fair-share manner, giving each VM a chance to receive an equal amount of CPU, up to the total CPU required by its application at the current time. CPU not used by one VM can be used by another, and any CPU amount required by a VM over and above the capacity of the host is not scheduled, resulting in application performance degradation.

7.3.7 Autonomic Managers & Policies

Our development of DCSim is focused on providing tools to support research on virtualized data centre management. The *Autonomic Manager* (AM) and related components provide a framework to allow quick development of new management systems, while taking care of some of the messaging and event handling details of DCSim automatically. The AM acts as a container for a set of *Capabilities* and *Policies*. A *Capability* is simply an object that stores data and provides methods for use in one or more policies. For example, the *HostManager* capability provides a reference to a host that is managed by an AM possessing the capability. This can be used by a *Policy* that is designed to manage a host in the data centre. Policies are installed into AM, and implement the actual management logic. A policy can only be installed in an AM that possesses the capabilities that the policy requires to function.

Policies can be triggered on a regular interval, or can respond to events sent to the AM by another policy or component. In order to design a policy that executes on a regular interval, we simply create a policy class that defines an `execute()` method, and pass the time interval to the AM when installing the policy. To trigger a policy on the arrival of a specific event class, we simply define an `execute(ConcreteEvent e)` method, and the AM will automatically detect that the policy accepts these events, and call the policy whenever one is received.

AMs do not need to be attached to any other component, and can simply run detached from the physical data centre infrastructure. However, they can also be attached to host objects, to indicate that the AM is running on that Host. When this configuration is used, the AM will only execute when the host is in the `on` power state.

Within this framework, it is a quick and simple process to define new policies, capabilities and events to build a desired management system or test a management algorithm.

7.3.8 Management Actions

Common management operations performed within a simulation can be encapsulated in a *Management Action*. DCSim currently features management actions for instantiating a new VM, migrating a VM, replicating a VM within an application task, and shutting down a host. Additional management actions can be created by extending an abstract class. It is possible to build a set of actions which can be executed either concurrently, in sequence, or in combinations of the two. If a sequence of management actions is executed, the preceding management actions must complete before subsequent ones can execute. This includes the case where some management actions, such as VM migration, may take some time to complete.

7.3.9 Metrics

DCSim includes a mechanism for recording metrics of interest in order to evaluate management systems and algorithms through simulation. All metrics are collected within the *SimulationMetrics* class, which is composed of specific *MetricCollection* objects responsible for collecting a group of related metrics. For example, the *ApplicationMetrics* metric collection contains a set of metrics related to the performance of applications deployed in the simulated data centre, while the *HostMetrics* collection contains metrics such as host utilization and power consumption. New metric collections can be defined to keep track of custom metrics of interest to a particular experiment by simply extending the MetricCollection class and adding it to the simulation metrics. Metric collections can output their set of recorded metrics in printed, human-readable form or as a list of *(name, value)* tuples.

7.4 Configuring and Using DCSim

In this section we describe some of what is required to configure and run DCSim.

7.4.1 Workloads

As discussed earlier in Section 7.3.5, the *Workload* component is responsible for specifying a dynamic workload level for applications running in the simulated data centre. In our simulations, we use normalized workload traces built from 5 real web server traces: the *ClarkNet*, *EPA*, and *SDSC* traces [3], and two different job types from the *Google Cluster Data* trace [9]. To ensure that VMs do not exhibit identical behaviour, we always start the trace for each VM at a randomly selected offset time.

In a data centre, the set of VMs is not static; VMs continuously arrive and depart the data centre. To model this, DCSim has a feature which dynamically adds new applications (see Section 7.3.5) to the data centre, which are submitted by sending a *Application Placement* event containing a description of the application to be instantiated in the data centre. Applications have a lifespan chosen randomly from a specified distribution, after which they terminate. This helps model not only changes in individual VM resource requirements, but also changes in overall data centre utilization over the course of a single simulation.

7.4.2 Default Metrics

DCSim provides a number of useful metrics in order to help judge the performance of data centre management systems and algorithms, including the following:

Average Active Host Utilization

The average CPU utilization of all hosts that are currently in the on state. The higher the value, the more efficiently resources are being used.

Max, Min, and Average Active Hosts

The maximum, minimum, and average number of Hosts in the on state at once.

Number of Migrations

The number of migrations triggered during the simulation, by each management component that triggers migrations. Migrations are also further broken down into migrations within a rack, across racks, and across clusters.

SLA Achievement

SLA (Service Level Agreement) Achievement is the percentage of time in which the SLA conditions are met (i.e. response time is below the threshold specified in the SLA). By default, DCSim reports the following regarding SLA achievement: mean and standard deviation; max and min; 95th, 75th, 50th, and 25th percentiles; the number of application with $\geqslant 99\%$ achievement, $\geqslant 95\%$, $\geqslant 90\%$, and $< 90\%$. Penalties can also be assigned to violating SLA on a per application basis, and various statistics on penalties are also reported.

CPU Underprovisioning

When a VM has more CPU demand than it is scheduled, we record the difference and report the total percentage of CPU not scheduled as *CPU underprovisioned*. This is done by recording the total CPU demand calculated in the first 'ideal' case, and comparing it with the final CPU scheduled. Note that computing this value would not be possible in a real-world deployment.

Power Consumption

Power consumption is calculated for each host, and the total kilowatt-hours consumed during the simulation are reported.

Message Counts

The number of message sent, for each subclass of MessageEvent used during the simulation.

Applications

In addition to SLA related metrics, a variety of metrics are reported regarding applications deployed during the simulation. The total number of applications created, placed, and completed are reported, as well as the number of applications that could not be placed (due to high data centre utilization). Response time and throughput values are also recorded.

7.4.3 Performing Experiments with DCSim

In order to make configuring and performing experiments with the simulator as clean and easy as possible, DCSim contains a set of helper classes for performing simulations. The *SimulationTask* class encapsulates a single simulation configuration, allowing the user to configure the simulator by implementing the setup() method, while taking care of the details of running the simulation automatically. Simulation name, and duration can be specified, as well as a period of time to wait before recording metrics. Finally, a seed for random number generation can be passed to the simulation task to be used to generate any random elements, such as workload configurations. This provides repeatable experiments, which is convenient both for debugging and for comparing management systems and algorithms. Once the task has been run, a collection of metrics recorded during the simulation is returned.

In order to run several simulations, either sequentially or concurrently, simulation task objects can be added to a *SimulationExecutor*. The simulation executor handles spawning threads for individual simulation tasks, waiting for all tasks to complete, and returning the resulting metric collections from each task.

7.4.4 Output & Logging

DCSim uses the logging library Apache log4j [2]. By default, only basic output is printed to the console, with other options available for more detailed logging (at the expense of processing time required for logging I/O). The DCSim configuration file contains several options specifying different logging output:

Enable Detailed Console

This will cause detailed, human-readable data on the execution of the simulation to be outputted to the console. This includes data on each Host and VM at every step in simulation, as well as data on management operations such as VM migration.

Enable Console Log File

Console output will also be written to a log file.

Enable Simulation Logging

Individual, detailed data on the execution of the simulation (the same data as enabled with the *Enable Detailed Console* option), will be written to a separate log file for each simulation task run, even if several simulation task objects are executed concurrently.

Enable Trace

This will enable a machine-readable version of the detailed simulation data, for use in graphing or visualizations.

7.4.5 Visualization Tool

When developing and evaluating data centre management techniques, it can be extremely helpful to have a tool to visualize what is happening within the simulated data centre. We have developed a visualization tool that makes use of the machine-readable trace output of DCSim to provide a set of graphs describing the simulation run in detail. Furthermore, it includes an animation, allowing the state of hosts and VMs in the data centre to be viewed as the simulation time progresses. Host and VM resource utilization are presented, and VM migrations and new instantiations are clearly shown. This allows the researcher to visually see how a management system or algorithm is operating, and to gain new insight into its behaviour.

7.5 Evaluation

In this section we demonstrate how DCSim can be used to implement and evaluate a management system, and use three different (though similar) management systems as working examples. We first describe the elements of these management systems (such as autonomic manager capabilities, policies, and events), discuss the changes that were made from one system to the next, and later compare the systems through simulation.

7.5.1 Data Centre Infrastructure

The target infrastructure consists of a collection of hosts and a DataCentre abstraction that contains all of the hosts. Each host has an associated *Autonomic Manager* (AM), as does the data centre. In the next sections we will discuss the capabilities of these managers and their associated policies.

7.5.2 Management Systems - Common Elements

Each host in the data centre has an AM associated with it. This manager possesses a capability, namely *HostManager*, that acts as a knowledge base for the manager, storing all relevant management information that the policies may need to successfully execute. One such policy is the *HostMonitoringPolicy*, which upon invocation collects the current status information of the host (resources in use or allocated, power consumption, number of incoming and outgoing VM migrations, etc.), packages the information in a *HostStatusEvent* message, and sends the message to the data centre's AM. The *HostMonitoringPolicy* requires the *HostManager* capability, so as to be able to access the host and collect the necessary status information.

Another policy installed in every host's AM is the *HostOperationsPolicy*. This policy defines the behaviour of the manager upon receiving the events *InstantiateVmEvent*, *MigrationEvent* and *ShutdownVmEvent*. These events trigger the allocation of the resources requested for the VM in the host, start a migration process, and stop and deallocate a VM, respectively.

At installation time, the *HostMonitoringPolicy* is configured to be triggered every 5 minutes. This behaviour is achieved by creating a *RepeatingPolicyExecutionEvent* with a periodicity of 5 minutes and specifying the host's AM as intended target. When the manager receives the event (once every 5 minutes), it triggers the associated policy.

The data centre's AM possesses the *HostPoolManager* capability, which serves to store information about a collection of hosts (in this case, all the hosts in the data centre). In the following sections we will discuss the policies that are installed in this AM.

7.5.3 Static Management System

The Static Management System allocates VMs in the data centre according to their expected peak resource demand, allocating to each incoming VM the total resources requested at creation time and never modifying that allocation. This is achieved through a single management policy, which is installed in the data centre's AM. This policy is a *VM Placement policy*, which defines how to perform the placement of incoming VMs onto hosts. Every time a *VmPlacementEvent* is received, the data centre's AM invokes the VM Placement policy. This policy implements a greedy algorithm to place the incoming VM in the first host that has enough resources available to fit the VM without over-committing resources. If one such hosts is found, then the search is terminated and an *InstantiateVmEvent* is sent to the host. Otherwise, the VM Placement fails and the client request is rejected. The policy relies on the manager's *HostPoolManager* capability to get status information about all the hosts.

Another policy installed in the data centre's manager is the *HostStatusPolicy*. This policy is invoked every time a *HostStatusEvent* is received. The policy stores the new host status information in a data structure in the *HostPoolManager* capability of the data centre AM.

7.5.4 Dynamic Periodic Management System

The Dynamic Periodic Management System maps VMs into hosts based on their current resource needs. Resources such as memory, bandwidth and storage are statically allocated and never change, but the CPU is *oversubscribed*, therefore allowing the system to map more VMs to a host than is possible with the Static Management System.

Like the Static Management System, the *VM Placement policy* installed in the data centre's AM is invoked upon reception of a *VmPlacementEvent*. This policy is similar to the one used in the Static Management System, but since this system leverages CPU oversubscription, the policy does not require the hosts to have unallocated CPU for the incoming VM, but the policy rather checks how much CPU is actually in use in the host, and if there is enough CPU not in use, then the VM can be placed on the host. As mentioned before, the system places VMs into hosts based on the their current resource needs. At creation time, the requested resources are taken as the current resource needs of the VM.

By oversubscribing resources, the management system can increase the resource utilization of the hosts, and therefore of the data centre as a whole. However, this strategy increases the risk of hosts becoming *stressed*. A *stress situation* occurs when the combined demand of the VMs co-located in a host exceeds the resource capacity of the host. When this happens, one or more VMs have to be migrated to another host, so as to free resources locally to satisfy the resource demand of the remaining VMs.

The management system uses a *VM Relocation policy* to determine which VMs to migrate away from a stressed host and to choose a new host for the migrating VMs. The policy is configured at installation time to run periodically every 10 minutes. When invoked, the policy first checks the set of hosts to determine which, if any, are stressed. For each stressed host, the policy follows a greedy algorithm to select VMs for migration and to find target hosts in which to place the migrated VMs.

The management system also uses a *VM Consolidation policy* to periodically consolidate VMs in the data centre, attempting to minimize the number of physical servers that need to be powered on to host VMs. This policy is installed in the data centre's AM and is configured to be invoked every hour. Upon invocation, the policy uses a greedy algorithm to migrate VMs away of underutilized hosts and into hosts with higher resource utilization. Hosts that are emptied of VMs are then suspended or powered off, to conserve power.

The same *HostStatusPolicy* used in the Static Management System is used here to process *HostStatusEvent* messages and maintain up-to-date status information about the hosts in the data centre.

7.5.5 Dynamic Reactive Management System

The Dynamic Reactive Management System is very similar to the Dynamic Periodic Management System, except that it triggers its *VM Relocation policy* on demand rather than periodically. The *VM Relocation policy* itself is essentially the same, with minor changes implemented to allow the policy to run as frequently as required rather than periodically.

The Reactive system attempts to detect stress situations and trigger VM migrations as soon as possible, so as to reduce the SLA violations suffered by VMs co-located in stressed hosts. In order to achieve this behaviour, a new *HostStatusPolicy* (i.e. different from the corresponding policy from the Dynamic Periodic Management System) is necessary. This policy, known as *ReactiveHostStatusPolicy*, is still invoked upon receipt of a *HostStatusEvent* and is still responsible for updating hosts' status information. However, once the status information of the host associated with the event is updated, the policy issues a *VmRelocationEvent* so as to invoke the *VM Relocation policy*.

Upon invocation, the new *VM Relocation policy* first queries the *VmRelocationEvent* to obtain identification information of the host whose status information was recently updated. The policy then performs a stress check on the host. If the host is stressed, the policy looks for VMs to migrate away from the host and for target hosts to receive the migrated VMs. If the host is not stressed, the policy terminates its execution.

7.5.6 Experimental Setup

The simulated data centre for these experiments consists of 200 hosts, divided equally between two types: *small* and *large*. The *small* host is modelled after the HP ProLiant DL380G5, with 2 dual-core 3GHz CPUs and 8 GB of memory. The *large* host is modelled after the HP ProLiant DL160G5, with 2 quad-core 2.5GHz CPUs and 16GB of memory. The different types of host have different power efficiency, which is calculated as *CPU capacity / power consumption at 100% utilization*. The power efficiency of the *large* host is 85.84 cpu/watt, while the power efficiency of the *small* host is 46.51 cpu/watt.

We use three types of VMs in these experiments. The *small* VM requires 1 virtual core with 1500 CPU units (minimum), plus 512MB of memory. The *medium* VM requires 1 virtual core with 2500 CPU units (minimum), plus 512MB of memory. The *large* VM requires 2 virtual cores with 2500 CPU units each (minimum), plus 1GB of memory. These descriptions correspond to the resource requirements of the VMs at creation time. Once a VM is running in

	Host Util.	# Migs	Power	CPU Underprovision	Failed Placement
Static	46%	0	7221kWh	0.0%	24%
Periodic	80%	10261	5056kWh	0.109%	0%
Reactive	79%	12508	5121kWh	0.059%	0%

Table 7.2: Management Systems Comparison

the data centre, further placement and allocation considerations are made based on the actual resource *usage* of the VM. These experiments include an equal number of each type of VM.

The experiments are configured to create 600 VMs in the first 40 hours of simulation. These VMs remain throughout the entire experiment, so as to maintain a minimum level of load in the data centre. In the third day of simulation, new VMs begin to arrive; they do so at a changing rate and last for about a day. The total number of VMs in the data centre changes daily, using randomly chosen values uniformly distributed between 600 and 1600. This second set of VMs provides for a dynamic load in the data centre.

We use the term *workload pattern* to refer to a randomly generated collection of VM instances with arrival, departure, and trace offset times. A *workload pattern* can be repeated by providing the random seed with which it was first generated.

7.5.7 Results

We evaluated the three proposed management systems through simulation using DCSim. We generated 10 different *workload patterns* and evaluated each management system under each of these *workload patterns*. The experiments lasted 10 simulated days, though only the last 8 days of simulation were recorded; the first 2 days were discarded to allow for the system to stabilize before recording results. Table 7.2 presents the results for each management system, averaged across the different *workload patterns*.

We can see that the Static Management System achieved the lowest host utilization by far, which translated also into the highest power consumption. However, given that VMs are statically allocated their total resource request (enough to meet their peak demand), the management system avoids CPU underprovisioning completely. It should be noted, however, that such a conservative approach to resource allocation resulted in an elevated percentage of failed placements, while the other management systems were able to accept every VM creation request.

Both Dynamic Management Systems achieved similar results, with Periodic showing slightly higher host utilization (and therefore less power consumption) and Reactive lowering CPU underprovisioning by about 40%. However, Reactive's reduction of CPU underprovisioning was achieved by triggering VM migrations as soon as hosts became stressed, which resulted in a 20% increase in the total number of VM migrations issued.

7.6 Conclusions and Future Work

Developing and evaluating data centre management techniques on the scale that they are ultimately required to perform at presents a significant challenge. As such, most work turns to simulation tools for their experimentation. We have presented DCSim (Data Centre Simulator), an extensible simulation framework for simulating a data centre operating an Infrastructure as a Service cloud. DCSim allows researchers to quickly and easily develop and evaluate dynamic resource management techniques. It introduces key new features not found in other simulators, including an interactive application model which allows the simulation of interactions and dependencies between VMs, VM replication as a tool for handling increasing workload, and the ability to combine these features with a work conserving CPU scheduler. The simulator can also be easily extended and customized for specific work in a range of key research topics in data centre management. Finally, we have presented an example use-case of the simulator, comparing three different VM management systems, to demonstrate the usefulness of the simulation results.

A number of additional features are planned for DCSim. An HPC/batch style application model should be included, as data centres typically host both interactive and HPC workloads. VM migrations are an important aspect to dynamic VM management, and their overhead needs to be considered in as accurate a manner as possible. We plan to include a more detailed modelling of migration bandwidth, and the impact of multiple simultaneous migrations on both migration time and SLA metrics, using our new model of data centre networking. Finally, the thermal state of the data centre should be considered and used to calculate cooling costs, as cooling power represents a significant cost for data centre operations.

Bibliography

[1] Amazon. Amazon web services. http://aws.amazon.com/, July 2014.

[2] Apache. Apache log4j. http://logging.apache.org/log4j/, July 2013.

[3] The Internet Traffic Archive. The internet traffic archive. http://ita.ee.lbl.gov/, July 2014.

[4] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010.

[5] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50, January 2011.

[6] Standard Performance Evaluation Corporation. Specpower_ssj2008 benchmark. http://www.spec.org/power_ssj2008/, July 2014.

[7] S.K. Garg and R. Buyya. Networkcloudsim: Modelling parallel applications in cloud simulations. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 105–113, 2011.

[8] S.K.S. Gupta, R.R. Gilbert, A. Banerjee, Z. Abbasi, T. Mukherjee, and G. Varsamopoulos. Gdcsim: A tool for analyzing green data center design and resource management techniques. In *Green Computing Conference and Workshops (IGCC), 2011 International*, 2011.

[9] Google Inc. Google cluster data. http://code.google.com/p/googleclusterdata/, July 2014.

[10] D. Kliazovich, P. Bouvry, Y. Audzevich, and S.U. Khan. Greencloud: A packet-level simulator of energy-aware cloud computing data centers. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, 2010.

[11] Seung-Hwan Lim, B. Sharma, Gunwoo Nam, Eun Kyoung Kim, and C.R. Das. Mdcsim: A multi-tier data center simulation, platform. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 2009.

[12] Ns-2. Ns-2. `http://nsnam.isi.edu/nsnam`, July 2014.

[13] Michael Tighe, Gastón Keller, Michael Bauer, and Hanan Lutfiyya. DCSim: A data centre simulation tool for evaluating dynamic virtualized resource management. In *SVM Proceedings, 6th Int. DMTF Academic Alliance Workshop on*, October 2012.

[14] Michael Tighe, Gastón Keller, Michael Bauer, and Hanan Lutfiyya. Towards an improved data centre simulation with DCSim. In *SVM Proceedings, 7th Int. DMTF Academic Alliance Workshop on*, October 2013.

[15] Sungkap Yeo and H. H.S. Lee. SimWare: A Holistic Warehouse-Scale Computer Simulator. *Computer*, 45(9):48–55, 2012.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The owner of a data centre operating an Infrastructure as a Service cloud (the *cloud provider*), has two primary goals: reduce costs, and provide service to their customers (the *cloud clients*). In this work, we have looked at reducing costs through minimizing power consumption, and providing service to customers by minimizing SLA (Service Level Agreement) violations. This basic premise leaves room for a large number of challenges, in both the implementation of these basic objectives and the addition of secondary goals. More specifically, we examine the use of a dynamic approach to management, constantly adapting resource allocations to match current workload demands.

This thesis addressed several challenges in dynamic virtualized cloud management. Each contribution examined a specific sub-problem, all falling into the scope of dynamic management and striving towards developing a more comprehensive set of techniques for cloud management. In our initial work, we evaluated a set of variations on an algorithm for dynamic management, and found that there is no single algorithm that can be considered "best". Rather, each variation has its strengths and weaknesses, representing trade-offs between different management goals. In order to address the problem of achieving more than one goal, we then proposed a method of dynamically switching between different management strategies. We also noticed that management required a large number of monitoring messages to be sent to a single central manager, which was responsible for managing the entire data centre. This represents a potential limitation for scalability. As such, we developed a distributed version of our dynamic management algorithm, which distributed the decision making among all hosts, and eliminated the requirement of monitoring messages. Up until this point, we considered applications running within a single virtual machine (VM). In reality, however, many applications are deployed in a set of VMs, with each VM running a component of the complete application.

Furthermore, in order to take advantage of pay-per-use pricing in the cloud, applications typically scale the number of VMs in use up and down to match their current workload. We looked at how this operation interacts with our dynamic management algorithms, and developed an integrated algorithm to handle both application scaling and dynamic management. Finally, once we examined deploying applications spanning multiple VMs, it became necessary to consider where in the data centre the components of the application were placed. This lead us to extend our algorithm to be topology-aware.

Chapter 2 defined the core problem, and presented a basic approach and set of algorithms for dynamic VM management. We define four states that a host can be in: *stressed*, when host CPU utilization is high and hosts VM performance is at risk; *partially utilized*, when host CPU utilization is in a normal, desirable range; *underutilized*, when host CPU utilization is low and therefore the host is not power efficient; and *empty*, when the host contains no VMs. Next, we defined three primary management operations: *VM Placement*, *VM Relocation*, and *VM Consolidation*. The *VM Placement* operation is responsible for the initial placement of a new VM in the cloud (data centre). The *VM Relocation* operation relieves *stressed* hosts, by selecting a VM from the host to migrate to another host. Finally, the *VM Consolidation* operation attempts to remove VMs from *underutilized* hosts, in order to consolidate load onto fewer hosts. We presented first-fit heuristic algorithms to perform these operations, with variations to tune their behaviour to achieve different goals. The remainder of the work presented here is based on these core concepts.

In Chapter 3, we introduced a method of pursuing more than one goal simultaneously. We look at two, conflicting goals for the cloud provider: minimizing power consumption, and minimizing SLA violation. Most current work focuses on a single goal, with other goals considered secondary, at best. We developed two management *strategies* (a set of management operation policies) designed to pursue a single goal. One focused on *SLA* and the other on *Power*. We then introduced a method of dynamically switching between these strategies, based on the current data centre state, in order to use the most appropriate strategy at a given time. This approach was able to improve overall performance when compared to individual strategies.

Most of the work on dynamic virtualized cloud management has implemented a centralized architecture. In Chapter 4, we presented an alternative, distributed approach to management. Given the large scale and highly dynamic nature of the cloud environment, a centralized manager with global knowledge is unlikely to be sufficiently scalable. Our distributed algorithm is capable of replicating the performance of the centralized version, while reducing the networking overhead of sending frequent status messages to a central manager and distributing the management computation load.

In Chapter 5, we looked at integrating the goals and operations of the *cloud client* into dynamic management. The cloud client deploys applications in the cloud, which are not simply contained in a single VM, but can consist of a set of communicating VMs. In order to reduce infrastructure costs in the face of a highly dynamic workload, the application should scale up and down (by adding and removing VMs) in order to provision only the resources required to meet demand, as required. This operation is referred to as *autoscaling*. We examined how applications performing autoscaling while the data centre performed dynamic VM management impacted performance. Finally, we presented a novel, integrated autoscaling and dynamic VM management algorithm, which uses some control over how autoscaling operations are carried out to assist dynamic management. Through the use of this algorithm, we were able to correct a drop in SLA performance incurred when running the two separate algorithms, and to significantly reduce the number of migrations required for management. Since migrations have a cost in terms of network bandwidth usage and VM performance overhead, minimizing the number of migrations performed is an important goal.

Typically, work on dynamic virtualized cloud management treats the data centre as a flat collection of hosts. When deploying applications consisting of multiple, communicating VMs, however, the network topology of the data centre becomes important. In Chapter 6, we proposed a topology-aware dynamic VM management algorithm, which takes the network topology of the data centre into consideration. The algorithm attempts to place all VMs of a single application within a single *rack* of the data centre, and to maintain this constraint through all dynamic management operations. This reduces the network latency between communicating VMs, and reduces network traffic in higher level links. Furthermore, it reduces the number of migrations that are performed between racks, again reducing the load on higher level network elements.

Finally, in Chapter 7, we presented a simulation tool, DCSim, for simulating a virtualized, multi-tenant data centre operating an Infrastructure as a Service cloud. Experimentation with a real system is not feasible, due to the scale and complexity of the target deployment. Therefore, simulation is used to evaluate algorithms and techniques for dynamic virtualized cloud management. The development of this tool was in response to a lack of available, open-source tools designed for this particular target environment. It is available online, as an open-source project, and we are actively encouraging other research groups to make use of DCSim for their own research.

Cloud computing is proving to be a huge driving force for change in computing today. Ever more applications are being deployed in the cloud, where infrastructure can be provisioned on-demand, in a pay-as-you-go fashion. There remains a good deal of work to perfect this approach, however. We have presented a number of novel advancements to managing a cloud data centre, and more still remain. There is no denying, however, that the move into the cloud has gained too much momentum to be stopped, and the cloud will continue to dominate in the near future. That is, until the *next thing* arrives.

8.2 Threats to Validity

There are several possible threats to the validity of the work and results presented in this thesis. First, a number of simplifications were made in order to reduce the complexity of the problem and focus on specific aspects. Namely, we did not consider the networking requirements of VMs running within the cloud, nor their storage requirements. Considering networking and storage may have an impact on VM placement decisions and will need to be addressed within our algorithms. We have also made some assumptions regarding the overhead of performing VM live migrations. We did not consider the networking costs associated with migration, and we applied a fixed penalty rate to all VMs and hosts involved in a migration. In reality, the overhead associated with a migration is dependent on the behaviour and workload patterns of the VM. This may have an impact on the number of migrations which can feasibly be performed for dynamic management, as well as how many can be performed concurrently. In this work, we always considered performing fewer migrations to be a desirable goal, in order to account for this possibility.

During our evaluation, we considered interactive applications, such as a typical multi-tiered web application. Other types of applications, such as media streaming, MapReduce, or other batch processing tasks, can and will also be run in the cloud. The algorithms developed in Chapters 2, 3 and 4 were developed considering a VM as a black-box, and as such should be applicable to other types of applications. This, however, was not tested. The algorithms in Chapters 5 and 6 specifically address management of interactive applications. As such, a different workload mix would have an impact on the performance of the algorithms, especially on the integrated autoscaling and dynamic management algorithm. This could be mitigated, however, by placing different application types within subsets of a data centre, under management algorithms designed specifically for that workload.

Finally, we evaluated our work using simulation. As such, actual values from results (such as host utilization and power consumption) will vary compared to a real-world implementation. However, the evaluations were based on comparisons between different approaches, and these comparisons should hold. Another possible threat to the validity of the results is possible problems or inaccuracies within the simulation tool, DCSim. The tool is available online, so that other researchers can make use of the tool and attempt to repeat our results.

8.3 Future Work

Several challenges remain in the area of dynamic virtualized cloud management, before a complete, comprehensive solution can be achieved. In terms of short term, more immediate additions to the work presented in this thesis, there are a few logical next steps to be taken:

- consider more advanced placement constraints on VMs, e.g. for fault-tolerance and availability purposes. This would put additional constraints on dynamic management, making achieving its goals more challenging;
- provide bandwidth guarantees between communicating VMs;
- learn parameter values via machine learning algorithms, rather than determining values through time consuming experimentation, which may not remain valid under changes in workload;
- develop a distributed implementation of the more advanced techniques presented in Chapters 5 and 6;
- develop a better understanding of migration overhead through experimentation, and develop methods to reduce the number of migrations performed by management.

Stepping a bit further away from immediate goals and into potential medium range future work, there are a few more avenues to pursue:

- investigate dynamic memory allocation in addition dynamic CPU;
- consider storage, and the location of network storage, in placement decisions;
- incorporate knowledge of the thermal properties of the data centre to consider cooling costs in decision making.

Finally, the use of multiple data centres and sites should be investigated. Applications deployed in the cloud do not need to live within a single data centre, or even a single geographical region. In many cases, application components (and the VMs running them) can and should be placed as close as possible to their users. This entails not only a static placement of application components in geographically diverse locations, but also the potential to *move* these components as user locations change, for example, due to the day-night cycle.

Curriculum Vitae

Name:	Michael Tighe
Post-Secondary Education and Degrees:	University of Windsor Windsor, ON 2002 - 2007 B.Sc. Computer Science
	University of Western Ontario London, ON 2007 - 2009 M.Sc. Computer Science
	University of Western Ontario London, ON 2009 - 2014 Ph.D. Computer Science
Honours and Awards:	OGS 2011-2014
Related Work Experience:	Teaching Assistant The University of Western Ontario 2007 - 2013

Publications:

Integrating Cloud Application Autoscaling with Dynamic VM Allocation, Michael Tighe and Michael Bauer, *Network Operations and Management Symposium (NOMS)*, 2014, Krakow, Poland.

A Hierarchical, Topology-aware Approach to Dynamic Data Centre Management, Gaston Keller, Michael Tighe, Hanan Lutfiyya and Michael Bauer, *Workshop on Management of the Future Internet (ManFI)*, 2014, Krakow, Poland.

A Distributed Approach to Dynamic VM Management, Michael Tighe, Gaston Keller, Michael Bauer and Hanan Lutfiyya, *International Conference on Network and Service Management (CNSM)*, 2013, Zurich, Switzerland.

Towards and Improved Data Centre Simulation with DCSim, Michael Tighe, Gaston Keller, Michael Bauer and Hanan Lutfiyya, *Workshop on Systems and Virtualization Management (SVM)*, 2013, Zurich, Switzerland.

The Right Tool for the Job: Switching Data Centre Management Strategies at Runtime, Graham Foster, Gaston Keller, Michael Tighe, Hanan Lutfiyya and Michael Bauer, *International Symposium on Integrated Network Management (IM)*, 2013, Ghent, Belgium.

DCSim: A Data Centre Simulation Tool for Evaluating Dynamic Virtualized Resource Management, Michael Tighe, Gaston Keller, Michael Bauer and Hanan Lutfiyya, *Workshop on Systems and Virtualization Management (SVM)*, 2012, Las Vegas, USA.

An Analysis of First Fit Heuristics for the Virtual Machine Relocation Problem, Gaston Keller, Michael Tighe, Hanan Lutfiyya and Michael Bauer, *Workshop on Systems and Virtualization Management (SVM)*, 2012, Las Vegas, USA.

Policies and Abductive Logic: An Approach to Diagnosis in Autonomic Management, Michael Tighe and Michael Bauer, *International Journal on Advances in Intelligent Systems*, 2010.

Mapping Policies to a Causal Network for Diagnosis, Michael Tighe and Michael Bauer, *International Conference on Autonomic Computing*, 2010, Cancun, Mexico.

Other Presentations & Posters:

DCSim: A Data Centre Simulation Tool, Michael Tighe, *International Symposium on Integrated Network Management (IM)*, 2013, Ghent, Belgium. *Demo*.

A Distributed Approach to Dynamic VM Management, Michael Tighe and Gaston Keller, *Consortium for Software Engineering Research (CSER)*, 2013. *Poster*.

DCSim: A data Centre Simulation Tool for Evaluating Dynamic Virtualized Resource Management, Michael Tighe, *Consortium for Software Engineering Research (CSER)*, 2012. *Presentation*.

DCSim: A Data Centre Simulator, Michael Tighe, *IBM Canada CAS Research Conference (CASCON)*, 2012. *Technology Demo*.

Academic Involvement:

Technical Program Committee (TPC), *Workshop on Systems and Virtualization Management (SVM)*, 2013
Editorial Board, *International Journal on Advances in Intelligent Systems*, 2012
TPC, *International Conference on Autonomic and Autonomous Systems*, 2011-2012
TPC, *International Conference on Adaptive and Self Adaptive Systems*, 2010
TPC, *International Conference on Cloud Computer, GRIDS, and Virtualization*, 2010
Session Chair, *International Conference on Autonomic and Autonomous Systems*, 2010