Improved Virtual Machine (VM) based Resource Provisioning in Cloud Computing

by

Md. Mahfuzur Rahman

A Thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science
University of Manitoba
Winnipeg

Thesis advisor

Peter Graham

Author

Md. Mahfuzur Rahman

Improved Virtual Machine (VM) based Resource Provisioning in Cloud Computing

Abstract

To achieve "provisioning elasticity", the cloud needs to manage its available resources on demand. A-priori, static, VM provisioning introduces no runtime overhead but fails to handle unanticipated changes in resource demands. Dynamic provisioning addresses this problem but introduces runtime overhead. To avoid sub-optimal provisioning my PhD thesis adopts a hybrid approach that combines static and dynamic provisioning. The idea is to adapt an initial static placement of VMs in response to evolving load characteristics. My work is focused on broadening the applicability of clouds by looking at how the infrastructure can be more effectively used to support historically atypical applications (e.g. those that are interactive in nature with tighter QoS constraints). To accomplish this I have developed a family of related algorithms that collectively improve resource sharing on physical machines to permit load variation to be better addressed and to lessen the probability of VM interference due to resource contention. The family includes three core dynamic provisioning algorithms. The first algorithm provides for the short-term, controlled sharing of resources between co-hosted VMs, the second identifies pairs (and by extrapolation larger groups) of VMs that are predicted to be compatible in terms of the resources they need. This

allows the cloud provider to do co-location to make the first algorithm more effective. The final, third, algorithm deals with under-utilized physical machines by re-packing the VMs on those machines while also considering their compatibility. This final algorithm both addresses the possibility of the second algorithm creating underutilized machines as a result of pairing and migration and also handles underutilization arising from "holes" left by the termination of short-duration VMs (another form of atypical VM application). I have also created a surprisingly simple static provisioning algorithm that considers compatibility to minimize VM interference that can be used before my dynamic algorithms. My evaluation is primarily simulation-based though I have also implemented the core algorithms on a small test-bed system to ensure correctness. The results obtained from my simulation experiments suggest that hybrid static and dynamic provisioning approaches are both feasible and should be effective supporting a broad range of applications in cloud environments.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to begin by thanking my advisor, my committee, my parents, my wife, and all the people who have supported me along the way.

Chapter 1

Introduction

The growth in popularity of clouds for providing IT services continues at a rapid pace. Cloud-based approaches are attractive because enterprises do not need to purchase, maintain and support their own computing infrastructure which often represents a significant and sometimes hard to predict operational cost. Further, the large scale of cloud infrastructure (tens of thousands of cores and beyond) allows cloud providers to optimize the efficiency of the systems, via the use of virtual machines (VMs) to permit effective sharing, so they can offer services more cost-effectively than end users can provide their own. This makes the use of clouds price-attractive as well. End users and cloud providers typically enter into contracts, quantified as service level agreements (SLAs), to describe the expected requirements, in terms of computing, etc. capacity, required for each user "application" (i.e. virtual machine). Cloud providers use this information to determine the amount of infrastructure they need to purchase and later allocate to users when their VMs run as well as the cost to the users. This allocation is done by "provisioning" algorithms that pack virtual machines

running end-user applications into physical cloud infrastructure (compute nodes) that the cloud provides. For applications that have predictable and generally unchanging demands/workloads, it is possible to use "static provisioning" (where placement decisions are made before VMs begin execution and are unchanged throughout execution) effectively. In this approach VMs are carefully packed into clouds as their machines are brought online and continue to execute effectively for potentially very long periods of time on the selected nodes. Using static provisioning, the cost of making the placement decision is incurred once, before the user's applications are running, and therefore may safely incur relatively significant overhead to achieve a better result without negatively impacting the end users.

With the growing adoption of clouds as elements of business and research computing solutions, the diversity of work done using clouds is also increasing. Much work is still typified by long-running VMs executing OLTP (OnLine Transaction Processing) and other standard workloads but there are also an increasing number of VMs exhibiting significant load variation over time as more interactive and shorter duration tasks are now using cloud infrastructure "on-demand". This greater percentage of what might be termed "dynamic" use presents new challenges to cloud provisioning. Packing well-characterized, "static" virtual machines (VMs) can be done very efficiently since more time may be devoted to the packing process (to minimize power consumption, resource usage and the like). As the VM mix becomes less regular and as VMs run for shorter duration, however, packing decisions need to be made quickly on the fly and we must be able to adapt to changing workload characteristics throughout the cloud. To achieve effective provisioning across the spectrum of VM

requirements, in this thesis, I describe a collection of static and dynamic provisioning algorithms that can be effectively employed together to meet the needs of both traditional and newer cloud-based applications.

I first developed a "static" VM provisioning strategy that addresses the challenges introduced by new cloud-based applications to the extent that a static algorithm can. This new algorithm makes clear the limitations of a conventional static approach but also serves as a starting point, filling a small part of a spectrum of cloud provisioning techniques that will need to be used concurrently to ensure effective resource utilization as well as quick response to evolving cloud workloads. The focus of the algorithm is on better handling of VMs with short life times and on being responsive to urgent VM requests (those that require early/immediate start times). Providing better support in these areas will increase the appeal of cloud environments for a broader range of user applications and will decrease the need for frequent dynamic re-provisioning later which will incur **run-time** costs. As such, the work is primarily concerned with making provisioning decisions quickly and effectively.

A strict focus on VM packing alone to maximize resource utilization as is common with conventional static provisioning algorithms ignores some aspects of the "compatibility" of co-hosted VMs such as which VMs will execute more effectively together on a physical machine (PM) given that they must compete for its resources. There are different forms of compatibility but most are focused on execution performance. For example, it is possible that two VMs on the same PM may have been packed together based on SLA values that indicate that they should run well together given the available resources but, due to unforeseen load variation, they collectively over-

consume one or more types of resources at some point(s) during their execution. This situation is known as 'VM interference' and is an important problem that needs to be addressed to ensure the efficiency of cloud operation and also user satisfaction. Thus, it is attractive to try to co-locate VMs that are likely to be compatible in terms of their **varying** resource requirements (i.e. short-term behaviour) as well as being able to be efficiently packed into the selected PM based on their SLA values. If this can be done then it is possible for the "multiplexed" VMs to more effectively share the available PM resources thereby offering the promise of greater user satisfaction without additional expense to the cloud provider. There are at least three possible approaches to such "compatibility-based" VM placement/ provisioning. When applied to a static placement algorithm, the cloud provider must rely on some sort of user characterization of VM workload behaviour to assess compatibility. As cloud users are often not technically sophisticated this means that a very simple characterization is necessary. This will limit the broad effectiveness of static compatibility-based placement algorithms but may still prove useful for a subset of VMs and users. When considering dynamic placement during VM execution, knowledge of current and some recent historical VM behaviour is available. This makes techniques that rely on predictions of future VM behaviour and that use live migration [24] of VMs to 'adjust' VM placements attractive. Such dynamic approaches are likely to be effective for VMs of reasonably long duration that are characterized by predictable and somewhat sustained changes in resource requirements over time. Between these two extreme approaches is a third approach based on recording the behaviour of a VM during one run (possibly a test run) and using it as feedback information to characterize that

VM's behaviour for a later run. This could be done using an automated mechanism that replaces the need for user characterization which is obviously desirable. Such an approach would have to be able to recognize re-runs of VMs for which information had previously been collected (i.e. some would require means of mapping VMs to the particular tasks they perform). This approach also has some limitations (for example some VM tasks may have different resource requirements based on their input data) and thus cannot hope to match the flexibility of a fully dynamic technique but can eliminate user involvement during static placement. This will prove useful in cases where multiple VMs are executed repeatedly that run the same task on similar data (e.g. end of day processing, periodic analysis of stream data, etc.). I therefore enhanced my initial static placement algorithm to support static 'compatibility based' VM placement using a very simple metric for determining "resource critical" (RC) virtual machines that should be co-located with non-RC virtual machines.

In cases where demand by applications may vary significantly over time, "dynamic provisioning" techniques have been suggested whereby VMs may be migrated on-the-fly to new compute nodes within the cloud if it is not possible to support the required load changes within existing physical machines. Workload changes can be categorized by expected duration into short-lived and longer-lived and both types of changes need to be handled efficiently. Dynamic provisioning allows the system to adapt to changing conditions but at the cost of runtime overhead and potential execution delays that may negatively affect end users. To minimize this overhead, restrictions on the complexity of provisioning algorithms are necessary. To effectively address scenarios with dynamically changing load, I introduce a hybrid provisioning approach

that starts from a good static placement that is subsequently tuned by dynamic re-provisioning (using the three dynamic algorithms mentioned earlier) as long as doing so is effective and not excessively costly. As the overhead of adjustment for workload changes in existing nodes or in new nodes increases, re-provisioning becomes less attractive. At that point, if needed, we can incur the greater cost of some form of static re-partitioning (of potentially all running VMs) based on the newly available demand information to return the system to a state of efficient sharing, execution, and better packing. This could be done concurrently with the ongoing execution of VMs in the cloud as long as there are available idle resources – a likely situation or one that can be planned for. Such overall static re-partitioning is not considered in this thesis.

The process of VM provisioning is particularly complicated when applications that are highly interactive in nature and have tight SLA constraints are hosted within a cloud. Consider, for example, an interactive consultation system where a patient and his/her doctor (in a remote location) meet electronically with a number of consulting specialists to discuss treatment plans and review diagnostic imagery (possibly of multiple types). In such a scenario the activities carried out are determined by the participants and their actions may significantly affect not only the SLA constraints of the interaction but also the demands on the cloud hosted application components. With such frequent and unpredictable workload changes, migrating the corresponding VM to a new node may prove to be either inappropriate since the demand change may be short-lived or incur unnecessary overhead. Thus, it would be better to handle such workload changes in existing compute nodes, if possible. Such short-lived

load changes can be more effectively handled if there exists sufficient under-utilized resources in the existing compute node. In a multiplexed environment not all VMs on a given PM may need to use their allocated resources at all times. Hence, any unused resources can be exploited by *temporarily* overloaded VMs to avoid VM migration due to a *short-term* SLA violation. To support this, I have developed a priority based resource sharing provisioning policy for scheduling VMs (or applications within a VM). The prioritization can be applied between the VMs (or applications) so that resources can be allocated/reallocated to VMs/ applications based on assigned priorities. Depending on the priority, the VMs (or applications in different VMs) either borrow resources from other VMs for a certain period (in the case of dedicated resource allocation) or may share the same resource (e.g. processor core, etc.) but each can be assigned a different (but fixed) time slice for use. Of course, the effectiveness of such a technique will depend on the compatibility of the VMs co-hosted on a given PM.

When load changes can not be supported in an existing node, it is necessary to find a suitable alternative compute node to which to migrate either the overloaded VM or any other multiplexed VM, if possible, to allow the overloaded one to execute more effectively in-place. In the selection of VM to migrate and also in selecting the destination node there is a trade-off between cost due to migration and long term benefit to the cloud system. When the existing VMs in a cloud are monitored for a longer period of time it may be found that some VMs' load increases occasionally and that some other VMs at the same time have available resources. In such a case, those VMs can be grouped together in the same physical machine(s) using what I will term "VM re-multiplexing" to better meet their collective needs and avoid

SLA violations. VM re-multiplexing always incurs some VM migration overhead but should be effective for those VMs that have reasonably continuous behavior and longer lease periods from the cloud provider. Finally, after monitoring VMs for a long period of time, if it is found that overall resource usage is not satisfactory (e.g. significant resources in different physical machines are unallocated), then the cloud provider can trigger dynmaic re-packing. Such re-packing can be done considering compatibility of VMs and will result in better PM utilization and the ability to, in some cases, power down unused PMs. The new dynamic provisioning techniques I have developed to deal with longer-lived load changes (i.e. VM re-multiplexing and re-packing collectively) enhance the effectiveness of VM provisioning in clouds.

To implement and assess hybrid static and dynamic provisioning strategies, it is necessary to have both a testbed environment for verifying the correctness and small-scale performance of the developed algorithms as well as a simulation system to assess larger-scale characteristics. I have made extensions to the Eucalyptus [1] open source cloud system to support hybrid static and dynamic provisioning. Later in this thesis, I will discuss my hybrid algorithm framework and then discuss my algorithm evaluations using CloudSim [18] based on the framework and results from my modified version of Eucalyptus. Through simulation assessment, I will then show that my algorithms are able to improve overall resource provisioning in cloud systems for workloads including VMs with highly dynamic load variation as welll as short-lived VMs.

This thesis makes several contributions towards broadening the applicability of clouds by looking at how the infrastructure can be more effectively used to support

what might be thought of as historically atypical applications (e.g. those that are interactive in nature with tighter QoS constraints). A family of related provisioning algorithms is presented where two of the algoritms are static and three are dynamic. The presented static provisioning algorithms consider packing efficiency, and compatibility to minimize forms of VM interference. Among the dynamic provisioning algorithms, the first algorithm provides for the short-term controlled sharing of resources between co-hosted VMs, the second identifies pairs (and by extrapolation larger groups) of VMs that are predicted to be "compatible" in terms of the resources they need. This provides a mechanism whereby appropriate co-location can be done to make the first algorithm effective. The final, third, dynamic algorithm is focused on dealing with under-utilized physical machines by repacking the VMs on those machines while also considering their compatibility. This final dynamic algorithm both addresses the possibility of the second algorithm creating underutilized machines as a result of pairing and migration and also handles underutilization arising from "holes" left by the termination of short-duration VMs (another form of atypical VM application). These algorithms are primarily evaluated via simulation-based scenarios using the Google traces though the core algorithms were also implemented on a small test-bed system to ensure correctness.

The rest of this thesis is organized as follows. Chapter 2 reviews work related to my research. Chapter 3 identifies the specific problem to be solved. Chapters 4 and 5 describe my proposed solution strategy and my algorithms respectively. My assessment and evaluations are discussed in Chapter 6. Finally, Chapter 7 concludes my thesis and suggests some directions for future research work.

Chapter 2

Background and Related Works

2.1 Cloud Computing

2.1.1 An Overview

The growth in popularity of clouds for providing IT services continues at a rapid pace. Cloud-based approaches are attractive because enterprises do not need to purchase, maintain and support their own computing infrastructure which often represents a significant recurring capital cost and also a sometimes hard to predict operational cost. The large scale of cloud infrastructure (tens of thousands of cores and beyond) allows cloud providers to benefit from economies of scale both in purchasing and operation. Cloud providers optimize the efficiency of their systems via the use of virtual machines (VMs) to permit effective sharing of the resources provided by the physical machines (PMs) in their cloud data centers, so they can offer services more cost-effectively than end users could. This makes the use of clouds price-attractive.

Cloud computing [4; 80; 101] is based on the use of cost-effective large-scale computing infrastructure and introduces an alternative way to efficiently deliver IT services. Clouds can offer hardware resources (Infrastructure-as-a-Service/IaaS), software platforms (Platform-as-a-Servce/PaaS) and/or hosted applications (Software-as-a-Service/SaaS) on an on-demand basis. Clouds are mainly built on a huge amount of hardware resources (CPU, memory, network, etc.) in the form of a large number of compute nodes (i.e. physical machines) networked together to form a massive, efficiently sharable computing resource. These physical resources in the cloud are then made available in the form of Virtual Machines (VMs) to run the applications of the clouds users. In clouds, application developers, who are sometimes referred to as SaaS (Software as a Service) providers or cloud clients, pay to allocate as much computational resources: memory space, disk storage, network bandwidth, etc. as they need to host their applications based on their current and/or expected load from their end-users. Depending on SaaS providers' requirements, cloud infrastructure providers (i.e. IaaS providers) allocate the necessary resources to them. IaaS providers are called PaaS providers when they offer resources in the form of a particular software development platform (e.g. LAMP - Linux/Apache/MySQL/PHP) instead of raw hardware resources. Most PaaS providers allow application developers (i.e. SaaS providers) to develop, run, and host applications on their cloud. SaaS providers may also have their own IaaS infrastructure to host their services.

Efficient resource utilization is a key requirement in clouds from the provider's perspective and VMs are used in cloud computing to enable efficient resource sharing as well as enhanced security due to virtual machine isolation. Cloud computing

systems deal with a large number of VMs. High availability, low cost and meeting SLAs (Service Level Agreements) are the most important issues for VM management in cloud computing. Clouds allocate resources on-demand with the ability to provide required resources (with the illusion of nearly unlimited resource availability) to the client and to quickly support changes in client's resource requirements. On-demand resource utilization is a very important and attractive feature in clouds which reduces the need for planning ahead by cloud users for resource purchases (which may be required in the future) and thus reduces unnecessary upfront investment and can avoid both overutilization and underutilization of resources. Usage based billing is also a desired feature in clouds where each client's usage of resources (e.g. processors, memory, storage, etc.) is monitored and metered by the cloud provider. Paying for only what you use reduces operational expenses for cloud clients.

Depending on the underlying infrastructure, clouds are normally classified into three deployment models: public, private and hybrid. In a public cloud, services and resources are offered off-site over the Internet and can be made available to any organization. A public cloud provides better utilization, greater "resource elasticity", lower costs and higher scalability. Alternatively, a private cloud is a small-scale cloud environment where infrastructure is maintained within a private network. A private cloud is established with the goal to obtain full control and utilization of computing resources with increased security and reliability, typically for a single organization. A hybrid cloud combines the features of both public and private clouds (allowing a private cloud to grow its capacity, if needed, using the resources of a public cloud) and provides a convenient and effective way to integrate them.

2.1.2 Noteworthy Existing Cloud Systems

There are a number of cloud providers who have established cloud infrastructure and offer cloud services (including resources, applications, tools etc.). Unfortunately, most current enterprise cloud computing systems are proprietary and thus are not well suited for experimentation and/or instrumentation thereby making empirical assessment of new management algorithms difficult but these systems are still worthy of mention. There is a considerable challenge related to the need for standardization (like Unified Cloud Interface [25]) and the open source community has been very active and has developed numerous cloud computing frameworks. Some popular cloud systems (both proprietary and open-source) include:

- Amazon Web Services (AWS): AWS (established by Amazon) is a complete IaaS system that offers its resources to clients through web services. AWS includes two major components: Amazon Elastic Compute Cloud (EC2) [28], and Amazon Simple Storage Service (S3) [83]. EC2, an IaaS cloud, provides compute resources (i.e. computing power, memory, bandwidth, etc.) to customers allowing them to create VM instances pay-as-you-go. EC2 allows the customer to change their resource requirements and, naturally, also to control settings of the entire software stack (i.e. platform, applications etc.) on the allocated resource(s). Amazon's Simple Storage Service (S3) provides scalable storage with a REST/SOAP [105] interface to host data for associated applications dynamically.

- Google App Engine: Google App Engine [86] offers a PaaS infrastructure which provides a fully featured environment for application development, hosting and

running. The client (application developer) doesn't need to purchase and maintain computing resources but just needs to build, maintain and upload the applications using a simple web application framework called "webapp". Google App Engine's current runtime environment is based on the Python programming language but other languages are planned for. If the application needs additional resources (e.g. traffic capacity and data storage) Google App Engine detects and administers the scaling whenever needed. Google's App Engine also provides distributed data storage with a query engine and a python based data modeling API [100].

- Microsoft Windows Azure: Microsoft Windows Azure [77] is primarily focused on providing PaaS infrastructure for developers to host, scale, and manage web applications. Windows Azure currently supports only .NET framework and REST [71], HTTP, and XML based APIs but will provide support for others in the future. Windows Azure also provides IaaS services allowing users to create and maintain VMs with desired resource requirements. Additionally, Azure provides data storage services which can be accessed by ADO.NET or LINQ [73].

- Salesforce: Salesforce [104] is one of the pioneers in offering SaaS cloud products and serves as a good, representative example. Salesforce allows enterprise customers to add new applications on top of its Customer Relationship Management(CRM) [3] product. Salesforce also has extended its services by offering PaaS services to developers that wish to create applications (which may integrate third party services) to execute on the salesforce cloud platform. Salesforce

currently supports the Eclipse integrated development environment (IDE) [14] and APEX [98] (a new programming language). Resource scalability for changes in application workload is solely maintained by salesforce.

- Eucalyptus: Eucalyptus [66] is an open-source cloud computing system which provides an environment where researchers are encouraged to perform experiments with new ideas and where they are also allowed to improve the software infrastructure. Eucalyptus started as a research project at the University of California, Santa Barbara and was initially targeted for a private company who wanted their own cloud for their own use and to avoid user malice. Node controller, cluster controller, storage controller, and cloud controller are the basic building blocks of a cloud using Eucalyptus. Eucalyptus assumes a cloud environment with one or more clusters where each cluster may have a number of nodes/ PMs. Node controllers are used to create a number of VMs given the available resources on a node. The cloud controller is responsible for all the scheduling and allocation tasks in Eucalyptus. The cluster controller maintains interconnectivity among different nodes in a cluster as well as different VMs by creating a virtual network overlay. All the storage features in Eucalyptus are served by the storage controller. In Eucalyptus, all the API's reflect the design of Amazon EC2 [28] and Amazon S3 [83] providing for the possibility of reasonable, if imperfect, high-level comparison with results based on those real-world systems.

- OpenNebula: OpenNebula [63] is another open-source cloud platfrom. It was developed under a research project of the European Union. OpenNebula was

primarily developed as a private cloud with a huge amount of customizability. OpenNebula is suitable for any organization interested in a cloud of its own using VM technology and is ideal for those who want to setup a cloud with a few machines quickly. OpenNebula currently supports several virtualization tools including VMware [47], KVM [55] , and Xen [7] hypervisors. OpenNubula provides a command line interface which is based on XML-RPC [19]. The provided functionalities of OpenNebula can be classified into three different layers: tools, core and driver. The first layer ("tools") contains functions for a system administrator and users. These functions include a command line for administrators and a planning module to place VMs. The second layer ("core") contains functions to process user requests to control resources. The third layer ("drivers") contains functions to regulate data transfer and to control VMs without the help of hypervisors. OpenNebula's unencrypted NFS based storage sharing may cause security issues but this is probably less important in the case of a private cloud.

- AbiCloud: AbiCloud [42] provides a standard, open-source and highly interoperable cloud platform where applications can be easily transferred among different clouds and is suitable for use in a hybrid cloud. Interestingly, AbiCloud also allows users to duplicate an entire cloud and implement it elsewhere. There are three main components in AbiCloud: AbiCloud server, AbiCloud Web Services and Virtual System Monitor. AbiCloud server provides functionalities to manage the computer center and user interface. AbiCloud Web Services manages the virtual applications and Virtual System Monitor is used as a plug-in monitor

manager. AbiCloud provides a very user-friendly web-based cloud management tool which includes "drag & drop" features.

- Nimbus: Nimbus [64] is an open source cloud system that enables clusters to be easily converted into a cloud platform and offers its resources in the form of IaaS. Nimbus integrates Globus certificate credentials [15] which makes it ideal for use by the scientific community. Nimbus is highly customizable. There are four major components in Nimbus: workspace service, workspace control, workspace resource management and workspace pilot. Workspace service maintains web services with GSI (Grid Security Infrastructure) authentication [15]. Workspace control constructs OS/system images and communicates with hypervisors. Workspace resource management provides a solution to manage VMs. Workspace pilot provides functionality to enable virtualization in clusters available in the cloud.

- OpenStack: OpenStack [72] is a collaborative project to create an open source and open standard cloud infrastructure. OpenStack has a large community enriched by a great number of registered developers and contributing companies. (OpenStack is released under the terms of the Apache License [33].) OpenStack's initial focus is on IaaS offerings where it has three major components: OpenStack Compute, OpenStack Object Storage and OpenStack Image. OpenStack Compute deploys VM instances while OpenStack Object Storage provides redundant storage, and OpenStack Image performs the necessary operations (e.g. discovery, registration, etc.) for managing virtual disk images.

2.2 Virtualization

Virtualization is the software abstraction of physical resources. Virtualization is a foundational and key enabling technique in cloud computing. Virtualization enables the creation of multiple virtual machines in a single physical machine all sharing the physical resources that are available. In clouds, virtualization is used for easy, fast and efficient resource provisioning. Virtualization also takes care of many of the risks associated with VM co-residence etc. Virtualization provides flexibility in resource configuration and can enhance the security of resource sharing. Virtualization techniques can be classified into three categories: full-virtualization, para-virtualization and hardware-assisted virtualization. Full virtualization provides emulation of hardware interfaces to "guest" OSes (those being virtualized on top of the "host" OS) and requires no changes to guest OSes. KVM [55], and VMware [47] provide full-virtualization. Para-virualization provides similar but not identical hardware interfaces to guest OSes and requires some modification to the "guest" OSes. Xen [7], and HyperV [92] provide para-virtualization. Hardware assisted virtualization occurs at the hardware level by extending the legacy or processor architecture, for example, Intel's Vanderpool Technology [23] and AMD's Pacifica [38].

Hypervisors

In most cloud systems, hosted VMs are managed using a particular Virtual Machine Monitor (VMM) or "hypervisor". The hypervisor is the core component of virtualization in a cloud. The hypervisor acts as an intermediary between VMs and the hardware resources in a physical machine and there must be at least one hyper-

visor in each host machine to monitor all of its VMs. Currently there are various approaches and solutions for VM monitoring, each of which supports specific virtualization environments. For flexibility, it could be useful to have a common API which can communicate with different types of VMs and which can offer developers a virtualized abstraction layer for a set of VMs across different hypervisors. In the libvirt project, Bolte et al. [11] introduced such a common management API which can manage different hypervisors and VMs over a "well defined" interface. Bolte et al. described different design issues and the use of libvirt [12] with commercial hypervisors such as VMware ESX [96] and Microsoft's Hyper-V [92].

In a cloud, CPU scheduling of VMs is normally managed by hypervisors using a CPU scheduler. There are two types of hypervisors: Type-1 (e.g. Xen, VMware ESX) and Type-2 (e.g. Virtual Box, VMware workstation, etc.). The Type-1 hypervisor (also called native hypervisor) has its own CPU scheduler whereas the Type-2 hypervisor (also called hosted hypervisor) doesn't have its own scheduler but rather depends on the host operating system task scheduler. CPU schedulers of most Type-1 hypervisors normally provide support for work-conserving (WC) mode and/or non work-conserving (NWC) mode. In WC mode, the CPU scheduler guarantees proportional CPU share for each VM and if there is any unused CPU resource available (due to unassigned CPU share or due to any inactive VM), then that amount is automatically distributed to active/running, collocated VMs. But, in NWC mode, the CPU scheduler ensures the desired CPU share for each VM but doesn't distribute unused or unallocated CPU shares to active/running, collocated VMs. A brief review of the key characteristics of some widely used hypervisors is given below:

- Xen: Xen [7; 23] is an open source hypervisor developed at the University of Cambridge. Xen uses para-virtualization to allow multiple VMs to run on a single physical machine. Xen hosts a linux device driver as the initial or host domain named "Domain0" and this Domain0 is used to manage (create, run, terminate, migrate) other guest domains. Xen uses synchronous and asynchronous events to maintain communication between guest domains and the host domain. Guest domains call synchronous events ("hypercalls") to send messages to Xen. Xen uses asynchronous events to send notifications to guest domains.

- KVM: KVM [55] is another open source hypervisor which uses full-virtualization. KVM integrates virtualization capability with the traditional kernel and user modes of linux. KVM uses a new process mode (named "guest") to provide the virtualization and allows guest OSes to be hosted. The new process mode also enables communication between KVM and guest OSes. KVM operates using two components: a kernel module and user-space component. The kernel module manages the virtualization of hardware and helps KVM to schedule VMs while the user space component simulates the behaviour of I/O operations and injects interrupts to guest OSes.

- VMware: VMware [47] is a proprietary hypervisor which can either run directly on host's hardware (i.e. does not need additional underlying operating system to run) or run within a host's conventional operating system. VMware virtualizes, aggregates and manages resources available on a host and creates a resource pool to provide a simple way to get control over those resources.

VMware creates VMs and uses a special driver "VMDriver" which is installed in the host operating system kernel and which allows guest VMs to have faster access to devices in the system. VMware uses "VirtualCenter" and "VMware DRS" to optimize resource allocation, and to monitor and balance resource utilization across the resource pool following resource allocation rules. VMware's VMotion [30] can relocate running VMs to a different physical machine without interruption. This is an implementation of "live migration" which will be discussed and applied later in this thesis.

2.3 The Provisioning Problem in Clouds

2.3.1 Virtual Machine (VM) placement strategies

Provisioning in clouds which is the process of assigning virtual machines to appropriate physical machines (also known as hosts) for execution is a placement problem. Placement problems are often presented as a variant on the classic bin packing problem [34] and they occur in operating systems to schedule processes among CPUs, to pack memory, or to place files on disks. They also occur in networking to place replicas of web contents in different geographical locations as well as in a wealth of other application areas.

Basic static VM provisioning algorithms can be categorized into online and batch placement algorithms. The former place VMs when they are first created and the latter gather/batch a number of VM allocation requests and then place them concurrently to achieve greater packing efficiency. All static placement algorithms make

the placement decision and then no changes to location are subsequently made. In dynamic placement, a VM may be moved after its initial placement if required. A discussion of key algorithms for both types of static placement as well as dynamic placement will be provided in Section 2.5.

2.3.2 Cloud Service Level Agreements (SLAs) and Violations

VM placement decisions are made, at least in part, using information contained in Service Level Agreements (SLAs). In general, there are three typical parties involved in the use of clouds: infrastructure resource providers (IaaS), business service providers and end-users. IaaS providers charge business service providers for renting computing resources and business service providers charge end-users for processing their requests. Service level agreements between the parties outline expected requirements, levels of performance and associated costs. Nguyen et al. [62] proposed a two-level architecture to separate SLAs related to computing resources and application services. In both cases, providers need to ensure quality, reliability, availability and performance of their resources or services to consumers. Normally, providers and consumers (i.e. either resource providers and service providers or service providers and end-users) create an agreement regarding resource usage or service consumption. Such a Service Level Agreement (SLA) may, if necessary, be achieved through negotiation between providers and consumers. The providers always need to satisfy consumer's expectations (to avoid violations) while provisioning resources or services. This requires an efficient SLA-aware resource provisioning strategy for IaaS providers.

SLA monitoring, which allows the detection of violations, is a continuous process

which should not be inclined towards either providers or consumers and which needs to incorporate and consider consumer feedback. Romano et al. [82] proposed a SLA monitoring tool - QoS-MONaaS ("Quality of Service MONitoring as a Service") to provide reliable monitoring facilities as a service. QoS-MONaaS allows providers and consumers to provide SLAs with key indicators and alerts both of them when it detects any SLA violation. CloudWatch [85], CloudKick [43], Monitis [46], and LogicMonitor [45] are also commercial SLA monitoring frameworks for clouds. Ayad et al. [5] proposed an agent based SLA monitoring framework that can monitor, detect and automatically cope with SLA violations. Emeakaroha et al. [29] proposed the DesVi architecture which can monitor and detect SLA violations and at the same time can take reactive actions automatically using a knowledge database. In general, monitoring performance relative to SLAs is required to support decisions regarding any adjustments to placements (e.g. via dynamic provisioning).

2.3.3 VM Multiplexing

VM multiplexing occurs whenever multiple VMs are co-hosted in the same physical machine. The resources of a physical host are allocated to each of its multiplexed VMs according to their requirements for usage. VM multiplexing is most beneficial if VMs are multiplexed in a host considering certain key characteristics of the resource usage of VMs (e.g. so that VMs having highly probable complementary CPU demands are multiplexed in the same host). It is not sufficient to consider only the past resource usage behavior of VMs individually. It is better to consider them all at the same time to find better multiplexing possibilities. Meng et al. [60] proposed a VM multiplexing

approach where multiple VMs are consolidated and provisioned in a host based on an estimation of their aggregated resource requirements. Chen et al. [20] proposed a new VM sizing approach called "effective sizing" which can be used to choose VMs for multiplexing in a host. The "effective sizing" of a VM depends not only its own resource usage but also on the usage of multiplexed VMs over a time period. In clouds, we are naturally interested in the SLA requirements and actual resource use of multiplexed VMs as well as individual VMs.

Of course, for both static and dynamic provisioning, "sizing" of the required resources must be done to determine the appropriate hardware resources needed. Meng et al. [60] introduced joint VM sizing which can aggregate the total capacity requirements of multiple VMs. Joint VM sizing is useful when *multiple* VMs are hosted in the same physical host and the unused resources, identified by previous resource usage patterns and/or future usage prediction, of one VM can then potentially be allocated to other VM(s). Total capacity measurement is used to select the appropriate VMs to host in a particular physical machine. In static VM provisioning, the resources that are allocated to a VM may not be fully utilized and in that case VM multiplexing helps to improve utilization of the hardware resources. As part of their work, Meng et al. also developed a workload forecasting model to determine compatible VMs that can be consolidated on a single physical machine without violating any of the VM's SLAs. Fito et al. [31] developed related criteria for cloud-based SLAs and a model to monitor SLA satisfaction in clouds. Verma et al. [93] considered workload correlation and dynamics for efficient VM placement.

2.3.4 Resource Usage Information and VM Management

VM resource usage information of interest in a cloud environment includes the CPU, memory, disk and network usage, etc. of a system as well as other, less obvious, resources. This information is needed to determine whether or not VM SLAs are being met. To collect the resource usage information, a number of metrics need to be defined. The metrics can be categorized into static and dynamic metrics. Static metrics may include processor (type and frequency), operating system (name and version), memory (size), disk (size, speed and type), network (speed/ bandwidth) while dynamic metrics may include information related to machines (boot time and respective uptime), processors (CPU percentage idle and busy time), memory (loaded and free space), disk (used and free space), and network (burst information). Static metrics do not vary with time and are often items specified in SLA requirements. Dynamic metrics reflect VM resource usages and naturally change over time and thereby need to be observed over time. Within an observation period, the values of dynamic metrics are typically saved after a certain fixed interval. The sampling time must be chosen considering a trade off between the benefits of frequent gathering, impact on resource load due to monitoring, and availability of storage space for collected information. Resource usage may vary somewhat within a sampling time interval, so average resource usage values are commonly calculated and stored for each sampling period thereby reducing storage requirements. Thresholds (both upper and lower) can be used to define whether a particular resource/system is in a normal, over-utilized or under-utilized state. If the dynamic metric value of a particular resource usage is higher than the upper threshold limit, then the corresponding system is considered

to be in an over-utilized state. On the other hand, if the dynamic metric value of a particular resource usage becomes less than the lower threshold limit then the corresponding system is considered to be in an under-utilized state. Otherwise the system is in the normal state. The fixing of upper and lower threshold values depends on each particular resource. VMs are often characterized as compute-bound, memory-bound, etc. [39] [103]. Considering levels of use for different resources and associated thresholds provides the basis for a potentially stronger form of characterization that can more readily adapt to as yet unforeseen patterns of VM behaviour that may not be correctly categorized simply as, for example, I/O bound.

In clouds, automatic management needs to be established for making decisions about resource allocation, detecting resource bottlenecks, detecting SLA violations and also triggering effective solutions for these problems. Padala et al. [69] worked on automatic resource management and introduced automated control (via a system named 'Auto-control') for multiple virtualized resources. Autocontrol can dynamically allocate and reallocate the available resources of a host among VMs depending on the requirements of the applications currently running on those VMs. Autocontrol depends on a model estimator and a resource controller (based on a multi-input, multi-output [MIMO] control approach [21]). Autocontrol provides dynamic VM provisioning that considers SLAs. Padala et al. identified several new challenges (i.e. complex, service level objectives or "SLO"s, time varying resource requirements, distributed resource allocation, and resource dependencies) with dynamic VM provisioning in datacenters and attempted to address those issues in their Autocontrol strategy. An SLO typically refers to higher-level performance metrics like transac-

tion throughput, response time, etc. of an application than an SLA does. When the applications in different VMs have different complex service level objectives, it is a challenge to multiplex available resources according to the total requirements of each VM. Padala et al. [69] experimented with the effectiveness of Autocontrol using three different applications (an online auction site, a benchmark application for an online auction site like RUBiS [2], and a secure media server) and their results showed that Autocontrol can successfully detect bottlenecks and adjust resource allocation using prioritization of applications.

2.3.5 Virtual Machine Migration

Sometimes the growing resource requirements of a VM may exceed the capacity of the current host. In that case, the VM needs to be migrated to another, more suitable host. (This is how dynamic VM provisioning is effected.) VMs can also be migrated off physical machines when the current host requires maintenance or to reduce power consumption by consolidating VMs into a smaller number of hosts so the unused hosts can be shut down. Clark et al. [24] described "live" migration of a VM across physical hosts where execution carries on either in the source or in the destination host during the transfer of memory contents. At a certain point, VM migration needs to stop execution in the VM at the source host and to start execution at the destination. The relatively short time between these events is when no progress can be made. In VM migration, any such downtime is important to minimize since it may degrade the system's ability to meet the relevant VM SLAs. In VM migration, CPU and network overheads may also degrade application performance. In spite of

this, VM migration is very useful to deal with runtime overload situations, especially in the case of sustained overload [2]. Hines et al. [37] compare pre-copy and post-copy based live migration of VMs. In pre-copy based live migration, the CPU state transfer takes place after the memory transfer phase and in post-copy based live migration, the CPU state transfer takes place before the memory transfer phase.

2.4 Predicting Future Behaviour in Systems

In clouds, as in other software systems, it may be beneficial to be able to predict likely future behaviour based on past behaviour. In the case of clouds, predicting changes in VM behaviour is of particular interest. This can be done using a variety of tools. Time series models [13] work with historical data and help to identify the trend, and variation (seasonal/cyclical variation and random variation) of some observed data. A trend normally exists in a time series when there is a long-term change in the mean level and a seasonal pattern exists when a time series is influenced by seasonal factors (e.g. month of the year, day of the week, hour of the day, etc.) [59]. To forecast future workload properly, it is necessary to understand the obseserved data and create an appropriate prediction model. A polynomial trend (with least squares) of a time series is normally constructed by finding an appropriate stationary An alternate method is to apply differencing operators repeatedly (i.e. replacing the original series X_t by Y_t : X_t - X_{t-d} for some positive integer d) until the differenced data construct a realization of some stationary time series [78]. Naturally, the more history information that is available the more accurate forecasting will normally be. The prediction of time series having seasonal data naturally performs well when there

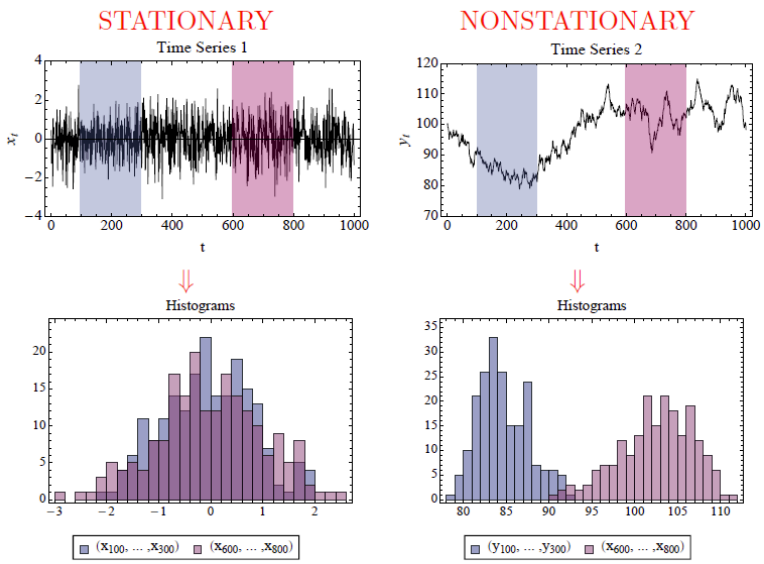

Figure 2.1: Stationary vs Non-Stationary Time Series [8]

is regular change in data over a season.

2.4.1 Exponential Smoothing

When pridicting future values from observed data how the observed values are weighted can have a great effect on the accuracy of prediction. There are a number of time series prediction models based on exponential smoothing of the observed data (e.g. Simple Exponential Smoothing, Holts Exponential Smoothing, and Holt-Winters Exponential Smoothing, etc.). Exponential smoothings make no assumptions about the correlations between successive observed data. An average weighted forecast provides equal weights on each observed data and performs the forecast but exponential smoothing based techniques provide more weight on the recent observations and the weights decrease exponentially for the older observations. For any fixed $\alpha \epsilon$ [0,1], the forecast estimation can be defined (as in [13]),

$$Y^{'}_{T+1}|T = \alpha Y_T + \alpha(1-\alpha)Y_{T-1} + \alpha(1-\alpha)^2 Y_{T-2} + ... \quad (2.1)$$

Exponential smoothings use the smoothed value at the present time to forecast the next value. The logic behind such smoothing is that more recent observations are more likely to be reflective of near future behaviour (hence their higher weighting) but that historical trends cannot be ignored (hence their inclusion in the model, albeit with lesser weight). Simple Exponential Smoothing is a time series model that is good for short term forecasts and when the data has no seasonality. Holts Exponential Smoothing is also not appropriate for data that exhibits seasonality but is good for the prediction of data having simple increase and decrease trends. Holt-Winters Exponential smoothing can provide prediction models for time series data with increase and decrease trends and also with seasonality and in this case, we may also get a classical decomposition model (as in [13]),

$$X_t = m_t + s_t + r_t \quad (2.2)$$

where m_t is known as a trend component (a slowly changing function), s_t is a seasonal component (that is a function with a known period d), and r_t is a random noise component. For example, if we have the observations $x_1, x_2, ..., x_n$ and if the seasonal period d is even (i.e. d=2q) then, firstly, the trend component can be estimated (as in [13]) by,

$$m^{'}_t = (0.5x_{t-q} + x_{t-q+1} + ... + x_{t+q-1} + 0.5x_{t+q})/d, \quad where \quad q < t \leq n-q \quad (2.3)$$

Secondly, for the seasonal component, the average w_k is measured from the deviations (x_{k+jd} - m'_{k+jd}), q¡k+jd ≤ n-q and the seasonal component, s'_k can be estimated as (in [13]) by,

$$s'_k = w_k - d^{-1}\sum_{i=1}^{d} w_i, \quad where \quad k = 1, ..., d \quad and \quad s'_k = s'_{k-d}, k > d \tag{2.4}$$

Finally, the noise component (and also its characteristics) can be determined by subtracting the estimated trend and seasonal components from the observed data (the random noise component is expected to be stationary),

$$r'_t = x_t - m'_t - s'_t \tag{2.5}$$

2.4.2 Auto-Regressive Integrated Moving Average (ARIMA) models

There are a number of prediction models based on autoregrssive processes where the current value of the time series is linearly dependent on its previous value with some white noise or error (as in [78]),

$$X_t = \alpha X_{t-1} + \epsilon_t \tag{2.6}$$

Here ϵ_t is a noise time series. Since X is regressed on itself, an autoregressive model

of order p, AR(p) can be defined (as in [78]) by:

$$X_t = \sum_{i=1}^{p} \alpha_i X_{t-i} + \epsilon_t, \tag{2.7}$$

There are also a number of prediction models that assume the current values of the series is a weighted sum of past noise terms,

$$X_t = \epsilon_t + \beta \epsilon_{t-1} \tag{2.8}$$

More generally, these are called moving average models and a moving average model of order q, MA(q) can be defined (as in [78]) by:

$$X_t = \epsilon_t + \sum_{j=1}^{q} \beta_j \epsilon_{t-j} \tag{2.9}$$

ARIMA (Autoregressive integrated moving average) is one of the most popular prediction models that integrates AR(p) and MA(q) models. An ARIMA(p,q) model can be defined (as in [78]) by:

$$X_t = \sum_{i=1}^{p} \alpha_i X_{t-i} + \sum_{j=1}^{q} \beta_j \epsilon_{t-j} \tag{2.10}$$

ARIMA models are assumed to be built on stationary time series and for this reason a differencing operator may need to be applied on the observed data. If the observed time series in not stationary, then the first order difference (δX), second order

difference ($\delta^2 X$), and so on, can be defined (as in [78]),

$$\delta X_t = X_t - X_{t-1} \tag{2.11}$$

$$\delta^2 X = \delta(\delta X_t) = X_t - 2X_{t-1} + X_{t-2} \tag{2.12}$$

Time series models of various types have been applied to a variety of scenarios in cloud systems management. Calheiros et al. [16] used an ARIMA model to predict the future application workload behaviour in a cloud environment. Saripalli et al. [84] used various versions of moving average models to detect hot spots in predicted workloads in clouds. Sergio et al. [68] performed Markovian workload characterization for the QoS prediction of cloud workload. Khan et al. [50] proposed clustering to identify groups of VMs that exhibit correlated workload patterns and then choose appropriate prediction models at the group level rather than at the individual VM level. Hyndman [41] used a number of accuracy metrics to find an appropriate prediction model for a given time series and proposed a number of ways to measure the forecast errors (e.g. scale dependent errors, percentage errors, relative errors and scale free errors). Hyndman et al. also proposed an automatic selection approach of an appropriate prediction model though this strategy has a limitation as it has greater computational complexity [40]. Wang et al. [97] identified a total of nine classical and advanced statistical features describing a time series (i.e. trend, seasonality, serial correlation, nonlinearity, skewness, kurtosis, self-similarity, chaos, and periodicity) and they introduced an automated rule induction system that chooses an appropriate forecasting

method depending on time series data characteristics. These characteristics were used to discover potential future peaks (i.e. maxima), valleys (i.e. minima), etc. in a time series and to cluster (i.e. to find similarity among) a number of time series observations in [94].

2.4.3 Changepoint Analysis

Changepoint analysis is the ability to detect changes considering mean and/or variance within observed or predicted data. There are various search algorithms for detecting changepoints with a variety of test statistics [52]. Given some predicted data, $X_{1:n}$=(X_1,X_1,.....X_n), a changepoint may be considered to exist at time k in this predicted data if the statistical properties of X_1,.....,X_k and X_{k+1},.....,X_n are different in some significant way. Similarly, there may present η changepoints with the positions $k_{1:\eta}$=(k_1,.....,k_η) splitting the data into η segments where the ith segment may contain data $X_{(k_{i-1}+1):k_i}$. Identifying changepoints in behaviour may provide an opportunity for adapting provisioning techniques in clouds.

2.5 Cloud-based VM Provisioning Techniques

VM provisioning is the assignment of cloud resources to cloud users' work (in the form of virtual machines). VM provisioning helps ensure efficient application execution, load balancing and proactive failure handling, and thus can increase the reliability and efficiency of the whole cloud system and the applications running in it. In clouds, VM provisioning can be categorized as either static or dynamic VM provisioning.

2.5.1 Static VM Provisioning

As described in the introduction (and repeated here for convenience), end users and cloud providers typically enter into contracts, quantified as service level agreements (SLAs), to describe the expected requirements, in terms of computing, etc. capacity, required for each user "application" (i.e. virtual machine). Cloud providers use this information to determine the amount of infrastructure they need to purchase and later allocate to users when their VMs run as well as the cost charged to the users. This allocation is done by "provisioning algorithms" that pack VMs running end-user applications into physical cloud infrastructure (i.e. compute nodes) that the cloud provides based on the values specified in the SLA. For applications that have predictable and/or generally unchanging demands/workloads, it is possible to use static provisioning effectively. In this approach, VMs are carefully packed into clouds as their machines are brought online and continue to execute effectively for potentially very long periods of time on the selected nodes. Using static provisioning, the cost of making the placement decision is incurred once, before the user's applications are running, and therefore may safely incur greater overhead, if necessary, to achieve a better result without negatively impacting the end users.

In static VM provisioning, the required hardware resources are selected as the VMs are requested based on agreed-to Service Level Agreements (SLAs) and remain fixed but in the case of dynamic VM provisioning, the resources allocated to a VM can be changed dynamically to handle unexpected fluctuations in workloads. Simple static algorithms track available resources on physical machines and allocate VMs to physical machines using techniques designed to achieve certain goals.

The main goal of many static provisioning algorithms is simply to pack the VMs in a minimum number of physical machines. Static VM provisioning is then an instance of the bin packing problem. A bin packing algorithm finds the minimum number of bins to allocate a list of given objects to where each object has a certain size and the size of each bin is fixed. Because of the proprietary nature of enterprise clouds, we do not know specially which static algorithms are being used.

Static algorithms, as mentioned previously, are commonly divided into online and batch techniques. The algorithms related to online and batch algorithms are now described.

Online Algorithms

There are a number of popular online algorithms which can be used for static provisioning of single VMs. Such online algorithms include Next Fit, First Fit, Best Fit, and Worst Fit which are generally well and widely known as they are also used in contexts other than clouds. All of these algorithms are one dimensional (for example considering CPU needs only) but there is no conceptual difficulty in considering multiple dimensions as well (for example memory, network, etc. requirements can be combined into a single measure). With an increase in the number of dimensions, the complexity increases, which can lead to increased run time or less accuracy as heuristics must be used. In all cases though, the goal is always to pack VMs into as few physical machines as possible. In all online algorithms, the requested VMs are placed in the physical machines in the order in which they have been requested to be created. The most basic online static VM provisioning techniques are now quickly

described:

- Next Fit: Next Fit places the next VM into the current physical machine if it fits. If it does not, Next Fit places it in a new physical machine.
- First Fit: First Fit places the next VM into the first available partially packed physical machine if it fits. If it does not fit into any partially packed physical machine, First Fit starts a new physical machine.
- Best Fit: Best Fit places the next VM into the physical machine which will have the least resources left over after the VM is placed in the physical machine. If it does not fit into any partially packed physical machines, Best Fit starts a new physical machine.
- Worst Fit: Worst Fit places the next VM into the physical machine which will have the most resources left over after the VM is placed in the physical machine. The logic behind this strategy is to leave large, available resources for other requests to make use of. If it does not fit into any partially packed physical machines, Worst Fit starts a new physical machine.
- Round Robin Algorithm: In Round Robin, VMs are evenly spread over the physical machines to achieve load balance and, presumably, good performance.
- Greedy Algorithm: Greedy is a technique for packing VMs onto the physical machines where the goal is to make the physical machines as fully packed as possible. This offers the potential benefit of achieving low power consumption since unused machines may be powered down.

Batch Algorithms

When the creation order of VMs does not need to strictly respect the request sequence, significant improvements can be achieved using what are called "batch algorithms". For example, if a group/batch of VM requests can be gathered and are then sorted according to their resource requirements before they are allocated into physical machines, the performance of the First Fit and Best Fit algorithms can be improved.

- First Fit Decreasing: In this strategy, VM requests are sorted according to their resource requirements and are placed in physical machines following the sorted order. First Fit Decreasing places the next VM into the first available partially packed physical machine if it fits. By placing large requests first, larger gaps cannot be occupied by unsuitably small VM requests. If the next VM to be placed does not fit or there are no partially packed physical machines, First Fit Decreasing starts a new physical machine. First Fit Decreasing (FFD) is a common benchmark algorithm used in assessing the performance of new placement algorithms.

- Best Fit Decreasing: In this strategy, VM requests are sorted according to their resource requirements and are placed in physical machines following the sorted order. Best Fit Decreasing places the next VM into the physical machine which will leave the least resources left over after the VM is placed in the physical machine. If it does not fit into partially packed physical machines, Best Fit Decreasing starts a new physical machine.

2.5.2 Dynamic VM Provisioning

In cases where demand by applications may vary, dynamic provisioning techniques have been suggested whereby VMs may be migrated, on-the-fly, to new compute nodes within the cloud if it is not possible to support their required load changes within existing compute nodes. This load variation mostly depends on the types of hosted applications/services in a VM and on the characteristics of the data being manipulated but also on certain characteristics (e.g. underlying data structures, etc.) of those applications. Every VM consists of a number of application processes (running in it) and the variation in resource utilization is mainly due to starting, stopping, and/or pausing such processes. Workload changes can be categorized by expected duration into, for example, short-term, medium-term and long-term, etc. and all categories need to be handled efficiently but not necessarily using the same mechanisms. Dynamic provisioning allows the system to adapt to changing conditions but at the cost of runtime overhead and potential execution delays during migration that may negatively affect some end users. To minimize this overhead, restrictions on the complexity and scale of dynamic provisioning algorithms may be necessary.

In dynamic VM provisioning, additional required resources can be dynamically allocated for a VM when its resource requirement exceeds the currently allocated resources. Also resources can be de-allocated when a VM doesn't need them. Kim et al. [54] proposed a prediction-based dynamic VM provisioning model based on the previous resource usage patterns of a VM. They introduced pattern based provisioning (where additional resources are provisioned depending on resource usage patterns) to avoid the negative effects of simpler threshold based provisioning techniques (where

additional resources are provisioned only when a certain threshold of resource use has already been crossed which they found sometimes leads to inappropriate provisioning requests).

Rolia et al. [81] also used historical resource use information to predict future resource requirements for applications. Though previous use information may not always provide accurate prediction, especially for web-based interactive applications and in the case of shared virtualized infrastructure where resource availability also depends on the requirements of co-hosted applications and their priorities, this approach can decrease the number of active hosts/nodes thereby decreasing cloud resource requirements and power consumption. Zhang et al. [102] proposed to use "ghost VMs" to quickly assign additional required resources to running VMs. Ghost VMs are spare VMs that remain active and are used to support the additional resource requirements of co-hosted VMs. Bobroff et al. [10] presented an approach to systematically identify the hosts that are good choices for dynamic provisioning, and also proposed a mechanism for dynamic migration of VMs based on a workload forecast. Finally, Khanna et al. [51] proposed a model to monitor the resources (CPU and memory) of physical and virtual machines and if the resource usage of a VM exceeds a predefined threshold or some SLA is at risk, then the system migrates a VM to another physical host. Naturally, a key challenge in dynamic provisioning is not just to decide which VMs to move and when but also where to move a VM to so that it will execute efficiently in the future.

2.5.3 Challenges in Resource Provisioning in Clouds

With the growing adoption of clouds as elements of both business and other (e.g. research, entertainment, gaming, etc.) computing solutions, the diversity of work done using clouds is increasing. Much work is still typified by long-running VMs executing OLTP (OnLine Transaction Processing) and other standard workloads but there is also an increasing number of shorter duration tasks for which clouds are now being used, on-demand. Further there is a growing desire to run a broader range of applications that may have much more with varying resource demands over time (e.g. interactive applications) in clouds. This greater percentage of what might be termed dynamic use presents new challenges to cloud provisioning. Packing well-characterized, static virtual machines (VMs) can be done very efficiently since more time may be devoted to the packing process (to minimize power consumption, resource costs and the like). As the VM mix becomes less regular with VMs exhibiting greater dynamic variation in resource requirements and as VMs run for shorter durations, however, placement and packing decisions need to be made more quickly and, potentially, using different strategies that can adapt to changing resource demands.

The process of VM provisioning is particularly complicated when applications that are interactive in nature and have tight SLA constraints are hosted within a cloud as such applications' resource demands may change unpredictably. For some applications, such as online gaming, this behaviour is obvious. For others it is less so. Consider an interactive consultation system where a patient and his/her doctor (in a remote location) meet electronically with a number of consulting specialists to discuss treatment plans and review diagnostic imagery (possibly from multiple

sources/modalities). In such a scenario, the activities carried out are determined by the participants and their actions may significantly affect not only the required SLA constraints but also the demands on the cloud hosted components (e.g. due to unanticipated manipulation of 3D fMRI imagery). With such workload changes, migrating the corresponding VM(s) to a new node may prove to be either inappropriate since the demand change may be of limited duration or may incur unnecessary management overhead. Thus, it would be better to handle such workload changes in existing compute nodes, whenever possible. In contrast, when the future workload is predictable and the demand changes are periodic and longer-lived, it would be better to group (or at least pair) a number of VMs that can share resources most effectively when co-located. Such dynamic re-provisioning may cause necessary migrations of the VMs (considered as a group) within already running nodes or into new compute nodes. As the overhead of adjustment for workload changes in existing nodes or in new nodes increases, such re-provisioning of course becomes less attractive. At that point, a broader form of dynamic re-provisioning could be considered that incurs the greater cost of some form of re-packing of (potentially all) the running VMs in underutilized physical machines based on the newly available load information to return the system to a state of efficient sharing, VM packing and therefore effective execution. This can be done concurrently with the ongoing execution of VMs in the cloud as long as there are available idle resources on underutilized physical machines. Efficient, low-cost algorithms need to be developed to meet these needs.

Chapter 3

Problem Description

As has been discussed, the provisioning of physical machines on which to run VMs is a fundamental problem in clouds, affecting both the utilization of the PMs and hence the cost of operation for cloud providers as well as the satisfaction of cloud users in terms of responsiveness and conformance to agreed-to Service Level Agreements (SLAs). Many different kinds of provisioning algorithms have been proposed, implemented and evaluated, and all have shortcomings, many in particular, with respect to handling VMs with unexpected time-varying workloads. Ideally, a provisioning solution would be able to offer the adaptability of dynamic techniques but with less VM management overhead than is currently the case using dynamic provisioning and thereby be able to effectively handle a broad range of cloud-based applications including those that are short-lived and those that exhibit significant changes in load over time. To accomplish this, a different approach is required that meets the needs of both cloud providers and clients for any sort of load varying work.

Introducing VMs into cloud workloads that either run for short durations or that

change their resource usage behaviour (and hence induced load) over time will be problematic for existing provisioning algorithms. Static techniques lack the ability to adapt to changes in demand but they might be improved somewhat to offer better initial grouping of VMs as well as to deal with urgent and short-duration VMs. Dynamic techniques offer support for adapting to changes in load but if such changes occur in many VMs and happen frequently, dynamic algorithms will need to be developed that can make effective and long-lasting changes in VM placements to avoid excessive runtime overhead due to the need to re-run dynamic provisioning algorithms. Further, making placement changes for VMs exhibiting only short-term changes in load does not make sense so some mechanism needs to be provided to handle such occurrences without VM migration. Finally, if many short duration VMs complete without more VMs arriving, PM utilization will go down on machines that have "lost" running VMs and hence have unused resources. This too needs to be handled.

There are, as just identified, multiple improvements needed to traditional cloud provisioning techniques to handle a broader range of VM types exhibiting less consistent and shorter-term load behaviours. All of these improvements cannot be effectively addressed using a single algorithm. Improved static techniques are needed to provide timely and appropriate initial placements of VMs that will enhance responsiveness for short duration VMs and help to minimize the need for dynamic techniques early in the execution of longer-lived VMs. New dynamic techniques are also needed to ensure fair and effective resource sharing on PMs during short-duration load fluctuations, to adjust the placement of running VMs in response to anticipated load changes of longer duration, and to react to observed inefficiency in the use of

the resources on physical machines.

In the next chapters, I propose two static provisioning techniques and a family of three related dynamic provisioning techniques to address these issues. In what follows I more directly point out the shortcomings of existing provisioning techniques to motivate the need for my algorithms.

As cloud usage grows to include more on-demand, shorter duration VM requests, balancing between efficient packing and responsiveness becomes more important. Existing algorithms currently tend to favour one over the other when what is needed is a new static algorithm that, whenever possible, provides the benefits of both online and batch techniques. Further such an algorithm needs to be able to effectively deal with the fact that VM requests are multi-faceted (including requirements for CPU, memory, storage, network, etc. resources) and thus require support multi-criteria packing. Batch provisioning is commonly done using algorithms which are variations on First Fit Decreasing (FFD) where the VM requests in a batch to be placed are sorted into decreasing order by SLA-specified resource requirements. With a single resource type this is straightforward. With multiple resource types, some sort of technique is often used to combine the requirements for each resource type into a single aggregate resource requirement for use by FFD. There is room for some improvement in this "combining" process.

Placing VMs in PMs based solely on packing efficiency focusing on SLA information alone can result in VMs being co-located that interfere with one another by, for example, concurrently requiring more CPU resources than expected. This is more likely to occur as the number of load-varying VMs increases. This situation is known

as "VM interference" and I refer to co-located VMs that result in interference as being "incompatible" [1]. For two (or more) VMs to be 'compatible' when placed on the same PM (i.e. to avoid interference issues) they must not collectively exceed the resources allocated to them, but they may temporarily share their resources if doing so provides a benefit. PM-local, fine granularity sharing of resources can be done implicitly as is supported by a number of VM hypervisors. Unfortunately, this resource sharing approach normally cannot be controlled leading to the possibility of a borrowing VM negatively impacting the performance of a loaning VM. So, there needs to be a new explicitly controlled approach for such intra-PM resource sharing. Given such a mechanism, to benefit from the ability to share resources on a PM, compatible VMs must be, of course, co-located there in the initial packing. Thus, by assessing VM compatibility, the cloud provider can then place compatible VMs on the same PMs, while still respecting packing efficiency (e.g. given a choice between two VMs in the packing process, the cloud provider can select the one which will be more compatible with existing VMs already on the target PM). This means that a cheap and simple mechanism to determine the expected compatibility of VMs for use with a **static** VM placement algorithm needs to be developed.

A controlled resource sharing approach is also needed among compatible VMs when considering dynamic provisioning. Such dynamic sharing needs to be limited (both in amount and duration) to avoid detrimental effects on a loaning VM caused by the corresponding borrowing VM. For short-duration increases in workload (such

[1] VM interference arises in a number of different ways including explicit resource use as described in SLAs and also competition for low-level hardware resources (such as shared cache lines). The focus in this thesis is on SLA-based resource use though "micro-architectural" interference could also potentially be addressed using techniques described later.

as those required to meet the requirements of certain "resource critical" VMs) safe dyanmic sharing of resources on a PM will avoid significant *unnecessary* overhead of VM migrations if other multiplexed VMs on the affected machine(s) are currently underutilizing the resources assigned to them.

Recognizing that resource demands by VMs are becoming increasingly variable over time as a wider range of applications (e.g. interactive, graphical and collaborative ones) are deployed using cloud infrastructure, a dynamic provisioning approach must be created to effectively address the changing VM requirements over time. Due to continuous demand changes in the longer-lived VMs, the compatibility criteria that could be used during initial packing may be inappropriate. In a dynamic context, it is necessary to address load variations periodically (after a certain, tunable, time interval) to co-locate newly compatible VMs together. This should be achieved by findings suitable low cost VM migration approach that has acceptable run-time overhead. Unfortunately, the use of live VM migration in the existing dynamic provisioning algorithms does not consider compatibility and introduces significant network overhead for data transfer which grows with the frequency of resource requirement changes. For this reason, a new dynamic provisioning approach is required that can predict future VM compatibility and then use VM migration to co-locate VMs that are expected to be compatible for significant periods of time.

Unfortunately, such potential 'compatiblity-aware' VM migrations may not always result in the best possible packing efficiency due to the need to sometimes bring new physical machines online and thus may reduce the overall resource utilization of physical machines. Combined with losses in PM resource utilization due to the

termination of VMs running for shorter-duration, there thus needs to be another dynamic provisioning approach for re-packing VMs on underutilized PMs to increase physical machine utilization. Done properly again considering VM compatibility, this might also reduce the need for future VM relocations.

Chapter 4

Solution Strategy

A hybrid provisioning approach combining the desirable features of static and dynamic techniques is certainly attractive. Such an approach must start with a good static placement that can be subsequently tuned by types of dynamic re-provisioning to address changing demands as long as doing so is effective and not excessively costly. I introduce a strategy to combine the best features of both static and dynamic VM provisioning via a hybrid approach - "HSDP" (Hybrid Static and Dynamic Provisioning) where dynamic tuning carefully refines a good initial static placement and, if necessary, may eventually trigger revised placement (i.e. re-packing). At least two challenges must be met in any such hybrid provisioning technique. First, it must be possible to determine when the migration of a VM should take place to avoid resource provisioning bottlenecks and to provide efficient and cost effective use of physical machines in the cloud. Second, it must be possible to determine whether or not a proposed migration will provide ongoing performance benefit (the challenge of dynamic provisioning is to ensure that the benefit of any migration significantly out-

weighs the overhead of doing so). Meeting these challenges will require an accurate prediction model. Making good dynamic placement decisions will depend on both the current and predicted future workload of the current host, current and predicted future workload of other hosts, power consumption (if the selected destination host needs to be powered on), memory usage patterns, network burst characteristics, and the compatibility of multiplexing with other VMs, etc. Further, it must be possible to identify characteristics of the dynamic provisioning done that can be used to trigger appropriate re-provisioning to limit the need for future dynamic provisioning and associated overhead thereby limiting the impact on VM performance and SLA conformance. Such re-provisioning can be done in parallel with ongoing operations in the cloud exploiting unused cloud resources (CPU cores, memory, etc.). Finally, selecting the order of VM migration to achieve a selected re-provisioning must be done carefully to minimize disruption to computations taking place in the cloud.

Underlying static provisioning, which provides the initial placement in HSDP, is a fundamental packing problem - assigning VMs with certain requirements (specified by their SLAs) to physical machines (PMs) with certain resources to minimize wasted resources within PMs and to minimize the number of PMs needed. With a single resource type, provisioning becomes an instance of the classic bin-packing problem. VM provisioning is therefore NP-hard [56] so heuristic methods are required that will give solutions that are hopefully good but necessarily without any guarantee of optimality. Considering multiple resource types adds complexity to the problem as it is harder to concurrently optimize the packing for multiple criteria (e.g. CPU, memory and storage requirements). Further, packing decisions typically need to be

made reasonably quickly so elaborate heuristics designed to get very close to the optimal packing in many cases may be impractical.

In this thesis, I introduce two new static provisioning approaches. The first combines online and batch techniques to improve the speed of scheduling time-sensitive VMs while maintaining good, overall, packing efficiency. The second focuses on longer-term VMs by trying to co-locate VMs that will be compatible to minimize the need for later dynamic re-provisioning. These two algorithms could be combined into one but in the thesis they are presented independently to make the contributions of each clear.

Recall that using an online provisioning approach, VMs are assigned to PMs as they arrive, one by one. While this provides quick response, the resulting packing is often sub-optimal. With a batch provisioning approach, VM creation requests are collected over a period of time to form a batch of requests that are handled concurrently. This allows for optimization of the packing using algorithms such as First-Fit, Decreasing (FFD). My first new static provisioning algorithm - "HBF" (Hybrid BackFill) provisioning supports both online and batch allocation of newly requested VMs (with various life times) and minimizes the number of total physical machines through efficient packing and backfilling (placing new VMs in existing PMs with free capacity, for example due to the completion of other VMs) while specifically providing quick placement for those VMs that require it. This provides better support for new cloud users whose workloads may consist of many short-duration "jobs" rather than long-running OLTP work.

I also introduce the "CSVP" (**C**ompatibility-based **S**tatic **V**M **P**lacement) al-

gorithm that takes a very simple approach to characterizing VM behaviours before packing so that "compatible" VMs within a batch can be co-located while still providing good packing efficiency. At the core of supporting VM-compatible placement is a mechanism for judging VM compatibility. To be able to effectively categorize VMs so that their "compatibility" may be considered in **static** placement decisions, the cloud provider must rely on either user-supplied information or feedback information characterizing VM behaviour on a previous run. Cloud users are not always technically sophisticated and thus may not be capable of providing an in-depth characterization of their VM's either due to lack of fundamental understanding of VM behaviour or due to lack of knowledge of prior VM behaviour. This makes an automated feedback mechanism that captures VM behaviour for use in future runs attractive. This approach, however, is also fundamentally limited by the possibility of changing behaviour both during and between VM runs. Further, significant changes in VM behaviour between runs, for example due to processing of different input data, is something that cannot be supported in a static placement scheme. Thus, a simple, overall characterization of VM behaviour is needed that can either be estimated by a user or easily and cheaply determined using an automated technique during a VM run. The goal is to characterize VMs into those that are resource critical (which are likely to need to borrow resources from other co-located VMs) and those that are likely to be able to loan resources to other co-located VMs. Certainly, any number of categories of resource criticality are possible. More categories provides for greater differentiation but also complicates the process. CSVP uses a characterization that is simple enough that a cloud user should be able to do a reasonably accurate ini-

tial characterization and that an automated monitoring system can easily generate without introducing significant overhead once a VM run has been completed.

Eventually, any static placement will lose effectiveness for a collection of VMs having time-varying resource/load requirements. Also, the completion of short-duration VMs may cause the efficiency of a static placement to decrease. Either gaps (of unallocated resources) are left in PMs causing under-utilization or when the gaps in PMs are filled with new VMs, a less dense packing of VMs or a less compatible group of VMs may be co-located on a PM. This means that dynamic provisioning techniques are needed to adjust VM placements to address changing VM requirements. In this thesis I also introduce a family of three related dynamic provisioning techniques: DTVM (**D**ifferential **T**ime-shared **V**M **M**ultiplexing) which provides for **controlled** sharing of resources between co-located VMs, DAVM (**D**TVM-**A**ware **V**M re-**M**ultiplexing) which groups VMs that are predicted to be highly compatible and effects their dynamic co-location (thereby providing DTVM compatible VMs with which to work) and DRVM (**D**ynamic **R**epacking of **VM**s) which identifies underutilized PMs and initiates compatibity-aware re-packing of the VMs on such PMs to improve utilization and allow the powering down of unneeded PMs in the cloud. To obtain the benefit of compatibility-based VM provisioning through a controlled resource sharing strategy, I introduce "DTVM". DTVM can be used to safely and effectively handle short-lived workload increases by a VM on a single physical machine if there exist sufficient under-utilized resources in the other VMs on the compute node where a resource-critical VM has a temporary increased need for resources. Not all VMs multiplexed in a PM may need to use all their allocated resources at all times. Hence, any unused

resources could be exploited by temporarily overloaded VMs to avoid the need for VM migration due to a short-term SLA violation[1]. Hypervisors support resource sharing but not in a controlled fashion. This means that a resource-critical VM might be able to "steal" too many resources (or resources for too long) from other co-located VMs thereby affecting their efficient execution. DTVM prevents this problem. To make DTVM as effective as possible in addressing shorter-lived SLA violations, the challenge is to ensure that compatible VMs are available on the same PM. To be effective, DTVM relies on a partner technique to ensure that compatible VMs are co-located so resources are available to share via DTVM. A static placement using CSVP ensures that compatible VMs should be co-located initially. As time passes and VM compatibility may decrease, however, DTVM's partner **dynamic** algorithm, DAVM, is used to proactively ensure ongoing compatibility of co-located VMs.

When predicted load changes cannot be supported using DTVM in a given PM, it is necessary to find a suitable alternative PM to which to migrate either the overloaded VM or any other multiplexed VM (to allow the overloaded VM to execute more effectively in-place using the additional resources freed-up). I therefore introduce a second new dynamic provisioning approach -"DAVM" used in the selection of the VMs to migrate and also for choosing the destination node to migrate VMs to.

There is a trade-off between cost due to migration and long-term benefit to the cloud system. This necessitates some sort of prediction of future VM resource need so that migration choices may be made that are effective over time. When the allocated VMs are monitored for a reasonably long period of time it may be found that

[1]Naturally, a VM receiving the temporary use of resources from another must be capable of exploiting those resources. Thus, for example, adding another physical core to a single-threaded application in a VM will not provide benefit.

some VMs' load increases predictably and that some other VMs at the same time have available resources. In such a case, those **compatible** VMs should be grouped together in the same physical machine(s) to better meet their collective needs and avoid SLA violations by being able to share resources using DTVM. Naturally selecting compatible VMs that are predicted to last the longest time gives the maximum potential benefit. Such compatibility-based VM re-provisioning does incur some VM migration overhead but should be effective for those VMs that have repeated behavior and longer lease periods from the cloud provider. DAVM provides benefit when

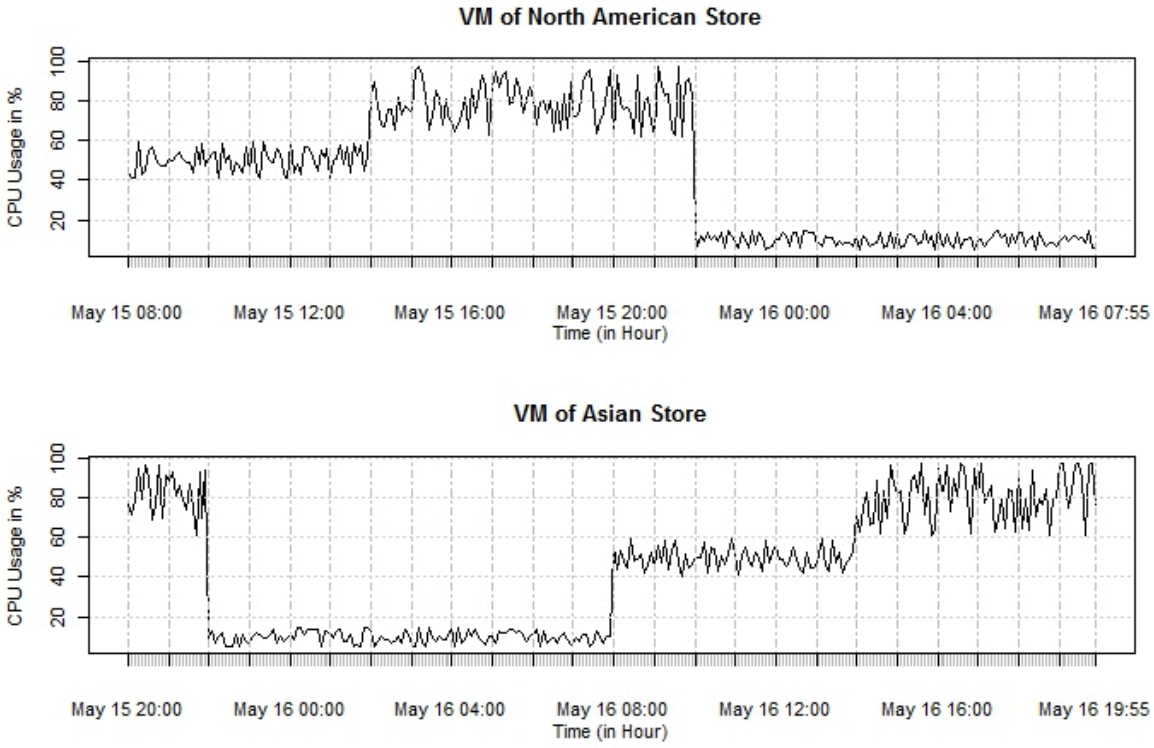

Figure 4.1: A simple example where DAVM is Useful

a group of VMs in the system have either up trending or down trending workload behavior that is predictable for a longer period of time. A very simple, coarse-grained

example illustrates the possibility of such compatible VMs. Consider two VMs where one is assumed to handle the workload of a store in North America (shown in the upper part of Figure 4.1) and the second is assumed to handle the workload of another store some place in Asia (shown in the lower part of Figure 4.1). If both stores are open only in the daytime and thus create workload in their corresponding VMs only during the daytime and if the time difference between the locations of the considered stores is around 12 hours, then DAVM could easily identify, group and co-locate these two VMs to make an effective very long-lasting pair for execution in the same physical machine. Of course, this idea generalizes to handling a wide range of situations where VM workload variations are larger or smaller and have longer or shorter time frames. Naturally, shorter term variations in VM requirements in clouds would, by sheer numbers provide the greatest potential for benefit from using DAVM with DTVM. Note also that unanticipated events can also lead to overload issues that could be addressed using DAVM. For instance, popular websites like TMZ.com, The Angels Times or Twitter were slowed to the point of being crippled with the news of Michael Jackson's death on June 25th 2009 [76]. This happened because there was an unpredictable, significant and sustained increase in workload on the servers. Using DAVM, such increases could be (at least partially) balanced by pairing such server VMs with other VMs having down trending workloads or by isolating them on their own physical machine(s).

If DAVM is forced to bring new physical machines online for co-hosting the compatible VMs it identifies, then the VMs will be distributed across more physical machines. With more physical machines there will be lower resource utilization in

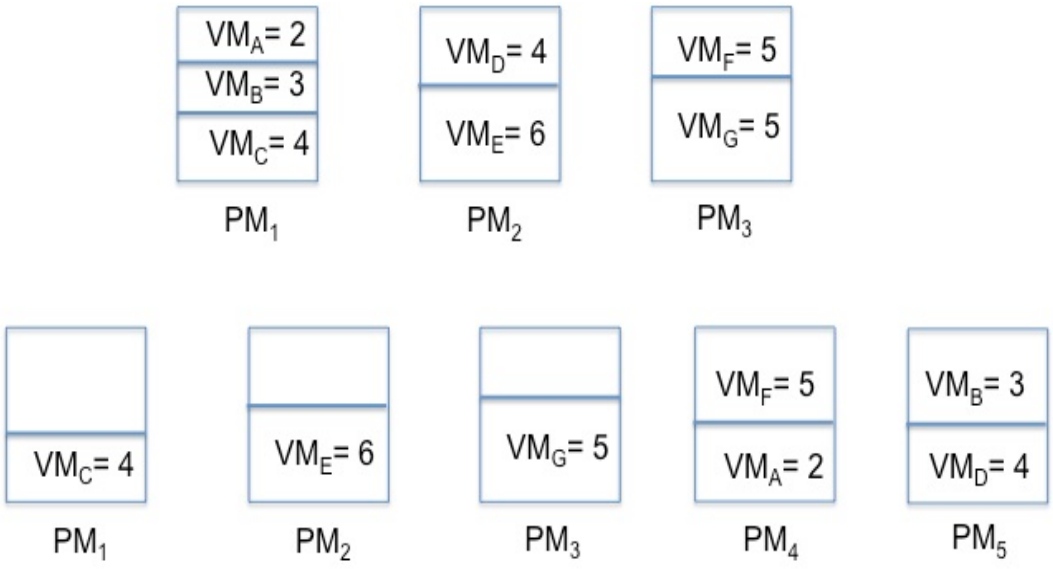

Figure 4.2: A simple example of the need for dynamic re-packing

the physical machines (mostly where VMs have been migrated from). The utilization of a physical machine may also be reduced due to the termination of short duration VMs (hosted in it). So, after monitoring allocated VMs and physical machines for a period of time, a scenario may arise where the resource usage in some PMs is not satisfactory (e.g. significant resources in some physical machines are unallocated or the number of overloaded machines is high). To handle the situation where the number of newly started physical machines or the number of under-utilized physical machines (detected by checking at regular intervals) is significantly high, I introduce another dynamic provisioning approach -"DRVM" with the goal to be able to shutdown under-utilized physical machines after efficiently re-packing the VMs from those under-utilized physical machines into other physical machines. In DRVM, some form of batch provisioning algorithm(s) are re-applied to improve the provisioning state. Although such dynamic re-packing may cause high migration costs and corresponding

overhead for a short period of time, it should be able to more efficiently increase the overall resource utilization and thus performance of a cloud system and should only need to be run occasionally. DRVM will, obviously, provide the most benefit when a large number of PMs in the system are underutilized since this will provide a large batch of VMs that can be re-packed together for efficiency. For example, referring to Figure 4.2, only three physical machines are required for packing VM_{A-G} (as shown in the upper part of Figure 4.2) where each physical machine is assumed to have 10 (ten) CPUs and each VM is shown with its associated SLA resource requirement. In this pathological example, if DAVM makes pairs (with the goal to achieve effective resource sharing) of VM_A with VM_F and of VM_B with VM_D and places them in PM_4 and PM_5 correspondingly (shown in the lower part of Figure 4.2), then due to bringing the new physical machines online to support the pairings made, some of the physical machines have become under-utilized. DRVM identifies the under-utilized physical machines in the cloud and re-packs the VMs of those under-utilized physical machines into as small a number of physical machines as possible. This returns the cloud to a state of dense packing meaning that the resources of the physical machines are being effectively utilized. This re-packing can be done both considering compatibility and considering past DAVM activity so that it maintains the groupings made earlier using DAVM to ensure the benefit of expected long-term VM execution efficiency on the physical machines hosting any DAVM-paired VMs.

Overall, combining controlled sharing (DTVM) with mechanisms to co-locate compatible VMs (CSVP then DAVM and DRVM) means that it is possible to more densely pack VMs into physical machines thereby achieving decreased resource wastage and

power consumption leading to lower cost for cloud providers. Importantly, this should be able to be done without incurring penalties for SLA violations (or the need to leave significant resource "headroom' in each host to avoid such violations).

Chapter 5

Algorithms

My hybrid provisioning framework - HSDP (Hybrid Static and Dynamic Provisioning) combines a number of static and dynamic provisioning algorithms. Recall that HBF (Hybrid BackFill provisioning) is focused on making static provisioning responsive to urgent VM requests while still ensuring good overall efficiency in packing. My other static provisioning algorithm, CSVP (**C**ompatibility-based **S**tatic **V**M **P**lacement), considers the compatibility of VMs while packing to make the static provisioning reflective of expected workload behaviour based on previous VM executions. I also developed three useful dynamic techniques that work together to deal with both short-lived and longer-lived load changes to enhance the effectiveness of provisioning in clouds. Many short-lived SLA-violations can be effectively handled by allowing **controlled** resource sharing among co-located VMs in a physical machine using my DTVM (**D**istributed **T**ime shared **V**M **M**ultiplexing) dynamic provisioning algorithm. Longer-term, periodic SLA violations are often predictable and can be minimized by pairing VMs that are predicted (using time-series analysis) to be com-

patible in terms of resource sharing and then locating the pairs together in the same physical machine. This is done using my DAVM (**D**TVM-**A**ware **V**M **M**igration) dynamic provisioning algorithm. DAVM selects and places paired VMs to new compute nodes within the cloud on-the-fly if it is not possible to place them in existing compute nodes. The co-located pairs can then effectively share resources using DTVM. The DAVM technique thus reduces SLA violations but may also have the side-effect of lessening packing efficiency. At this point, my DRVM (**D**ynamic **R**e-packing of **VM**s) technique can be used to perform re-packing of, potentially, all the VMs running in underutilized physical machines based again on their predicted resource requirements. DRVM reduces the number of underutilized physical machines whether or not they arise strictly due to DAVM pairings while respecting resource sharing efficiency.

I will shortly present the various algorithms and discuss their implementation. Before doing so, however, I will describe how they fit into my higher-level, general hybrid provisioning framework.

5.1 Hybrid Provisioning Framework

Given the pre-requisite ability to collect performance information on VM instances and PM nodes, the first component of a hybrid static and dynamic provisioning scheme, assuming an existing static placement approach, is a mechanism to decide whether any required changes can be resolved without VM migration (using DTVM in each node) or, if not, when and which VMs should be migrated (for DAVM). A key challenge to implementing a heuristic that assesses the cost and benefit of VM migration is that re-provisioning decisions must not be made based on transient

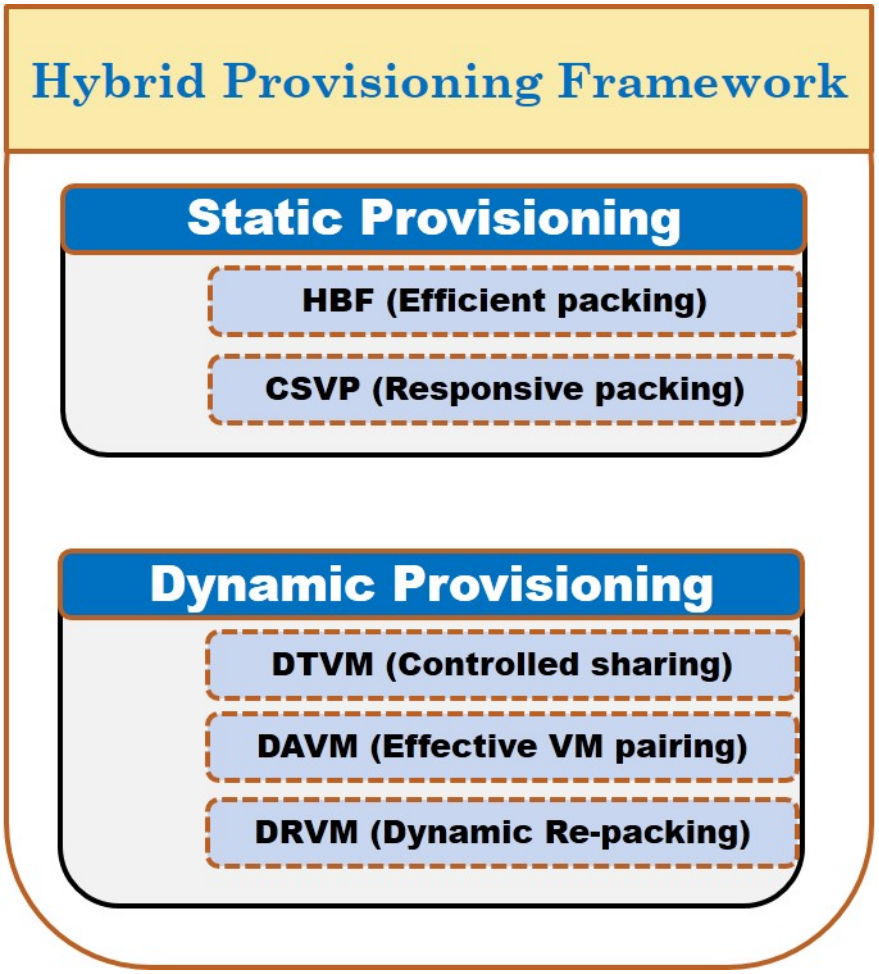

Figure 5.1: Hybrid Provisioning Framework

changes in system load and hence performance. Reacting too quickly to changes in load may result in unnecessary and ultimately non-beneficial VM migrations. An appropriate heuristic must be able to ensure that the benefits of re-provisioning will continue to be effective over time. Ultimately, if we encounter a situation where we are unable to meet this criteria then that suggests DRVM (i.e. re-provisioning *all/some* existing VMs) is necessary. Figure 5.1 shows these algorithms that form a part of my hybrid provisioning framework.

I now describe my initial framework for supporting the use of a mix of static and

Algorithm 1: Initial Framework: Hybrid Static and Dynamic Provisioning (HSDP)

1. Repeat in each unit time
2. clusterHistories ← getNodeResourceUsageHistories( )
3. cloudHistory ← getClusterResourceUsageHistories(clusterHistories)
4. VmUsageRecords ← extractRecords(cloudHistory)
5. VmPredictedUsage ← predictUsageByRecords(VmUsageRecords)
6. **IF** VM_n requests arrive **then** //New VM requests
7. **Foreach** VM_i in VM_n **do**
8. **IF** VM_i.requestType equals urgent **then** onlineVmList ← VM_i
9. **ELSE** batchVmList ← VM_i
10. **End for**
11. **IF** onlineVmList.size is not zero **then**
12. **IF** isConsideringCompatibility equals FALSE **then** call **HBF**(onlineVmList)
13. **ELSE** call **CSVP**(onlineVmList)
14. **IF** batchPackingDeadline has past **then**
15. **IF** isConsideringCompatibility equals FALSE **then** call **HBF**(batchVmList, UsageHistory)
16. **ELSE** call **CSVP**(batchVmList, UsageHistory)

    //After static placement, DAVM and DRVM become effective on a periodic basis
17. **Foreach** record r in VmPredictedUsage **do**
18. **IF** r.predictedUsageSummary ≥ DAVMthresholdUsage **then**
19. selectedVms ← r.InstanceId
20. **End for**
21. **IF** DAVMpairingDeadline has past and possibilityForFindingDAVMpairs(selectedVms) = TRUE **then**
22. performDAVMmigrations(selectedVMs)
23. underUtilizedNodes ← findUnderUtilizedNodes (cloudHistory)
24. **IF** DRVMpairingDeadline has past and count(underUtilizedNodes) ≥ re-packingThreshold
25. selectedVMs ← benefitAnalysis(underUtilizedNodes, cloudHistory, VmPredictedUsage)
26. performDRVMMigration (selectedVMs)
27. // DTVM remains activated on all nodes
28. End Repeat

End Algorithm 1

dynamic algorithms to provide effective provisioning. My basic framework for DTVM or migrating a single VM based on a threshold being exceeded over some period is shown in Algorithm 1. I have implemented this framework in Eucalyptus [66] to verify its correctness. In that implementation, each Cluster Controller (CC) collects

performance information from each of its registered nodes and the VMs running in them using `getNodeResourceUsageHistories()` [line 2, Algorithm 1]. The collected information includes resource usage records of VM instances (in the corresponding node) and the information is forwarded to the Cloud Controller (CLC) by each CC via `getClusterResourceUsageHistories()` [line 3]. The VM information from the CC's need to aggregated at the CLC to discover better opportunities regarding provisioning and to apply any provisioning decision across clusters. After aggregating and collecting the performance information from the CCs, the load/utilization of each VM is extracted and the resulting information is used to predict the future resource usage of each VM. The CLC uses one or more prediction models and identifies the usage records corresponding to over-utilized instances it predicts will occur in the future and performs the necessary migrations to co-locate pairs of VMs that should run effectively together. (Pairs of VMs that are predicted to run well together could be determined by some form of VM behaviour characterization based on history information.) Information in `VmUsageRecords` permits assessment of past variation in use from the expected use which can then be used in predicting future use. This prediction is likely to be more accurate if longer-term history information is available [lines 4–5]. HBF handles the arrival of new VM requests. If any newly arrived VM request needs to be packed urgently then it is placed in onlineVmList otherwise the request is kept in batchVmList until the execution of next batchPackingDeadline. If the compatibility of VMs is to be considered during packing then the VMs are packed using my CSVP algorithm [lines 6–16]. After a certain time interval, using the predicted resource usage of already running VMs, the possibility of cre-

ating new DAVM pairs (for the resource critical VMs) is examined and a number of selected VMs are migrated if beneficial [lines 21–22]. Each such migration may generate a set of potential `underUtilizedNodes` and if the number of underutilized nodes exceeds a threshold then the impacted nodes are identified [line 23] using the history of actual resource usage information (`cloudHistory`). To reduce the number of underutilized VMs in the system, a list of `selectedVMs` is also generated that would be chosen for DRVM migration considering a cost-benefit analysis. The list of potential migration targets is ordered by the expected benefit of migration. The function `benefitAnalysis` that produces the list considers such additional factors as predicted longer-term resource sharing, network bandwidth and latency available, the cost of memory state migration, file migration and network re-configuration as well as the benefits of power savings etc. Migration is only done if there is a benefit [lines 24–26]. DTVM is continually activated in all nodes for balancing load fluctuations corresponding to short-lived changes [line 27].

In my framework, resource requirements, resource usage and threshold information are all compound representations encompassing multiple factors such as CPU cores, memory, storage, network, etc. need, use and limits, respectively. While the pseudo-code in Algorithm 1, presents operations on these values as simple comparisons, they are not. In different situations, the operations may require differences in behaviour to ensure appropriate outcomes. Generally, comparisons are done component-wise (e.g. CPU need vs. CPU available, memory need vs. memory available, etc.). Simple absolute comparisons of components, however, may be inappropriate in some cases so a "fuzzy" form of comparison can be used where the degree of flexibility may vary

between component values during comparisons. (E.g. The need for CPU capacity may be harder than for expected available network capacity.) The best form of fuzzy comparison, if actually used in practice, will need to be determined experimentally. Similarly, the threshold referred to in the algorithm (`DAVMthresholdUsage`) appears to be absolute and arbitrary. Initially such values are absolute but must also be carefully chosen experimentally if employed in an actual implementation. It may also be possible to set thresholds dynamically to adapt to changing system conditions.

5.2 Static Provisioning Algorithms

Online provisioning algorithms offer quick response at the price of less efficient packing of VMs to PMs. Batch algorithms improve packing efficiency but introduce delays in provisioning hardware for VMs. In many cases, it is not critical that requests by cloud clients to start new VMs be satisfied immediately. Many VM requests can tolerate short delays to allow a batch to accumulate and some would prefer even larger delays to achieve cost savings (e.g. the provision of "spot intstances" in Amazon EC2). As "on-demand" use of clouds increases for smaller duration work, however, the importance of being able to satisfy urgent VM requests increases. This suggests that online provisioning should be preferred but the decreased packing efficiency offered by online algorithms is a deterrent to their use in practice. My new HBF algorithm balances packing efficiency and responsiveness to urgent requests. It also incorporates an improved mechanism for packing VMs based on multiple criteria (e.g. CPU, memory and storage). Supporting effective multiple criteria based packing also has increasing importance as a wider range of applications are run in clouds. Typical

cloud applications often have a single predominant criterion that may be used for packing (e.g. a strong CPU focus). For new applications, it may be of increasing importance to balance packing based on multiple criteria. By supporting urgent VM requests HBF meets a new requirement of cloud users. By offering improved support for packing based on multiple criteria, HBF will also result in improved initial static packings that, in some cases, will lessen or delay the need for dynamic adjustments in VM placement.

Another way in which static placements may decrease the need for dynamic provisioning is by trying to ensure that the VMs that are placed together on PMs are unlikely to interfere with one another as they execute. VMs that interfere with other co-hosted VMs will require migration and it would be better if the associated overhead could be avoided. I therefore considered the compatibility of VMs in packing when I created my CSVP (Compatibility-based Static VM Placement) provisioning algorithm. I now first describe my HBF and then my CSVP static provisioning algorithms.

5.2.1 HBF (Hybrid BackFill)

Like static batch provisioning algorithms, HBF has a pre-defined HBF window/interval during which time it gathers VM requests into a list: vmList. At the end of each HBF interval, requests remaining in vmList are efficiently packed into available PMs using a variant on FFD. Unlike existing batch algorithms, however, HBF always respects VM request start times by provisioning resources for urgent requests that need to start prior to the end of the current interval by their requested start times. If multiple

urgent requests are available to be placed at the same time, their placement is done collectively to avoid obvious non-optimalities. Placing urgent requests before the end of the current interval improves responsiveness and thus HBF is better able to handle more short-duration, on-demand type VM requests. Such requests are, naturally, removed from the vmList once their provisioning is done. In the handling of urgent requests, HBF's behaviour is similar to online provisioning. HBF also differs from common batch algorithms in its aggressive use of backfilling (assigning VM requests to available resources in existing PMs rather than to those in a new PM). This is done not just when processing urgent requests but also prior to doing any batch provisioning. While this may have the effect of decreasing the number of VMs in a batch, it improves the likelihood of filling holes in existing PMs and, based on simulations (reported in the next chapter of this thesis), still commonly leaves a sufficiently large number of VM requests to permit efficient packing into new PMs. When backfilling, whether for handling urgent or batched requests, HBF also checks for any possible near perfect fit[1] in an existing hole before applying another algorithm (e.g. first fit), if needed, to determine the VM placement. Since backfilling is ongoing, there are relatively few available holes and it is easy to track them and therefore efficient to search for a near perfect fit.

In general, the HBF algorithm supports both online and batch operation to provide the important advantage of immediate packing (for urgent VM requests) and efficient packing for others. For efficiency, a cloud provider may want to queue VM requests for an extended period of time after which the VMs are packed as a batch. But

[1]Perfect packing is hard to achieve, particularly for multiple criteria. In general, perfection within some precision for each criterion is used.

there may be some VMs requests (e.g. for running time-sensitive applications like on-demand broadcasting, video editing, analytics, etc.) in the queue that need to start immediately or at least well before the next batch packing begins (at the end of each hour in the simple example below). In such cases, HBF performs online packing for the VMs that have strict time constraints. For example, referring to Figure 5.2, if a number of VM requests (VM_{1-10}) arrive in the queue during the batching period and three of them (i.e. VM_4, VM_7 and VM_9) need to start immediately (shown in red in the upper part of Figure 5.2), then HBF uses individual online packing for those 3 VMs (VM_4, VM_7, VM_9) and the other VMs are packed using batch packing (as shown in the lower part of Figure 5.2).

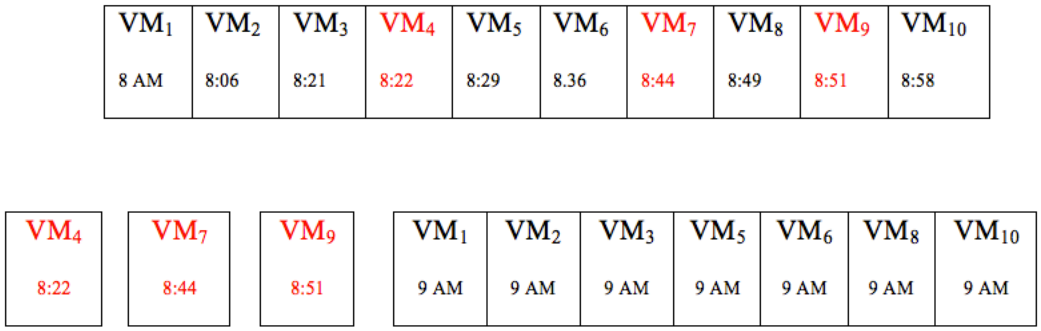

Figure 5.2: Example of HBF

For simplicity, my HBF algorithm is first described, without loss of generality, for a single criterion only (e.g. CPU need). Thus, I simply refer to the "size" of a VM request. Extending the algorithm to support multiple criteria can be done by changing any code that considers the size of a VM request (including underlying operations such as "first fit", "best fit", etc.). My improved technique with multi-criterion provisioning is described in detail after the basic algorithm is presented.

Algorithm 2: HBF(vmList)

1: Repeatedly checks vmList for new VM requests at fixed intervals (μ)
2: At current time t_i, compute $totalVMs_{t_i}$ = size($vmList_{t_i}$)
3: **IF** $totalVMs_{t_i}$ equals 0 then // no VM requests at this time
4: schedule next check at time $t_i + \mu$ **or on receipt of urgent request**
5: **IF** $totalVMs_{t_i}$ equals 1 then // Only one new VM request
6: BackFill if possible (try perfect fit then use First Fit if needed)
7: **IF** (unsuccesful) then
8: ForwardFill into new PM using First Fit
9: **IF** $totalVMs_{t_i}$ equals 2 // Two new VM requests (R1, R2) - optimize this case
10: set R1 = vmList[0] and R2 = vmList[1]
11: set N = size of each physical machine // all machines are identical
12: Find space S1 in a partially filled PM that meets the criteria:
13: $S1 < |R1| + |R2|$ and $S1 \geq |R2|$ // |Rx| denotes the size of Rx
14: Find space S2 in another partially filled PM that meets the criteria:
15: $S2 < |R2|$ and $S2 \geq |R1|$
16: **IF** S1 or S2 not available
17: BackFill when possible (try perfect fit then use First Fit if needed)
18: **IF** (unsuccessful) then
19: Forward fill using First Fit
20: **IF** S1>S2 & S1.HostID<S2.HostID // S1 placed earlier(First Fit)
21: BackFill with First Fit is sub optimal so use Best Fit
22: **ELIF** S1 > S2
23: BackFill with Worst Fit is sub optimal so use Best Fit
24: **ELSE** // S1 < S2
25: Try to BackFill with Worst Fit
26: **IF**(unsuccessful) then
27: ForwardFill using First Fit
28: **IF** $totalVMs_{t_i}$ equals 3 // Three new VM requests (R1, R2, R3) - optimize this case
29: set R1 = vmList[0], R2 = vmList [1] and R3= vmList [2]
30: set N = Total size of each physical machine
31: Find space S1 in a partially filled PM that meets the criteria:
32: $S1 < |R2| + |R3|$ and $S1 \geq |R1| + |R2|$
33: Find space S2 in another partially filled PM that meets the criteria:
34: $S2 < |R1| + |R3|$ and $S2 \geq |R3|$
35: **IF** S1 or S2 not available
36: BackFill if possible (try perfect fit then use First Fit if needed)
37: **IF** (unsuccessful) then
38: Forward fill using First Fit
39: **IF** S1 > S2 & S1.HostID > S2.HostID
40: BackFill with First Fit is sub optimal so use Best Fit
41: **IF** S1 > S2
42: BackFill with Best Fit is sub optimal so use Worst Fit
43: **ELSE** // S1 < S2
44: Try to BackFill with First Fit
45: **IF** (unsuccessful) then
46: ForwardFill using First Fit
47: **IF** $totalVMs_{t_i}$ is greater than 3
48: BackFill if possible (try perfect fit then use First Fit **Decreasing**)
49: ForwardFill remaining VMs in vmList using First Fit **Decreasing**

End Algorithm 2

Algorithm 2 is reasonably straightforward. The variable `totalVMs` stores the number of requests in `vmList` for the current window [line 2]. If, at the end of the window at time t_i, $\texttt{totalVMs}_{t_i}$ is zero, HBF waits for new VM request(s) to arrive in the next window [lines 3–4]. If there is a single VM request available in `vmList`, HBF tries to BackFill (if possible) using perfect fit first and then first fit, if necessary, into existing PMs. If BackFill is not possible, HBF ForwardFills (into a new PM) using First Fit [lines 5–8]. If there are two VM requests available in a window, HBF first attempts backfilling for one or both requests and, if unsuccessful for one or more then does forward filling. Filling is done carefully to avoid sub-optimal situations involving the two requests (R1,R2) [lines 9–27]. Suboptimal cases may occur in various scenarios if a single online algorithm (e.g. first/best/next fit) is always applied. Thus, I consider the specific requests being made and choose the best "fit" algorithm for the current situation. For example if two outstanding requests, |R1| = 1 and |R2| = 5 (out of 10) must be allocated to spaces S1 = 5 and S2 = 2 , then FirstFit would not perform well since it will not provide optimal packing as R1 will be assigned to S1 leaving space too small for R2. If there are three VM requests in the window, HBF again checks for sub-optimal situations, this time specific to the three requests (R1, R2, R3) and chooses the most suitable fit option for both backward and forward filling [lines 28–46]. Finally, if there are more than three VM requests, HBF first tries to BackFill each request with perfect fit and then ForwardFills the remainder using First Fit Decreasing (FFD) [lines 47–49]. The approach described in Algorithm 2 is also used for dealing with a group of urgent requests when they can be handled concurrently. Thus, they too benefit from the special handling to avoid sub-optimal provisioning.

This is important as it is likely that any batches of urgent requests will be small compared to normal requests. Handling of sub-optimal cases for small batches could also be applied for a larger number of specially handled requests (e.g. 4, 5, 6). I chose to consider only three requests or fewer since four requests provides sufficient potential for effective packing using FFD (in the final stage of the algorithm).

Supporting Multi-Criteria Packing

Provisioning of cloud resources must concurrently consider multiple criteria such as CPU cores, memory space and network capacity requirements, among others. Multi-criteria optimization is a complex problem and a number of techniques (e.g. [89; 53; 70; 99]) have been developed to try to combine criteria to reduce the problem back to single criterion packing. It is important to be able to quickly assess the "fit" of a VM request based on multiple criteria and, as such, the combining techniques need to be relatively simple. This naturally leads to potentially sacrificing some packing quality for efficiency of operation. Good results needs to achieve a balance between simplicity and packing quality.

Most approaches to multi-criteria packing involve combining a VM's requests for individual resources (CPU, memory, etc.) in some simple way. Resource requests are normally specified using numeric values (e.g. number of cores, gigabytes of RAM, etc. and are often normalized to a convenient range). A common way of combining such requests is to simply take their product and then sort the request sequence according to those products from largest to smallest and assign the VMs to PMs using FFD in the resulting order (e.g. as in [99]). This method is cheap and relatively effective

Table 5.1: Sub-optimal Performance of the Product Algorithm

Range of Values	Sub-optimal Cases
10	4
12	9
14	11
16	17
18	24
20	31

in terms of the resulting packing but there are specific situations where clearly sub-optimal results are obtained that could be easily avoided.

The range of values that may be specified for a particular resource varies depending on the type of resource in question. For example, the number of cores requested might be constrained to be between 1 and 8 on certain hardware while the number of megabytes of RAM might vary significantly more (e.g. from 512 to 16384). The wider the range of possible values for certain resources, the greater the possibility that the use of the "product algorithm" will produce sub-optimal results. A simple manual enumeration of the cases where the product algorithm produces sub-optimal results for small ranges shows that there is the potential for improvement, particularly when VMs request resources near the middle of their associated ranges. Table 5.1 shows six cases (for small ranges of values) where the product algorithm will produce sub-optimal results when combining two resource requests (e.g. CPU and memory). Notice that the number of cases grows with the size of the range. This is an undesirable side-effect of the product algorithm. A specific example of this problem occurs for the following sequence of requests (where the CPU and memory requirement of VM requests are both in the normalized range 1 to 10).

$$R_1 = (2, 6)$$
$$R_2 = (3, 4)$$
$$R_3 = (4, 3)$$
$$R_4 = (6, 2)$$

In this example the product of the two resource requirements of all the requests (R_is) is the same, 12, so they can appear in any order, relative to one another, when sorted by product alone. Thus, it is possible that R_1 will be packed with either R_2, R_3 or R_4. If R_1 is packed with R_2 or R_3, the packing will be sub-optimal. This is because little or no "space" would be left in the second (memory) dimension as the aggregate value of the packed requests for memory would be either 10 or 9. This would leave little opportunity for another request to be added and thus the available CPU in this example would be wasted. The host utilization for both resources (CPU and memory) can be better *balanced* by packing R_1 and R_4 in the same host, thereby leaving usable space (in both dimensions) for a later request.

Another example of this problem occurs for the following sequence of requests (where the CPU, storage and memory requirement of VM requests are all in the range 1 to 10) .

$$R_1 = (2, 4, 6)$$
$$R_2 = (3, 4, 4)$$
$$R_3 = (4, 3, 4)$$
$$R_4 = (6, 4, 2)$$

By packing R_1 and R_4 in the same host, the host utilization for all resources (CPU, storage, memory) is better balanced.

To achieve better balance of resource usage during packing in HBF, I simply sort by the sum of the resource requirements within those requests having the same product of resource requirements thereby grouping more "SLA compatible" requests together so that when FFD is applied in order they will be placed together resulting

Algorithm 2M: Improved Multi-Criteria Provisioning - mHBF(vmList)
1: Repeatedly Scan vmList for new VM requests at fixed interval (μ)
2: At time t_i, compute $totalVMs_{t_i} = size(vmList_{t_i})$
3: **IF** $totalVMs_{t_i}$ equals 0
4: then check again after at $t_i + \mu$ or on receipt of urgent request
5: **IF** $totalVMs_{t_i}$ equals 1 // Only one outstanding request
6: BackFill if possible (perfect fit else First Fit if needed)
7: If (unsuccessful) then
8: Forward fill using First Fit
9: **IF** $totalVMs_{t_i}$ is greater than 1
10: Sort $vmList_{t_i}$ based on sum within product of requirements
11: BackFill if possible (perfect fit else First Fit Decreasing if needed)
12: ForwardFill remaining VMs in vmList using First Fit Decreasing
End Algorithm 2M

in more efficient packing. Despite being very simple, this approach is, to the best of my knowledge, not discussed in the literature and is integrated into provisioning as shown in Algorithm 2M ("M" for Multi-criteria). HBF's improved multi-criteria packing ensures that more "compatible" VMs will be packed together to ensure better overall PM resource utilization. This type of compatibility is only related to static resource requirements and does not consider possible workload changes over time. To handle such possible workload changes while performing VM packing I created CSVP which is discussed in the following sub-section.

5.2.2 CSVP (Compatibility-based Static VM Placement)

Recall that to be useful in a static provisioning algorithm, any metric for assessing VM compatibility must be very simple. (This allows a non-expert cloud user to provide an initial assessment and very low overhead VM monitoring to assess VM behaviour during a "profiling" run.) At the most fundamental level, before packing

the VMs, my CSVP algorithm simply divides them into those that are assumed to be resource critical and those that are not. This is done using a tuneable threshold value. If a VM is expected to execute at or above the threshold, it is declared to be resource critical, otherwise, it is not. To bootstrap the packing, a user is expected to try to estimate whether or not a new VM will exceed this threshold during its execution to determine if it is resource critical. In most cases, this will be feasible but even if some users mis-characterize their new VMs, a negative performance impact is very unlikely (relative to the use of FFD alone without any compatibility consideration) and will be very short lived, since after the first run profile/feedback information on actual behaviour will be available. While a user is only expected to estimate whether or not a new VM is resource critical, it is straightforward for the feedback mechanism to be somewhat more specific. Thus, my CSVP algorithm actually divides all VMs into three categories: low, medium and high resource criticality and the highly resource critical VMs above the key threshold into a small number of sub-categories representing more refined degrees of criticality. This allows the algorithm to more aggressively target the VMs that, based on trace information, are known to have been more highly resource critical in prior run(s). Obviously, a feedback based approach cannot be used until a run has been done to provide feedback information. Thus, the initial user "guess" at resource criticality is needed to bootstrap the packing.

Knowing that VMs from the Google traces [74] exhibit both coarser and finer grained variation in resource demand, this very simple characterization of VMs is a reasonable approach. Cloud users will commonly only be aware of the coarse variations and thus will use them to determine SLA values for the VM. This leaves the

Algorithm 3: CSVP(vmList)

```
1. Repeatedly scan vmList for new vmRequests at fixed interval (μ);
2. Sort vmList in descending order depending on SLA requirements;
3. Foreach VM_i in vmList
4.       IF isResourceCritical(VM_i) equals TRUE then
5.            add VM_i into RCList
6.       ELSE add VM_i into notRCList
7. End for
8. Foreach VM_A in RCList
9.       findHost(VM_A) and remove VM_A from RCList
10.      size_A ← VMA.SLArequirements and size_B ← 0
11.      Foreach VM_B in notRCList
12.           size_B ← size_B + VM_B.SLArequirements
13.           findHost(VM_B) and remove VM_B from notRCList
14.           IF (size_B ≥ size_A) break
15.      End for
16.End for
17. Foreach VM_B in notRCList do
18.      findHost(VM_B)
19. End for
End Algorithm 3
```

finer grained variations as an opportunity for improvement through resource sharing between co-located VMs. To determine the characterization of VM workloads, I manually explored a significant random sample of the Google Traces [74] to try to understand what the behavioural characteristics of actual workloads were. I naturally saw a wide range of behaviours, but in looking for **simple** metrics for resource criticality I realized that a relatively crude characterization based on the time spent by a VM at or above a given threshold for resource demand would, for a large number of VMs, be a reasonable predictor of resource criticality that, for the purposes of judging VM compatibility, might lead to a static placement algorithm that would offer better performance. This has since been confirmed by my simulation results (presented in the following chapter) and is the basis of my feedback-driven CSVP approach.

Algorithm 3 describes CSVP which attempts to balance the aggregate resource requests for critical and non-critical VMs in each host. During a fixed interval, VM allocation requests are gathered into `vmList` [Algorithm 3, line 1]. The vmList is then sorted in descending order depending on the SLA requirement of the VM requests [line 2]. The VM requests in vmList are then categorized considering the criticality of an SLA requirement (e.g. CPU) and arranged into two different lists: RCList and notRCList. The RCList contains the resource critical VMs only and the remaining VMs are placed into the notRCList [lines 3-7]. The algorithm toggles between the RCList and notRCList lists and finds a suitable host for each VM. (That is, it finds non resource critical hosts for resource critical VMs and resource critical hosts for non resource critical VMs.) The toggling is triggered after allocating each resource critical VM and also after allocating a sufficient number of non resource critical VMs to match (i.e. the total SLA requirements of successive non resource critical VMs will be close to the SLA requirement of the just packed resource critical VM) [lines 8-16]. If there are not enough VMs (in any category) to toggle, then the remaining VMs must simply be co-allocated in a suitable host [lines 17-19].

Algorithm 3.1 finds a suitable host for each VM_i depending on the aggregated resource-criticality of the already allocated VMs in a host and of VM_i. **PMList** is a list that contains the information about already used **PMs/hosts** [Algorithm 1.1, line 1]. The hosts in PMList are characterized using the aggregated resource-criticality and are categorized into two different lists: RCpmList and notRCpmList. The RCpmList contains the hosts that have a positive value for the aggregated resource-criticality and the hosts having a negative value are placed into the notRCpmList [lines 2-11].

Algorithm 3.1: findHost(VM_i)

1. create PMList containing all physical machines having at least one virtual machine
2. **Foreach** PM_k in PMList
3. aggregatedRC(PM_k) ← 0
4. **Foreach** VM_i in PM_k
5. aggregatedRC(PM_k) ← aggregatedRC(PM_k) + VM_i.SLArequirements * isResourceCritical(VM_i)
6. // isResourceCritical(VM_i) method returns +1 if VM_i is resource critical else returns -1
7. **End for**
8. **IF** aggregatedRC(PM_k) greater than or equals 1 **then**
9. add PM_k into RCpmList
10. **ELSE** add PM_k into notRCpmList
11. **End for**
12. Sort RCpmList in descending order depending on aggregated resource-criticality
13. Sort notRCpmList in ascending order depending on aggregated resource-criticality
14. **IF** isResourceCritical(VM_i) equals 1 **then** //backfilling
15. try to pack VM_i into notRCpmList
16. **IF** unsuccessful **then** try to pack VM_i into RCpmList
17. **ELSE**
18. try to pack VM_i into RCpmList
19. **IF** unsuccessful **then** try to pack VM_i into notRCpmList
20. **IF** VM_i.isAlreadyPacked equals unsuccessful **then** //forwardfilling
21. pack VM_i into a new physical machine

End Algorithm 3.1

The RCpmList is then sorted in descending order and the notRCpmList in ascending order depending on the aggregated resource-criticality values [lines 12-13]. If VM_i is resource critical then the algorithm tries to find a host firstly in the notRCpmList and if there is not enough space in any of the hosts of notRCpmList then the RCpmList is searched to find a suitable host. On the other hand, if VM_i is not resource critical then the algorithm tries to find a host firstly in the RCpmList and if there is not enough space in any of the hosts of RCpmList then the notRCpmList is searched to find a suitable host [lines 14-19]. Finally, there may arise a need to bring additional hosts online to allocate VM_i in the cases when none of the already used hosts have enough space for VM_i [lines 20-21].

Algorithm 3M: mCSVP(vmList)

1. Repeatedly scan vmList for new vmRequests at fixed interval (μ)
2. Sort vmList in descending order depending on SLA requirements
3. **Foreach** VM_i in vmList
4. **IF** isCpuResourceCritical(VM_i) equals 1 and isIOResourceCritical (VM_i) equals 1 **then**
5. add VM_i into bothRCList
6. **ELIF** isCpuResourceCritical(VM_i) equals 1 and isIOResourceCritical (VM_i) equals 0 **then**
7. add VM_i into cpuOnlyRCList
8. **ELIF** isCpuResourceCritical(VM_i) equals 0 and isIOResourceCritical (VM_i) equals 1 **then**
9. add VM_i into ioOnlyRCList
10. **ELSE** add VM_i into notRCList
11. **End for**
12. **Foreach** VM_A in bothRCList
13. mFindHost(VM_A)
14. remove VM_A from bothRCList
15. $size_A \leftarrow VM_A$.SLArequirements and $size_B \leftarrow 0$
16. **Foreach** VM_B in notRCList
17. $size_B \leftarrow size_B + VM_B$.SLArequirements
18. mFindHost(VM_B)
19. remove VM_B from notRCList
20. **IF** $size_B \geq size_A$ **then** break
21. **End for**
22. **IF** bothRCList is empty or notRCList is empty **then**
23. **Foreach** $VM_B remaining$ in notRCList add $VM_B remaining$ into ioOnlyRCList **End for**
24. **Foreach** $VM_A remaining$ in bothRCList add $VM_A remaining$ into cpuOnlyRCList **End for**
25. break
26. **End for**
27. **Foreach** VM_A in cpuOnlyRCList
28. mFindHost(VM_A)
29. remove VM_A from cpuOnlyRCList
30. $size_A \leftarrow VM_A$.SLArequirements and $size_B \leftarrow 0$
31. **Foreach** VM_B in ioOnlyRCList
32. $size_B \leftarrow size_B + VM_B$.SLArequirements
33. mFindHost(VM_B)
34. remove VM_B from ioOnlyRCList
35. **IF** $size_B \geq size_A$ **then** break
36. **End for**
37. **End for**
38. **Foreach** VM_B in ioOnlyRCList
39. mFindHost(VM_B)
40. **End for**

End Algorithm 3M

Algorithm 3M ("M" for Multi-criteria) presents a version of CSVP (mCSVP) that considers the resource criticality of VMs addressing two SLA requirements (e.g. CPU and IO) at the same time. The principle behind Algorithm 3M is almost identical to Algorithm 3, but the VM allocation requests in vmList are now categorized and arranged into four (not two) different lists: bothRCList, cpuOnlyRCList, ioOnlyRCList and notRCList. The bothRCList contains the VMs that are resource critical considering both CPU and IO. The cpuOnlyRCList contains the VMs that are resource critical considering CPU only and similarly the ioOnlyRCList contains VMs that are resource critical considering IO only. The remaining VMs are placed into the notRCList [lines 3-11]. The algorithm intially tries to toggle between the bothRCList and notRCList lists and tries to find a suitable host for VMs from both lists. The toggling is triggered after allocating each resource critical VM and also after allocating a sufficient balancing/off-setting amount of non resource critical VMs (i.e. again the total SLA requirements of successive non resource critical VMs is close to the SLA requirement of the just packed resource critical VM) [lines 12-21]. If there are insufficient VM requests in bothRCList and/or in notRCList (to toggle) then again toggling takes place for the VMs of cpuOnlyRCList and ioOnlyRCList [lines 22-37]. If there are not enough VMs (in any category) to toggle then the remaining VMs must simply be co-located in a suitable host [lines 38-40].

Algorithm 3M.1 finds a suitable host for each VM_i depending on the aggregated resource-criticality of the already allocated VMs in a host and of VM_i. In this case, the resource-critically is measured for more than one SLA (i.e. CPU and IO individually) [Algorithm 3M.1, lines 1-9]. The **hosts** in PMList are characterized using the

aggregated resource-criticality of both CPU and IO and are, in this case, categorized into four different lists: bothRCpmList, cpuOnlyRCpmList, ioOnlyRCpmList and notRCpmList. The RCpmList contains the hosts that have a positive value for the aggregated resource-criticality for both CPU and IO and the hosts having a negative value for both of them are placed into the notRCpmList. The cpuOnlyRCpmList and ioOnlyRCpmList contain the hosts that have a positive value for the aggregated resource-criticality for CPU only and IO only, respectively [lines 10-18]. If VM_i has specific type(s) of resource criticality then the algorithm tries to find a space in a suitable compatible host [lines 19-26]. There may arise a need to bring additional hosts online to allocate VM_i in the cases when the already used hosts do not have enough space for VM_i [lines 27-28].

In general there should be no negative side-effects of using the CSVP algorithm (or mCSVP). As described in the case of the initial user characterization, a mis-characterization (e.g. due to changing behaviour across VM runs) should be no worse than placement without considering compatibility at all, except possibly for rare pathological placements. Our simulation results (shown in the next chapter) using a representative sample of the Google traces suggest that this is not likely to be a problem. The efficiency obtained through static methods will be necessarily inferior to that using dynamic methods for VMs with workloads that vary significantly over time but there is no overhead during VM execution. Since a goal of this thesis is to broaden the range of applications that can be run successfully in a cloud and since this would certainly include VMs with varying workloads, I also created a number of dynamic algorithms that are described in the following section.

Algorithm 3M.1: mFindHost(VM_i)

1. create PMList with the physical machines having at least one virtual machine
2. **Foreach** PM_k PMList
3. aggregatedCpuCritical(PM_k) ← 0 and aggregatedIOCritical(PM_k) ← 0
4. **Foreach** VM_i in PM_k
5. aggregatedCpuCritical(PM_k) ← aggregatedCpuCritical(PM_k) +
6. VM_i.SLArequirements * isCpuResourceCritical(VM_i)
7. aggregatedIOCritical(PM_k) ← aggregatedIOCritical (PM_k) +
8. VM_i.SLArequirements * isIOResourceCritical (VM_i)
9. **End for**
10. **IF** aggregatedCpuCritical(PM_k) greater than 0 and aggregatedIOCritical(PM_k) greater than 0 **then**
11. add PM_k into bothRCpmList
12. **IF** aggregatedCpuCritical(PM_k) greater than 0 and aggregatedIOCritical (PM_k) less or equal 0 **then**
13. add PM_k into cpuOnlyRCpmList
14. **IF** aggregatedCpuCritical(PM_k) less or equal 0 and aggregatedIOCritical (PM_k) greater than 0 **then**
15. add PM_k into ioOnlyRCpmList
16. **IF** aggregatedCpuCritical(PM_k) less than 0 and aggregatedIOCritical (PM_k) less than 0 **then**
17. add PM_k into notRCpmList
18. **End for**
19. **IF** isCpuResourceCritical(VM_i) equals 1 and isIOResourceCritical(VM_i) equals 1 **then**
20. newPMList ← merge(notRCpmList, ioOnlyRCpmList, cpuOnlyRCpmList, bothRCpmList)
21. **ELIF** isCpuResourceCritical(VM_i) equals 1 and isIOResourceCritical(VM_i) equals 0 **then**
22. newPMList ← merge(ioOnlyRCpmList, notRCpmList, bothRCpmList, cpuOnlyRCpmList)
23. **ELIF** isCpuResourceCritical(VM_i) equals 0 and isIOResourceCritical(VM_i) equals 1 **then**
24. newPMList ← merge(cpuOnlyRCpmList, bothRCpmList, notRCpmList, ioOnlyRCpmList)
25. **ELIF** isCpuResourceCritical(VM_i) equals 1 and isIOResourceCritical(VM_i) equals 0 **then**
26. newPMList ← merge(bothRCpmList, cpuOnlyRCpmList, ioOnlyRCpmList, notRCpmList)
27. Try to pack VM_i into newPMList //backfilling
28. **IF** unsuccessful **then** pack VM_i into a new physical machine //forwardfilling

End Algorithm 3M.1

5.3 Dynamic Provisioning Algorithms

I developed a family of three new dynamic provisioning techniques (DTVM, DAVM and DRVM). The challenge of dynamic provisioning is to ensure that the benefit of any change significantly outweighs the overhead of doing so and my family of algorithms collectively seeks to achieve this goal. A significant part of ensuring ef-

Figure 5.3: Checking Points

ficiency is simply running the various algorithms at appropriate times, In general, we want each algorithm to run frequently enough to avoid long-term negative conditions but infrequently enough to ensure that the information available about recent VM behaviour is sufficiently large to make good decisions. The run frequency for each algorithm is independently tunable but may be related to the run frequencies for other algorithms. Each of my dynamic techniques potentially takes place at specific but different time intervals, which I have termed **checking points**. The actual execution of the dynamic techniques occur at **execution** points. So, there are DTVM checking points, DAVM checking points, and DRVM checking points that determine the times when the particular techniques may possibly be applied and also DTVM, DAVM and DRVM execution points that occur when the various algorithms are actually applied. In the sample timeline shown in Figure 5.3, DTVM checking (and, potentially, execution) points are shown in blue which is also the basic unit of simulation time used in the evaluations presented in the next Chapter. DAVM checking and execution points are in red and DRVM execution points are shown in green. DTVM is executed locally in each physical machine. DAVM and DRVM checking points are periodic and are related to making global decisions regarding the migrations of VMs. Though DRVM checking points are also periodic, DRVM execution points are not since DRVM execution depends on specific conditions being fulfilled (i.e. the number of under-utilized

physical machines in the cloud must exceed a threshold) when the DRVM checks are done.

5.3.1 DTVM

To support controlled resource sharing among multiplexed VMs on a single PM, I developed the Differential Time shared VM Multiplexing (DTVM) algorithm - a short-lived resource sharing mechanism for co-hosted VMs in a physical machine. The level of workload variation in a VM depends on the type of application(s) hosted by the VM. Fluctuation of workload is very common, for example, when the VMs are used for hosting web sites, social media, IPTV services, etc. since the number of users and what they are doing fluctuates continually over fine grained timescales. The DTVM algorithm can effectively handle smaller, short-term workload variations in resource critical VMs by allowing other non resource critical VMs in the same physical machine to share resources with the resource critical ones. This sharing is closely controlled by DTVM and occurs only for short periods of time ensuring that one VM cannot abuse its access to the resources originally allocated to another.

The sharing of a resource between VMs depends on the type of resource. In the case of a dedicated resource (e.g. a CPU core), the VMs lend entire resources to other VMs for a certain period of time. If a resource (e.g. CPU core, IO resource) is not dedicated exclusively to a VM, then to share the resource with another VM, a relative amount of time is allocated for each VM to access the resource. The relative access time is measured depending on the level of sharing resources among VMs. For example, in Figure 5.4, two virtual machines: v1 and v2 have been assigned equal

execution times on some CPU resource (upper part of the figure). This is an example of simple time-shared multiplexing which is common in many cloud systems. If v1 needs to obtain a higher percentage of the execution time for a short time period, it cannot. Using DTVM (lower part of Figure 5.4), however, v1 can be given higher priority than v2 and can thus be treated preferentially. In this example, DTVM then provides greater execution times (larger time slices) for v1. Providing higher priority to a VM or providing more importance to certain VMs at runtime can improve overall execution performance for resource-critical cloud VMs and avoid unnecessary VM migrations. This assumes that there are other co-hosted VMs available that do not require all of their allocated resources. It is not acceptable to give extra resources to a resource-critical VM if it causes the donor VM to violate its SLA.

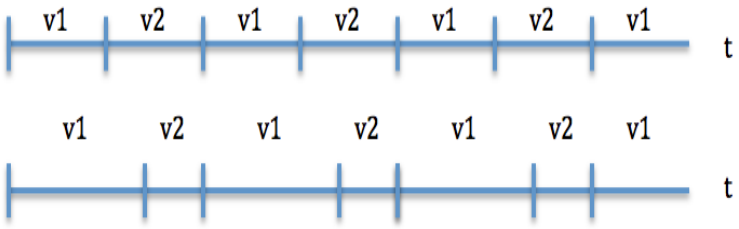

Figure 5.4: Resource sharing flexibility using DTVM

If the current CPU usage value of a resource-critical VM is higher than the DTVM CPU load threshold (maxUsageThreshold), then that VM is considered to currently be in an over-utilized state (with respect to CPU). For an over-utilized VM, DTVM checks for the possibility of obtaining resources from other non resource critical VMs (in the same physical machine). If any VM is found that is under-utilized (i.e. which has CPU usage value lower than the DTVM CPU load threshold) then DTVM calculates the amount of CPU resources that can be obtained (i.e. "borrowed") from the

under-utilized VM (assuming, of course, that the application(s) in the VM are able to use multiple cores). The amount is determined in such a way that the under-utilized VM does not itself become an over-utilized VM after providing the CPU resources. If the amount of CPU resources obtained from the under-utilized VM is not sufficient for the over-utilized VM then DTVM checks all other under-utilized VMs sequentially to see if it can obtain additional resources from them, if possible. In the same way, considering IO resources, DTVM tries to discover under-utilized VMs to share IO resources with each over-utilized VM.

At each DTVM checking point, the resource critical VMs are considered in each physical machine and DTVM is used to provide resource sharing in a controlled manner. My DTVM technique is presented in Algorithm 4 where pmList contains all the information for all the physical machines (PMs) in the system, vmList contains all the information about the hosted VMs in each physical machine, and all the already created VM pairs due to other provisioning techniques (i.e. DAVM, DRVM or CSVP) are contained in vmPairsAll list [lines 1-4]. The pairs created by other provisioning approaches are placed into separate lists to allow them to be managed differently but those lists can be easily merged as vmPairsAll list for DTVM. The list vmListRC is used to hold the resource critical VMs and vmListNotRC is used to hold the non resource critical VMs in each physical machine [line 5]. The resource critical VMs (in vmListRC) are sorted in descending order according to their resource requirements to consider the most critical VMs first. The non resource critical VMs (in vmListNotRC) are also sorted but in ascending order [lines 6-7]. For a resource critical VM, in vmListRC, if the current resource requirement crosses the maxUsageThreshold then a

number of non resource critical VMs in vmListNotRCNew are considered for resource sharing. If the resouce critical VM is paired with any other non resource critical VM then that VM (the non resounce-critical one) is placed in front of the vmListNotRCNew list (to be considered first). Otherwise vmListNotRCNew contains only those VMs that are not paired (due to DAVM, DRVM or CSVP) with other VMs [lines 8-16]. From each VM of vmListNotRCNew list, DTVM sequentially tries to obtain a particular amount of resources for the resource critical VM. But in this case, DTVM always try to obtain resources from those VMs that won't become overloaded themselves because of resource sharing [lines 17-25]. All the current resource utilizations of the VMs are recorded in resourceUsageHistory which is used with my DAVM and DRVM techniques [lines 23,26].

5.3.2 DAVM

DAVM (**D**TVM-**A**ware **V**M re-**M**ultiplexing) has the goal of making effective VM pairs (using necessary migrations) so that the co-located VMs can share the PM resources efficiently. To be effective, a pair of VMs needs to be compatible for as long a time as possible. Of course, pairings can be done repeatedly (including with other pairs) to produce larger groupings of compatible VMs. To make effective VM pairs, **predicted** resource usage information is used. Prediction of expected resource usage helps to identify those VMs that are likely to be over-utilized and under-utilized in the future. At each DTVM checking point, the current resource usage information of each VM is saved to create a historical dataset so that the recent resource usage information of a VM can be used for making future workload predictions by my DAVM algorithm.

Algorithm 4: DTVM (maxUsageThreshold, pairsCSVP, pairsDAVM, pairsDRVM)

```
1. pmList ← getCurrentlyRunningPhysicalMachineLists( )
2. Foreach PM_i in pmList do
3.     vmList_i ← getVirtualMachineLists(PM_i)
4.     vmPairsAll ← getAllCurrentPairs (parisCSVP, pairsDAVM, pairsDRVM)
5.     vmListRC_i ← getRcVMs(VM_i) and vmListNotRC_i ← getNotRcVMs(VM_i)
6.     vmListRC_i ← sort(vmListRC_i, descending) // sorted depending on current resource requirements
7.     vmListNotRC_i ← sort(vmListNotRC_i, ascending)
8.     Foreach VM_i,j in vmListRC_i do
9.         IF VM_i,j.ResourceUsagewithOwnResource > maxUsageThreshold then
10.            PairofVM_i,j ← searchPairofVM_i,j(vmPairsAll)
11.            IF PairofVM_i,j is not null then
12.                vmListNotRCNew_i ← PairofVM_i,j //place as first element in vmListNotRC_i
13.            Foreach VM_n in vmListNotRC_i do
14.                IF isPaired(VM_n,vmPairsAll) equals FALSE then
15.                    vmListNotRCNew_i ← VM_n //including all non-paired and non resource critical VMs
16.            End for
17.           Foreach VM_i,k in vmListNotRCNew_i do
18.                borrowingResource ← VM_i,k.hasAvailableResourcesToShare
19.                IF borrowingResource > 0 AND
20.                VMi,j.ResourceUsagewithCurrentResource+borrowingResource ≤ maxUsageThreshold then
21.                    VM_i,j.currentResource ← VM_i,j.currentResource + borrowingResource
22.                    VM_i,k.currentResource ← VM_i,k.currentResource − borrowingResource
23.                    resourceUsageHistory ← recordResourceUsage (VM_i,k )
24.                    break
25.            End for
26.        resourceUsageHistory ← recordResourceUsage (VM_i,j )
27.    End for
28. End For
29. Return resourceUsageHistory
End Algorithm 4
```

A DAVM interval is the time period between two consecutive DAVM checking points. At each DAVM checking point, a resource usage prediction for each VM is made that covers the period until the next DAVM checking point using a time-series prediction model. The impact of larger DAVM interval may degrade the accuracy of predictions but can be considered for discovering potential long lived pairings.

DAVM defines a set of load thresholds for each resource type, currently CPU and

IO, termed DAVM load thresholds. If a predicted/expected CPU usage value for a VM at any time in the following, future, DAVM interval is higher than the DAVM CPU load threshold, then the VM is considered to be in a resource critical state at that particular time with respect to CPU resources. All those VMs that have at least one predicted CPU usage value higher than the DAVM CPU load threshold are considered to be resource critical VMs. Similarly, those VMs that have predicted CPU usage always lower than the DAVM CPU load threshold are considered to be non resource critical VMs. I have also defined an "urgency parameter" for each resource critical VM that is a measure of the total extra (above threshold) resource requirements of a VM until the next DAVM checking point. Intuitively this is an indication of the resource criticality of the VM and can be used to help identify the most appropriate VMs to pair with, in order to achieve "balance" in resource sharing when the VMs are co-located. This is measured based on the total differences between the predicted values and the DAVM CPU load threshold in a DAVM interval (that is the sum of the differences between the predicted and threshold values at each sampling point in the interval). VMs with high resource criticality are also the ones that will benefit most from pairing and migration, if needed, and therefore are the ones that should be processed first.

The example in Figure 5.5 shows the predicted CPU usage percentage (based on a synthetic workload) of a VM for the time period 8 AM - 11 AM during a day. The scale used in this example is large. Of course, the technique is equally applicable at finer, potentially more realistic, scales as well. In this example the DAVM CPU load threshold is 73% and the predicted above threshold amounts are shown in rose color

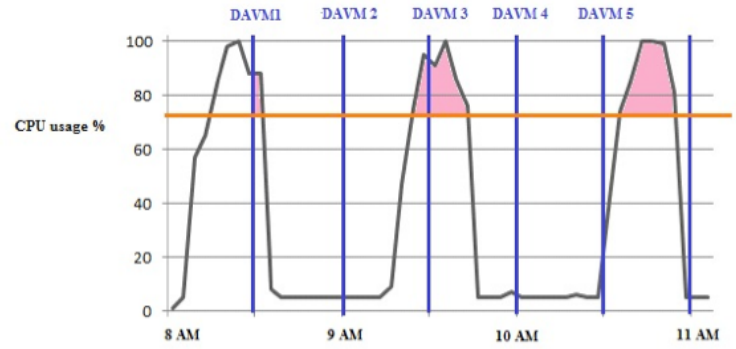

Figure 5.5: Example of DAVM

(for different DAVM intervals).

To consider the most highly resource critical VMs first, the algorithm sorts them considering their urgency parameter. To identify a proper match for each resource critical VM, my DAVM algorithm looks for one non resource critical VM among all the available non resource critical VMs where the extra resource requirements of the resource critical VM (considering CPU first and then IO) can be fulfilled through resource sharing. If there are multiple non resource critical VM solutions for making a pair (for a resource critical VM) that offer the same benefit, then DAVM always prefers a local VM (which is running in the same physical machine as the resource critical VM, if any) to make the pair so as to reduce migrations. If any resource critical VM finds a VM to pair with, then DAVM marks them as a potential pair and places them together by finding an appropriate host where the VMs in the pair can be co-located. If both VMs of a pair are currently running in the same physical machine, then they remain in the same physical machine. If, however, the VMs of a pair are currently running in different physical machines then DAVM looks for a possibility to place them in either of the existing physical machines (preferably, where the resource

critical or non resource critical VM is currently running). Only if the pair cannot be co-located in an already running physical machine then DAVM migrates the pair to a new physical machine, effectively preferring VM resource sharing compatibility over physical machine packing density.

Algorithm 5: DAVM (maxResourceUsageThreshold, usageHistory, iDAVM)

```
1. MigrationMap ← null; vmListRC ← null and vmListNotRC ← null
2. pmList ← getPhysicalMachineLists( )
3. Foreach PM_i in pmList do
4.     vmList_i ← getVirtualMachineLists(PM_i)
5.     Foreach VM_{i,j} in vmList_i do
6.         VM_{i,j} AboveThresholdCount ← predictWithHistoryData (usageHistory(VM_{i,j}), iDAVM, maxResourceUsageThreshold)
7.         IF VM_{i,j}.AboveThresholdCount >0 then
8.             vmListRC ← VM_{i,j}
9.         ELSE vmListNotRC ← VM_{i,j}
10.     End for
11. End for
12. Foreach VM_A in sort(vmListRC)
13.         possiblePairingListForVM_A ← null
14.         Foreach VM_B in vmListNotRC
15.             benfetOfPairingVM_{A,B} ← CheckPossibilityAndAmountofSharing (VM_A, VM_B)
16.             IF benfetOfPairingVM_{A,B} ≥ benefitThreshold then add VM_B into possiblePairingListForVM_A
17.         End For
18.         IF possiblePairingListForVM_A.size >0 then
19.             SelectedVm_BPairingForVM_A ← SelectBest(possiblePairingListForVM_A.Benefit)
20.         IF SelectedVm_BPairingForVM_A is not null then
21.             Remove SelectedVm_BPairingForVM_A from vmListNotRC
22.             MigrationMap ← AddIntoMigrationMap(VM_A, SelectedVM_BPairingForVM_A)
23. End for
24. IF MigrationMap equals not null then
25.     Perform all migrations
End Algorithm 5
```

Algorithm 5 presents the DAVM algorithm and is reasonably straightforward. The input parameter *usageHistory* contains the resource usage information, *maxResourceUsageThreshold* contains the load threshold, and *iDAVM* contains the time interval between two consecutive DAVM checking points. The *usageHistory* information is used by the predictWithHistoryData method invoked by the algorithm to predict future resource usage. This method must implement an accurate and inex-

pensive prediction model so that it can accurately predict likely future load behaviour without incurring excessive cost at runtime. Time-series prediction was selected as an appropriate strategy because of the nature of the problem. A specific time-series technique had to be chosen to meet the algorithm criteria. While an ARIMA model had been used previously for similar predictions in clouds [16], its potential runtime overhead seemed too high for practical use in the DAVM algorithm assuming a reasonable DAVM interval. Experimenting using data from the Google Traces [74] I discovered that Holt-Winters Exponential Smoothing [13], which has lower overhead, could be used effectively for making the necessary predictions despite the fact that VM workloads may not always be expected to have clearly defined seasonality.[2]. I found that my test results gave good predictions in approximately 95% of the cases.

MigrationMap is used to store the information regarding all the migrations to perform; *vmListRC* and *vmListNotRC* are two different lists used to hold the information for resource critical and non resource critical VMs correspondingly [Algorithm 5, line 1]. The variable *pmList* cotains the information for all the physical machines and *vmList* contains the information for all the VMs in each PM. For a certain DAVM checking point, the *usageHistory* of each VM is used to predict the expected resource usage information until the next DAVM checking point (i.e. for an *iDAVM* interval). The *predictWithHistoryData* method formally makes a call to a routine for Holt-Winters prediction as needed. From the predicted resource usage, the VMs are categorized into either resource critical (placed into *vmListRC*) or non resource critical (placed into *vmListNotRC*) depending on the predicted resource usage exceeding

[2]Applications running in VMs may exhibit repeated behaviour due to iteration at various levels of granularity and VMs may also be run at certain times of day but there is no guarantee of seasonality for all VMs.

maxResourceUsageThreshold [lines 2-11]. *vmListRC* is then sorted to consider the most highly resource critical VMs first. For each resource critical VM in *vmListRC*, DAVM creates a *possiblePairingListForVM* considering the possibility of resource sharing with the underutilized VMs in *vmListNotRC*. The amount of resource sharing possible is used as a metric of benefit and this benefit is used to identify an effective pair (where the benefit of pairing crosses a certain benefitThreshold) [lines 12-23]. After finding all effective pairs, the necessary VM migrations take place in the system [lines 24-25].

5.3.3 DRVM

If DAVM has to bring many new physical machines online for co-hosting its pairs, then the VMs in the cloud will be distributed across more physical machines. With more physical machines there will be fewer VMs per physical machine so there will be lower resource utilization in the physical machines (mostly where VMs have been migrated from). If the number of newly started physical machines (detected by checking at regular intervals) exceeds a threshold, then DRVM (**D**ynamic **R**e-packing of **VM**s) is initiated with the goal to be able to shutdown under-utilized physical machines after efficiently re-packing the VMs of those under-utilized physical machines into new physical machines. To identify under-utilized physical machines, a host resource utilization history dataset is maintained for each host that contains the resource usage information of the host for the time duration from the last DRVM checking point - the "DRVM history window". As DRVM can perform re-packing individually and apart from DAVM, DRVM and DAVM may have individual checking points. The

frequency of DRVM checking can also be made relative to number of DAVM pairs created. If there are more DAVM pairs created then there are more chances of getting under-utilized physical machines due to migrations and then it needs more frequent checking for DRVM. At a particular DRVM checking and chosen execution point, DRVM classifies the VMs in under-utilized hosts into 4 different classes:

- Over-utilized VM Class (OVC)
- Under-utilized VM Class (UVC)
- Single Stable Class (SSC)
- Paired Stable Class (PSC)

Among those VMs, if there are VMs that have been paired during the immediately preceding DAVM checking point (prior to the current DRVM checking point), then both VMs of the DAVM pairs are classified into PSC leaving them untouched so DRVM does not inappropriately "undo the work" of DAVM. DRVM then predicts resource usage for the remaining VMs that have not been classified yet. DRVM uses a DRVM prediction window (i.e. the time between two consecutive DRVM checkings) as the time period for which DRVM predicts resource usage for the unclassified VMs. DRVM also defines a set of load thresholds for both CPU and IO, termed DRVM load thresholds. If there are VMs (not paired yet) whose utilizations are predicted to be stable, stationary, or at a satisfactory level in the upcoming DRVM window, those are classified into class SSC. If a predicted CPU usage value for a VM at any time in the following, future DRVM interval, is higher than the DRVM CPU load threshold, then the VM is considered to be in a resource critical state at that

particular time. All those VMs that have predicted resource (e.g. CPU, IO) usage higher than the corresponding DRVM load threshold for a significant time period (i.e. at least 10 % of the predicted period) are considered to be in OVC. If there is not sufficient (i.e. less than 10 % of the predicted period) predicted resource critical states for a particular VM in the future DRVM window then the VM is classified into UVC. After classifying the VMs of under-utilized physical machines, they are re-packed in a particular sequence to obtain better packing and also for compatibility to address anticipated resource sharing. This sequence is described further below. DRVM checking is performed regularly after a fixed time interval but the necessary DRVM executions (i.e. re-packing) do not take place until there is a sufficient number (an absolute minimum of at least two) under-utilized PMs found during a DRVM checking.

VMs in SSC and PSC are re-packed after the VMs in OVC and UVC (and also after finding potential pairs among them). Re-packing is done in an order so that DRVM can best place compatible VMs together and also can achieve better packing (as the new pairs made with OVC and UVC are based on future predictions and also the combined SLAs would be higher compared to SSC, so they will be packed together and won't create underutilized physical machine in future checking). For each VM in OVC, DRVM tries to find a VM in UVC for making a potential DRVM pair. While creating any DRVM pair, the information related to effective pairing time (i.e. minimum predicted duration for a DRVM pair), for how long the pair will be considered, is associated with each VM in the pair. If there is no suitable pair for any VM in OVC, it is re-packed (sequentially with other VMs in newly started

PMs). After re-packing all VMs in OVC, if there remain any VMs in UVC then they are also re-packed (sequentially with other VMs in newly started PMs). VMs in SSC and PSC are then considered to create a new sorted VM list (depending on the SLA requirements; for paired VMs, the aggregated SLA requirements are considered) the contents of which are re-packed sequentially. To reduce the number of migrations, if any SSC VM or PSC VM pair is found alone in an existing physical machine, the VM(s) are not migrated elsewhere instead, that under-utilized physical machine is used for re-packing the rest of the VMs in the list (instead of shutting down that physical machine and potentially only having to start another). After all re-packing, any empty PMs are powered down to save energy costs. DRVM processing, of course, has the effect of improving the overall physical machine utilization.

In algorithm 6 the parameter *usageHistory* contains the resource usage information, *iDRVM* contains the time interval between two consecutive DRVM checking points, *pairsDAVM* contains the already existing VM pairs created by DAVM, and *thrDRVM* contains a set of load thresholds for the various resource types, CPU and IO, as presented in the algorithm. *MigrationMap* is initialized to store the information regarding all the migrations to perform; *listVmToRepack*, *listOVC*, *listUVC*, *listSSC*, *listPSC* are all initialized to contain VMs of the different categories that need to be considered when re-packing [Algorithm 6, lines 1-2]. The *pmList* contains the information on all the currently running physical machines and by analyzing their previous resource usage information the underutilized PMs are identified [lines 3-6]. All the VMs in each underutilized PM are considered for re-packing and those VMs are classified as overUtilized, underUtilized, singleStable, and pairedStable and

Algorithm 6: DRVM(usageHistory, iDRVM, pairsDAVM, thrDRVM)

1. MigrationMap ← null;
2. listVmToRepack ← null; listOVC ← null; listUVC ← null; listSSC ← null; and listPSC ← null;
3. pmList ← getPhysicalMachineLists( ) //currently running physical machines
4. **Foreach** PM_i in pmList **do**
5. PM_i.resourceUsageHistory ← getHistory(usageHistory (PM_i), iDRVM)
6. **IF** PM_i.resourceUsageHistory < pmResourceUsageThreshold **then**
7. **Foreach** $VM_{i,j}$ in PM_i **do**
8. $VM_{i.j}$.Category ← predictWithHistoryData (usageHistory($VM_{i,j}$), iDRVM, pairsDAVM, thrDRVM) //predictWithHistoryData method is to call Holt-Winters prediction routine
9. **IF** $VM_{i,j}$.Category equals overUtilized **then** listOVC ← $VM_{i,j}$
10. **ELIF** $VM_{i,j}$.Category equals underUtilized **then** listUVC ← $VM_{i,j}$
11. **ELIF** $VM_{i,j}$.Category equals singleStable **then** listSSC ← $VM_{i,j}$
12. **ELIF** $VM_{i,j}$.Category equals pairedStable **then** listPSC ← $VM_{i,j}$
13. **End for**
14. **End for**
15. **IF** listOVC.size >0 **then**
16. listOVC ← sort(listOVC) ; listUVC ← sort(listUVC)
17. **Foreach** VM_A in listOVC **do**
18. possiblePairingListForVM$_{A,B(s)}$ ← null;
19. **Foreach** VM_B in listUVC **do**
20. **IF** alreadyExistedinDAVMPairs(pairsDAVM, VM_A, VM_B) equals false **then**
21. possiblePairingListForVM$_{A,B(s)}$ ← FindPossibilityAndAmountofResourceSharing (VM_A, VM_B, thrDRVM)
22. **End for**
23. **IF** possiblePairingListForVM$_{A,B(s)}$ .size >0 **then**
24. SelectedVm$_{B(s)}$PairingForVm$_A$ ← SortAndSelect(possiblePairingListForVM$_{A,B(s)}$.Benefit)
25. MigrationMap ← AddPairIntoMigrationMap(VM_A, SelectedVm$_{B(s)}$PairingForVm$_A$)
26. Remove SelectedVm$_{B(s)}$PairingForVm$_A$ from listUVC and remove VM_A from listOVC
27. **ELSE** MigrationMap ← AddVmIntoMigrationMap(VM_A) and Remove VM_A from listOVC
28. **End for**
29. **IF** listUVC.size >0 **then** add remaining VMs into listVmToRepack
30. **IF** listPSC.size >0 **then** sort(listPSC) add each Vm into listVmToRepack
31. **IF** listSSC.size >0 **then** sort(listSSC) add each Vm into listVmToRepack
32. **Foreach** VMp in listVmToRepack **do**
33. MigrationMap ← AddVmIntoMigrationMap(VMp)
34. Remove VMp from listVmToRepack
35. **End for**
36. **IF** MigrationMap is not null **then**
37. Perform all migrations specified in MigrationMap

End Algorithm 6

correspondingly inserted into listOVC, listUVC, listSSC, and listPSC. From the predicted resource usage, the VMs are easily categorized into either over-utilized (and

thus placed into *listOVC*) or under-utilized (placed into *listUVC*) depending on the predicted resource usage exceeding *thrDRVM*. The predicted resource usages are also used to identify the stable workload VMs and to place them into either *listSSC* or *listPSC*. The VMs in listOVC and listUVC can be either single or paired and, for the paired VMs, aggregated resource usages are used for categorization [lines 7-13]. *listOVC* and *listUVC* are sorted (in descending order) depending on the SLA requirements of the corresponding VMs so that the VMs with the highest SLA requirements are processed first [lines 15-16]. For each over-utilized VM in *listOVC*, DRVM creates a $possiblePairingListForVM_{A,B(s)}$ considering the possibility of resource sharing with the underutilized VMs in *listUVC*. The amount of resource sharing possible is used as a metric of benefit and this benefit is used to identify an effective DRVM pair that should be migrated to the same physical host (this pairing approach is very similar to my DAVM technique and so, the resulting DRVM pairings can also reduce potential migrations during upcoming DAVM checking) [lines 17-28]. The VMs, that are already in DAVM pairs, are not considered for any DRVM pairing. After considering all the VMs in *listOVC* for **MigrationMap**, the remaining VMs in *listUVC* are used to create a new, *listVmToRepack*, and then the VMs of both *listPSC* and *listSSC* are included in that list sequentially after sorting them (in descending order considering SLA requirements) individually. The VMs in *listVmToRepack* are finally added into **MigrationMap** and the necessary VM migrations take place [lines 29-37].

Chapter 6

Evaluation

Assessment of cloud systems (including VM provisioning techniques) is complicated by the scale of cloud infrastructure. Few researchers have access to large-scale cloud facilities or data from them that they can use to do assessments. Small scale prototype systems can and should be used to demonstrate algorithm correctness but such systems cannot be used to assess scale-related effects. To address this shortcoming, some form of simulation is therefore also required. Most cloud systems software is also commercial and hence proprietary and unavailable to academic researchers. One exception is the Eucalyptus [67] open source cloud system which provides a basic but largely complete cloud testbed environment supporting cloud infrastructure consisting of clusters containing compute nodes running a variety of operating systems under a number of different virtual machine management packages. A number of simulation systems have also been developed for assessing cloud-based systems. Perhaps the most common and widely used of these is the Cloudsim [17; 87] simulator which supports the simulation of both the system and behaviour of cloud components including

VMs, resource provisioning policies, etc. Of importance to my research, CloudSim also provides custom interfaces for implementing new provisioning techniques.

6.1 Test-bed Setup and Results

I created a private Eucalyptus cloud testbed using dozens of physical (host) machines for the purpose of algorithm verification. The evaluation done using my testbed implementation was focused on the possibility of integrating static and dynamic VM provisioning in cloud infrastructure and on whether the integration can improve overall resource utilization. Thus, determining correctness of interactions between provisioning algorithms was of importance. I extended Eucalyptus with my new static and dynamic VM provisioning techniques and verified their integration and correctness. Additionally, I created tools in Eucalyptus to gather and save the resource (CPU, memory, etc.) usage information of VMs and hosts. I also added techniques to perform and control necessary migrations based on commands from the Cloud Controller (CLC) level of Eucalyptus.

An issue closely related to cloud test-beds is the generation of cloud workloads for testing with test-beds. As many cloud applications are web-based, the use of tools such as Olio [32] and the Faban web load generator [6; 32] are common. I used both these tools to create reasonably realistic, albeit synthetic, workloads for doing the basic testing of my algorithms.

6.1.1 Eucalyptus System Architecture

Eucalyptus implements IaaS (Infrastructure as a Service) level cloud computing functionality which is very similar to Amazon's EC2 [28] and which can be easily extended due to its simple organization and modular design. There are three major high level components in Eucalyptus: the Node Controller (NC), the Cluster Controller (CC) and the Cloud Controller (CLC) as shown in Figure 6.1. The VMs are provided by the underlying hypervisor used such as Xen [26], etc.

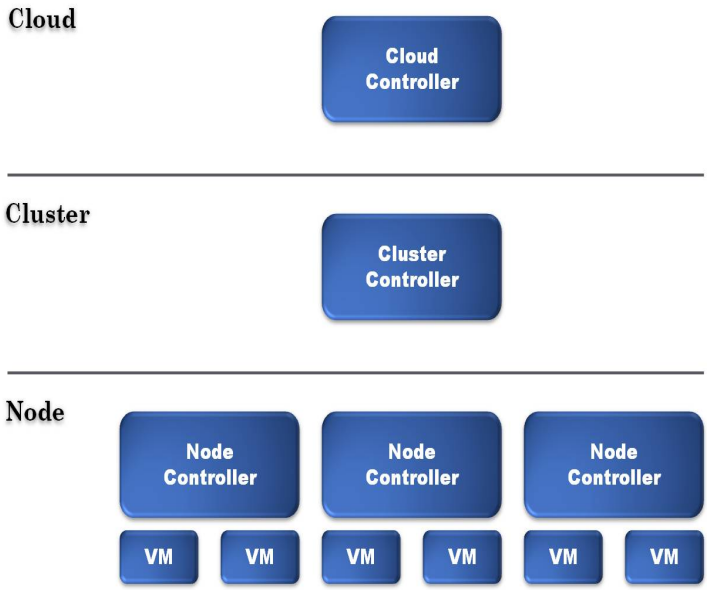

Figure 6.1: Different Levels of Control in Eucalyptus (adapted from [44])

Node Controllers (NCs) are installed on each physical node/host machine that is part of the Eucalyptus cloud infrastructure. Each NC makes the physical resources (i.e. computing core(s), memory, storage etc.) of the associated node available to the Eucalyptus cloud. When any new node is attached to a cloud, the Cloud Con-

troller (CLC) can then allocate and manage its physical resources in the form of VM instances via the NC of that particular node. Nodes, having the same type of hypervisor, are arranged under a cluster where each cluster contains a Cluster Controller (CC). The single CLC communicates with NCs through each particular cluster's CC. Each NC can query, control, allocate and/or manage the host operating system, the hypervisor of the node, VM instance(s) on the node, etc. upon request from the CLC.

Cluster Controllers (CCs) act as intermediaries between the CLC and the NC(s) registered in the corresponding cluster. Each CC gathers VM instance information from its associated nodes. Each CC also handles requests for new VM instance creation (from the CLC) by finding a node that has enough free resources to satisfy the new VM instance's resource requirements (i.e. CCs do the VM provisioning). Additionally, each CC passes VM instance termination, migration, or reboot requests from the CLC through to the relevant node's NC. The interfaces that maintain the communication between each CC and its NCs are described using WSDL [22]/SOAP [27].

The Cloud Controller (CLC) is the management entry point and major decision making component in Eucalyptus. The CLC is accessed using a number of web-services that can be categorized into: a resource management service, a data service and an interface service. Resource management services interact with CC(s) to allocate and de-allocate physical resources and to create and control VM instances. The data services create, integrate, modify and contain information related to users, systems and data storage. This information normally includes key-pairs, security groups, network definitions, management and monitoring information of VM instances, etc. The interface services are provided to allow communication with internal Eucalyptus

objects using standard web services security policies and to initiate high-level system management tasks.

There are three static VM provisioning techniques implemented (by the CCs) and distributed with Eucalyptus: the Round Robin, Greedy and PowerSave algorithms.

The Round Robin algorithm allocates new VMs to the host machines in a circular fashion effectively distributing the load across the available nodes. The Round Robin algorithm maintains a list of host machines and also a counter for identifying the next host machine in that list to be used when allocating resources for a new VM. The counter is incremented modulo the number of nodes after each new VM allocation to point to the next host machine in the list where the next VM allocation will take place. The Round Robin algorithm can prevent starvation as it considers each host machine equally and it is very good for balancing load (i.e. approximately equal distribution of total workload among the host machines in the cloud).

The Greedy algorithm maintains a sequence of host machines with available capacities and when a new VM needs to be allocated, the Greedy algorithm uses that sequence to find possible VM allocation locations. The Greedy algorithm allocates the new VM to the first host machine meeting the needs of the new VM, thereby packing VMs onto as few nodes as possible. The Greedy algorithm provides locally optimal allocation but often fails to achieve the best overall solution as it always considers the first possible solution without considering other possible allocation locations. Though the Greedy algorithm cannot balance the total VM workload among the host machines, it is reasonably fast.

The PowerSave algorithm enhances the Greedy algorithm by putting host ma-

chines to sleep when they are not running any VM(s) and reawakens a host machine when it is needed for allocating new VM(s). By shutting down the unused host machines, the PowerSave algorithm can reduce overall power consumption but it also decreases the speed of allocation due to the need to reawaken a node if no capacity exists on other nodes.

6.1.2 Implemented Changes to Eucalyptus

In Eucalyptus, the CLC contains 'euca2ools' which are used to query and display information about currently running VM instances in the cloud. The information produced provides VM instances' IDs along with their IP addresses and the relevant cluster's name. This information can be used to access any particular VM instance, through its IP address, to terminate or to reboot the VM instance, but the query result doesn't include information related to the node where the VM instance is currently running. The query result also doesn't provide any resource usage information for VM instances. Both of these pieces of information are required to support a cloud-wide hybrid provisioning approach. I therefore enhanced Eucalyptus so that a query result now also provides the corresponding nodes' performance information, considering available and used resources, along with their hosted VM instances' performance information, also considering allocated and used resources. I also modified the data structures maintained by the CLC, CCs, and NCs and also the message passing among them to permit collection and storage of the required performance information. This performance information provides the basis upon which we can identify over-utilized and under-utilized nodes and VM instances in the cloud that may need to be migrated

as part of a hybrid provisioning technique.

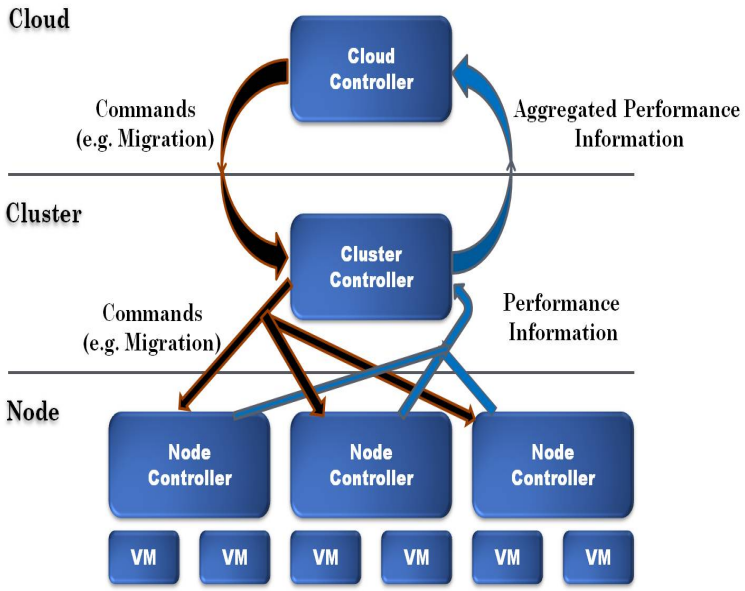

Figure 6.2: Modified Communications (blue arrows) and Control (brown arrows) in my Enhanced version of Eucalyptus (compare to Figure 6.1).

Currently in Eucalyptus, the VM provisioning sub-module occurs in each CC and provides various static online provisioning algorithms (such as Round Robin, Greedy, etc.) to identify a suitable node to place a new VM instance on. This obviously does not permit provisioning across the cloud as a whole and also prevents the possibility of migration across clusters in the cloud (due to lack of overall placement information) thereby limiting choices in re-provisioning. I therefore moved the VM provisioning sub-module to the CLC level instead of the CC level. This rearrangement provides the underpinnings to allow us to allocate new VM instances to support migrating existing VM instances to a node across clusters in the cloud. I also added my static algorithms to the set of existing static provisioning algorithms in Eucalyptus. Since

both HBF and CSVP use a number of online algorithms (i.e. First Fit, Best Fit, Worst Fit) and a batch algorithm (i.e. First Fit Decreasing and its updated version to handle compatibility), I also added those algorithms to the CLC of Eucalyptus.

Additionally, in the CLC, I also introduced a new communication interface permitting **dynamic** resource reallocation decisions or any migration decisions to be delivered to the affected NCs. To do this, each CC was modified to pass-thru VM instance migration or resource reallocation requests from the CLC to the relevant node's NC and I made each NC capable of directing its hypervisor to perform resource reallocation or live migration of a VM instance. (Refer to the control communication paths on the left side of Figure 6.2). I also enhanced the NCs to communicate with their underlying hypervisors to query and collect VM instances' performance information and I modified the CCs so that when they gather VM instance information from their associated nodes they also collect the additional resource usage information of the nodes and the VM instances running on them. Additionally, each CC now summarizes the information collected and forwards the information up to the CLC, as shown in the data communication paths on the right side of Figure 6.2. Finally, I made it so each NC periodically refreshes and updates the CCs about the resources used by their nodes and VM instances. As a result, dynamic provisioning algorithms (e.g. DAVM, DRVM) now always have access to updated resource usage information available for making change decisions. I also modified the NC code to handle controlled resource sharing for DTVM. I implemented and/or verified some basic functionalities (e.g. creating history dataset, VM migrations, etc.) of DAVM and DRVM in Eucalyptus for manual verification but didn't make them automated

and fully functional.

6.1.3 Test-bed Results

I followed a common practice in cloud performance assessment and used Olio [32] along with the Faban load generator [6; 32] to create reasonably realistic synthetic workloads to drive my basic testing. Olio allows the creation of simple Web 2.0 applications to evaluate the performance of web based cloud applications. Faban was used along with Olio to create a Markov-chain based workload for the VMs (this approach was previously used in [95]).

Once I had implemented my algorithms I first ran small-scale, manually generated tests to verify their correctness, I then evaluated the effectiveness of my HBF static algorithm and DTVM dynamic algorithm together. As the arrival frequency of new VM requests increased, HBF naturally became effective. However, if the workload changes were more significant and/or as changes compounded over time the overhead of HBF grew to the point where it represented a significant drain on the cloud system's capacity. As the workload variation in the allocated VM instances increased over time, the effectiveness of dynamic resource sharing increased as would be expected. If the changes in workload were short-lived, I often found that DTVM alone remained effective for some time. There were, however, also frequent cases when DTVM could not solve short-lived additional resource requirements that would have required a VM migration. In Eucalyptus, I also verified the effectiveness of my algorithms over existing ones, in terms of overall resource utilization. I found that my algorithms were providing better resource utilization compared to existing algorithms.

My experiments using the test-bed system were mainly designed for assessing the correctness of my algorithms and were necessarily limited in terms of the types of applications run and the size of the cloud system involved. I therefore realized that I would have to do further, scale-related experiments in a simulated cloud environment. These were done with Cloudsim using the widely used Google Trace data to create model VM applications with various characteristics. The setup for my simulation experiments is described in the next subsection.

6.2 Simulation Setup

Having confirmed the correctness of my implementation and its small scale behaviour, the algorithms were then added as extensions to CloudSim [17; 87] to assess their effectiveness for more realistic scenarios. Specific, handcrafted, workloads were initially given to Cloudsim to ensure that my algorithm implementations were behaving as I expected them to. This served as a means to provide confidence in the later simulations without the development and use of a complex analytical model. I then used Cloudsim to create a much larger simulated cloud environment for my assessment of any scale related effects. From my initial study of Cloudsim, I determined that the Cloudsim simulator would support the relatively straightforward addition of the necessary hybrid provisioning policies and VM migration behaviour. Using my simulated cloud environment, I have assessed the effectiveness of my static (HBF, CSVP) and dynamic (DTVM, DAVM, DRVM) algorithms in a large-scale, realistic cloud scenario.

6.2.1 CloudSim : Changes made

I first created extensions to the CloudSim simulation system [18] by modifying Cloudsim's **PowerDataCenter** class and **Helper** class to support including the different "Fit" algorithms. I also created a new **MyConstants** class to hold the constant values (e.g. total simulation time, total number of VMs, etc.) which were used in cloudsim at runtime. The **updateCloudProcessing** function in the **PowerDataCenter** class was previously used only for scheduling new events. I modified this function to additionally check the arrival and departure queue of VM requests at a fixed interval. I then created an arrival queue to contain the arriving VM requests and their expected start times. Similarly, a departure queue was created to contain the already started VM requests and their expected completion times. The VMs are placed in both queues in sorted order considering their execution start time and execution finish time, respectively. A number of algorithms : first, next, worst, best fit, and first fit decreasing along with HBF were then implemented in the **PowerDataCenter** class. When the **updateCloudProcessing** function is used to schedule the upcoming events it looks at the arrival and departure queues and uses a specific fit algorithm to allocate or de-allocate VM instance(s). The **MyConstants** class was used to hold the parameter values that remain constant during the experiments used to evaluate the fit algorithms. The **Helper** class was modified to print the required results related to resource usage during the experiments.

CloudSim already included policies to execute tasks/applications (obtained from multiplexed VMs in a physical machine) both sequentially and in parallel. The existing implementation didn't support dedicated resource allocation for VMs rather it

always assumed equal time sharing of resources for multiplexed VMs. I first considered the resource requirements of each VM request as priorities (which is the percentage of maximum usage allowed for a multiplexed VM for a specific resource) and then added support to associate priorities with each VM request. In priority based scheduling, I then developed a new class, **CloudletSchedulerDifferentialTimeShared** as a part of the DTVM algorithm implementation, that could provide the desired share of resources for VMs. I modified the **VM** class in CloudSim which was used to create VMs and I added **vmPriority** (a new floating point parameter) to specify the priority of VMs for a specific resource. In DTVM, there was also support for assigning priorities to applications running in each VM instance. I included **taskPriority** (a new floating point parameter) with each application running in a VM instance of the **Cloudlet** class (which is used to create applications in CloudSim). These associated priorities (**taskPriority** and **vmPriority**) were then used to calculate the specific time slices allotted to each end user application when simulating the scheduling activities. I also integrated my time series prediction model in the **PowerDataCenter** class to predict future workloads of each hosted VM using the collected resource usage information. The resource usage history and predicted information are used in the simulation of the DAVM and DRVM provisioning approaches.

I also modified the **PowerDataCenter** class to record **VM** utilization which I use as a measure of VM performance since CloudSim does not provide execution time information in a way that can be effectively used to judge VM performance. The logic behind using VM utilization to judge VM performance is that if VMs are utilizing a high percentage of their assigned resources then they have what they need

to be performing well. While this is an indirect measure (compared to, for example, VM execution time) it is reasonable as an aggregate measure and is relatively easy to derive in the simulations. The **PowerDataCenter** class was further extended to record **PM/Host** utilization that is used as a measure of cloud system efficiency. If the physical resources (CPU cores, memory, etc.) are being heavily utilized, while **VM** utilization is also high, then both the cloud provider and cloud users needs are being met.

For the experimental results presented later in this chapter, I have used **VM** utilization, amount of resources allocated and **host** utilization (both absolute and relative to a VM's SLA request[1]) to assess the performance benefit of my algorithms. I use **aggregate VM** utilization as a measure of overall VM execution performance. The higher the overall VM utilization is the better use of the allocated resources so the faster the VM applications will execute. This value is normally reported only for those VMs affected by a change made by a particular algorithm to make the impact clear when improvements are made for only a small percentage of all the VMs being run in a cloud. The *resources allocated* per physical machine tells us how efficiently the VMs are packed in the hosts. The higher the percentage for resource allocation, the more densely the VMs are packed into the hosts. I use aggregate **host** utilization (with and without SLAs considered) as a measure of the effective use of the resources available on hosts. The higher the overall **host** utilization is the better the allocated resources are being used. These four metrics are used throughout the experimental

[1]Absolute measure of **host** utilization considers the load actually introduced by the VMs running on the PM at some point in time. This can be significantly lower than the host utilization when considering the SLA requests made by the VMs since a VM may request (via the SLA) more resources than it may actually be using at some point in time.

process and are shown in the result graphs for most of the algorithms and experimental scenarios. The basic scenarios and, briefly, the technologies underlying them and the developed experiments and simulation results are described separately in the sections following a discussion of the real-world Google trace data which I use in many of my simulations.

6.2.2 Google TraceData

I base many of my algorithm simulation results using the extensions to CloudSim just described on data from the widely used Google Trace Data (second version) [74]. I chose to use the Google trace data because it provides a real-world source of data characterizing a large collection of running jobs doing a variety of tasks over long time frames. As described earlier, traces from commercial cloud systems are generally unavailable. Because of this, the Google trace data has often been used in assessing algorithms for cloud provisioning, etc. For testing, I randomly selected 1000 workload patterns from the long-running Google traces for use (an example of four sample whole day workload patterns is presented in Figure 6.3) that shows both the wide range of resource usage variation in some of the traces and the regularity in trends in resource load that suggests predictability through the use of time-series methods. This is large enough to serve as a reasonable sample for the entire set of traces and hence, the results I present should be representative of the trace data as a whole.

It is important to understand that the Google traces provide us only with patterns of resource utilization (i.e. 'more at this time, less at that time') not specifics about the actual resources used (e.g. number of cores) or the precise usage of the allocated

resources (e.g. 78% of 4 cores). Thus, I use the 'normalized' trace information provided (without loosing any information) and scale it to various levels of resource usage for varying numbers of resources, as needed.

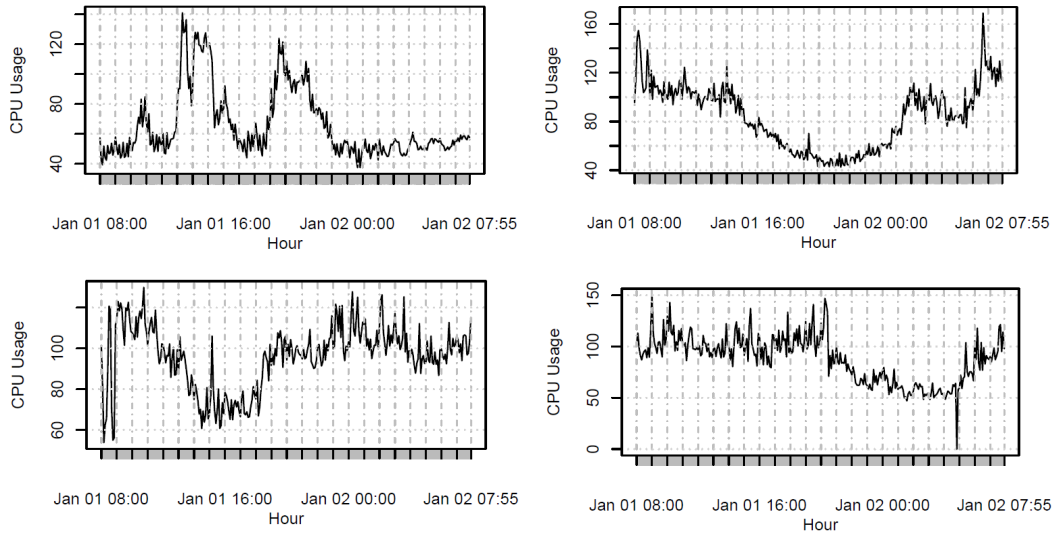

Figure 6.3: Example Google Traces

Of course, there are different ways to measure the time that a VM spends above some threshold and many could be suitable. In our assessments, described later, we simply tested that a VM was above threshold at least 50% of the time to determine resource criticality, though this is certainly not the only possible approach. What is required is to pick a criterion that will select VMs that will perform well when co-located with *non* resource critical VMs. Figure 6.3 shows four, day-long samples from the Google traces. At varying threshold levels, different numbers of these traces could be considered to be Resource Critical (RC). Note that there is significant variance in the resource usage patterns in the four example traces. Using the criterion of being above a threshold of 70% for half the sample time or more, all four VMs are RC

though clearly the first trace spends less time being RC than the others and therefore will need to borrow more resources from the VMs it is co-located with during its peaks periods and less or none at other times. Nevertheless, a VM running the first trace will still benefit, overall, from being identified as RC and thus being co-located with non-RC VMs.

I initially focus on a single resource type for the presented results (considering only CPU usage) before broadening my work to consider two resource types (CPU and I/O requirements) for the assessment of multi-criteria versions of my algorithms. CloudSim does not support the notion of overuse of resources so the absolute maximum that a VM can consume is 100% of the resources assigned to it. This does not permit recognition of overuse as an indication of resource criticality. To solve this problem, I simply adjust the scale of what is considered "full" utilization downward. As long as this is done consistently in all cases, it does not affect results. Therefore, in my simulation, I define a threshold of 70% or more of SLA requirements to define "resource critical" VMs [2]. Naturally, a given VM may be resource critical at one point in time and not at another. For the CSVP algorithm, since I need a simple metric, for the simulations presented, I simply chose to look at the average daily utilization and see whether or not it exceeded 70% of the SLA. For all algorithms, as described earlier, I then further divided the resource critical VMs into three sub-groups representing the lesser (70-80%), moderately (80-90%) and highly (90-100%) resource critical VMs.

[2] A threshold in the range of 70% is also appropriate since it is normally desirable to leave some free resource capacity for each VM so that minor usage spikes can be easily accommodated.

6.3 Simulation Results - Static Algorithms

Recall that I created the HBF and CSVP static provisioning algorithms. The algorithms both have the goal to improve packing efficiency and CSVP does this while considering a simple form of VM compatibility. I have chosen FFD as the baseline algorithm to compare how my static algorithms provide improvement in packing as well as resource utilization.The simulation results for each algorithm are described separately in the following two sub-sections.

6.3.1 HBF

Recall that HBF was designed to better handle VMs with short life times and to be responsive to urgent VM requests (those that require early/immediate start times). Providing better support in these areas should increase the appeal of cloud environments for a broader range of user applications. I compared HBF with the batch algorithm FFD (First Fit Decreasing) which sorts the batch of outstanding VM requests acquired during a window of time into decreasing order by size and then applies First Fit. This well known heuristic for the bin packing problem typically produces good (much better than just First Fit) but still necessarily non-optimal solutions. Given the relatively small size of the queue of outstanding VM requests, the added overhead of sorting is not normally significant.

I now present the results of six experiments I performed to assess various characteristics of the HBF algorithm relative to FFD. Five of the six experiments consider only a single criterion in packing VM requests (Algorithm 2). The final experiment considers, without loss of generality, two specific criteria though more can be sup-

ported (Algorithm 2M). Simulations were run for a range of VM request patterns (varying percentage of various request sizes, order of requests, etc.). Results are reported for six different patterns representing a range of possible VM requests. The six patterns, labelled $P1$ through $P6$, are described in Table 6.1. The resource requirements of VM requests were classified into three different categories: Large(L), Medium(M) and Small(S). Different patterns were generated first by considering VM request sets/series having different percentages of large, medium and small resource requirements and secondly by considering the sequence of such request arrivals. In patterns P1 and P2, all Large requests come first followed by all smalls and then mediums (labelled "LSM" for Large-Small-Medium). In the remaining patterns, all large requests come first followed by all mediums and then smalls ("LMS") but with varying percentages of each. Having large requests come first reflects the reasonable assumption that large, OLTP-like applications, will likely start early and run continuously. The patterns vary the percentage of such VMs and then vary the percentage of small and medium VMs within what is left. The smaller VMs are of primary interest as they are more likely to represent short duration requests which will normally (though not necessarily always) also require fewer resources. The percentages of large, medium and small VM requests in each pattern are shown in Table 6.1.

Table 6.1: VM Request Patterns Used in the Experiments

Pattern ID	Large(L)	Medium(M)	Small(S)	Sequence
P1	40	30	30	LSM
P2	34	33	33	LSM
P3	10	10	80	LMS
P4	10	80	10	LMS
P5	80	10	10	LMS
P6	20	20	60	LMS

All experiments were run multiple times and average results are reported. Simple variance bars are included in the graphs so that likely trends in algorithm behaviour may be correctly recognized.

Figure 6.4 shows how HBF compares to FFD for the request patterns described as the number of VM requests increases. This has the effect of increasing the number of batched VM requests available in each interval which is a benefit to both FFD and HBF. In all cases, HBF requires fewer physical machines (PMs) than FFD. Further, the number of fully used hosts (those with all resources utilized) is significantly higher with HBF than with FFD. Naturally, the larger the number of VM requests, the more resources are needed. Within these general observations, both the number of required hosts and the number of fully used hosts varies based on the VM request pattern. No VMs completed execution during the simulations (i.e. they were all long-lived).

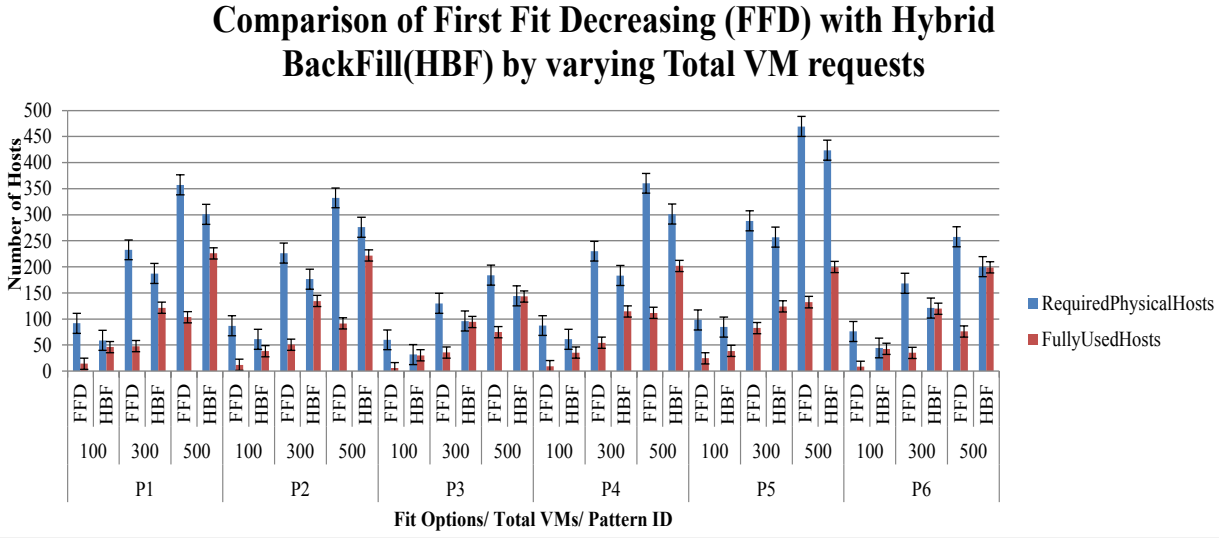

Figure 6.4: HBF vs FFD, Varying Request Patterns and Number of Requests

Figure 6.5 compares HBF and FFD for the various request patterns as the window size is varied. Intuition suggests that larger window sizes should allow greater packing

efficiency in both algorithms. Window sizes of 10, 50 and 150 simulation time units were tested. The results show that the number of physical hosts required for FFD decreases modestly as the window size increases due to backfilling characteristics of HBF, confirming the intuition for FFD. In general, HBF appears to pack more efficiently than FFD for smaller window sizes though this is due to its aggressive backfilling even when processing a **batch** of VM requests. Based on the trends in the graph and considering the variance bars shown, changes in the number of hosts required and fully used hosts as the window size increases do not appear to be significant. This reflects the fact that HBF already does a good job, even for relatively small window sizes. Again, no VMs completed during the simulations.

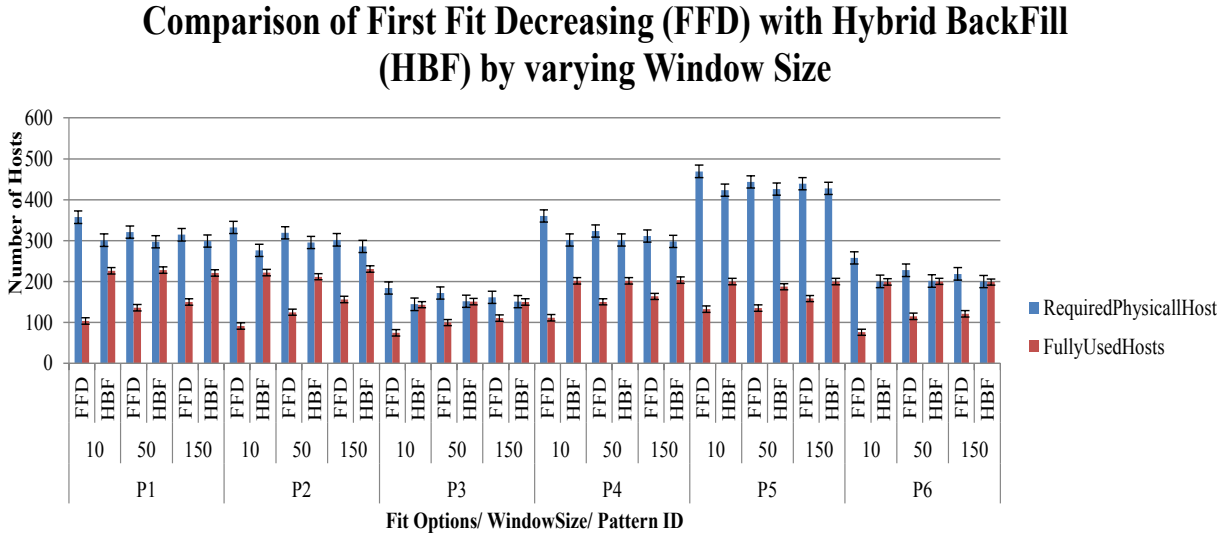

Figure 6.5: HBF vs FFD, Varying Request Patterns and Window Size

The results so far correspond to experiments where VM executions were long lived. While such long running VMs are, naturally, still common in clouds, increasingly, cloud infrastructure is also now being used for shorter duration VMs (for running "on-the-fly" applications and, in some cases, for handling spikes in load by provisioning

additional VMs rather than by increasing resources to existing ones). Figure 6.6 compares HBF and FFD as the percentage of "early terminating" (i.e. shorter-lived) VMs varies up to 20 percent. In such cases, HBF's aggressive backfilling allows it to out-perform FFD. The number of hosts needed by HBF decreases as the percentage of completing VMs grows while there is no real change for FFD. The number of fully used hosts also decreases with an increase in terminating VMs. This is not unexpected as backfilling is less likely to result in perfect fits than is batch filling.

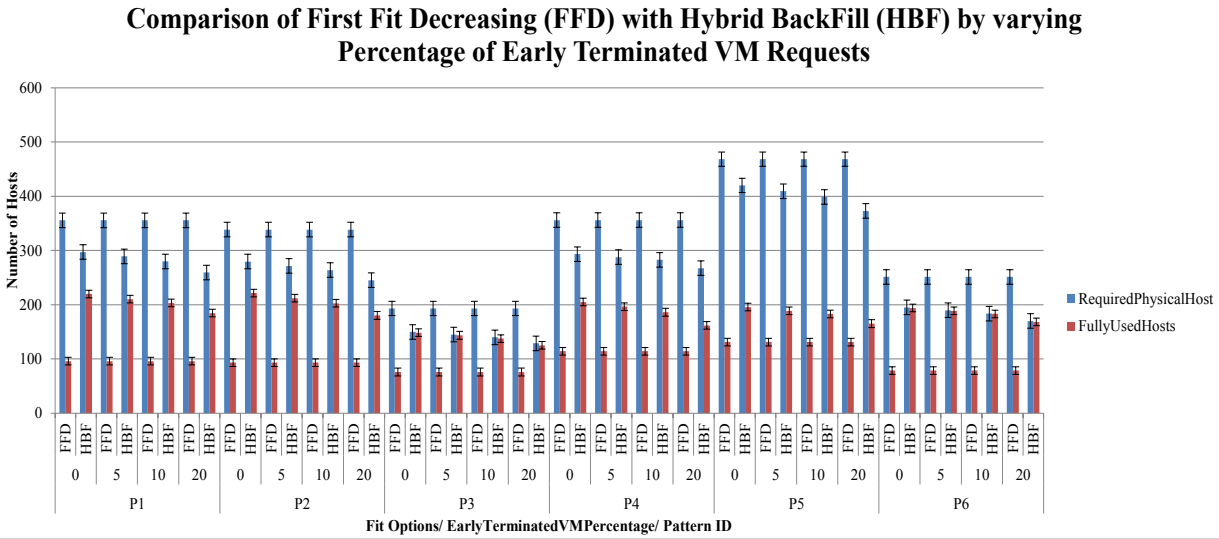

Figure 6.6: HBF vs FFD, Varying Patterns and VM Completion Rate

Another important advantage of HBF over FFD is its ability to provision resources for "urgent" VM requests that specify start times within the current batch window. Figure 6.7 confirms the obvious fact that FFD wait times (the time VMs must wait after their requested start time before they can actually execute) grows with the percentage of VM requests specifying an early start time. Naturally, there are no such wait times for HBF so delays are only shown for FFD. Note also that delays are, as expected, largely unaffected by VM request pattern.

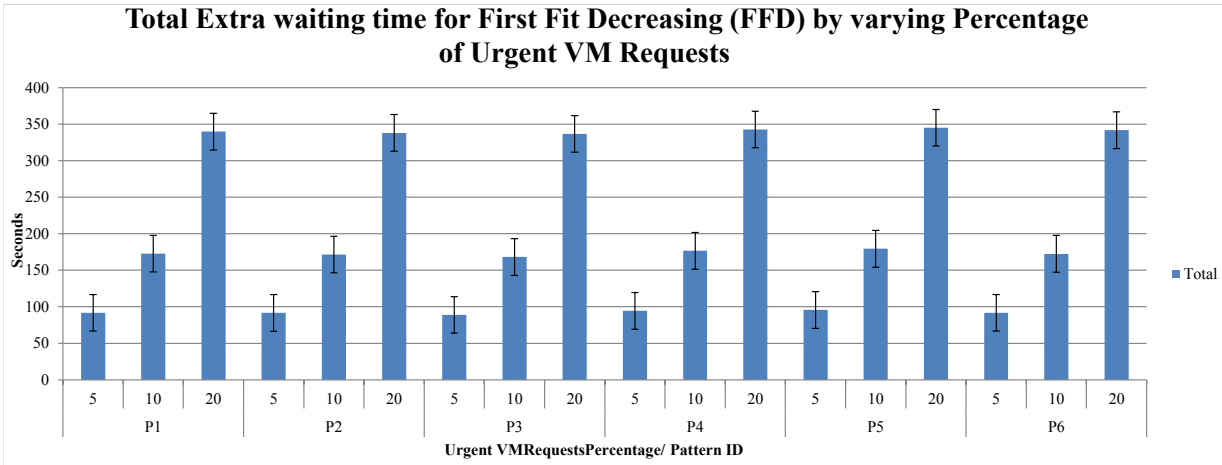

Figure 6.7: HBF vs FFD, Varying Patterns and Waiting Time

My final single-criterion experiment looked at the impact of the number of requests per window. Figure 6.8 shows the impact of this on packing efficiency. Decreasing the number of VM requests per window has little discernible impact on HBF. In comparison, FFD requires slightly more hosts when there are fewer requests per window since it has fewer opportunities to pack new PMs as densely. As in some other experiments, while the trends are consistent, the extent of the impact varies with the VM request pattern considered. FFD is certainly disadvantaged when the request pattern provides greater opportunity for HBF to optimize by more effective backfilling (e.g. pattern P5). Overall, HBF packs better than FFD despite having fewer available requests.

Figure 6.9 effectively reconsiders the experiments reported in Figure 6.6 for the case of multiple provisioning criteria. In these new experiments there were two criteria (e.g. CPU and memory requirements) with request values for both normalized to a range of 1 to 10. My enhanced technique for multi-criteria packing (Algorithm 2M) was applied to both HBF (now M-HBF) and FFD (now M-FFD). Not surprisingly

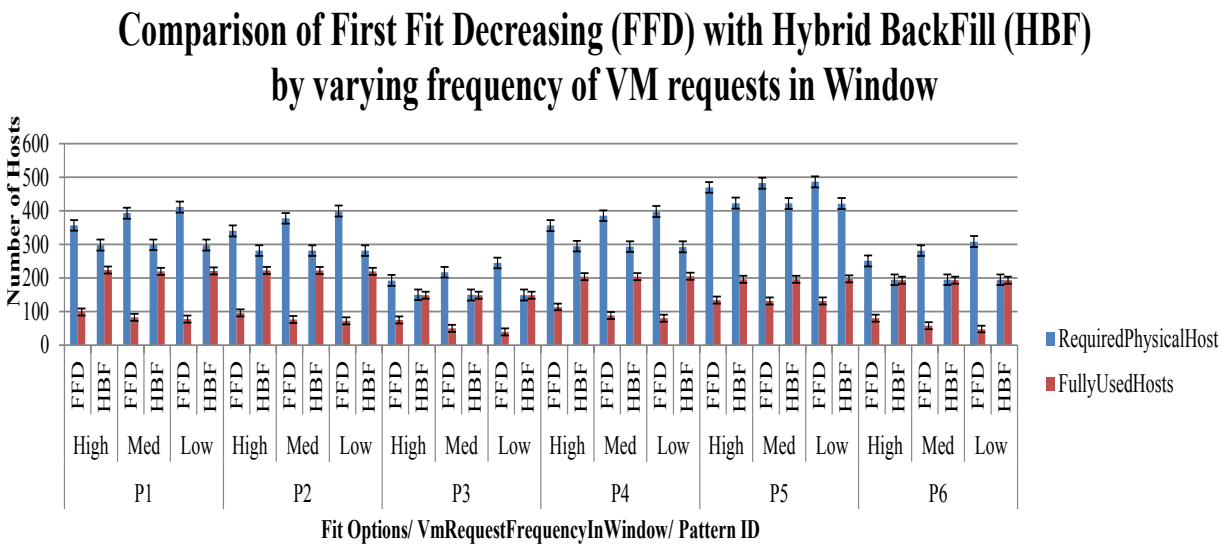

Figure 6.8: HBF vs FFD, Varying Patterns and Requests in Window

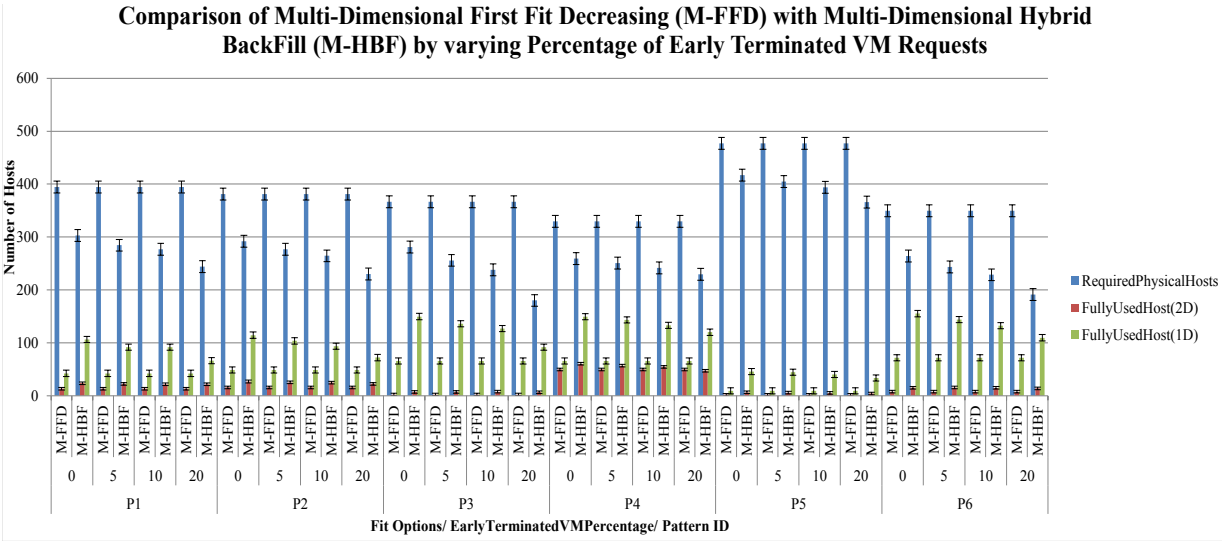

Figure 6.9: HBF vs FFD, Varying Patterns and VM Completion Rate: Multi-Criteria

the results are generally consistent with those from Figure 6.6. The number of required hosts and fully used hosts both decrease with an increase in percentage of early terminating VM Requests for M-HBF but are not significantly changed for M-FFD. Generally, M-HBF requires fewer physical hosts compared to M-FFD as VMs termi-

nate. Note that the number of fully used hosts reported differs depending on whether a host uses its full capacity in only one ("1D") or both ("2D") dimensions/criteria. This is shown by the correspondingly labelled bars in Figure 6.9.

6.3.2 CSVP

To assess the effectiveness of my CSVP approach (Algorithms 3 and 3.1), I based my simulations on data from the widely used Google Trace Data (second version) [74]. As mentioned earlier, details of actual cloud usage is generally unavailable so being able to use a source of real-world data that reflects actual workloads adds credence to the results presented. Again, I initially focus on a single resource type (CPU usage) for the presented results before broadening my work to consider two resource types. I present two sets of single resource results, one where the number of resource-critical VMs is comparable to the number of non resource critical VMs and those VMs in between (thus representing an anticipated future with more resource critical VMs) and another set where the number of resource critical VMs is smaller, reflecting the current Google Trace distributions as reported by Moreno et al [61] who found that only a small percentage (about 1.5%) of the Google traces were very resource critical (i.e. above 95% utilization). I then present results for two resource types to illustrate the effectiveness of my mCSVP technique (Algorithms 3M and 3M.1) when considering multiple resource types. For the testing, I again randomly selected 1000 workload patterns from long-running Google traces which should be representative of the trace data as a whole. Recall that the Google traces provide only patterns of resource utilization not specifics about the actual resources used. Thus, I use the

"normalized" trace information and scale it to various levels of resource usage for varying numbers of resources, as needed.

For the CSVP experiments, I define a threshold of 70% or more of SLA requirements to define "resource critical" VMs due to the limits of CloudSim as described previously. Going over 70% in the simulation represents a given VM being resource critical at one or more points in time. I chose subsets from the long running VMs and then simply looked at the average daily utilization to see whether or not it exceeded 70% of the SLA. As described earlier, I then further divided the resource critical VMs into three sub-groups representing the lower (70-80%), moderately (80-90%) and highly (90-100%) resource critical VMs. For all experiments I chose the VMs having SLA requirements such that they would pack **perfectly** into the available physical machines thereby eliminating any effects due to greater or lesser efficiency in VM packing when comparing with FFD. For example, the CPU SLA requirement of each VM was set to 5 cores ("to provide more opportunity for resource sharing") while each PM was assumed to have 10 CPU cores [3].

In the first CSVP experiment I consider a single resource type, CPU cores, and use a uniform distribution of VMs drawn from our pool of 1000, one third of which are resource critical (including low, medium and high), one third of which are very non resource critical and the remaining third of which are in between. I compare CSVP to FFD for low, medium and highly resource critical VMs to assess the benefit of CSVP. In all experiments I present four sub-graphs showing: VM utilization (as

[3]Though not presented in the thesis, I also ran simulations of CSVP where the packing was imperfect. This had the expected effect of lessening the quality of the packing somewhat but had relatively little impact on the benefit offered by co-locating compatible VMs in all but pathological cases. This too was expected since, for example, if we must place an RC VM with a non-RC VM having fewer resources to loan, th RC VM still benefits from the resources that can be borrowed.

a measure of VM performance since a VM that uses more of its resources can run faster), the amount of resources allocated per host (which will be consistent in this case due to perfect packing), and two measures of the PM utilization, one absolute and one relative to the VMs' aggregate SLA requirements. The VM utilization results are reported **only** for the VMs for which a benefit can be obtained. (There are no negative effects of using CSVP for other VMs.) The physical machine measures are shown for **all** VMs in each simulation.

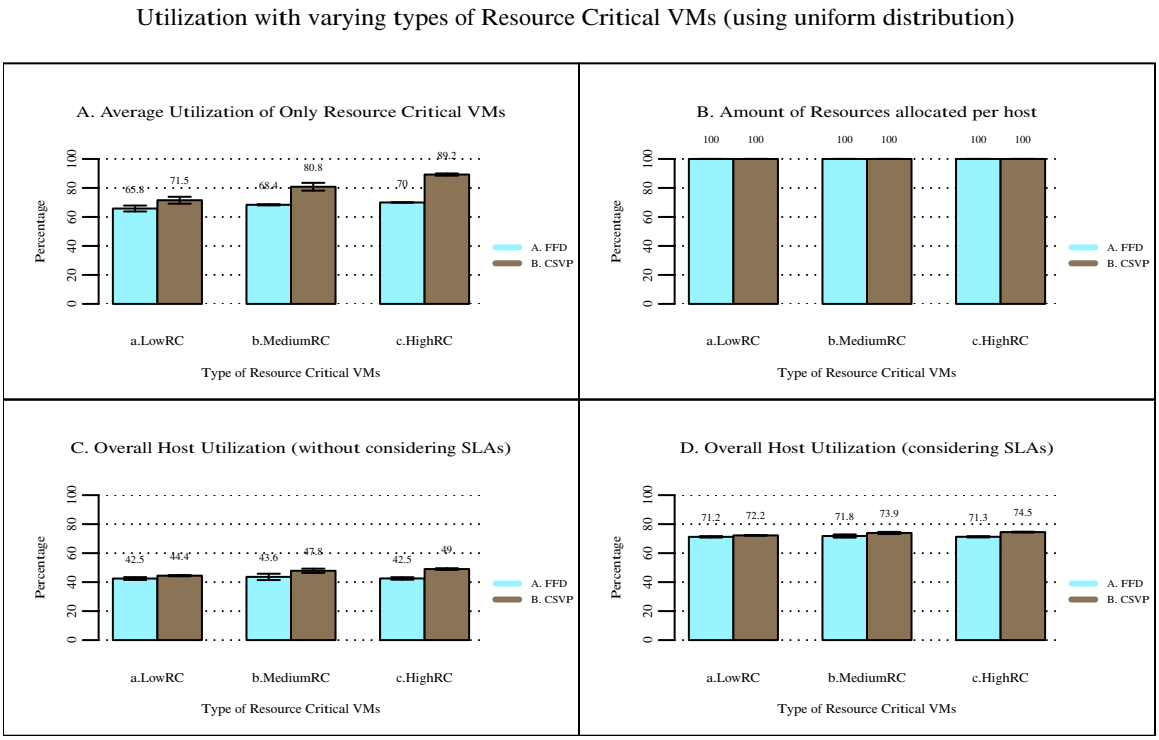

Figure 6.10: Single Criterion, FFD vs CSVP, uniform distribution for Resource Critical VMs)

The results in Figure 6.10 show that CSVP consistently outperforms FFD in terms of VM utilization and the benefit improves for more highly-resource critical VMs ('c.HighRC' in Figure 6.10A). This benefit is realized because more compatible

VMs are co-located permitting more effective sharing of resources between them using DTVM. With a uniform distribution of resource critical, non-critical and in-between VMs, as was used in this experiment, there is a high probability that compatible VMs will be found in each batch. It is also worth noting that the host utilization also improves with the improvement in VM utilization. Since the host (PM) utilization figures are for **all** VMs, this difference in the graphs is naturally smaller.

In the second CSVP experiment I changed the distribution of the resource critical VMs to follow that discovered in the Google Trace data by Moreno et al. [61]. The traces represent a workload that is quite conservative in terms of what might now, or certainly in the near future, be run in a cloud. As such, these can be viewed as very conservative results. Again, FFD is used as a baseline for comparison.

The results in Figure 6.11 show that CSVP again outperforms FFD although, naturally, by less than in Figure 6.10. Co-location of compatible VMs still provides a benefit as seen in Figure 6.11A. Of course, this is for a smaller number of VMs benefitting from the use of CSVP since Moreno's distribution includes a smaller percentage of resource critical VMs. A correspondingly smaller but still consistent benefit is seen in the host utilizations, Figure 6.11C&D. This is to be expected since the benefits provided to a small number of VMs are amortized over a relatively larger number of PMs.

As described, my CSVP approach is also applicable when considering multiple resource types for placement. In my third CSVP experiment I considered, without loss of generality, two criteria: CPU and I/O requirements, again using both the Google Traces and Moreno distribution. As with the CPU-only experiments, I chose

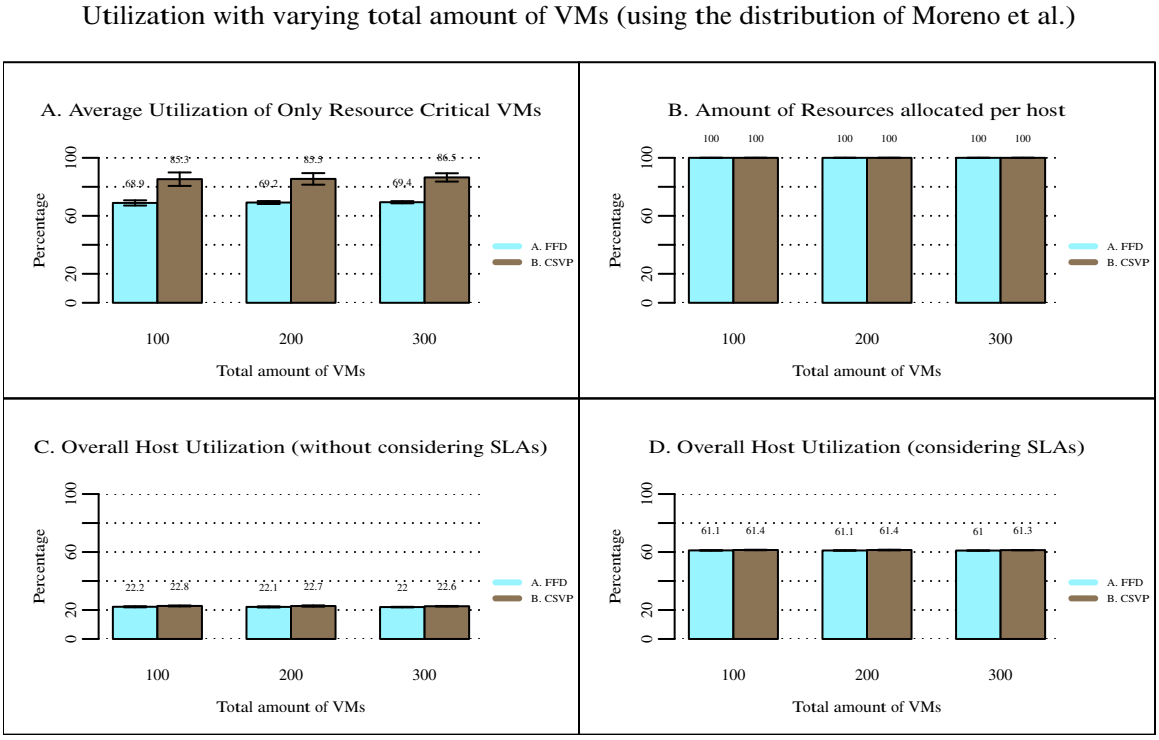

Figure 6.11: Single Criterion, FFD vs CSVP, Moreno distribution for Resource Critical VMs

to assign VM I/O requirements so that they would pack perfectly into, in this case, the I/O capacity of each PM in the cloud. The results described below show that, my multi-criteria version of the algorithm can effectively handle multiple criteria concurrently.

In my first mCSVP experiment, I explore the ability of the multi-criteria version of the algorithm to deal with VMs that are primarily CPU resource critical. In the second, I do the same for VMs that are primarily I/O resource critical and then in the third I consider the critical case where some VMs are either CPU or I/O resource critical or both. By considering these three scenarios, I show that my multi-criteria algorithm is effective for workloads with varying forms of resource criticality and that

there is no undue bias in the algorithm to any of the three cases. In these sets of experiments, I am interested in showing that increasing benefits are achieved as the degree of resource criticality increases. No direct comparisons are made with FFD, but can be inferred indirectly based on the results seen in the earlier experiments.

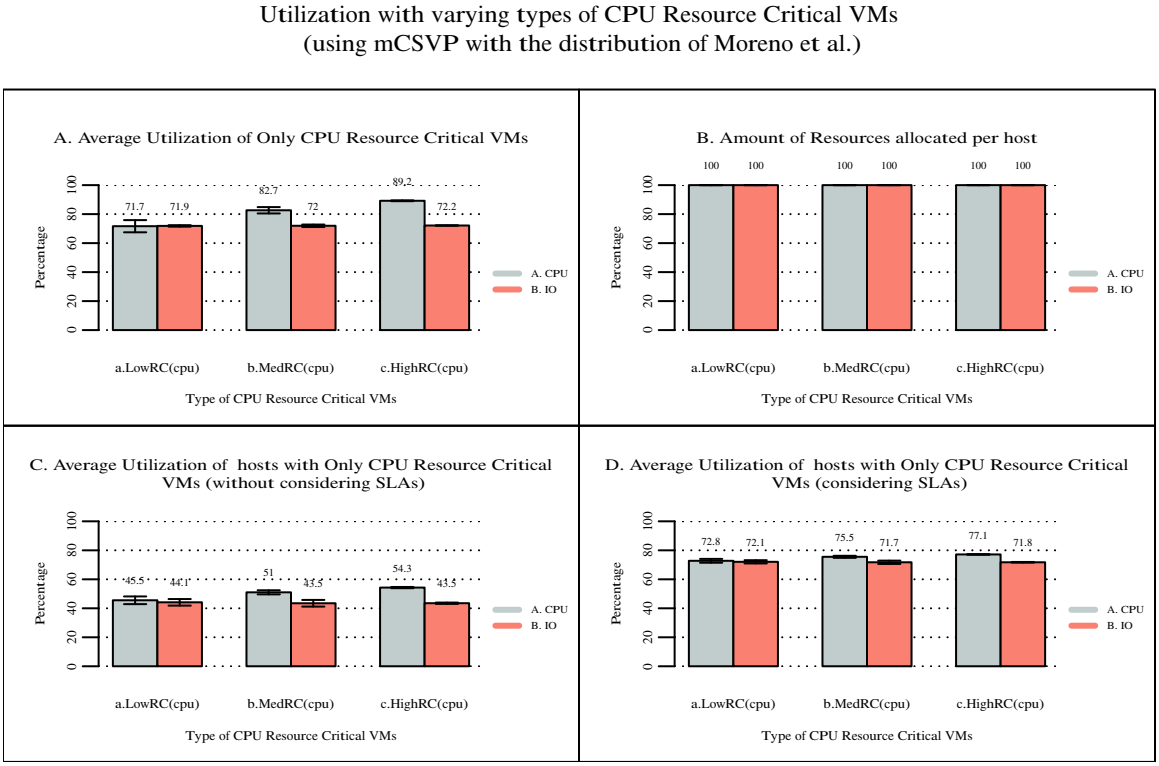

Figure 6.12: Multiple Criteria, FFD vs CSVP, Moreno distribution, focusing on CPU

The results shown in Figure 6.12 show how my algorithm successfully optimizes the placement of the CPU resource critical VMs offering consistently improving CPU utilization (i.e. performance) for more resource critical VMs without negatively impacting I/O efficiency. Note also that the host utilizations (shown for machines hosting CPU resource critical VMs only) also show modest but consistent improvement reflecting the fact that those PMs are expending more **committed** resources to pro-

vide the improved utilization seen in the CPU resource critical VMs.

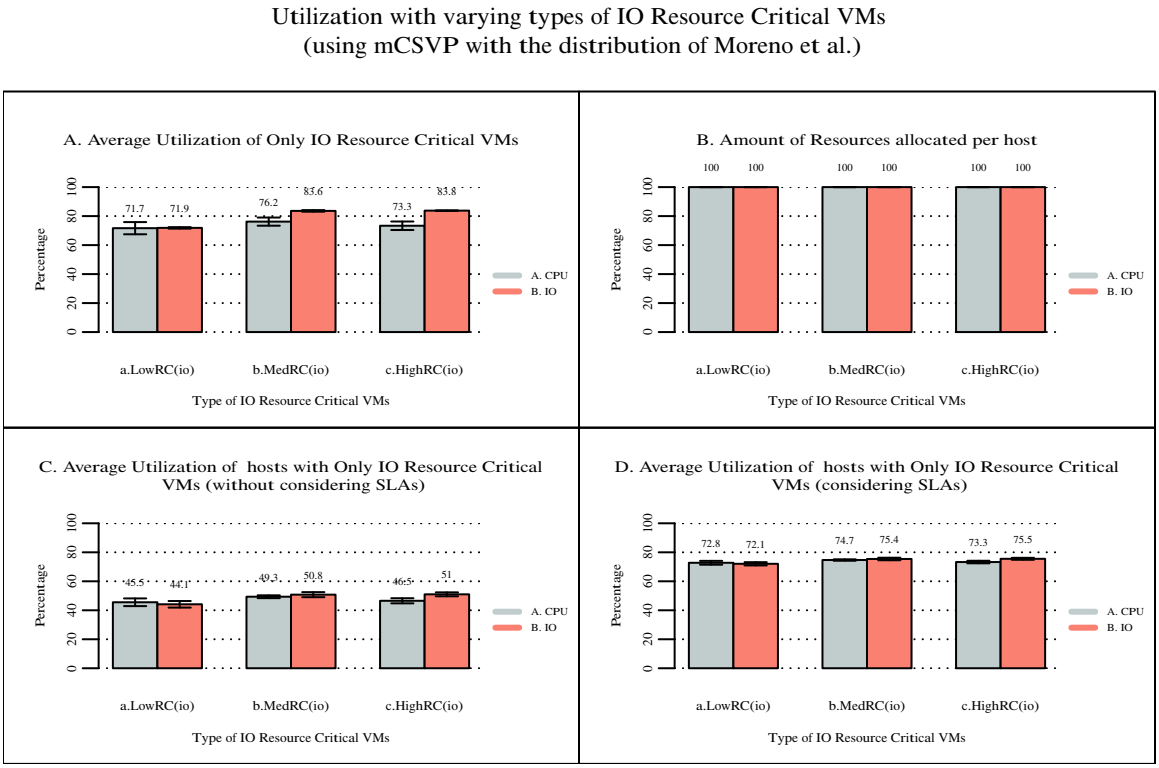

Figure 6.13: Multiple Criteria, FFD vs CSVP, Moreno distribution, focusing on I/O

The results shown in Figure 6.13 parallel the results from Figure 6.12 but are focused on I/O resource critical VMs rather than CPU resource critical VMs. The improvements offered by our technique as the I/O resource criticality increases are, naturally, in line with those when the focus was on CPU resource criticality. The mCSVP algorithm does not favour one resource type over another in these cases.

Finally, the results shown in Figure 6.14 show how mCSVP successfully handles situations where there are a mix of CPU resource critical and I/O resource critical VMs that need to be co-located. In general there are some cases where resource critical VMs of both types may need to be dealt with concurrently, introducing the

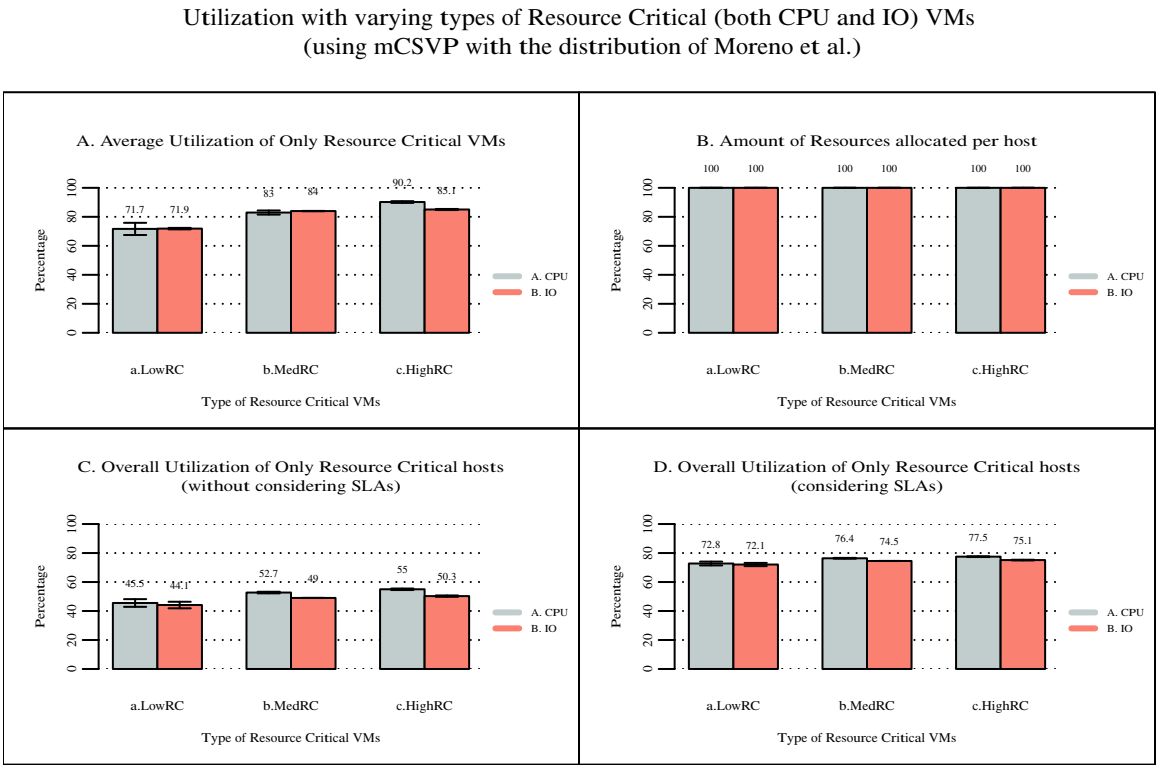

Figure 6.14: Multiple Criteria, FFD vs CSVP, Moreno distribution, CPU and I/O

possibility that attempting to optimize one type could negatively impact the other. As discussed, my mCSVP algorithm, when necessary, favours CPU over I/O for optimization purposes. Despite this, the results show that both CPU and I/O resource critical VMs can be concurrently optimized. Note that the VM performance improves for both CPU and I/O resource critical VMs and that this improvement is proportional to the resource criticality. Note that both the CPU and I/O bars in the figure are trending upwards as criticality increases.

Note that for the multi-criteria experiments using the Moreno distribution to determine the number of both CPU and I/O criticial VMs it is unlikely that there will be individual VMs that are acutely resource critical for both CPU and I/O. The

code in mCSVP will attempt to co-locate such VMs with others that are non resource critical for both CPU and I/O. Thus, even with a more uniform distribution such as that in the first single-criterion experiment we would expect to find suitable VMs for co-location that would offer performance benefits consistent with those shown in the preceding graphs.

All the simulation experiments show that despite using a very simple metric for resource criticality (one that can easily be initially estimated by a user or trivially collected from a prior VM execution) it is possible to significantly improve VM performance by statically co-locating **compatible** VMs from within batches. This is true when the compatibility of both single and multiple criteria are considered. In general, as long as each scheduled batch contains sufficient non resource critical VMs to "balance" the resource critical ones, performance should improve noticeably. Until the percentage of resource critical VMs increases significantly beyond that reported by Moreno, our technique should remain effective.

6.4 Simulation Results - Dynamic Algorithms

I created the DTVM, DAVM and DRVM dynamic provisioning algorithms which form a family of related algorithms where DTVM and DAVM work particularly closely together (DAVM ensuring that DTVM can be effective). When simulating these algorithms in the cloudsim [18] environment, I consider a number of physical machines which have identical physical resources (e.g. ten CPU cores, etc.). I then create a number of VMs (with various configurations) to be run on those physical machines. The assessment of my family of algorithms (DTVM, DAVM and DRVM) is done at

different scales of resources. The specific behaviour of the algorithms was first confirmed using hand-crafted groups of VMs that exhibit characteristics of interest to the specific algorithms. Using these handcrafted tests, I found that the algorithms and simulation system behaved as expected. These purely synthetic test cases were then expanded to larger groups of such hand-crafted VMs to further understand algorithm behaviour. The results from these basic assessments are not all described herein. What is presented are a small number of larger-scale assessments based either on VMs modeled after real-world applications exhibiting different types of load variation or based on captured trace data for applications that could be run in cloud environments. For the primary, large-scale assessment, I again used the Google trace-data [74]. A potential shortcoming of using the Google traces is that we do not know what applications were being run to create the workloads seen. Thus, we cannot associate specific application characteristics with good (or poor) performance using my algorithms. As a first attempt to address this problem, I developed synthetic scenarios based on the characteristics of two possible future cloud applications. Specifically, I created an eHealth scenario and a ubiquitous computing scenario that exhibit specific types of load variation that can be linked to the behaviour of my provisioning algorithms. The results associated with these two synthetic scenarios are discussed in greater detail with additional information in a later section in this chapter.

Recall that DTVM is designed to increase overall **VM** and **host** utilization by sharing unused resources among co-located VMs but the benefit of DTVM is subject to the characteristics of actual workloads depending on how frequently under-utilized VMs happen to be co-located with over-utilized VMs during VM packing. DAVM

offers the ability to find good pairs of compatible VMs and do migrations to bring such VM pairs together to provide longer-term execution benefits by supporting effective resource sharing via DTVM. (DAVM is restricted to pairing for reasons of efficiency. Considering the number of VMs, n, in a large cloud and the growth rate of $\binom{n}{k}$ as k increases it would be impractical to explore triples, quadruples, etc. Of course, a pair created by DAVM might also be paired (as a unit) with another VM to create a "triple" and so on. Alternatively, an overutilized VM might try to find more than one underutilized VMs to be co-hosted together.) DAVM should always improve overall VM utilization (pairings and migrations that do not provide benefits are not done) but in some cases it may reduce overall PM utilization due to the need to bring additional hosts online to support the needed groupings. In such a case, either the new hosts may be under-utilized if the grouped VMs do not require all the PM resources or existing hosts from which VMs were migrated away by DAVM may be under-utilized since the resources that were previously assigned to the migrated VMs are no longer in use. The main goal of DRVM is to improve **host** utilization by re-packing the VMs in under-utilized hosts together into fewer PMs. DRVM, of course, can be used to address **any** source of PM under-utilizations, including those created by DAVM. I now present assessments of my three algorithms in turn. As they are a family, some assessments require more than one of the algorithms to be active at a time. (For example, assessment of DAVM without DTVM does not make sense.)

6.4.1 DTVM

My DTVM algorithm is not, by itself, a provisioning algorithm since it does not decide where VMs are to be run. It is, instead, a dynamic VM resource management mechanism that underpins both my static CSVP provisioning algorithm and also my DAVM and DRVM provisioning algorithms Assessment of DTVM alone is straightforward and designed only to verify that DTVM does exert control over the resources assigned to each VM (and that it does so successfully in the simulation environment). The DTVM results therefore do not provide insight into improved provisioning but are necessary and do confirm DTVM's anticipated behaviour. Accuracy in the priority based scheduling provided by DTVM is determined by whether or not the scheduling respects the priorities assigned to VMs, thereby achieving temporary, controlled resource sharing between VMs to manage load fluctuations. I ran two simple simulation experiments using purely synthetic data as part of my evaluation of DTVM. In these experiments, I created a number of VMs with fixed requirements in terms of processor cores, RAM, hard disk and network capacity. I then experimented with collections of co-located VMs varying the resources assigned to each by varying DTVM priorities and gathered results on the progress made by each VM (measured by VM utilization, as described earlier). Overall, I found that the DTVM strategy is effective in terms of accuracy and speed of execution and as a result increased the throughput of the simulated cloud system as a whole. I present the results of two simple experiments using DTVM alone.

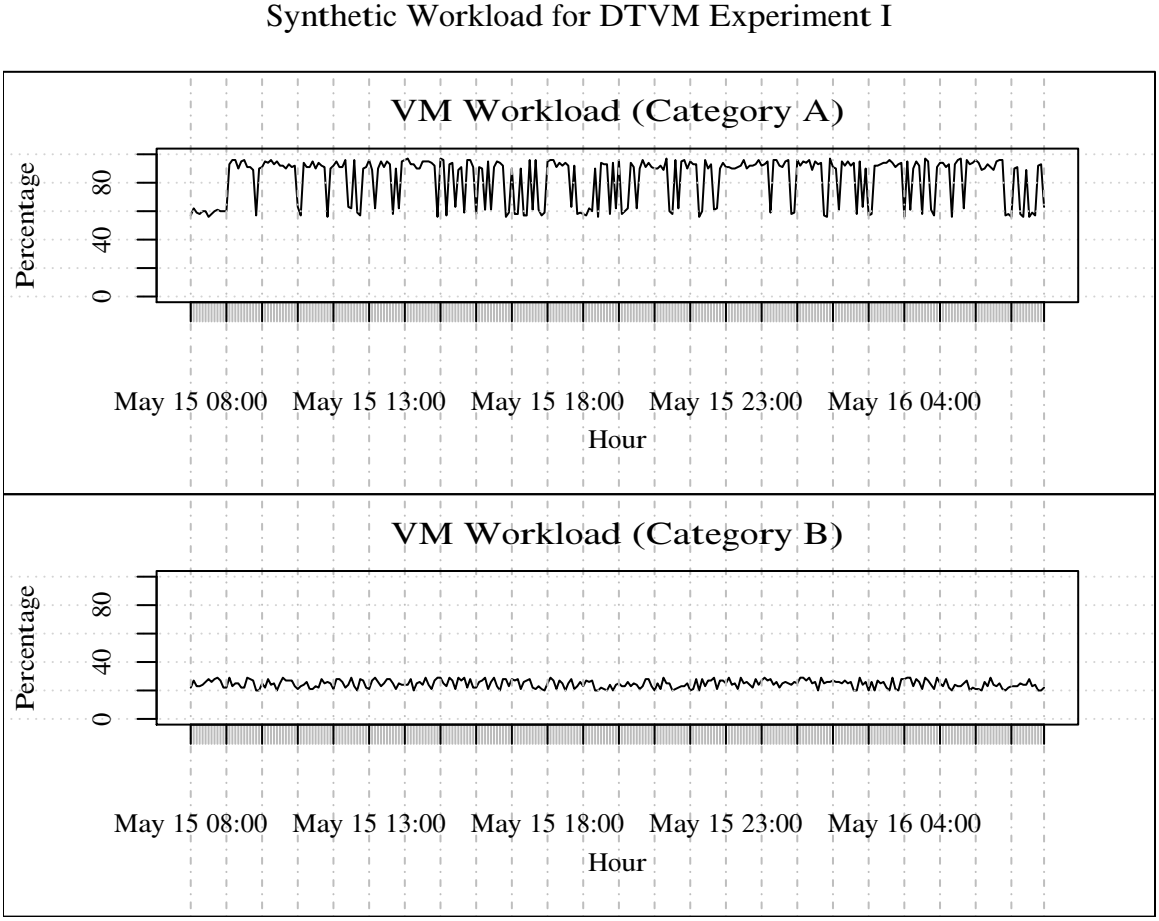

Figure 6.15: Synthetic Data for DTVM Experiment 1

DTVM-Experiment 1

In this experiment, I have used synthetic data to verify the benefit of DTVM over NoDTVM. There are VMs of two categories: category A and category B (refer to Figure 6.15). VMs of Category A have only one cpu and in the normal period, the workload on the CPU is between 56% to 64%. VMs of category A become highly overloaded (having spikes) for some fixed period of time and during these times, the workload remains between 81% to 99%. VMs of Category B have two CPUs and the

workload always remains between 20% to 30%. I consider a single physical machine (since DTVM runs on a single PM at a time) where the machine has ten CPUs and those are assigned to VMs of both categories.

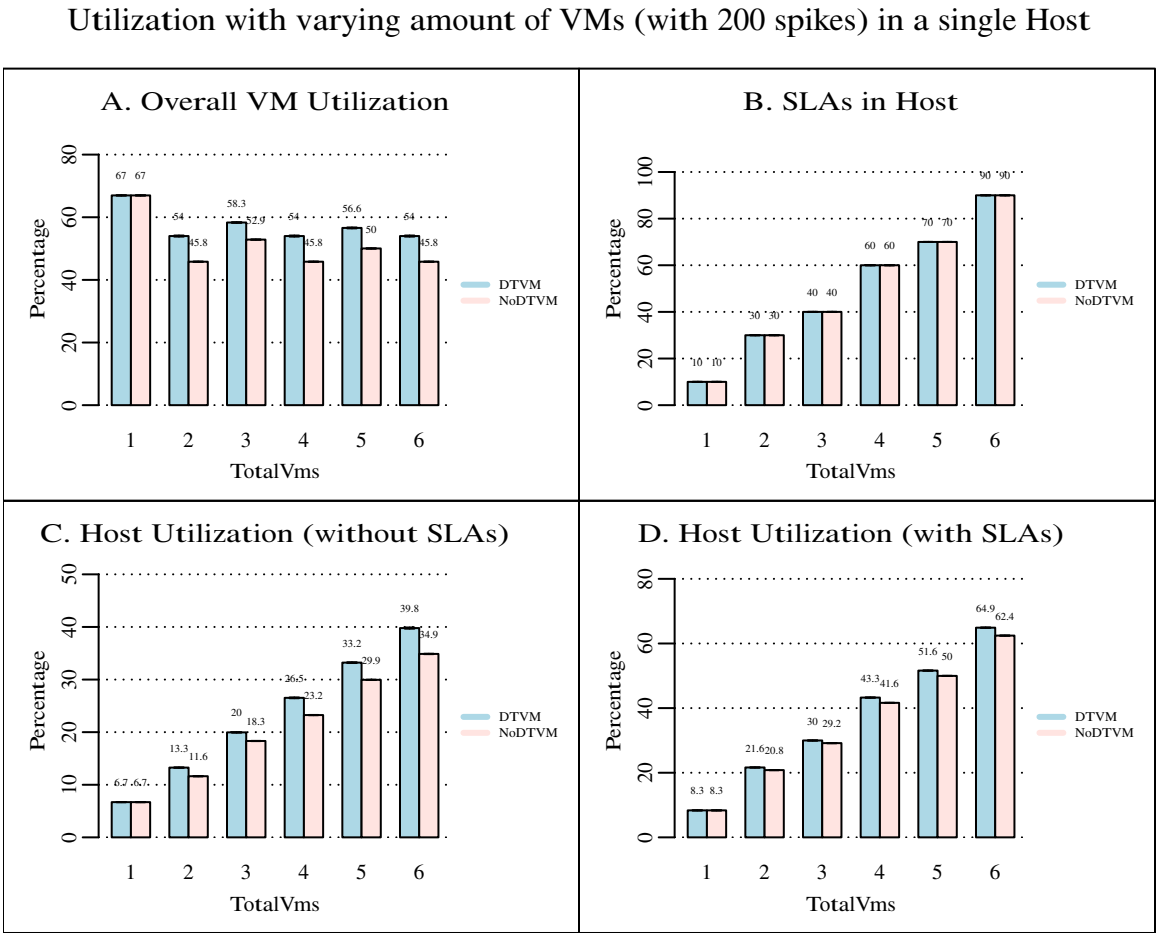

Figure 6.16: DTVM-Experiment 1-Result1

I verified the effectiveness of DTVM when there are varying numbers of VMs in a single physical machine and the corresponding results are shown in Figure 6.16. When there is only one VM (of category A) in the physical machine then the overall VM utilization is 67% and the host utilization is 6.7% (without SLA) and 8.3% (considering SLA). Since the single VM of category A has only one CPU, it uses at

most only 10% of the available CPUs of the host (based on its SLA). In this case, DTVM provides the same amount of utilization as NoDTVM since the VM does not have any underutilized VM to obtain resources from during its overloaded period.

Still referring to Figure 6.16, when there are two VMs (one of category A and one of category B), the overall VM utilization is 54% for DTVM and 45.8% for NoDTVM. At first glance, this seems like an unexpected result but the VM of category B always has low workload so, the **overall** VM utilization is lower compared to the case when there is only one VM (of category A) in the physical machine. In this case, DTVM provides more benefit in overall VM and host utilization than NoDTVM as the VM (of category A) does find an underutilized VM (of category B) to balance some of its overload during the peak period. With increasing numbers of VMs in a physical machine, resources requested in the VMs' SLAs and both types of host utilization also increases but the overall VM utilization varies depending on the newly added VM's type. When there are three VMs (two of category A and one of category B), the overall VM utilization increases but only one VM (of category A) gets the opportunity to balance its peak workload. When there are four VMs (two of category A and two of category B) the overall VM utilization difference between DTVM and NoDTVM is larger compared to the case of three VMs as two VMs get opportunity to balance their peak workloads. The pattern of changes is identical for the cases when there are five VMs (three of category A and two of category B) and six VMs (three of category A and three of category B), all shown in Figure 6.16.

I also verified the effectiveness of DTVM when there are different amounts of peak workload in the VMs (of category A) in a single physical machine (refer to

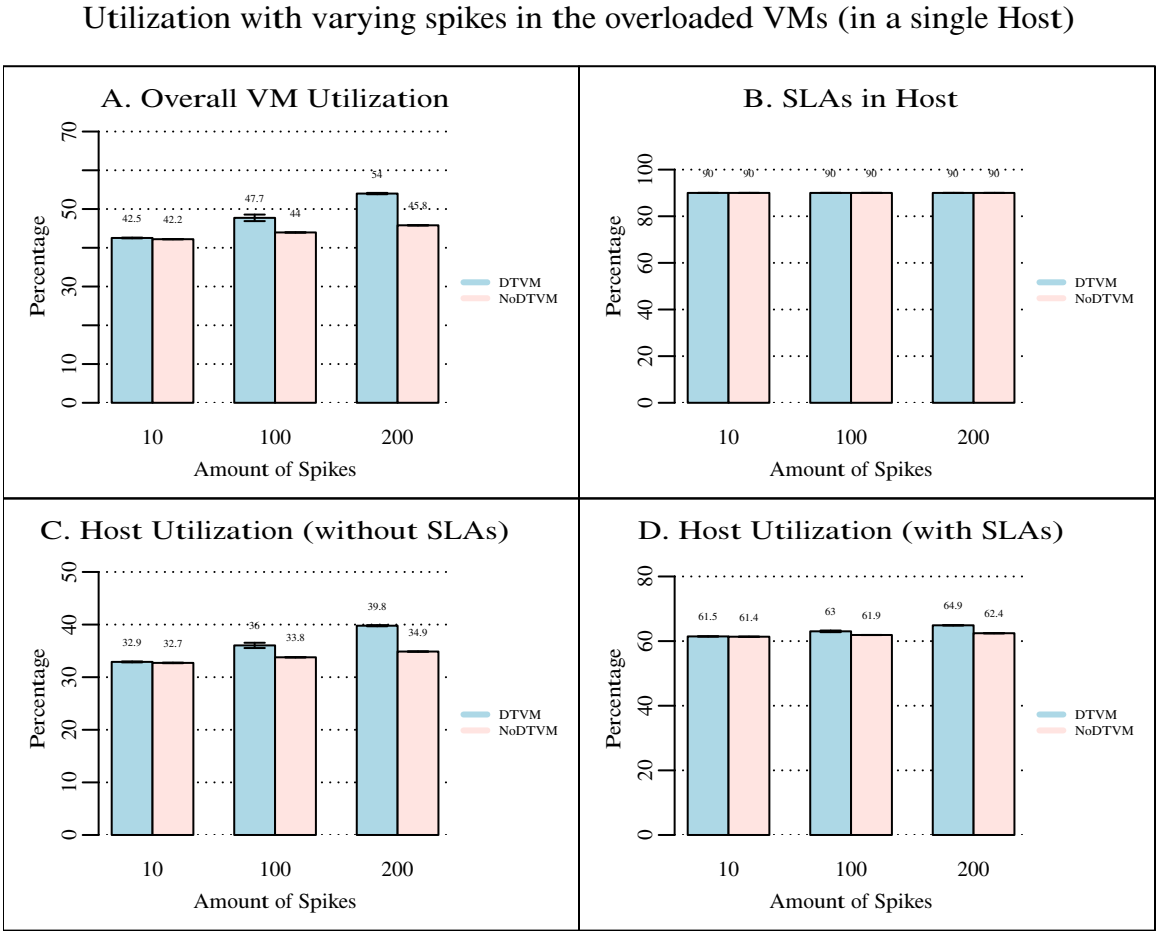

Figure 6.17: DTVM-Experiment 1-Result2

Figure 6.17). In this scenario, I considered six VMs (three of category A and three of category B) which are co-hosted in the same physical machine. The three VMs of category A get opportunity to balance their peak workload with the other three VMs of category B. With an increase in the number of spikes (i.e. period of time the VM has peak workload) in each VM of category A the overall VM utilization and Host utilization (both considering and not considering the amount of resources requested in the SLAs) increases but the increase is higher for DTVM compared to NoDTVM since DTVM allows the VMs of category A to obtain resources from other co-hosted VMs

of category B. So, the obtained results confirm that DTVM becomes more effective and provides increased benefit with greater spikeiness in the workload and also when there are more non-critical VMs to share resources with resource-critical VMs in a PM.

DTVM-Experiment 2

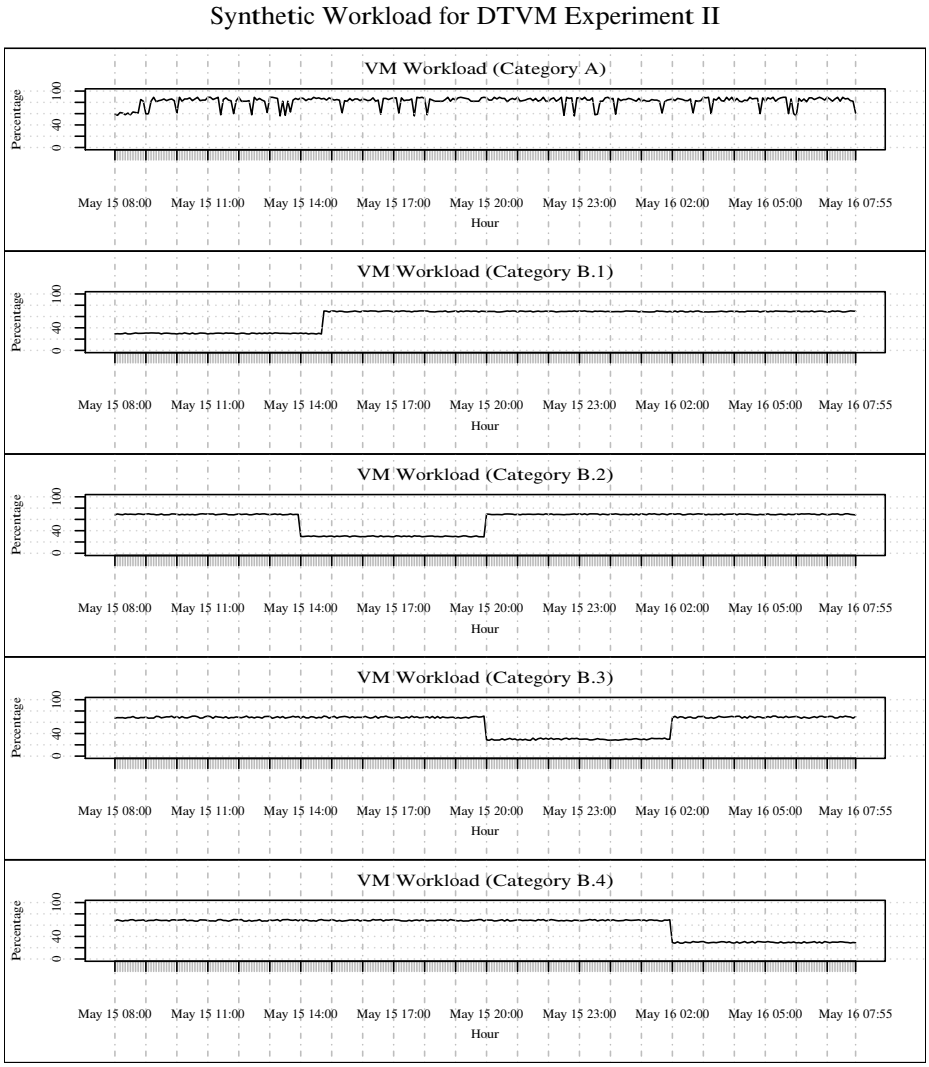

Figure 6.18: Synthetic Data for DTVM Experiment 2

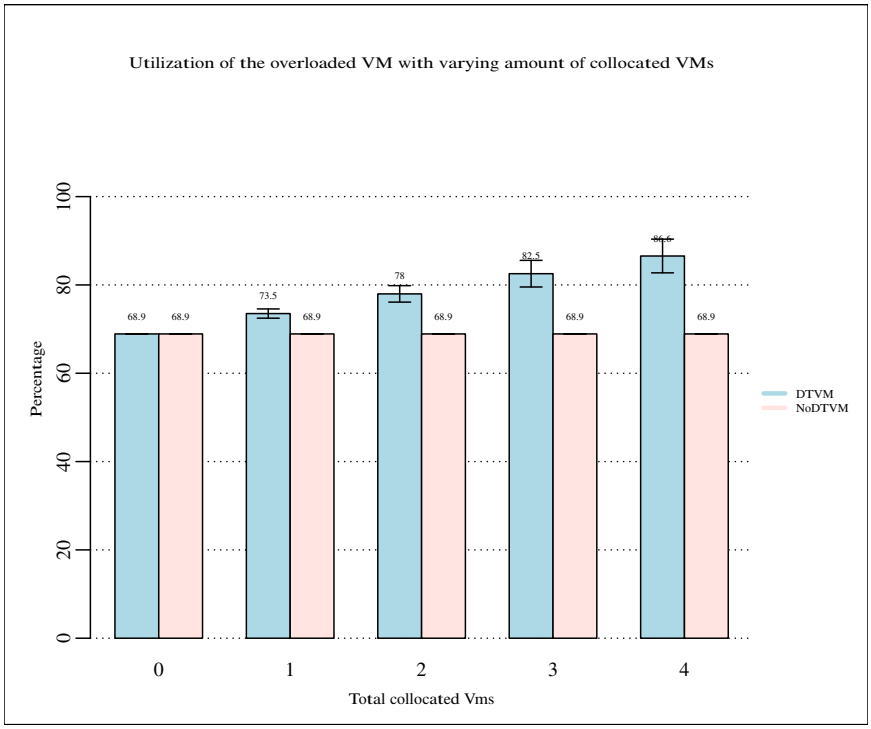

Figure 6.19: DTVM-Experiment 2-Result

In my second DTVM experiment, I created and used synthetic workload data (for a whole day) to further explore the benefit of DTVM over NoDTVM. There are, again, VMs of two categories: category A and category B. VMs of Category A require only one CPU and in the normal period, the workload on the CPU is between 56% to 64%. It is assumed that VMs of category A become highly overloaded (having spikes) for most of the time and during the peak period the workload remains between 81% to 99%. In this experiment, there are 4 sub-types of low-demand (Category B) VMs having different periods where their demand is low. The Category B VMs have moderate demands at other times. Thus, the potential to loan resources across the simulation period is limited for each Category B VM. VMs of Category B require two CPUs each and the workload in their normal period remains between 68% to 70%

but the workload decreases to 28% to 32% at specific times as shown in Figure 6.18. I have again considered a single physical machine where the physical machine has ten CPUs that are assigned to VMs of both categories.

I then verified the effectiveness of DTVM when there are different numbers VM_{Bs} (of category B) co-hosted with a VM_A (of category A) in a single physical machine (refer to Figure 6.19). When there is only a single VM_A in the physical machine then the utilization of VM_A is 68.9%. In this case, DTVM and NoDTVM of course provide the same amount of utilization for VM_A since there is no opportunity for resource sharing. When there is one VM_B (having underutilized workload from 8 am - 2 pm) co-located with VM_A the utilization of VM_A increases to 73.5% from 68.9% for DTVM but the utilization of VM_A remains the same in case of NoDTVM. When there are two VM_Bs (one having underutilized workload from 8 pm - 2 pm and the second one from 2 pm - 8 pm) co-hosted with VM_A, then the VM_{Bs} can share resources with VM_A at different time periods and thus further increase the utilization of VM_A. So, as expected, with an increase of the number of co-hosted VMs (having underutilized resources) the resource utilization of VM_A increases when using DTVM but not in the case of NoDTVM (Figure 6.19).

The obtained results confirm that DTVM increases overall **VM** and **host** utilization but the benefit of DTVM is subject to the characteristics of actual workloads depending on how frequently under-utilized VMs are co-located with over-utilized VMs during VM packing. To improve the effectiveness of DTVM, more "compatible" VMs need to be co-located. This was done statically by my CSVP algorithm. As workloads change, however, adaptation may be needed to achieve and then main-

tain such compatible co-locations. This is where DAVM, discussed next, is needed. Naturally, DAVM is assessed together with DTVM.

6.4.2 DAVM

In my first, simple, DAVM experiment I created a uniform distribution of VMs (using the Google tracedata), half of which were resource critical (including low, medium and high), and the remaining half of which were non resource critical (refer to the"RC-50" distribution in Figure 6.20). Each VM requires 5 cores and each PM provides 10 cores thereby, again, eliminating any effects due to imperfect packing which allows us to focus on the benefits of compatibility-based placement. I then compare a placement done using simple First Fit Decreasing (FFD), with one employing DTVM (which supports controlled resource sharing within a physical machine) [75] and my DAVM algorithm which, as described, tries to co-locate resource critical VMs with other types of VMs so that the pairings will be compatible and thereby provide better VM efficiency by being able to share resources more effectively between the co-located VMs using DTVM.

When the VMs are less resource critical (i.e. in the LowRC case), comparing DAVM and FFD, we see in Figure 6.20 that there is some improvement (5.8%) in the average VM utilization (of the resource critical VMs only) but the improvement increases to 11.6% when the VMs are moderately resource critical (i.e. in the MedRC case) and to 15.4% when the VMs are highly resource critical (i.e. in the HighRC case), refer to Figure 6.20A. The reason is that the higher resource critical VMs find opportunities for effective pairings so the resource utilization increases significantly.

I also compared the DAVM approach with the DTVM technique by itself. Since DTVM doesn't incorporate effective pairings in resource sharing it cannot provide as much improvement as DAVM does and this too is seen in the results.

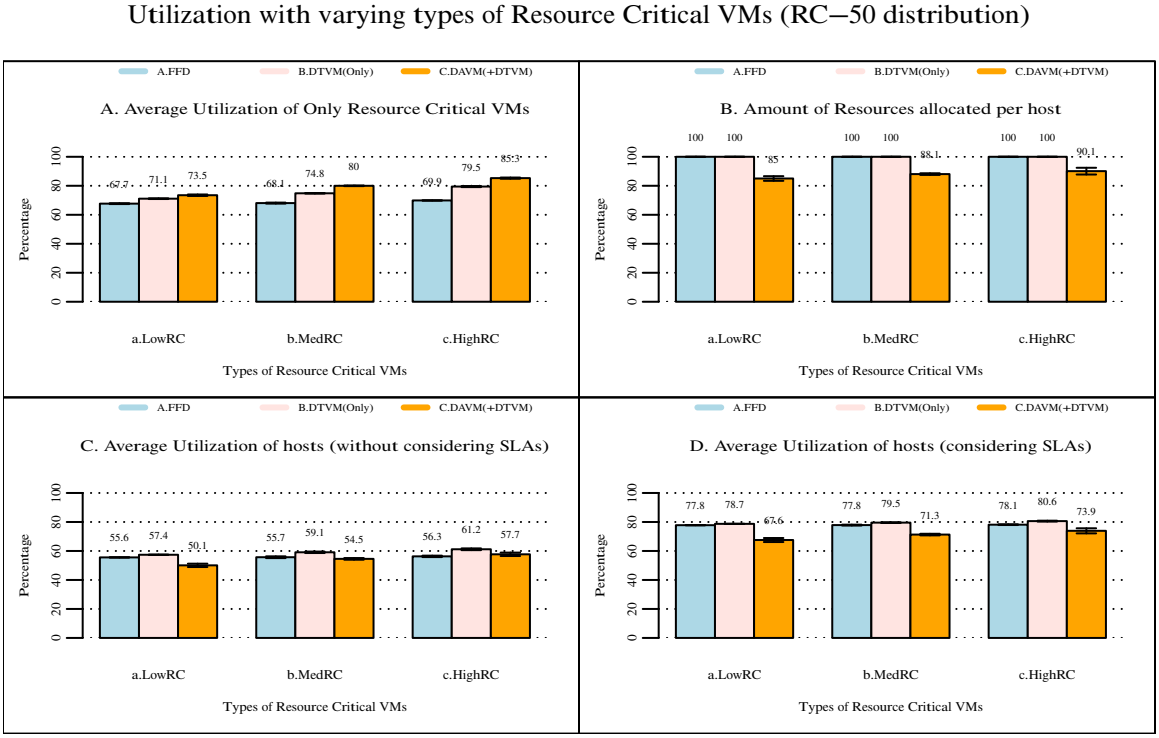

Figure 6.20: DAVM-Experiment 1(a)-Results

The overall host utilization[4] (both considering SLAs and without considering SLAs) and packing density degrade somewhat when DAVM becomes active. The amount of resources allocated per host (i.e. packing density) in the overall experimental period is shown in Figure 6.20B. Both FFD and DTVMOnly do not do any VM migrations and that's why they provide more efficient packing. Of course, they

[4]This absolute measure of host utilization considers the load actually introduced by the VMs running on the PM at some point in time. This can be significantly lower that the host utilization when considering the SLA requests made by the VMs since a VM may request (via the SLA) more resources than it may actually be using at some point in time.

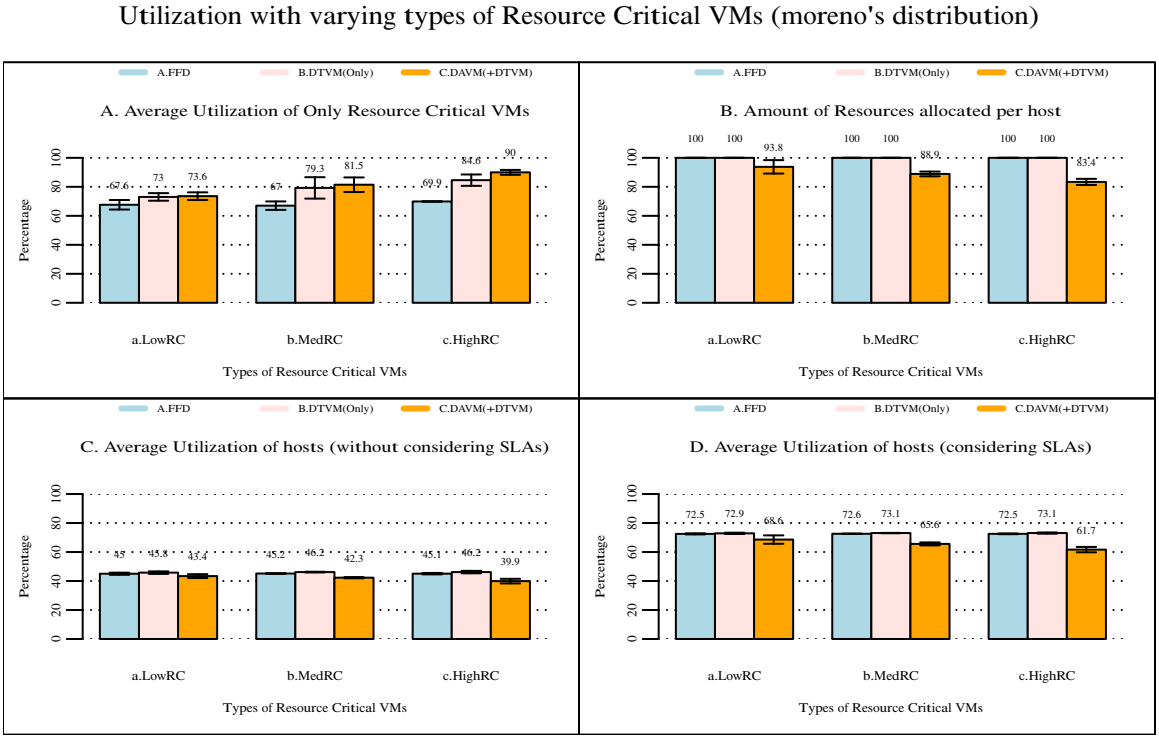

Figure 6.21: DAVM-Experiment 1(b)-Results

provide far less benefit in terms of VM performance when compared to DAVM which, after making effective pairings, must do the necessary migrations to co-locate the compatible VMs and thus DAVM is less efficient in packing VMs. For the same reason, the overall host utilization (refer to Figures 6.20C and 6.20D) in cases using DAVM is less compared to FFD and DTVMOnly.

In my second, follow-up, DAVM experiment, the distribution of the resource critical VMs was changed to reflect that found by Moreno et al. [61] in the Google Traces as a whole (results using Moreno's distribution are shown in Figure 6.21) and I found similar behaviour in the results. The Moreno distribution corresponds to a very conservative view of cloud workloads with few resource-critical VMs while the

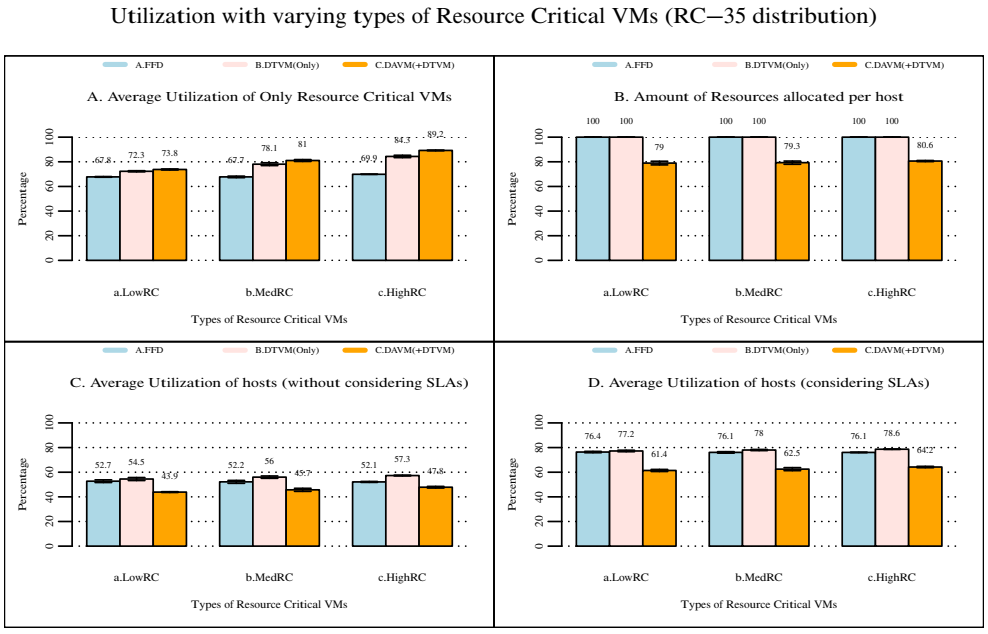

Figure 6.22: DAVM-Experiment 1(c)-Results

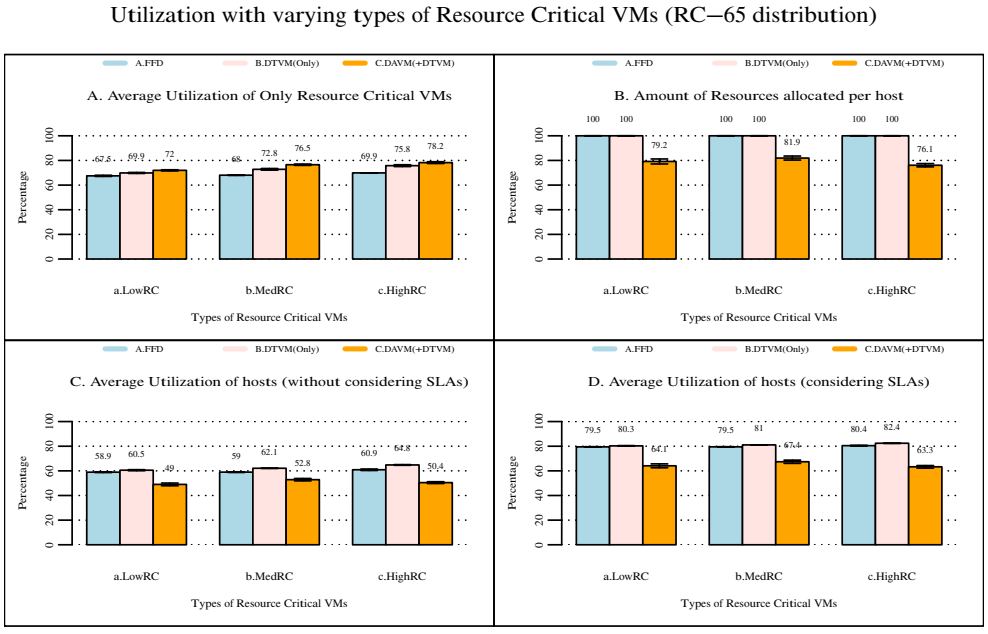

Figure 6.23: DAVM-Experiment 1(d)-Results

distribution used in my initial DAVM experiment had a much higher percentage of resource-critical VMs.

I also created two more distributions of VMs (to better understand DAVM's behaviour with different distributions having a significant percentage of resource-critical VMs). In the two new distributions, almost one third of the VMs (still drawn from the Google traces) are resource critical (including low, medium and high), and the remaining two thirds are non resource critical (i.e. "RC-35" distribution) and vice versa (i.e. "RC-65" distribution). I again found generally consistent behaviour (refer to Figures 6.22, 6.23), but in the case of the RC-65 distribution, the improvement in the average VM utilization (of the resource critical VMs only) is less compared to all other experiments. This is because some of the resouce critical VMs did **not** find any opportunity to create effective pairs due to the lack of sufficient non resource critical VMs to act as donors.

The obtained results from my DAVM experiments show the effectiveness of DAVM when there are sufficient and appropriate (i.e. compatible) VMs available. This permits DAVM to make pairings and perform the needed migrations to provide DTVM with co-hosted VMs that it can facilitate controlled sharing between to improve VM performance.

6.4.3 DRVM

Recall that DRVM (Dynamic Re-packing of VMs) is focused on addressing the problem of underutilized physical machines. It has the goal of combining the VMs from underutilized PMs onto fewer PMs so some machines may be shut down to con-

serve power. DRVM, like CSVP and DAVM, is compatibility-aware so unlike simply re-packing using an algorithm like FFD, it attempts to co-locate compatible VMs as it re-packs them into fewer PMs. DRVM solves the problem of PM underutilization regardless of the cause of the underutilization. Of interest in this thesis are underutilization due to side-effects of the DAVM pairing process and underutilization resulting from the completion of short-duration VMs without the arrival of new ones to backfill into the underutilized PMs. Accordingly, I undertook two DRVM related experiments, one to understand the algorithm's behaviour in each case.

DRVM-Experiment 1

In the first experiment I created a uniform distribution of VMs, half of which are resource critical (including low, medium and high), and the remaining half of which are non resource critical (refer to the "RC-50" distribution in Figure 6.24). Each VM requires 5 cores and each PM provides 10 cores thereby again eliminating any effects due to imperfect packing which allows us to focus on the benefits of compatibility-based re-packing. I then compare a placement done using simple First Fit Decreasing (FFD), with one employing my DTVM, DAVM, and DRVM algorithms.

The primary benefit of DRVM is, of course, in improving PM utliization. This, however, does not mean that we can ignore any possible impacts on VM utilization. Thus, I begin by reporting the results of running DRVM as well as DAVM and DTVM on VM utilization. Little change is anticipated since DRVM respects pairings made by DAVM which are the primary source of enhanced VM performance. When the VMs are less resource critical (i.e. in the LowRC case), when comparing DRVM

and DAVM with FFD to assess VM utilization, there is again some improvement (5.8%) in the average VM utilization (of the resource critical VMs only) but the improvement increases to 11.6% when the VMs are moderatly (i.e. in the MedRC case) and to 15.4% when the VMs are highly resource critical (i.e. in the HighRC case), refer to Figure 6.24A. The reason is still that the higher resource critical VMs find opportunities for effective pairings so the resource utilization increases notably. I also compared DRVM and DAVM with the DTVM technique by itself and since DTVM doesn't have a way of ensuring effective pairings for resource sharing, DTVM cannot provide the improvement that DAVM and DRVM together do.

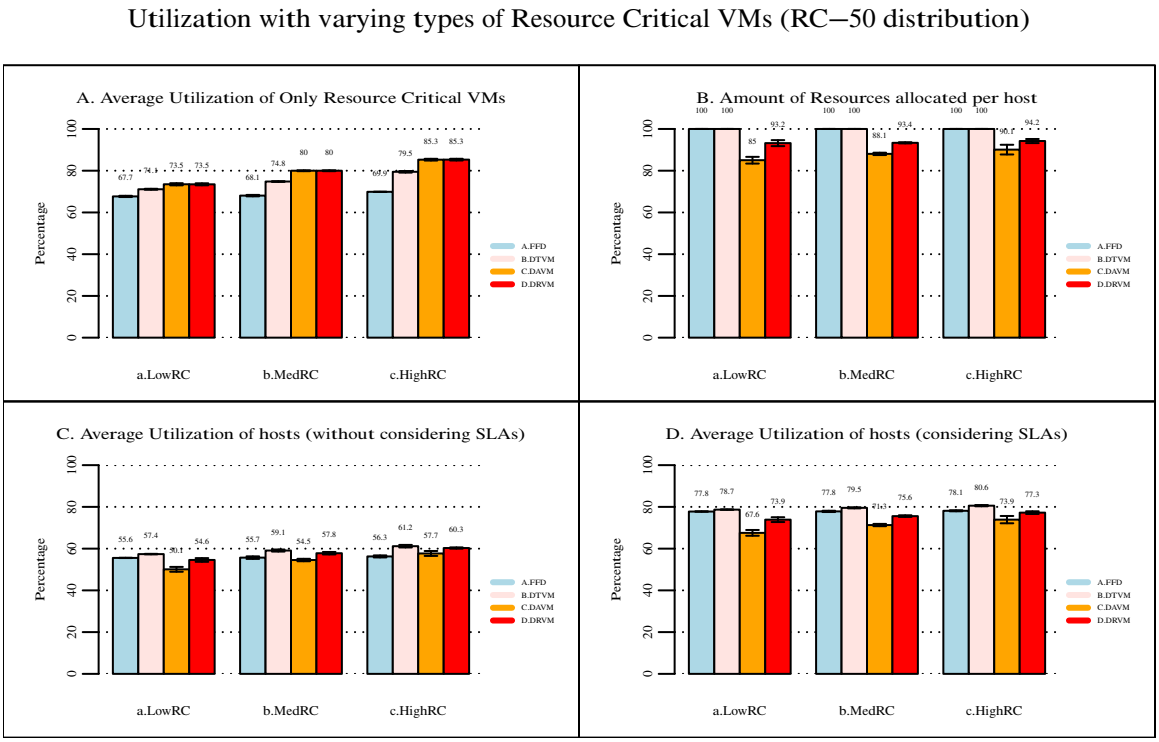

Figure 6.24: DRVM-Experiment 1(a)-Results

The overall host utilization (both considering SLAs and without considering SLAs)

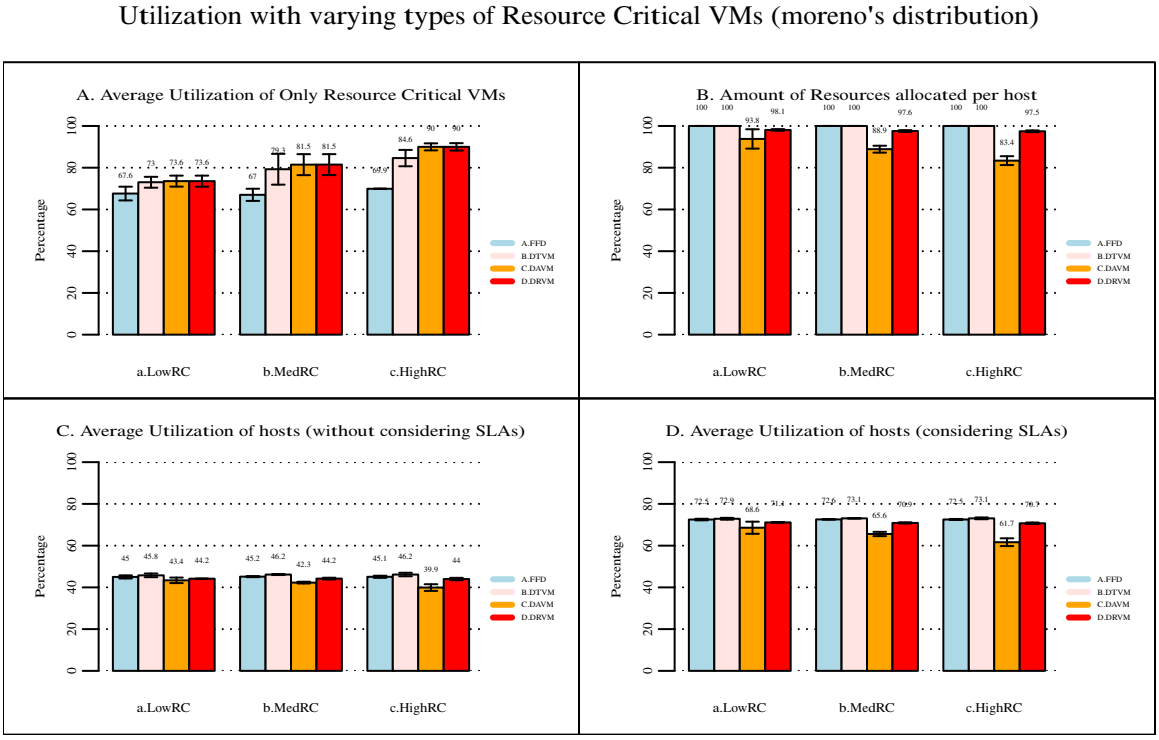

Figure 6.25: DRVM-Experiment 1(b)-Results

and packing density degrades due to the side-effects of DAVM but DRVM provides significant improvements in these cases. The amount of resources allocated per host (i.e. packing density) for the overall experimental period is shown in Figure 6.24B. Both FFD and DTVMOnly do not do any VM migrations and that's why they provide efficient packing whereas due to effective pairings and necessary migrations DAVM is less efficient in packing VMs in PMs but, due to its support for compatibility-aware re-packing, DRVM makes the packing more efficient while still ensuring the compatibility of co-hosted VMs. For the same reason, the overall host utilization (refer to Figures 6.24C and 6.24D) when using DAVM without DRVM is less compared to FFD and DTVMOnly, but when also using DRVM better overall host utilization is provided.

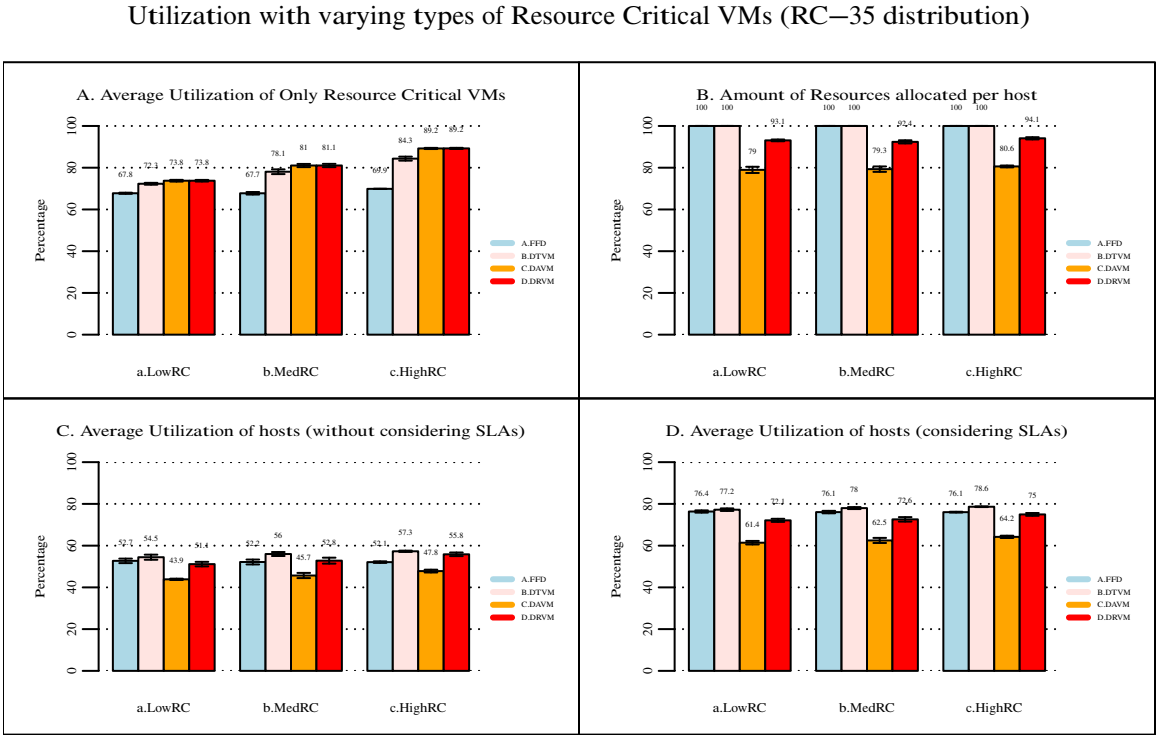

Figure 6.26: DRVM-Experiment 1(c)-Results

As in the DAVM experiments, I then changed the distribution of the resource critical VMs to reflect that found by Moreno et al. [61] in the Google Traces as a whole (refer to the results of Moreno's distribution in Figure 6.25). I again found very similar behaviour in the results. Finally, as in the DAVM assessment I then created two more distributions of VMs, almost one third of which were resource critical (including low, medium and high), and the remaining two thirds of which were non resource critical (i.e. RC-35 distribution) and vice versa (i.e. RC-65 distribution). I again found similar VM utilization behaviour (refer to Figures 6.26, 6.27) including in the case of the RC-65 distribution, where the improvement in the average VM utilization was less. In the cases of the RC-35 and RC-65 distribution, DRVM found a need to re-

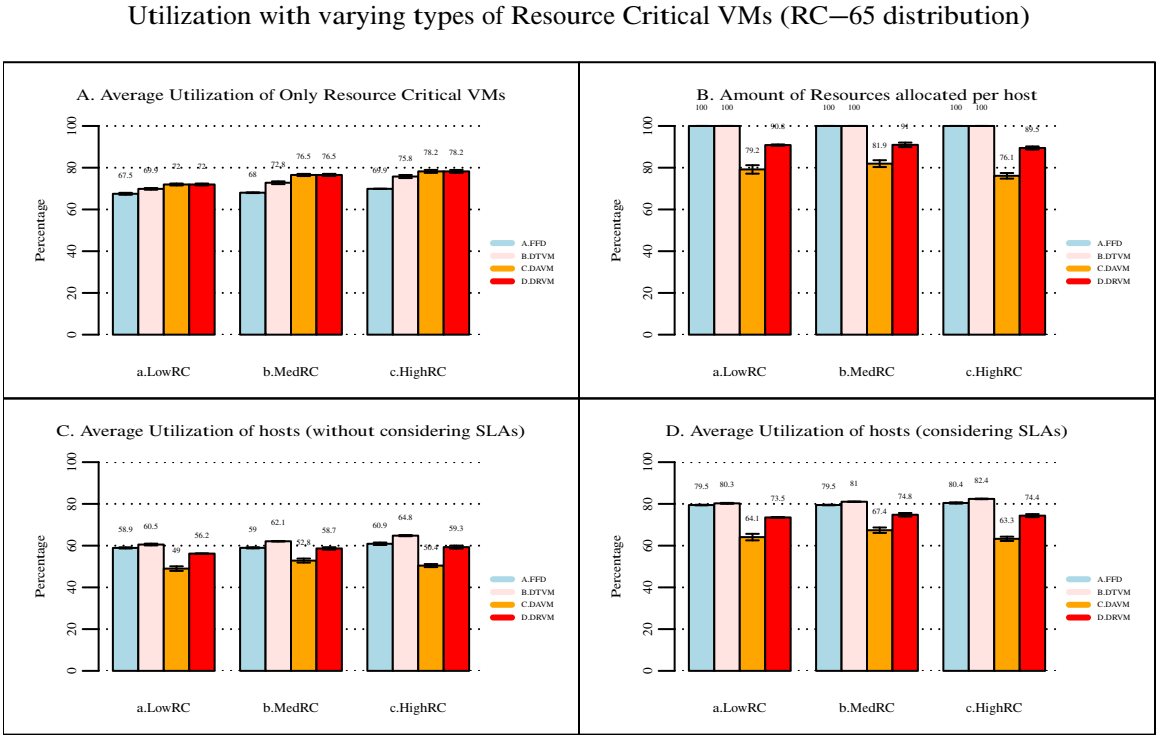

Figure 6.27: DRVM-Experiment 1(d)-Results

pack a larger number of VMs since DAVM did more pairing and migrations. DRVM thus provides greater improvement in the overall host utilization in these cases.

DAVM and DRVM both improve the overall VM utilization (specifically the utilization of the resource critical VMs) but in some cases DAVM reduces overall host utilization due to the need to bring additional hosts online to support the needed groupings and side-effects of the corresponding migrations. Due to the re-packing of VMs, DRVM improves the overall host utilization in response to this challenge.

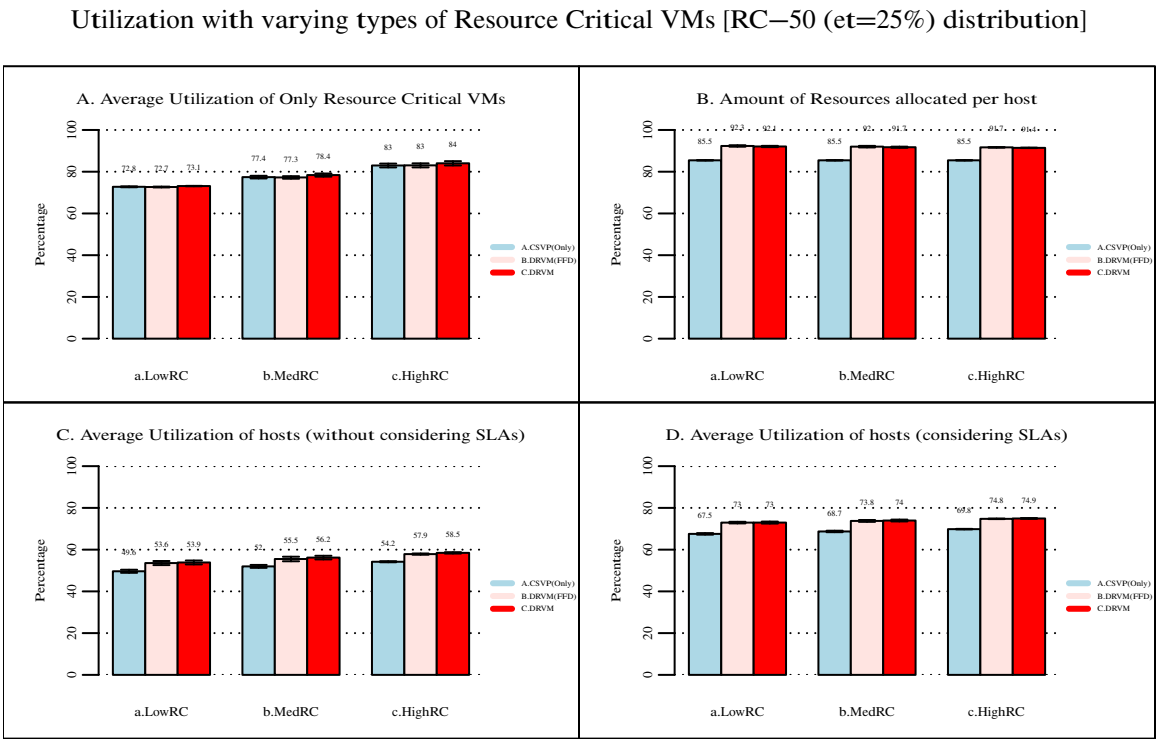

Figure 6.28: DRVM-Experiment 2(a)-Results

DRVM-Experiment 2

In the second DRVM experiment, I first created a uniform distribution of VMs (the same as for Experiment 1) half of which are resource critical (including low, medium and high), and the remaining half of which are non resource critical (refer to the results for the RC-50 distribution shown in Figure 6.28). In this experiment, I also introduce VMs which execute for a shorter duration (i.e. that terminate early) to demonstrate the ability of DRVM to also improve physical machine utilization when there are early-terminating ("et") VMs. I consider three different different cases for short-duration/early terminating VMs where 25% [in Experiment 2(a)], 50% [in Experiment 2(b)], and 75% [in Experiment 2(c)] of the VMs (both resource critical and

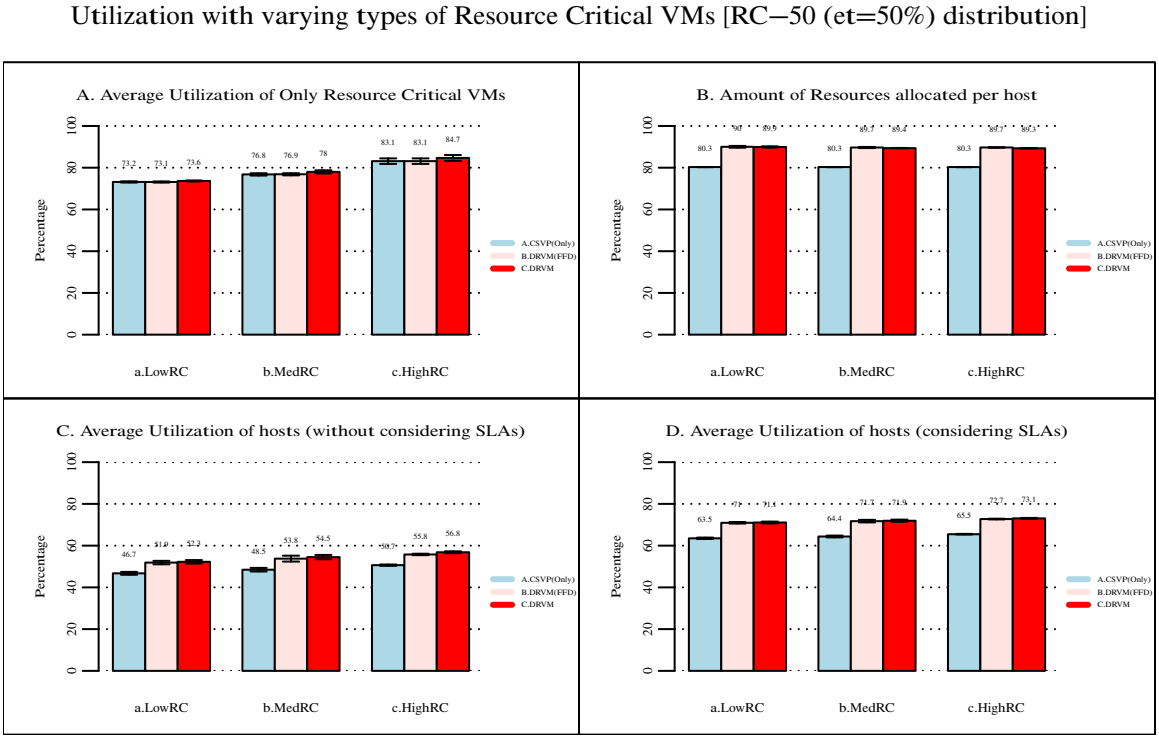

Figure 6.29: DRVM-Experiment 2(b)-Results

non resource critical) terminate early. Each VM again requires 5 cores and each PM provides 10 cores thereby once again eliminating any effects due to imperfect packing which allows us to focus on the benefits of compatibility-based placement and re-packing. I compare a placement done using simple **static** compatibility based VM packing (using CSVP) (where VMs are initially packed depending on CSVP's simple assessment of VM compatibility), with one employing DRVM-FFD (this version of DRVM only allows re-packing using FFD and does not consider compatibility in re-packing), and also with my version of DRVM that tries to co-locate resource critical VMs with **compatible** VMs while re-packing the VMs of underutilized PMs so that the pairings will be compatible and thereby provide better VM efficiency while main-

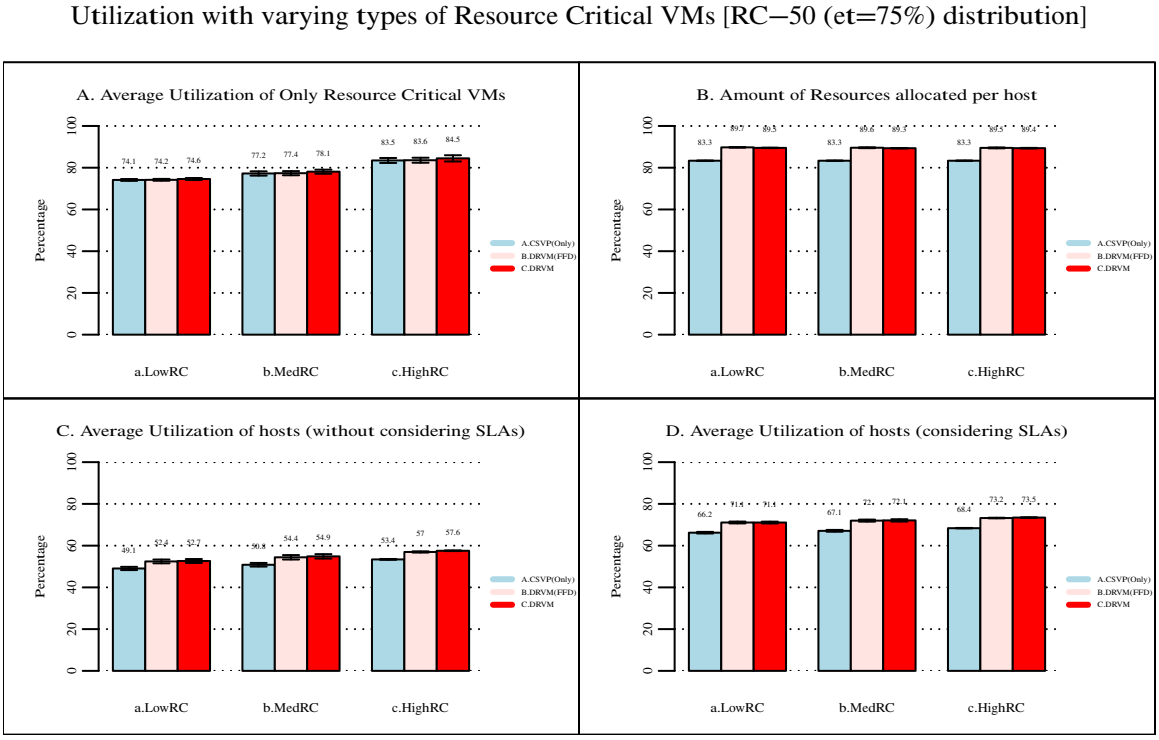

Figure 6.30: DRVM-Experiment 2(c)-Results

taining packing efficiency. In this experiment DRVM and DRVM(FFD) both intially pack the VMs using CSVP. DTVM (which allows resource sharing in a controlled manner) [75] is effective in all cases at improving VM utilization as there are always some compatible VMs in each host.

When there are 25% short duration VMs, Experiment 2(a) shown in Figure 6.28 and there are different levels of resource critical VMs (i.e. LowRC, MedRC and HighRC) DRVM, when compared with DRVM(FFD) and CSVP(Only), always provides some improvement in the average VM utilization (of the resource critical VMs only) but the improvement only increases by a modest 1% even when the VMs are highly resource critical (i.e. the HighRC case) [refer to Figure 6.28A]. The reason

that little difference is seen between the three algorithms is because compatibility based placement has already been done. Despite this, some improvement is seen with DRVM since DRVM considers compatibility while re-packing and if the higher resource critical VMs find opportunities for additional effective pairings then the resource utilization increases. DRVM(FFD) doesn't incorporate compatibility while re-packing so it cannot provide the improvement that DRVM does.

The amount of resources allocated per host (i.e. the packing density) for the overall experimental period with 25% early terminating (ec) VMs is shown in Figure 6.28B. Both DRVM and DRVM(FFD) have the goal to reduce the amount of under-utilized hosts. In this experiment this is also caused due to the termination of short duration VMs and that's why they both provide better packing when compared with CSVP(Only). DRVM(FFD) provides **slightly** more efficient packing than DRVM as DRVM(FFD) is only concerned with packing efficiency whereas DRVM considers compatibility of VMs on top of packing efficiency.

The overall **host** utilization (both considering SLAs and without considering SLAs) is improved using either DRVM or DRVM(FFD) but DRVM provides additional improvements in overall host utilization, compared to DRVM(FFD), due to its consideration of VM compatibility while re-packing.

I also did experiments where I increased the percentage of short duration VMs to 50% [refer to Figure 6.29] and 75% [refer to Figure 6.30] and observed the same type of improvements for DRVM over DRVM(FFD) and CSVP(Only). I also changed the distribution of the resource critical VMs to reflect that found by Moreno et al. [61] in the Google Traces as a whole and, as would be expected, found very similar behaviour

in the results. Thus, those results are not reported here in.

In conclusion, my assessments show positive results for all the algorithms presented in this thesis. My static HBF algorithm provides small but still useful improvements over FFD generally and specifically supports early starting of "urgent" VMs. It also improves packing in the case where multiple criteria must be considered. The results for my static CSVP algorithm show that surprisingly significant improvements in VM utilization can be achieved by a very simple static estimate of compatibility followed by only static compatibility-aware placement. The results obtained using the sample workloads from the Google traces for my dynamic algorithms indicate the overall effectiveness of the use of the three techniques together to improve VM utilization through careful controlled sharing of resources while still maintaining high efficiency of VM packing.

As mentioned earlier, I also created two synthetic scenarios intended to model new applications that might move into the cloud and require effective, dynamic compatibilty-aware provisioning of VMs. These two scenarios, one representing a set of e-health applications and the other a set of ubiquitous computing applications and the associated simulation results are described in the following section.

6.5 Simulation Results - Synthetic Scenarios

Many health care applications like radio image archiving, health system management, patient records, etc. have successfully been migrated into "the cloud" [57; 58; 35]. Considering the benefits offered by cloud computing a similar transition can easily be foreseen for ubiquitous applications (e.g. Rightscale, Kaavo and KOALA

project [49], Jingzhou project [48], etc.). Due to their load varying and on-demand characteristics, such applications introduce further challenges to successful cloud-based implementation. I created synthetic scenarios for eHealth and ubiquitous applications intended for further evaluation of my proposed algorithms to associate specific behaviours with benefits seen from my algorithms. The two scenarios and the results of some basic experiments using them are described below.

6.5.1 eHealth Scenario

I created six eHealth related applications and considered VMs running an eHealth application workload for a whole day [shown in Figure 6.31]. The eHealth applications and their components are:

- Stroke Assessment application I (MRI, Angiography, Ultrasound, Echocardiogram, Electrocardiogram)
- Stroke Assessment application II (CTScan)
- Telemedicine application I (video conferencing, teleradiology, telecardiology)
- Telemedicine application II (telesurgery)
- Patient monitoring application (pulse and oxygen sensor, breathing sensor, temperature sensor, pressure sensor, sweating sensor, electromyogram sensor),
- E-billing management application (patients billing, hospital accounting, patients appointment)

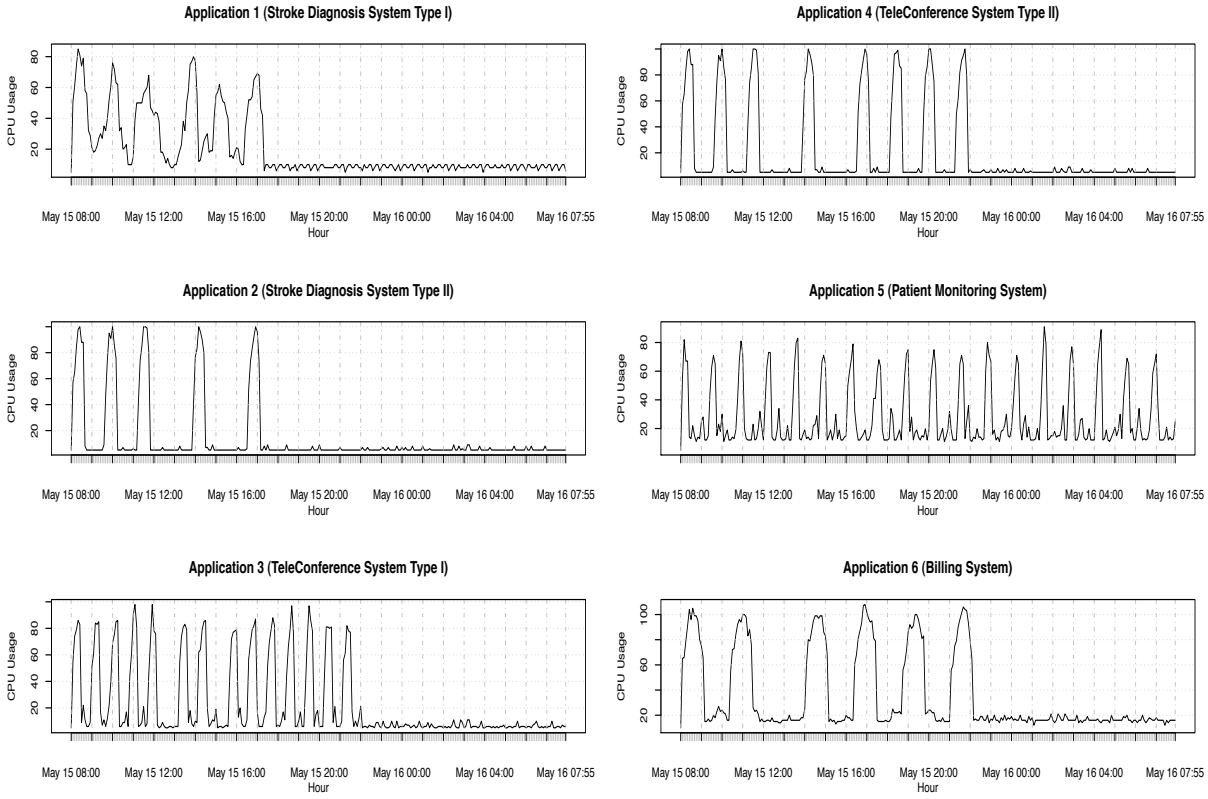

Figure 6.31: eHealth Workloads

Table 6.2: eHealth Applications distribution in VM

Application Type	Number of Application Instances in a VM
Stroke Assessment application I	1
Stroke Assessment application II	3
Telemedicine application I	2
Telemedicine application II	1
Patient monitoring application	25
E-billing management application	5

eHealth Systems, Application Components and Workloads

I have modelled a diagnostic center (whose IT needs could be met using a cloud) where patients come to have a number of diagnostic tests performed. I assume that

Workloads in eHealth System

Application 1 (Stroke Diagnostic System Type I)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	MRI	30-100	5	3D Images	3500
2	Angiography	60-120	6	Video	1500
3	Ultrasound	15-30	1	Live Images	1800
4	Echocardiogram	45-50	3	2D Images	4500
5	Electrocardiogram	5-10	1	Image	700

Application 2 (Stroke Diagnostic System Type II)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	CT scan	20-60	4	3D Images	4000

Application 3 (Teleconference System Type I)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	Video Conferencing	10-30	2	Audio, Video	4500
2	Teleradiology	2-5	1	Image	600
3	Telecardiology	8-10	1	Image	900

Application 4 (Teleconference System Type II)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	Telesurgery	30-60	4	Interactive Video	12000

Application 5 (Patient Monitoring System)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	Pulse and Oxygen sensor	1-5	2	Text	10
2	Airflow, breathing sensor	2-10	2	Text	20
3	Body temperature sensor	1-5	1	Text	10
4	Blood pressure sensor	2-10	1	Text	50
5	Sweating sensor (GSR)	3-15	3	Text	100
6	Electromyogram sensor(EMG)	5-25	4	Image	300

Application 6 (Billing System)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	Patient's billing	20-60	4	Text	800
2	Hospital Accounting	30-90	5	Text	800
3	Patient's Appointment	60-120	6	Text	500

Figure 6.32: eHealth Application Workload Assumptions

the number of patients visiting the diagnosis center varies in a day. Depending on the number of patients and their individual test requirements, the physicians in the diagnostic center have different levels of interactions with the eHealth applications

running in the cloud and thus produce different amounts of workload across the day. Among the six eHealth applications, two applications are for Stroke Diagnosis, two for Telemedicine, one application is for Patient Monitoring and one for Appointment and Billing Management (traditional health system functions that may already be done in the cloud). The eHealth applications and their components are described below and the assumed workloads of eHealth applications are shown in Figure 6.31 which were obtained from Figure 6.32 which determines a workload estimate based on the volume of data manipulated, the duration of manipulation and the number of instances of each type of data (e.g. MRI/CT/...) being manipulated. Some brief background on the work done in each system is now provided.

Stroke Diagnosis System

MRI uses non-ionizing radio frequency (RF) signals (without X-rays or dyes) to acquire images and it produces very accurate pictures of the brain (and its arteries). In MRI, a magnetic field is generated and the response to this magnetic field is then used to create an image of the brain (to detect a wide variety of brain and blood vessel abnormalities such as those occurring during strokes). MRI can provide direct views of 3D images (allowing the doctor to manipulate the imagery to detect tiny changes in structures) from almost any direction. In contract, CT scans only provide images in a single axial orientation [88]. An MRI test takes between 30 and 90 minutes to complete.

Cerebral angiography [90] is used to detect abnormalities in the brain's blood vessels (such as narrowing or blockage) by taking images of the blood vessels of the brain and the blood flowing through them. It is usually done after another test (such

as a CT scan) that has already detected an abnormality. Regular x-ray is used to image a radioactive dye (a contrast medium which is injected into carotid arteries) that flows through the blood vessels. X-rays absorbed by the injected dye become one of many images which finally are composed to form a movie of the blood flow through the vessels. The entire procedure can take from one to two hours to complete.

In carotid *ultrasound*, narrowing of the carotid arteries caused by cholesterol deposits can be detected in patients who have had a stroke or who might be at high risk for a stroke. Carotid ultrasound does not use dyes or X-rays but rather uses a transducer to emit high-frequency/ultrasound waves that are reflected by the carotid arteries and the red blood cells moving through them differently. The reflections of the sound waves are then measured and converted into live pictures of the arteries and the blood flow. This procedure usually takes 15 to 30 minutes to complete and the results are immediately known by the doctor [65].

In a stroke evaluation, *echocardiogram* can be used to find out if there is an abnormality of the heart that could lead to a stroke. This test makes images with higher resolution of the hearts walls, valves, internal structures of the heart and gives information about heart motion, the size of heart chamber relative to suspected blood clots. An ultrasound transducer is positioned against the skin and the picture of the heart can be viewed from several different angles by varying the transducers position. The images are typically stored as a 15-minute recording to be viewed later by the physician and the whole procedure requires about 45 minutes to finish [9].

Electrocardiogram(ECG) is done often and routinely and is considered as a regular test to learn about any irregularity of a patients heartbeat, the size and position of

the heart chambers and/or to discover any damage to the heart. About 12 electrodes (which contain wires that can detect the electrical signals of the heart through the skin) are attached to various parts of the upper body and the computing device traces the heart rhythm (a snapshot of the electrical activity) producing a graph. The doctor can instantly know the basic vital characteristics of the heart. The procedures take about 5 minutes [91].

CT scan is one of the important tests done in a stroke evaluation. CT scan is a medical imaging procedure that produces a three-dimensional image of a specific area of the body (in this case the brain) which is generated from a large series of two-dimensional X-ray images taken around a single axis of rotation [36]. In stroke evaluation, CT scan is usually used to diagnose areas of abnormalities in the brain, and can help to determine the reasons behind them. In CT scanning, the patient lies in a tunnel-like machine while the inside of the machine rotates and takes X-rays of the head (hundreds of snapshots are taken) from different angles which are later used by computers to make an image of a cross-section of the brain. The procedure usually takes between 20 minutes and an hour.

Telemedicine System

In Telemedicine, communications (along with the transmission of medical and health information) between patient and medical staff takes place because they are not physically in the same location. *Videoconferencing* involves simultaneous two-way video and audio transmissions or live exchange/sharing of video, still images, audio, and documents related to health using appropriate encryption and decryption techniques. *Telesurgery* enables a doctor to perform certain routine surgeries on a

patient remotely. Telesurgery generally involves elements of robotics, real time and high-speed data connections. Telesurgery makes the expertise of specialized surgeons available to patients worldwide without the need for patients or doctors to travel. The duration of each surgery depends on the type of surgery and is generally from 30 minutes to 1 hour [36]. *Teleradiology* is the most popular use for telemedicine which allows the sending of radiographic images (X-rays, CT, etc.) of appropriate resolution from one location to another to retrieve needed radiology images with the desired quality to analyze for clinical purposes [79]. *Telecardiology* can be used to monitor patients with pacemakers in remote areas. Telecardiology is typically where Electrocardiographs (ECGs) are transmitted from a remote location using radio networks with special algorithms for eliminating noise [79].

Patient Monitoring System

Pulse and Oxygen sensors indicate the patients heart rate and arterial oxygen saturation of functional hemoglobin which determines the amount of oxygen dissolved in blood. It is typically used to help diagnose conditions where a patient is suspected of having breathing related issues. *Airflow, breathing measurements* are used to measure the breathing rate of a patient and can detect major physiological instability. A *temperature sensor* is used to detect the temperature of a specific part of the body. This is widely used since a number of conditions are accompanied by specific, localized changes in body temperature. A *blood pressure sensor* provides automatic measurements of the pressure/force that the heart creates on arteries when pumping blood. This sensor identifies high/low pressure conditions that may correspond to serious heart problems. *Sweating sensors (GSR)* that measure galvanic skin response pro-

vide measurement of the moisture level of skin by testing the electrical conductance of the skin. GSR helps to identify the psycho galvanic reflex of the body. An *electromyogram sensor (EMG)* senses electrical activity produced by skeleton muscles. EMG activates the cells and records cell activity electrically or neurologically. Any combination of these sensors may be used remotely to unobtrusively gather important information on patient health.

e-Appointment and e-Billing Management System

This system provides solutions for booking patients appointments (to visit physicians and diagnosis centers); handling patients overall billing information pertaining to various services including lab tests, food and beverages, bed charges, etc. based on specific hospital/clinic rules. It also involves income and expense analysis of the hospital, pharmacy center, etc.

eHealth Scenario - Results

I used the synthetic eHealth workloads running in VMs to further assess the performance of my algorithms. Depending on the number of instances of each application[as shown in Table 6.2], I assigned a corresponding collection of resources to each VM. I then verified the effectiveness of my techniques for a varying number of VMs, offering more or fewer opportunities to benefit from more effective resource sharing between VMs. Figure 6.33 shows a comparison of the NoDTVM, DTVM alone, DAVM and DRVM techniques. With a total of 40 (forty) VMs, the overall **VM** utilization is 16% and the **host** utilization is also 16% (not considering SLAs) but 58% (when considering SLA requests) for the case when none of my algorithm

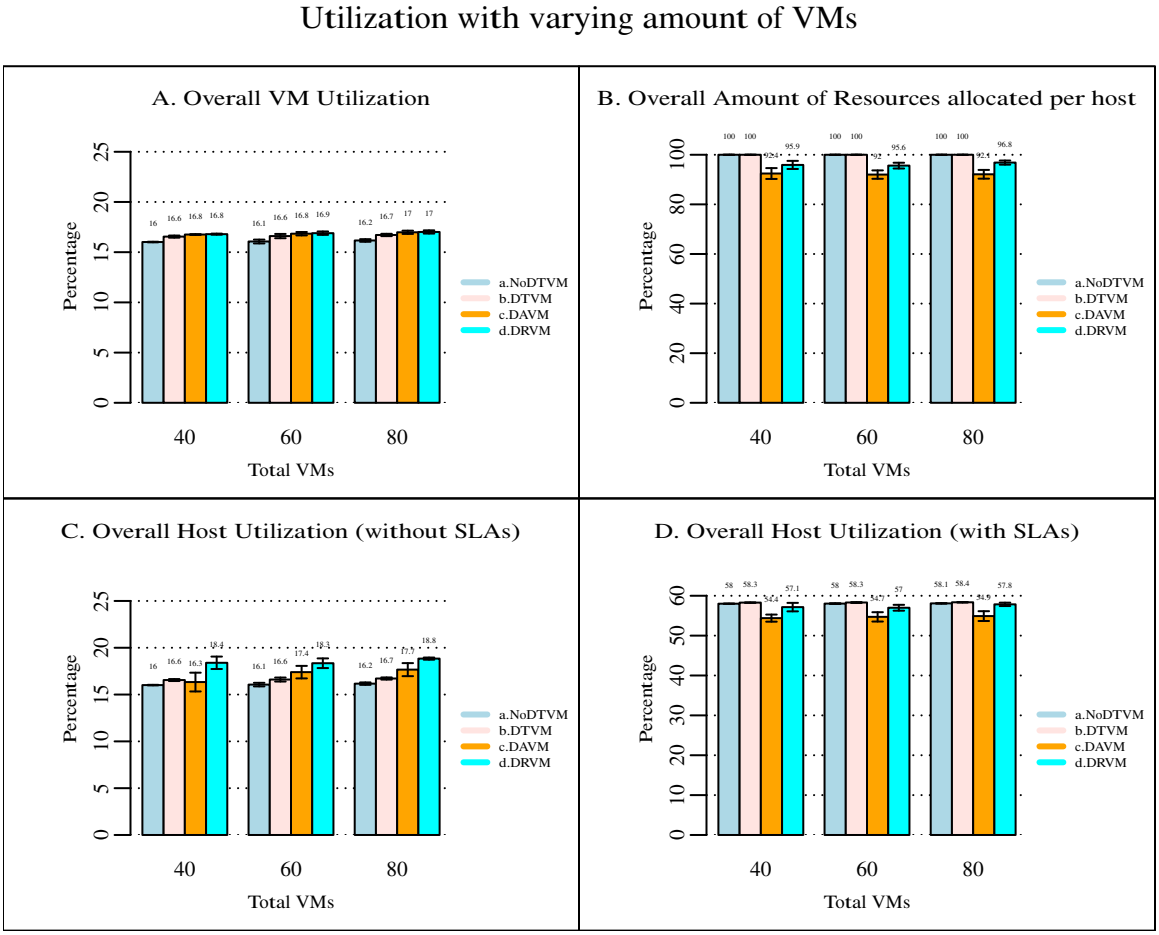

Figure 6.33: Results (with eHealth Application workloads)

is used (labelled "noDTVM") in Figure 6.33 - (A,C,D). When DTVM is used, the overall **VM** utilization increases to 16.6% from 16% and the overall **VM** utilization is further increased to 16.8% in the cases of DAVM and DRVM [refer to Figure 6.33 - (A)]. The overall **host** utilization (not considering SLAs) is also increased to 16.6% from 16% for DTVM but is reduced to 16.3% for DAVM, since additional host(s) had to be brought online to accomodate DAVM pairings. Finally, the use of DRVM improves the utilization to 18.4% [refer to Figure 6.33 - (C)] by re-packing under-utilized VMs together in fewer hosts. The overall **host** utilization (when con-

sidering SLAs) increases slightly to 58.3% from 58% for DTVM but is reduced to 54.4% for DAVM. DRVM improves the utilization to 57.1% [refer to Figure 6.33 - (D)]. As before, we are trading off improved **VM** utilization (i.e. VM performance) against slightly reduced packing efficiency. For different numbers of VMs, I found that DRVM provides similar improvements in overall **VM** utilization as DAVM but provides significant improvements on overall **host** utilization. DAVM performs better than DTVM alone considering overall **VM** utilization and DTVM by itself performs better than NoDTVM considering both **VM** and **host** utilization. Because no migration takes place in the cases of NoDTVM and DTVM alone, the overall VM packing is better for them when compared to DAVM and DRVM [refer to Figure 6.33 - (B)].

The reason for the difference in corresponding utilization for different techniques is that, in the eHealth scenario, there are some VMs running workloads with shorter and longer duration load variation leading to SLA violations. The shorter duration SLA violations are satisfied using DTVM. The differences in utilization become more significant with DAVM or DRVM as a greater percentage of the SLA violations are avoided due to good pairings and/or efficient packing.

6.5.2 Ubiquitous Computing Scenario

Ubiquitous computing provides the ability for individuals to easily, intuitively and effectively interact with devices in the environment to accomplish daily functions. Ubiquitous applications are very user-friendly, highly responsive, and are commonly designed to be aware of preferences and to adapt to the users current context (location,

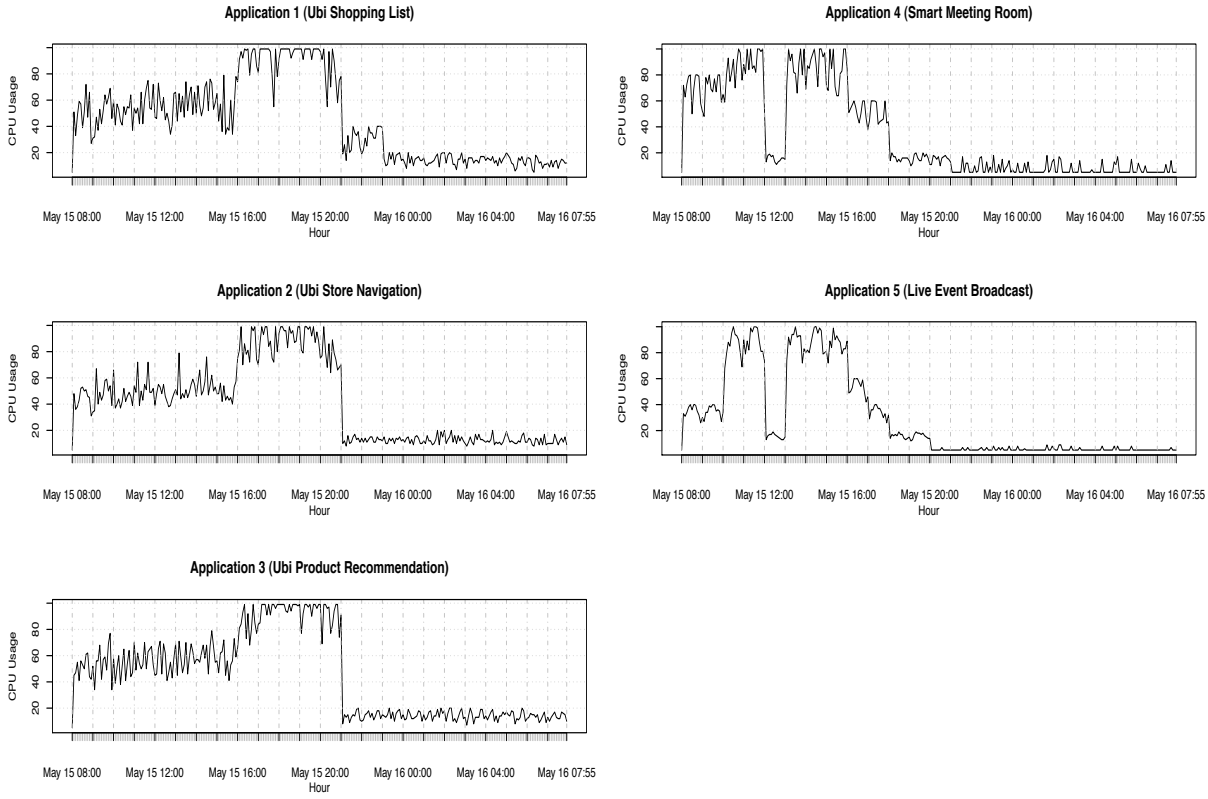

Figure 6.34: Ubiquitous Workloads

activity, etc.). As such they represent another class of workloads that are atypical in current cloud usage and that might make an interesting test case for my algorithms.

I also created a small family of ubiquitous computing scenarios where sensors, cameras, display, etc. are used to collect information and interact with users. Specifically, I created five separate synthetic ubiquitous applications (three of which are related) and consider VMs running the resulting ubiquitous application workloads. The workload of the applications depends both on the number of users served concurrently, which varies depending on the time of the day, and on the users actions. I have plotted a whole day's workload [shown in Figure 6.34] obtained based on the ap-

Workloads in Ubi System

Application 1 (Ubi Shopping List)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	User's Context (shopping frequency, User's travel schedule, Upcoming invitations, shopping mall enrooted, etc.)	1-5	1	Text	35
2	Item's Context (Items available and unavailable, Items which will be finished very soon, etc.)	1-5	1	Text	20
3	Environment Context (Sensing Space in Refrigerator,etc., weather forecasts, etc.)	1-5	1	Text, 3D Image	3000
4	Suggested Information (Shopping list, Shop address, etc.)	1-5	1	Text, Image	600

Application 2 (Ubi Stores Navigation)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	User Context (Current Location, Time Constraints, Schedule of the day, trip planning, etc.)	1-5	1	Text	10
2	Product Context (Items to purchase, optional items, store selection (pricing & preferences), etc.)	1-5	1	Text	50
3	Store Context (Store preference & location, distance, traffic, store hours, etc.)	1-5	1	Text	100
4	Suggested Information to user (Navigation, alternate routes, etc.)	1-5	1	Text, Image and Video	1600

Application 3 (Ubi Product Recommendation)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	Customer's Context (Food Habit, Nutrition Constraints, Salary Structure, Price Constraints, Food Allergy, Number of family members, Purchase history, etc.)	1-5	1	Text	20
2	Product's Context (Items available, Quality of Products, Expiry Information, Products on Sale current and in future, etc.)	1-5	1	Text	25
3	Suggested Information (product list, etc.)	1-5	1	Text, Image and Video	1600

Applications 1 - 3 Usage Frequency

8:00 AM -4:00 PM (1-96): Medium 4:00 PM-9:00 PM (96-156): High 9:00 PM-8:00 AM (156-288): low

Figure 6.35: Ubiquitous Workloads Assumptions

plication characteristics shown in Figures 6.35 and 6.36. The ubiquitous applications are described below:

- Ubi Shopping List Application (parameterized based on user, item and environment context): This application collects user information, items for purchase and environment context information, processes them and suggests a shopping list to its users. The user context includes information such as the shopping frequency of the user, user's planned travel schedule, upcoming invitations, etc.. The item context includes information related to the items required, items available or unavailable for purchase, items frequently used, items close to being consumed, etc. The environment context includes information like available space in the refrigerator/cupboards, weather forecasts, for travel, etc. that may influence normal shopping behaviours.

- Ubi Store Navigation Application (parameterized by product, user and store context): This application collects user information, the shopping list (products to purchase) generated by the previous application and stores' context information, processes them and suggests a navigation plan (i.e. a route) to its users so that they can visit the store(s) that they need to visit in an optimal way to efficiently purchase the products on the shopping list. The user context includes information like current location, time constraints, schedule, etc., the product context includes information related to product pricing and quality preferences, etc., and the store context includes information like store location, hours, distance, nearby traffic, etc.

- Ubi Product Recommendation Application (parameterized by customer and

Workloads in Ubi System (continued)

Application 4 (Smart Meeting Room)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	Participant's Context (Current Location, Time Constraints, Schedule of the meeting, media preferences, etc.)	4-5	2	Text	20
2	Meeting Room's Context (Participant Monitoring, Local/Remote participant's communication, Meeting Archive, content sharing, "out-of-band" personal inter-communication, etc.)	30-120	1	Text, Image, Audio, Video	4500
3	Suggested Information (related topic, etc.)	30-120	1	Text, Image and Video	1800

Application 4 Usage Frequency

8:00 AM -10:00 AM (1-24): high
10:00 AM -12:00 PM (24-48): very high
12:00 PM -1:00 PM (48-60): very Low
1:00 PM -4:00 PM (60-96): very high
4:00 PM -6:00 PM (96-120): medium
6:00 PM-9:00 PM (120-156): very Low
9:00 PM-8:00 AM (156-288): No

Application 5 (Live Event Broadcasting)

SL	Components	Duration (in min)	Relative Occurrences	Media Type	Workload
1	Live Broadcast (bandwidth, room space, number of cameras used, commentators, archive information, etc.)	30-120	1	Text, Audio, image, Video	2000
2	Viewer (resolution requirements)	30-120	1	Video	3500
3	Suggested Information (breaking news, advertisement, etc.)	30-120	1	Text, Image and Video	1500

Application 5 Usage Frequency

8:00 AM -10:00 PM (1-24): Low
10:00 AM -12:00 PM (24-48): very high
12:00 PM -1:00 PM (48-60): very Low
1:00 PM -4:00 PM (60-96): very high
4:00 PM-5:00 PM (96-108): medium
5:00 PM -6:00 PM (108-120): Low
6:00 PM-8:00 PM (120- 144): Very Low
8:00 PM-8:00 AM (144-288): No

Figure 6.36: Ubiquitous Workloads Assumptions (continued)

product context): This application collects customers' purchase preferences, shopping list, available alternative products in stores, processes the context information and recommends a number of related products to each customer while

visiting the store. The customer context includes information about nutrition, allergy, and price constraints, etc., the product context includes information related to the quality, sale history, expiry dates of the products, etc.

- Smart Meeting Room Application (parameterized by participant and meeting room context): This application (which is unrelated to the first three) helps to achieve smart collaboration among the participants in a meeting room. The application collects participants' context information and meeting room context information, processes them and recommends related and useful information to the participants. Each participant's context includes information related to media preferences (based on devices being used), current location, etc., the meeting room's context includes information like content sharing abilities, meeting archive, etc.

- Event Broadcasting application (parameterized by live broadcast and viewer context): This application broadcasts any live event to a number of users. The application collects viewers' context information and the event's context, processes them and recommends related and useful news, advertisements, etc. to the viewers. The broadcast context includes information related to bandwidth, environmental characteristics, cameras used, etc. while viewer context information includes such things as device resolution requirements, etc.

Ubiquitous Scenario - Results

In my ubiquitous computing experiment, I used ubiquitous workloads running in VMs to assess performance. Depending on the number of instances of each appli-

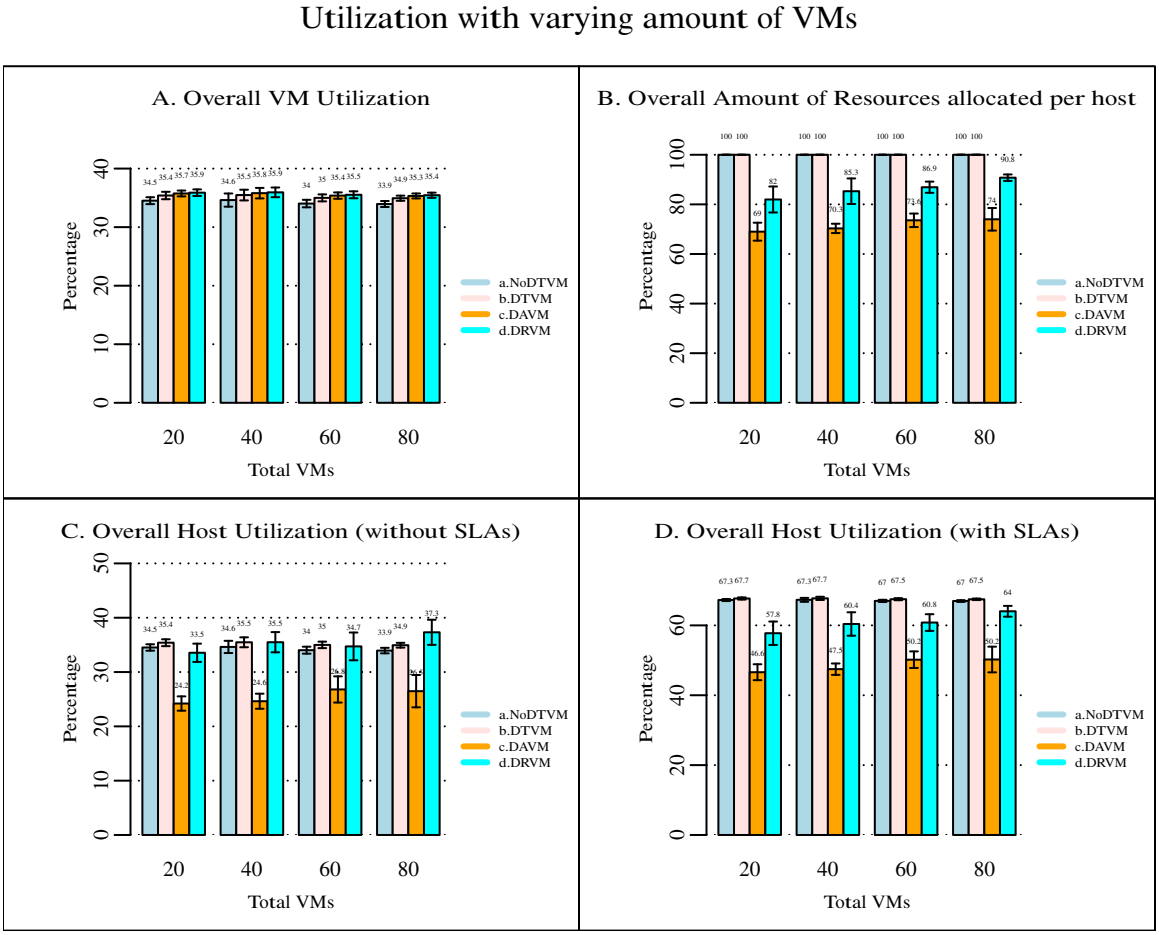

Figure 6.37: Results (with Ubiquitous Application workloads)

Table 6.3: Ubiquitous Applications in VM

Application Type	Number of Application Instances in a VM
Ubi Shopping List application I	2
Ubi Store Navigation application II	5
Ubi Product Recommendation application I	4
Smart Meeting Room application II	1
Event Broadcasting application	1

cation [as shown in Table 6.3], I assigned required resources to each VM. I, again, verified the effectiveness of my techniques when there is a varying number of VMs reflecting opportunities to benefit from more effective resource sharing. Figure 6.37

shows a comparison of the NoDTVM, DTVM only, DAVM and DRVM techniques. When there are 20 (twenty) VMs, the overall **VM** utilization is 34.5% and the **host** utilization is 34.5% (not considering SLAs) and 67.3% (when considering SLAs) for the case when DTVM is not in use. When DTVM is used, the overall **VM** utilization increases to 35.4% from 34.5% and the overall **VM** utilization is further increased to 35.7% in the case of DAVM and to 35.9% in the case of DRVM. The overall **host** utilization (when not considering SLAs) is increased from 34.5% to 35.4% for DTVM but is significanly reduced to 24.2% for DAVM (as migrations take place which cause the powering up of additional hosts). Finally, DRVM improves the utilization to 33.5% compensating for much of the lost PM utilization induced by DAVM migrations. The overall **host** utilization (when considering SLAs) is increased to 67.7% from 67.3% for DTVM but is reduced to 46.6% for DAVM and finally DRVM improves the utilization to 57.8%. In all the cases, it was found that DRVM provides similar improvements in overall **VM** utilization to DAVM but provides significant improvements in overall **host** utilization. DAVM performs better than DTVM considering overall **VM** utilization only and DTVM performs better than NoDTVM considering both **VM** and **host** utilization.

The reason for the difference in corresponding utilization for the different techniques is that, again, in the Ubiquitous scenario, there are some VMs having workloads with both shorter duration and longer duration SLA violations. The shorter duration SLA violations are satisfied using DTVM. The differences in utilization again become more significant with DAVM or DRVM as a greater percentage of SLA violations are avoided due to good pairings and/or efficient packing.

6.5.3 Comparison with Google Trace Results

The results seen for the eHealth and ubiquitous computing applications generally reflect those seen for the Google trace based results but we now further know that variability in resource demand due to computational peaks is the source of behaviour that leads to opportunities for resource sharing. Further experiments varying the number and distribution of the synthetic VMs would allow case-by-case verification of algorithmic details and workload mix factors that influence the effectiveness of DAVM and DRVM.

The trend of VM utilization gain (our primary concern) is certainly consistent in all three scenarios (eHealth, Ubiquitous and GoogleTracedata). In all cases there are some VMs with shorter and longer duration SLA violations and also some VMs under-utilizing their assigned resources to offset the SLA violations. It is interesting to note that the VM utilization gain in the experiments with the Google tracedata is more significant compared to the two other scenarios. The reason is that the usage of most of the applications in the eHealth and Ubiquitous scenarios at night is comparatively low, and there are fewer SLA violations compared to the VMs in the Google tracedata. When there are more SLA violations and also sufficient other VMs to offset the violations, there is more opportunity for effective pairing or packing that improve utilizations significantly.

Chapter 7

Conclusion

The goal of the research presented in this thesis is to address increased variance in the requirements of VMs running in cloud environments and is intended to thus broaden the range of applications for which clouds are useful. To this end, I have created two new static provisioning algorithms (HBF and CSVP) and three new dynamic provisioning algorithms (DTVM, DAVM, and DRVM) designed to collectively and concurrently support VMs with significantly different and varying resource needs. I began by exploring the characteristics of more "dynamic" applications that could be run in clouds (e.g. those with longer-lived vs shorter-lived changes in resource requirements, frequent vs occasional changes, stable vs unstable types of changes, VMs of permanent or temporary and regular or periodic clients, etc.) that will affect cloud provisioning, and considered these factors in developing my algorithms. I also made extensions to the Eucalyptus open source cloud system to support my hybrid provisioning techniques and I implemented my hybrid provisioning algorithms in CloudSim and ran a set of experiments to evaluate the effectiveness of my algorithms

and thus of my approach generally. Based on the experimental results, it appears that my strategy is effective (i.e. my algorithms can induce the necessary provisioning behaviour to offer enhanced speed of execution for a broader set of applications run in VMs than is currently common). I now describe the specific contributions of my thesis in somewhat greater detail.

I have developed and presented a new hybrid static provisioning algorithm, HBF (Hybrid BackFill), supporting both online and batch provisioning in clouds. The algorithm provides responsiveness when it is needed and superior packing (resulting in decreased infrastructure need and operational costs) when it is not. The algorithm also takes steps to optimize placement of VMs particularly during online operation and aggressively back-fills into existing physical machines resulting in fewer machines being needed and a greater percentage of in-use machines being "fully" utilized. Additionally, I have developed an optimized multi-criteria packing technique which outperforms related heuristics (by eliminating a class of easily avoidable sub-optimal packings) and which has been integrated into my hybrid provisioning algorithm. My approach is particularly well suited to running utility computing based work in cloud environments since it better support shorter-duration and on-demand VM requests.

I also developed and simulated single and multiple criteria versions of my Compatibility-based Static VM Placement (CSVP) algorithm. Recognizing the significant negative impact of VM interference in clouds, the goal of CSVP was to see if the use of a **very** simple metric for judging compatibility could provide benefit when used as a basis for static intelligent co-placement. Based on my simulation results, using the Google Trace data, the answer is a definitive yes. Significant improvements over FFD (our

baseline for comparison) were seen in the simulations for the single criterion case. Further, the simulations also show that the multi-criteria version of the algorithm is capable of providing optimized placement considering multiple resources without any negative performance impacts on potentially problematic VMs (e.g. those which are resource critical in more than one way).

Of greatest significance, I have implemented a family of three related dynamic provisioning algorithms: DTVM (**D**istributed **T**ime shared **V**M **M**ultiplexing) [75], DAVM (**D**TVM-**A**ware **V**M re-**M**ultiplexing) and DRVM (**D**ynamic **R**e-packing of **VM**s). DTVM is designed to provide safe, controlled sharing of resources between VMs for short time periods within a single physical machine. This underpins the other algorithms by providing a means for compatible VMs that are co-located to share their resources safely and efficiently. Based on my assessment of DTVM and its use with DAVM and DRVM, it is clear that my DTVM strategy is effective (i.e. the algorithm can support the necessary sharing behaviour and this will result in enhanced speed of execution for VMs while handling short-term load variation). When compatible VMs with varying load demands are co-located, then we can use the DTVM algorithm to temporarily "re-provision" available resources in the affected physical machine(s) on demand to meet VM needs without the expense of undertaking VM migration. Further, since the effects of DTVM are local, controlled, and short-lived, there will be no inadvertent starvation of donor VMs introduced. If multiple VMs on a single machine **all** experience load increases concurrently, DTVM will have no effect and the problem will have to be dealt with through more costly live migration of one or more VMs.

Workload changes cannot always be handled using DTVM. For example, when the additional resource requirements of a VM cannot be fulfilled by borrowing from other VMs on the current host machine. This happens because the co-located VMs are incompatible. To address this problem I have introduced DAVM for clouds with the goal of better supporting varying VM resource requirements by grouping compatible VMs and co-locating (i.e. re-multiplexing) them. Such VM re-multiplexing can be achieved by migrating overloaded VM(s) to a suitable host where the additional resource requirements can be satisfied by borrowing from one or more available, compatible VMs using DTVM. DAVM is run periodically to potentially re-multiplex selected VM(s) based on predictions of future behaviour done using simple time-series analysis. In this case, currently co-hosted VMs are considered for migration with a compatible partner VM if there is more benefit to doing so then leaving them where they are. By predicting the behaviour of applications, via monitoring of prior behaviour (and using time series models), it can be determined when increases in resource demand are likely to occur (and cause SLA violations) and then the DAVM algorithm can be used to do compatibility-based re-provisioning of a VM requiring additional resource with a VM underutilizing its assigned resources. In a large scale cloud with many VMs, the DAVM approach should be generally applicable for a range of application types and my simulation results show clear improvements in VM performance when using DAVM if compatible VMs exist to be grouped.

I have also introduced DRVM (Dynamic Re-packing of Virtual Machines) for cloud environments with the goal of better supporting both varying VM workloads and ensuring **ongoing** packing efficiency despite reductions in packing efficiency arising

from DAVM migrations and/or completion of short-duration VMs. Through periodic checking of physical machines, DRVM can determine the underutilized hosts and then re-pack the VMs they contain to improve overall PM utilization and decrease the number of PMs that need to be powered on. DRVM considers the "compatibility of VMs" in its re-packing of the VMs of under-utilized hosts into fewer suitable physical machine(s) thereby further enhancing (over DAVM alone) the pool of compatible VMs on each PM that DTVM can exploit. It does this without loosing significant packing efficiency.

Future Work

In the future, my work could certainly be extended with further simulations to better characterize the conditions under which my approach is most effective. This would naturally include extending my assessments to consider a wider range of algorithm parameter values (i.e. thresholds and intervals). It would also be desirable (though perhaps difficult) to do some large-scale assessments in a commercial cloud system, particularly for new and developing cloud applications such as cloud-based gaming.

I would also like to do another set of simulations to assess the benefit of using DTVM, DAVM and DRVM in terms of the decrease in number and severity of SLA violations and/or the required resource "headroom" needed in each host to avoid SLA violations.

When deciding on the prediction methods to use in my dynamic provisioning algorithms, I was focused on seeing how good I could make prediction without introducing

expensive mechanisms that could preclude the ability to do predictions at a suitably frequent interval. I ran tests with various time-series based predictors using sample workloads from the Google traces to assess their effectiveness. I found that quite effective short-term predictions could be made for most example VMs using the relatively inexpensive Holt-Winters technique. There were, however, a small number of VMs for which predictions were sometimes inaccurate. I did some preliminary explorations of the use of changepoint analysis [52] to try to detect when my predictions might be poor so I could avoid those VMs. (The situations with poor results seemed to be tied to sudden changes in behaviour and my hope was to be able to "see those coming".) My results were not ideal so this work was not included in the thesis. It would still be good though to find a mechanism to anticipate poor predictive performance and either omit the affected VMs (leaving them in place) or ideally, select an alternative but still low cost approach for making prediction.

Recognizing that even a relatively low cost prediction technique will become problematic as the scale (number of of VMs) of a cloud system increases, it would be interesting to look for opportunities to decrease the number of VM's for which we must predict future behaviour. A number of strategies are possible. I already have code that identifies VMs whose load requirements are a stable and could choose to lessen the frequency at which those VMs are included in the pool of VMs that have predictions made prior to DAVM pairing. (i.e. the prior predictions for stable VMs would be reused when it appears safe to do so). It would also be possible to consider the stability of previously created VM pairs and avoid re-prediction (and possibly migration) of the pairs that are anticipated to be stable. This might be based on a

separate predictor running at a longer time scale. Pairs confirmed by the larger time scale predictor would be ignored by the normal DAVM process, There might also be practical benefit to sub-dividing the set of VMs for which predictions are done into groups in some way and then process each group separately at different times. This would have the benefit of distributing the cost of prediction over time (thus requiring fewer cloud resources for this administrative function) but would have to be done carefully to avoid loss of potential pairing opportunities since VMs in different groups would not be considered for creating possible pairs.

My current VM pairing and re-packing algorithms both consider pairs of two compatible VMs only. (This is sensible since the overhead of considering larger groupings increases rapidly with the group size (consider $\binom{n}{2}$ vs. $\binom{n}{3}$ for large n). A created VM pair might also be paired (as a unit) with another VM to create a "triple" and so on. I need to add support for pairing pairs with other VMs, pairs or groups of pairs to my simulations to verify that the results are as effective as increasing the size of groupings made by DAVM in the first place. Over aggressive pairing by DAVM could also be detrimental, incurring overhead for unnecessary migrations and possibly also resulting in additional PM under utilization. I need to do additional simulations to understand when/if this may occur and how to deal with it.

VM re-packing (i.e. DRVM) also currently only considers a fixed prediction period that could be made dynamic depending on some characteristics of the VMs (e.g. some high level prediction of VM's workload change frequency, etc.). It is important not to undertake a high-cost re-packing before it is necessary to do so. Triggering checking for the need to re-pack based on VM activities could decrease unnecessary overhead

in the existing algorithm which currently checks on a regular basis.

VM compatibility could also be broadened to include additional characteristics (e.g. completion time, etc.) which could result in better pairings/groupings. It would certainly seem preferable, if possible, to group VMs that should be compatible **and** run for about the same period of time. Additionally, I plan to assess the potential benefits of a simple user-based characterization of VMs into "performance conscious" and "cost conscious" groups allowing the pairing of cost and performance-conscious VMs so that cost conscious VMs can serve as resource donors to performance conscious ones. Again, the setting of the considered thresholds could be be made dynamic based on sampled system behaviour to adapt to changing global conditions in the cloud system and thereby improve the overall performance benefit of the migrations selected.

Also, considering the static algorithms that I developed, combining HBF and CSVP should be possible and useful. This would provide a single algorithm that could be used to "bootstrap" the entire provisioning system that offered the benefits of both algorithms as described in this thesis. Finally, it would be interesting to explore the potential for addressing VM interference arising due to the sharing of micro-architectural components such as last-level caches in multi-core processors. A challenge in doing this will be to measure and assess the potential use of such components by VMs so that compatibility between VMs can be judged.

Bibliography

[1] Eucalyptus homepage. http://www.eucalyptus.com.

[2] C. Amza, A. Chanda, A.L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*, pages 3 – 13, 2002.

[3] Jon Anton and Natalie Petouhoff. *Customer relationship management*. Prentice Hall, 1996.

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

[5] A. Ayad and U. Dippel. Agent-based monitoring of virtual machines. In *Information Technology (ITSim), 2010 International Symposium in*, volume 1, pages 1–6, 2010.

[6] Arshdeep Bahga and Vijay Krishna Madisetti. Synthetic workload generation

for cloud computing applications. *Journal of Software Engineering and Applications*, 4:396 – 410, 2011.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[8] Jozef Barunik. Econometrics. http://staff.utia.cas.cz/barunik/econometrics.htm.

[9] George A Beller. Tests that may be overused or misused in cardiology: The choosing wisely campaign. *Journal of Nuclear Cardiology*, pages 1–3, 2012.

[10] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE, 2007.

[11] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehorster, and A. Brinkmann. Non-intrusive virtualization management using libvirt. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 574 –579, 2010.

[12] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehorster, and Andre Brinkmann. Non-intrusive virtualization management using libvirt. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 574–579. European Design and Automation Association, 2010.

[13] Peter J Brockwell and Richard A Davis. *Introduction to time series and forecasting*. Springer Science & Business Media, 2006.

[14] Ed Burnette. *Eclipse IDE Pocket Guide*. O'Reilly Media, 2009.

[15] Randy Butler, Von Welch, Douglas Engert, Ian Foster, Steven Tuecke, John Volmer, and Carl Kesselman. A national-scale authentication infrastructure. *Computer*, 33(12):60–66, 2000.

[16] Rodrigo Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. Workload prediction using arima model and its impact on cloud applications' qos. 2014.

[17] Rodrigo N. Calheiros, Rajiv Ranjan, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services. *CoRR*, abs/0903.2525, 2009.

[18] Rodrigo N. Calheiros, Rajiv Ranjan, César A. F. De Rose, and Rajkumar Buyya. *CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms*. Wiley Press, 2010.

[19] Ethan Cerami. *Web services essentials: distributed applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Incorporated, 2002.

[20] Ming Chen, Hui Zhang, Ya-Yunn Su, Xiaorui Wang, Guofei Jiang, and Kenji Yoshihira. Effective VM sizing in virtualized data centers. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 594–601. IEEE, 2011.

[21] Xinkai Chen. Adaptive sliding mode control for discrete-time multi-input multi-output systems. *Automatica*, 42(3):427 – 435, 2006.

[22] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 1.1, 2001.

[23] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. In *USENIX Annual Technical Conference, FREENIX Track*, pages 135–144, 2004.

[24] Christopher Clark, Keir Fraser, and H. Steven. Live migration of virtual machines. pages 1–11, 2005.

[25] Reuven Cohen. Unified cloud interface project. https://code.google.com/p/unifiedcloud/.

[26] Xen Community. The home of xen project. http://www.xen.org.

[27] Francisco Curbera, Frank Leymann, Tony Storey, Donald Ferguson, and Sanjiva Weerawarana. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR Englewood Cliffs, 2005.

[28] Amazon EC2. Amazon elastic cloud computing. http://aws.amazon.com/ec2/. Accessed on March 15, 2011.

[29] Vincent C. Emeakaroha, Marco A. S. Netto, Rodrigo N. Calheiros, Ivona Brandic, Rajkumar Buyya, and César A. F. De Rose. Towards autonomic

detection of SLA violations in cloud infrastructures. *Future Gener. Comput. Syst.*, 28(7):1017–1029, July 2012.

[30] Xiujie Feng, Jianxiong Tang, Xuan Luo, and Yaohui Jin. A performance study of live vm migration technologies: Vmotion vs xenmotion, 2011.

[31] Josep Oriol Fito, Iiigo Goiri Presa, and Jordi Guitart. SLA-driven elastic cloud hosting provider. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:111–118, 2010.

[32] Apache Software Foundation. Olio. http://incubator.apache.org/olio/. Accessed on April 25, 2011.

[33] Logic Software Foundation. Apache license. http://www.apache.org/licenses/LICENSE-2.0.html.

[34] Teofilo F Gonzalez. *Handbook of approximation algorithms and metaheuristics*. CRC Press, 2007.

[35] Mohammad Mehedi Hassan, Mohammod Shamim Hossain, Atif Alamri, Mohammad Anwar Hossain, Muhammad Al-Qurishi, Yousuf Aldukhayyil, and Dewan Tanvir Ahmed. A cloud-based serious games framework for obesity. In *Proceedings of the 1st ACM multimedia international workshop on Cloud-based multimedia applications and services for e-health*, pages 15–20. ACM, 2012.

[36] Gabor T Herman. *Fundamentals of computerized tomography: image reconstruction from projections*. Springer, 2009.

[37] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[38] AbiCloud Holdings. The Enterprise Open Source Cloud Computing. http://wiki.abiquo.com/ display/abiCloud/Home/.

[39] Chi-Chung Hui and Samuel T Chanson. Improved strategies for dynamic load balancing. *IEEE concurrency*, (3):58–67, 1999.

[40] Rob J Hyndman and Yeasmin Khandakar. Automatic time series for forecasting: the forecast package for r. Technical report, Monash University, Department of Econometrics and Business Statistics, 2007.

[41] Rob J Hyndman and Anne B Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, 22(4):679–688, 2006.

[42] Advanced Micro Devices Inc. AMD IO-virtualization Technology IOMMU. http://support.amd.com/.

[43] CloudKick Inc. Rackspace cloud monitoring. https://www.cloudkick.com/home.

[44] Eucalyptus Systems Inc. Learn about cloud computing. http://www.eucalyptus.com. Accessed on April 25, 2011.

[45] Logic Monitor Inc. Logic monitor. http://www.logicmonitor.com/monitoring/storage/netappLfilers/.

[46] Monitis Inc. Monitis monitoring services. http://portal.monitis.com/.

[47] VMware Infrastructure. Resource management with VMware DRS. *VMware Whitepaper*, 2006.

[48] Qiao Jingzhou. On the construction and realization of ubiquitous library based on the cloud computation [j]. *New Century Library*, 6:026, 2010.

[49] Steffen Kächele, Jörg Domaschka, and Franz J Hauck. Cosca: an easy-to-use component-based paas cloud system for common applications. In *Proceedings of the First International Workshop on Cloud Computing Platforms*, page 4. ACM, 2011.

[50] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1287–1294. IEEE, 2012.

[51] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 373–381, 2006.

[52] Rebecca Killick and Idris Eckley. changepoint: An r package for changepoint analysis. *Journal of Statistical Software*, 58(3):1–19, 2014.

[53] Hongjae Kim, Munyoung Kang, Sanggil Kang, and Sangyoon Oh. A Novel Adaptive Virtual Machine Deployment Algorithm for Cloud Computing. In

2012 International Conference on Information Science and Industrial Applications (ISI 2012), pages 264–269, 2012.

[54] Hyukho Kim, Woongsup Kim, and Yangwoo Kim. Predictable cloud provisioning using analysis of user resource usage patterns in virtualized environment. In Tai-hoon Kim, Stephen S. Yau, Osvaldo Gervasi, Byeong-Ho Kang, Adrian Stoica, and Dominik, editors, *Grid and Distributed Computing, Control and Automation*, volume 121 of *Communications in Computer and Information Science*, pages 84–94. Springer Berlin Heidelberg, 2010.

[55] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[56] Bernhard Korte and Jens Vygen. Combinatorial optimization: theory and applications, 2006.

[57] Alex Mu-Hsing Kuo. Opportunities and challenges of cloud computing to improve health care services. *Journal of medical Internet research*, 13(3), 2011.

[58] Emil Lupu, Naranker Dulay, Morris Sloman, J Sventek, Steven Heeps, Stephen Strowes, K Twidle, S-L Keoh, and A Schaeffer-Filho. Amuse: autonomic management of ubiquitous e-health systems. *Concurrency and Computation: Practice and Experience*, 20(3):277–295, 2008.

[59] Spyros Makridakis, A Andersen, Robert Carbone, Robert Fildes, Michele Hibon, Rudolf Lewandowski, Joseph Newton, Emanuel Parzen, and Robert Win-

kler. The accuracy of extrapolation (time series) methods: Results of a forecasting competition. *Journal of forecasting*, 1(2):111–153, 1982.

[60] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. Efficient resource provisioning in compute clouds via VM multiplexing. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 11–20, New York, NY, USA, 2010. ACM.

[61] Ismael Solis Moreno, Peter Garraghan, Paul Townend, and Jie Xu. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *Cloud Computing, IEEE Transactions on*, 2(2):208–221, 2014.

[62] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. Autonomic virtual resource management for service hosting platforms. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[63] Nimbus. Open-source toolkit on providing IaaS. http://nimbusproject.org.

[64] Nimbus. Open-source toolkit on providing IaaS. http://nimbusproject.org.

[65] Robert A Novelline and Lucy Frank Squire. *Squire's fundamentals of radiology*. La Editorial, UPR, 2004.

[66] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM Inter-*

national Symposium on Cluster Computing and the Grid, CCGRID '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

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

[68] Sergio Pacheco-Sanchez, Giuliano Casale, Bryan Scotney, Sally McClean, Gerard Parr, and Stephen Dawson. Markovian workload characterization for qos prediction in the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 147–154. IEEE, 2011.

[69] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM.

[70] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for Vector Bin Packing. Microsoft Research.

[71] Cesare Pautasso. RESTful web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866, 2009.

[72] Ken Pepple. *Deploying OpenStack*. O'Reilly Media, 2011.

[73] Paolo Pialorsi and Marco Russo. *Introducing microsoft LINQ*. Microsoft Press, 2007.

[74] Google project hosting. Traces for google workloads. https://code.google.com/p/googleclusterdata/.

[75] Md Mahfuzur Rahman, Ruppa Thulasiram, and Peter Graham. Differential time-shared virtual machine multiplexing for handling QoS variation in clouds. In *Proceedings of the 1st ACM multimedia international workshop on Cloud-based multimedia applications and services for e-health*, pages 3–8. ACM, 2012.

[76] Linnie Rawlinson and Nick Hunt. Jackson dies, almost takes internet with him. http://www.cnn.com/2009/TECH/06/26/michael.jackson.internet/.

[77] Tejaswi Redkar and Tony Guidici. *Windows Azure Platform*. Apress, 2011.

[78] Gesine Reinert. Time series. http://www.stats.ox.ac.uk/ reinert/.

[79] Cheze-Le Rest et al. Integrated telemedicine applications and services for oncological positron emission tomography. *Oncology reports*, 15(4):1091–1100, 2006.

[80] B.P. Rimal, Eunmi Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51, aug. 2009.

[81] Jerry Rolia, Ludmila Cherkasova, Martin Arlitt, and Artur Andrzejak. A capacity management service for resource pools. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, pages 229–237, New York, NY, USA, 2005. ACM.

[82] Luigi Romano, Danilo De Mari, Zbigniew Jerzak, and Christof Fetzer. A novel approach to qos monitoring in the cloud. In *Data Compression, Communications and Processing (CCP), 2011 First International Conference on*, pages 45–51. IEEE, 2011.

[83] Amazon.com S3. Amazon simple storage service. http://docs.amazonwebservices.com/AmazonS3/latest/dev/. Accessed on March 17, 2011.

[84] Prasad Saripalli, GVR Kiran, R Ravi Shankar, Harish Narware, and Nitin Bindal. Load prediction and hot spot detection models for autonomic cloud computing. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 397–402. IEEE, 2011.

[85] Amazon Web Services. Amazon cloudwatch. http://aws.amazon.com/cloudwatch/.

[86] Charles Severance. *Using Google App Engine*. O'Reilly Media, 2009.

[87] Yuxiang Shi, Xiaohong Jiang, and Kejiang Ye. An energy-efficient scheme for cloud resource provisioning based on cloudsim. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 595 –599, sept. 2011.

[88] W.C. Shiel. Magnetic Resonance Imaging (MRI scan). MedicineNet.com.

[89] Shekhar Srikantaiah. Energy Aware Consolidation for Cloud Computing. In *USENIX Workshop on Power Aware Computing and Systems*, 2008.

[90] M.D. Timothy Johnson. Arteriograms, venograms are angiogram territory. http://articles.chicagotribune.com/.

[91] Eric J Topol and Robert M Califf. *Textbook of cardiovascular medicine*, volume 355. Lippincott Williams & Wilkins, 2007.

[92] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[93] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 28–28. USENIX Association, 2009.

[94] Michail Vlachos, S Yu Philip, and Vittorio Castelli. On periodicity detection and structural periodic similarity. In *SDM*, volume 5, pages 449–460. SIAM, 2005.

[95] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 254–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[96] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[97] Xiaozhe Wang, Kate Smith, and Rob Hyndman. Characteristic-based clustering for time series data. *Data mining and knowledge Discovery*, 13(3):335–364, 2006.

[98] Craig D Weissman and Steve Bobrowski. The design of the force.com multi-tenant internet application development platform. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 889–896. ACM, 2009.

[99] Bo Yin, Ying Wang, Luoming Meng, and Xuesong Qiu. A Multi-dimensional Resource Allocation Algorithm in Cloud Computing. *Information and Computational Science*, 11(9):3021–3028, 2012.

[100] Alexander Zahariev. Google app engine. *Helsinki University of Technology*, 2009.

[101] Shuai Zhang, Shufen Zhang, Xuebin Chen, and Xiuzhen Huo. Cloud computing research and development trend. In *Future Networks, 2010. ICFN '10. Second International Conference on*, pages 93 –97, jan. 2010.

[102] Wei Zhang, Hangwei Qian, Craig E Wills, and Michael Rabinovich. Agile resource management in a virtualized data center. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 129–140. ACM, 2010.

[103] Xiaodong Zhang, Yanxia Qu, and Li Xiao. Improving distributed workload performance by sharing both cpu and memory resources. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 233–241. IEEE, 2000.

[104] Andris A Zoltners, Prabhakant Sinha, and Sally E Lorimer. *The complete guide*

to sales force incentive compensation: how to design and implement plans that work. AMACOM Div American Mgmt Assn, 2006.

[105] Michael Zur Muehlen, Jeffrey V Nickerson, and Keith D Swenson. Developing web services choreography standards - the case of REST vs. SOAP. *Decision Support Systems*, 40(1):9–29, 2005.