DISSERTATION

AUTONOMOUS MANAGEMENT OF COST, PERFORMANCE, AND RESOURCE UNCERTAINTY FOR MIGRATION OF APPLICATIONS TO INFRASTRUCTURE-AS-A-SERVICE (IAAS) CLOUDS

Submitted by

Wes J. Lloyd

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2014

ABSTRACT

AUTONOMOUS MANAGEMENT OF COST, PERFORMANCE, AND RESOURCE UNCERTAINTY FOR MIGRATION OF APPLICATIONS TO INFRASTRUCTURE-AS-A-SERVICE (IAAS) CLOUDS

Infrastructure-as-a-Service (IaaS) clouds abstract physical hardware to provide computing resources on demand as a software service. This abstraction leads to the simplistic view that computing resources are homogeneous and infinite scaling potential exists to easily resolve all performance challenges.

Adoption of cloud computing, in practice however, presents many resource management challenges forcing practitioners to balance cost and performance tradeoffs to successfully migrate applications. These challenges can be broken down into three primary concerns that involve determining *what*, *where*, and *when* infrastructure should be provisioned. In this dissertation we address these challenges including: (1) performance variance from resource heterogeneity, virtualization overhead, and the plethora of vaguely defined resource types; (2) virtual machine (VM) placement, component composition, service isolation, provisioning variation, and resource contention for multi-tenancy; and (3) dynamic scaling and resource elasticity to alleviate performance bottlenecks. These resource management challenges are addressed through the development and evaluation of autonomous algorithms and methodologies that result in demonstrably better performance and lower monetary costs for application deployments to both public and private IaaS clouds.

This dissertation makes three primary contributions to advance cloud infrastructure management for application hosting. First, it includes design of resource utilization models based on step-wise multiple linear regression and artificial neural networks that support prediction of better performing component compositions. The total number of possible compositions is governed by Bell's Number that results in a combinatorially explosive search space. Second, it includes algorithms to improve VM placements to mitigate resource heterogeneity and contention using a load-aware VM placement scheduler, and autonomous detection of under-performing VMs to spur replacement. Third, it describes a workload cost prediction methodology that harnesses regression models and heuristics to support determination of infrastructure alternatives that reduce hosting costs. Our methodology achieves infrastructure predictions with an average mean absolute error of only 0.3125 VMs for multiple workloads.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Dr. Shrideep Pallickara, for his indispensable guidance and invaluable support and advice in conducting the research described herein. I've been *very* lucky to have found a great mentor to work with throughout the execution of this research and the compilation of its results. I must also equally express gratitude to my colleague, Dr. Olaf David, from the Department of Civil Engineering. I've been lucky to work with Olaf's modeling/software lab at the USDA supported by various grants and cooperative agreements. I strongly appreciate his support for helping me identify and execute this dissertation research in the context of the OMS/CSIP workgroup at the USDA. I must also equally express gratitude to Ken Rojas. Ken a former project manager, now acting director of the USDA-NRCS Information Technology Center in Fort Collins has also been very generous in his support of this research. I must also equally express gratitude to Dr. Mazdak Arabi. Mazdak was very generous in offering the use of his private "Erams" cluster for supporting experiments in this dissertation. I would also like to acknowledge two M.S. students working with Dr. Arabi who provided the CSIP CFA and SWAT-DEG modeling workloads as part of their research: Tyler Wible, and Jeff Ditty. Additionally I must thank a number of my colleagues at the USDA who encouraged my work over the years including: Dr. James Ascough II, Dr. Tim Green, Jack Carlson, George Leavesley, and Frank Geter.

I would also like to acknowledge the support of Dr. Bieman for his long term encouragement and support for my graduate research. Within the computer science department at Colorado State University I would also like to thank Dr. Daniel Massey for his service and contributions on my committee. I would also like to thank Dr. Sudipto Ghosh, Dr. Robert France, and Dr. Adele Howe for encouragement and support of my graduate studies.

I would like to thank the employees and founders of the Eucalyptus. Much of the work in chapters 3, 4, 5, 6, and 7 was performed using local Eucalyptus private clouds. Their support over the years helped our CSU/USDA team implement and sustain cloud systems in support of CSIP and the research described in this dissertation.

Finally, I would like to thank all of my family, friends, and mentors who provided emotional support and fellowship over the years as a student at Colorado State University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Cloud computing strives to provide computing as a utility to end users. Three service levels delineate cloud computing: software-as-a-service (SaaS), platform-as-a-service (PaaS) and infrastructure-as-a-service (IaaS). Each offers an increasing level of infrastructure control with less infrastructure abstraction to the end user. S***oftware-as-a-service (SaaS)*** hosts computational services such as computational libraries, modeling engines, and/or application programming interfaces (APIs) making them more easily accessible to end users. ***Platform-as-a-service (PaaS)*** provides a hosting framework that allows developers freedom to design and deploy services so long as they adhere to specific platform(s). PaaS provides specific relational databases, application servers, and vendor/platform specific programming APIs while abstracting, hosting, of the underlying infrastructure. Developers are freed from the burden of infrastructure management enabling them to focus on the design and development of application middleware, which can be deployed to PaaS containers. PaaS cloud providers then optimize hosting and scaling of these containers to minimize cost and optimize performance. ***Infrastructure-as-a-service (IaaS)*** provides maximum freedom to developers enabling control of the middleware design, as well as the underlying application infrastructure stack. Developers can freely choose databases, application servers, cache/logging servers as needed. IaaS enables diverse application stacks to be supported through the virtualization of various operating systems using hypervisors such as Xen, KVM, or VMWare ESXi or operating system containers such as OpenVZ, LXC, or Docker [1]–[4].

IaaS provides many benefits including easier application migration to clouds as existing applications can often be deployed as-is without extensive refactoring or new development which may be required when deploying applications to PaaS clouds that provide vendor specific infrastructure. Legacy infrastructure can often be run under IaaS minimizing the need to rearchitect systems enabling a faster approach towards cloud migration. Avoiding lock-in to vendor specific application infrastructures and APIs can improve software maintainability throughout the application's life-cycle as vendor specific APIs may incur special costs and have limited support if abandoned due to business reasons. IaaS clouds enable application specific granular scaling as individual application components can be scaled as needed to meet demand. The number of public IaaS cloud offerings has grown extensively of late with a recent cloud evaluation website currently identifying 57 distinct providers, but only 14 PaaS providers [5].

Given the advantages, IaaS clouds have become very attractive for hosting applications with service oriented architectures. In this dissertation we refer to applications with service oriented architectures as service oriented applications (SOAs). Deploying to IaaS clouds involves deployment of each SOA's application stack across virtual machines (VMs). Application stacks consist of the unique set of components that constitute an application's infrastructure including: web server(s), proxy server(s), database(s), file server(s), distributed cache(s) and other server(s)/services. But how should SOAs be rearchitected to take advantage of the unique characteristics of IaaS? How can the costs of application hosting be minimized while maximizing application performance? Research to investigate these questions forms the basis of this dissertation.

1.1. KEY RESEARCH CHALLENGES

SOA deployment to IaaS clouds incurs many resource management challenges which can be broken down into three primary concerns: (1) Determining WHEN infrastructure should be provisioned, (2) Determining WHAT infrastructure should be provisioned, and (3) Determining WHERE infrastructure should be provisioned. Management challenges vary for practitioners depending on if they are deploying their application to a private or public IaaS cloud. In private cloud settings, practitioners and system administrators have some ability to influence resource management leading to improved application deployments. In public cloud settings, practitioners only have limited ability to influence the management of physical infrastructure. For public cloud application deployments, our research efforts focus on introspection of infrastructure management to improve awareness in helping identify scenarios that produce unwanted resource contention and application performance degradation.

WHEN server infrastructure should be provisioned to address service demand is informed by ***hotspot detection*** [6], [7]. Determining when to scale-up resources is complicated by the ***launch latency*** of virtual machines (VM). In some cases, the time required to provision and launch new infrastructure exceeds the duration of demand spikes [7]. Analysis of historical service usage trends can support future ***load prediction*** to anticipate demand to enable pre-provisioning server infrastructure. Load prediction can be difficult particularly for applications with stochastic load behavior. Care must be exercised as poor prediction can result in overprovisioning and higher hosting costs, or underprovisioning and poor performance. ***Prelaunching VMs*** in anticipation of future service demand can help mitigate launch latency and support service availability. Additional VMs provisioned to address service demand spikes can be preserved in resource pools for future use when service demand drops.

WHAT server infrastructure should be provisioned concerns the size and type (***vertical scaling***) and quantity (***horizontal scaling***) of VM allocations. Vertical scaling involves modifying resource allocations of existing VMs. Changing VM resource allocations including

CPU core, memory, disk, and network bandwidth may alleviate poor performance when possible. When vertical scaling is unavailable or insufficient to address service demand, horizontal scaling can be used. Additional service capacity is provisioned by launching new VMs and the service workload is balanced across the expanded pool of VMs. A key challenge lies in determining how many VMs should be provisioned, and with what resource allocations?

Vertical scaling is frequently unavailable in public clouds because to achieve economies of scale vendors fix VM resource allocations to provide a limited number of ***virtual machine types***. Focusing efforts on providing a limited set of resources helps vendors optimize hardware deployments and resource allocations for customer requests. For example in the spring of 2014 Amazon Elastic Compute Cloud (EC2) provided 34 fixed VM types [8], while Hewlett Packard's (HP) Helion Cloud provided 11. Quantifying the performance expectation of cloud resource is difficult. Public cloud vendors typically provide only vague qualitative descriptions of VM capabilities. These ***qualitative resource descriptions*** can be considered as ordinal scale measures [9]. Ordinal scale measures provide an empirical relation system which preserves the ordering of classes with respect to each other. Ordinal measures provide ranking only. Comparisons involving calculation of differences or ratios involving ordinal values are not valid. Amazon EC2 describes VM performance using elastic compute units (ECUs), where one ECU (1.0 ECU) is stated to provide the equivalent CPU capacity of a 1.0 – 1.2 GHz 2007 AMD Opteron or Intel Xeon processor. HP Cloud Compute Units are advertised to be roughly equivalent to the minimum power of 2/13th of one logical CPU core of an Intel 2.6 GHz 2012 Xeon CPU. Amazon employs approximate categories to describe network throughput of VM types. Categories include: very low, low (250 Mbps), moderate (500 Mbps), high (1000 Mbps), and 10 Gigabit. Attempts to calculate resource differences fail because these descriptions are at best ordinal measures expressing only relative approximations of resource capabilities. The lack of quantitative resource descriptions makes it exceedingly difficult for practitioners to interpret how their SOAs will run in the cloud.

Hardware heterogeneity in private cloud settings is common when system administrators lack the resources to procure significant amounts of identical hardware infrastructure. Heterogeneous hardware leads to performance variation when application deployments are deployed to heterogeneous servers. The problem of heterogeneous hardware has been shown to pervade in public cloud settings as well [10], [11]. Prior work has demonstrated that public cloud VM types can be implemented using different backing hardware. In 2011 Ou et al. identified no less than 5 different hardware implementations of the Amazon EC2 m1.large VM. Further, these "homogeneous" m1.large implementations led to application performance variance up to 28%. We have replicated their results by demonstrating up to 14% performance variance for an erosion modeling service application on heterogeneous implementation variants of Amazon's m2.xlarge VM.

Virtualization enables the resources of physical hardware to be partitioned for use by VMs. Memory is physically reserved and not shared, while CPU time, Disk I/O and network I/O are multiplexed and shared by all VMs running on a PM. The VM hypervisor either fully simulates physical devices using software, a practice known as full virtualization, or virtual device requests are passed directly to physical devices using para-virtualization. Full and para-virtualization of disk and network devices both incur overhead because underlying devices are shared amongst multiple guests. WHAT resources are provisioned also involves the choice of ***virtualization hypervisor*** to provide cloud-based VMs in private cloud settings, and the awareness of the vendor's hypervisor choice in public cloud settings. Different hypervisors have been shown to exhibit different degrees of ***virtualization overhead*** depending on the workloads being virtualized [3], [12], [13]. Our research has generally found that CPU bound workloads can perform better using KVM, while I/O bound workloads benefit from Xen. HP's Helion cloud uses kernel-based-virtual machines (KVM) while Amazon EC2 uses the Xen hypervisor. To our knowledge vendors do not mix hypervisor types when providing VMs of the same types.

WHERE server resources are provisioned- and the decision making processes involved- are abstracted by public IaaS clouds. Representing VMs as tuples and using them to pack physical machines (PMs) can be thought of as an example of the multidimensional bin-packing problem that has been shown to be NP-hard [14]. Consequently in practice simplified heuristic based approaches to ***VM placement*** are typically used.

SOAs consist of unique sets of components including: web server(s), proxy server(s), application server(s), relational and/or NoSQL database(s), file server(s), distributed caches, log services and others. These components comprise the ***application stack***. Application deployment requires components to be deployed and scaled as service demand requires. Components are distributed to virtualization containers using a series of machine images to instantiate virtual machines (VMs), a concept known as ***component composition***. The number of images and the composition of components vary. Ideal SOA compositions exhibit very good performance using a minimum number of images/VMs. Component aggregations typically deliver superior application performance when resource contention is avoided. ***Service isolation*** involves hosting components of the application stack separately so they execute using separate virtual machine (VM) or operating system container instances. Isolation provides components explicit sandboxes, not shared by other systems. Service isolation supports easy resource ***elasticity*** as the quantity, location, and number of VM deployments for particular application components can scale dynamically to meet varying system loads, improving agility to add and remove hardware resources to address service demand. A lighter weight alternative to using full VMs is to segment the host operating system using operating system containers to provide isolated sandboxes that simulate separate physical computers.

Using brute force performance testing to determine optimal component compositions is only feasible for applications with small numbers of components. If considering an application as a set of (n) components, then the total number of possible component compositions is Bell's number (k).

Bell's number is the number of partitions of a set (k) consisting of (n) members [15]. An exponential generating function to generate Bell numbers is given by the formula:

$$\sum_{n=0}^{\infty} \frac{B_n}{n!} x^n = e^{e^x - 1}.$$

Table 1.1 shows the first few bell numbers describing the possible component compositions for an application with (n) components. Beyond four components the number of compositions grows large demonstrating that brute force testing to identify optimal compositions becomes an unwieldy, arduous task. Further complicating testing, public IaaS clouds typically do not provide the ability to introspect or control VM placements making it difficult, if not impossible, to even infer where components have been deployed across physical hardware.

WHERE VMs are provisioned in a public cloud is not only uncontrollable, but difficult to discern as well [16]. End user determination of VM location and co-location remains an open challenge. Previous efforts using heuristics to infer VM co-residency and launching probe VMs for exploration are both expensive and only partially effective at determining VM locations [17]. Resource contention from VM multi-tenancy has been shown to degrade performance and is of concern for SOA application hosting [16], [18]. Public clouds often pack VMs onto as few physical hosts as possible to reduce hardware idle time and save energy [19]. Forcing multi-tenancy in this way to save energy often leads to a tradeoff in performance.

Table 1.1. Number of SOA Component Compositions

Number of components (n)	Number of compositions (k)
3	5
4	15
5	52
6	203
7	877
8	4140

Provisioning variation, the variability of where application VMs are deployed across physical hosts of a cloud, results in performance variation and degradation [16], [18], [20]. Unwanted ***multi-tenancy*** and interference occurs when multiple VMs that intensively consume the same resource (CPU, Disk I/O, and Network I/O) reside on the same physical host computer leading to resource contention and performance degradation. Given an application with 4 components and 15 possible component compositions, VM provisioning variation yields 46 variations on how the 15 compositions can be deployed across physical hosts. Component composition and VM provisioning variation result in an explosion of the search space, making brute force testing to quantify performance implications of provisioning variation an arduous task.

Key resource management challenges for SOA deployment to IaaS clouds broken down by problems concerning WHEN, WHAT, and WHERE to provision infrastructure are summarized in table 1.2.

Table 1.2. IaaS Cloud Resource Management Challenges

WHEN to provision	WHAT to provision	WHERE to provision
Hot spot detection	Vertical scaling	VM placement
Launch latency	Horizontal scaling	Component composition
Load prediction	Virtual machine types	Service isolation
Prelaunching VMs	Hardware heterogeneity	Provisioning variation
	Virtualization	Multi-tenancy
	Virtualization hypervisor	
	Virtualization overhead	
	Qualitative resource descriptions	

1.2. KEY RESEARCH QUESTIONS

This dissertation broadly investigates the following research questions:

DRQ-1: [Chapter 3] What factors must be accounted for when migrating and then scaling SOAs on IaaS clouds for high performance?

DRQ-2: [Chapter 4] Which resource utilization variables are the best independent variables for predicting application performance? Which modeling techniques are most effective?

DRQ-3: [Chapter 5] How does resource utilization and application performance vary relative to component composition across VMs? What is the magnitude of performance variance resulting from the use of different component compositions across VMs?

DRQ-4: [Chapter 7] What performance implications result from VM placement location when dynamically scaling cloud infrastructure for SOAs?

DRQ-5: [Chapter 7] How can we detect the presence of noisy neighbors, multi-tenant VMs that cause resource contention and what are the performance implications for SOA hosting?

DRQ-6: [Chapter 8] How effectively can we predict required infrastructure for SOA workload hosting by harnessing resource utilization models and Linux time accounting principles?

1.3. RESEARCH CONTRIBUTIONS

This dissertation contributes three primary contributions to advance IaaS cloud resource management for SOA hosting. These contributions include:

1. Resource utilization modeling to predict performance of SOA deployments on IaaS clouds
2. Resource management techniques to improve VM placement to reduce resource contention for SOA deployments on both public and private IaaS clouds
3. A workload cost prediction methodology which offers infrastructure alternatives to reduce SOA hosting costs on IaaS clouds

Detailed research contributions of this dissertation from the individual chapters include:

Chapter 3: An exploratory investigation on the implications of SOA migration to IaaS cloud is presented. Key results identified through the study include: (1) the requirement of application tuning to address distinct system bottlenecks when scaling up server infrastructure, (2) the importance of careful component composition to avoid resource contention, and (3) relationships between application profile characteristics (e.g. CPU bound vs. I/O bound) and virtualization overhead.

Chapter 4: Resource utilization models to predict performance of SOA deployments to IaaS clouds are proposed. The best independent variables are identified and modeling techniques are identified.

Chapter 5: An empirical investigation on the implications of SOA performance based on component composition across virtual machines. Characteristics of compositions that provide the best performance are identified. Overhead is quantified from deploying components in isolation using separate VMs on the same physical host. Performance implications of increasing VM memory allocations (vertical scaling) and the use of different hypervisors (KVM vs. XEN) are studied.

Chapter 6: The Virtual Machine Scaler (VM-Scaler), a REST/JSON based web services application which supports IaaS cloud infrastructure provisioning and management is described. VM-Scaler provides a platform for conducting IaaS cloud research by supporting experimentation with hotspot detection schemes, dynamic scaling approaches, VM management/placement, job scheduling/proxy services, SOA workload profiling, and SOA performance modeling.

Chapter 7: Multiple techniques are presented to reduce resource contention from multi-tenancy to improve SOA performance. These include: A load-aware VM placement/job scheduler, an empirical evaluation of performance implications of VM placement for dynamically scaling application infrastructure, evaluation of performance implications from VM type heterogeneity, and an approach to detect noisy neighbors in cloud settings using the *cpuSteal* CPU metric.

Chapter 8: This chapter presents a workload cost prediction methodology to predict hosting costs of SOA workloads harnessing resource utilization models. This methodology provides infrastructure alternatives that provide equivalent performance allowing the most economical infrastructure to be chosen for application hosting.

1.4. NON-GOALS

Autonomic service composition and resource provisioning for hosting SOAs using IaaS clouds is a compound problem that crosscuts many existing areas of distributed systems research. Related research problems which are NOT the primary focus of this work are described below. These research problems can be as considered related research and potential future work, but are generally considered as non-goals of this work.

1.4.1. Stochastic Applications

This research focuses on service composition and resource provisioning to support hosting non-stochastic service oriented applications which exhibit stable resource utilization characteristics. The primary focus is to support application deployment and infrastructure management for service-based applications which provide modeling or computational engines as a service to end users. Applications with non-deterministic stochastic behavior are not the focus of this work.

1.4.2. Simultaneous Deployment of Multiple Applications

Public IaaS clouds and private IaaS clouds hosting multiple applications may experience interference when these applications simultaneously share the same physical hosts. Interference from external applications, particularly stochastic applications, can cause unpredictable behavior. The research focuses on service composition and resource provisioning for one SOA at a time. Future work could investigate support for deploying multiple sets of non-stochastic applications simultaneously. We do investigate resource contention for VM multi-tenancy from our own application hosting in chapter 5, and from external cloud users extensively in chapter 7.

1.4.3. Fault Tolerance

This research does not focus exclusively on fault tolerance of the virtual infrastructure hosting SOAs. Fault tolerance is considered an autonomic resource provisioning system feature, but is not a primary focus of this research. Fault tolerance support and investigation of fault tolerance research questions related to autonomic resource provisioning systems is considered as future or related work.

1.4.4. Hot Spot Detection

This research does not focus specifically on development of novel hot spot detection algorithms. Hot spot detection scheme(s) are required for autonomic resource provisioning and appropriate methods are chosen as needed for investigations of dynamic scaling in Chapter 7.

1.4.5. Heuristic Based Approach for Component Composition

This research investigates the use of performance models to predict performance of SOA component compositions. Our approach relies on execution of training workloads to train performance models. We do not develop a heuristic based approach to guide component compositions which avoids training regression based performance models. This exercise is considered as future or related work.

1.5. ORGANIZATION

The remainder of this dissertation is organized as follows. Chapter 2 provides an overview of research gaps and related work. Chapter 3 provides an exploratory investigation on the migration of service oriented applications to IaaS clouds. Chapter 4 explores the development and use of resource utilization performance models to predict the performance of SOA component deployments across virtual machines. Our resource utilization based approach to performance modeling for SOAs deployed to IaaS clouds is harnessed later in this dissertation to tackle a myriad of resource management challenges in chapters 5, 7, and 8. Chapter 5 investigates performance implications of WHERE SOA components are deployed across VM and presents resource utilization based performance models which predict performance of component compositions. Chapter 6 introduces the Virtual Machine Scaler (VM-Scaler), a REST/JSON Java-based web services application to support cloud infrastructure management for SOA deployment. Chapter 7 investigates implications of WHERE VM placement occurs in both private and public clouds settings. Multiple management approaches are presented to improve SOA performance deployed to both public and private IaaS clouds. Chapter 8 harnesses resource utilization performance modeling to provide infrastructure alternatives to address WHAT infrastructure should be provisioned to balance both performance and cost tradeoffs. Chapter 9 provides overarching conclusions and Chapter 10 summarizes references cited in this dissertation.

CHAPTER 2

BACKGROUND

2.1. PREVIOUS WORK

Placement of application components across a series of VM images can be envisioned as a bin packing problem. The traditional bin packing problem states that each bin has a size (V), and items ($a_1,...,a_n$) are packed into the bins. For our problem there are a minimum of five bins describing dimensions of: CPU utilization, disk write throughput, disk read throughput, network traffic sent, and network traffic received. To treat component composition as a bin packing problem both the resource capacities of our bins (PMs) as well as the resource consumption of our items (components) must be quantified. Determining resource utilization and capacities is challenging particularly for stochastic applications and heterogeneous hardware where resource consumption and performance of resources varies.

Several approaches exist for autonomic provisioning and configuration of VMs for IaaS clouds. Xu et al. have identified two classes of approaches in [21]: *multivariate optimization* (performance modeling), and *feedback control*. Multi-variate optimization approaches have a specific optimization objective, typically improving performance, which is achieved by developing performance models which consider multiple system variables. Feedback control approaches based on process control theory attempt to improve configurations by iteratively making changes and observing outcomes. Formal approaches to autonomic resource provisioning attempted include: *integer linear programming* [22], [23], *knowledge/case-based reasoning* [24], [25], and *constraint programming* [26]. Integer linear programming techniques attempt to optimize a linear objective function. Case based reasoning approaches store past experiences in a knowledge base for later retrieval which is used to solve future problems by applying past solutions or inferring new solutions from previous related problems. Constraint programming is a form of declarative programming which captures relations between variables as constraints which describe properties of possible solutions. Feedback control approaches have been built using reinforcement learning [21], support vector machines [27], neural networks [28] [29], and a fitness function [30]. Performance models have been built using regression techniques [31], artificial neural networks [21], [32][21], and support vector machines [27]. Hybrid approaches which combine the use of performance modeling with feedback control include: [21], [27], [29].

Feedback control approaches apply control system theory to actively tune resources to best meet pre-stated service level agreements (SLAs). Feedback control systems do not determine optimal configurations as they often consider a smaller subset of the exploration space as they use actual system observations to train models. This may result in inefficient control response, particularly upon system initialization. Multivariate optimization approaches model system performance with larger or complete training data sets enabling a much larger portion of the exploration space to be considered. Performance models require initialization with training datasets which can be difficult and time consuming to collect. Models with inadequate training data sets may be inaccurate and ineffective for providing resource control. Time to collect and analyze training datasets results in a trade-off between *model accuracy vs. availability*. Additionally performance models with a large number of independent variables or a sufficiently large exploration space exhibit the *accuracy vs. complexity* trade-off. Difficulty of collecting training data for models with a large search space, and a large number of variables leads to increased model development effort possibly forcing a tradeoff with model accuracy to keep model building tractable. Hybrid autonomic resource provisioning approaches combine the use of performance models with feedback control system approaches with an aim to provide better control decisions more rapidly. These systems use training datasets to inform control decisions immediately upon initialization which are further improved as the system operates and more data is collected. Hybrid approaches often use simplified performance models which trade-off accuracy for speed of computation and initialization.

Wood et al. developed Sandpiper, a black-box and gray-box resource manager for VMs [31]. Sandpiper was designed to oversee server partitioning and was not designed specifically for IaaS. "Hotspots" are detected when provisioned architecture fails to meet service demand. Their approach was limited to vertical scaling which includes increasing available resources to VMs, and VM migration to a less busy PMs as needed. They did not perform horizontal scaling by launching additional VMs and load balancing. Sandpiper acts as a control system which attempts to meet a predetermined SLA. Xu et al. developed a resource learning approach for autonomic infrastructure management [21]. Both application agents and VM agents were used to monitor performance. A state/action table was built to record performance quality changes resulting from state/action events. A neural network model was later added to predict reward values to help improve performance after initialization when the state/action table was only sparsely populated. Kousiouris et al. benchmarked all possible configurations for different task placements across several VMs running on a single PM [32]. From their observations they developed both a regression model and an artificial neural network to model performance. Their approach did not perform resource control, but focused on performance modeling to predict the performance implications of task placements. Niehorster et al. developed an autonomic resource provisioning system using support vector machines (SVMs) for IaaS clouds [27]. Their system responds to service demand changes and alters infrastructure configurations to enforce SLAs. They performed both horizontal and vertical scaling of resources and dynamically configured application specific parameters.

A number of formal approaches for autonomic resource management appear in the literature and commonly they've been built and tested with simulations only and not tested with physical clouds. Lama and Zhou proposed the use of self-adaptive neural network based fuzzy controllers in [28], [29] and was limited to controlling the number of VMs. Addis et al. model resource control as a mixed integer non-linear programming problem and apply two main features of classical local search exploration and refinement in [22]. Maurer et al. propose using a knowledge management system which uses case based reasoning to minimize SLA violations, achieve high resource utilization, conserve time and energy and minimize resource reallocation (migration) [24], [25]. Van et al. treat virtual resource management as a constraint classification problem and employ the choco constraint solver [26]. Their approach is model agnostic as individual applications must provide their own performance model(s). Li et al. propose a linear integer programming model for scheduling and migration of VMs for multi-cloud environments which considers the costs of VM migration and cloud provider pricing [23]. Bonvin et al. propose a virtual economy which considers the economic fitness of the utility provided by various application component deployments to cloud infrastructure [30]. Server agents implement the economic model on each cloud node to help ensure fault tolerance and adherence to SLAs. Unlike above mentioned approaches Bonvin et al. evaluated their approach using applications running on physical servers, but their approach did not consider server virtualization but simply managed the allocation/deallocation of services across a cluster of physical servers.

2.2. RESEARCH GAPS

Table 2.1 provides a comparison of autonomic infrastructure management approaches described in [21], [27], [31], [32]. These approaches are compared because of the similarity to our proposed approach(es) described later in this research proposal. Each of the reviewed approaches has built a performance model and validated it benchmarking physical hardware in contrast to theoretical approaches which were validated using only simulation [22]–[26], [28], [29]. The complexity of cloud based systems makes validation using only simple economics-like simulations of questionable value. Table 2.1 shows features modeled, controlled, and/or considered by each of the approaches. Analyzing these approaches helps identify gaps in existing research. None of the approaches reviewed address composition of application components, as components were always deployed separately using full VM service isolation. Service isolation enables easier horizontal scaling of resources for specific application components but requires the largest number of VMs and may not provide better performance versus more concise deployments [13], [33]. Several issues were considered by only one approach including: horizontal scaling of VMs, tuning application specific parameters, determination of optimal configurations, and live VM migration. None of the approaches in Table 2.1 consider many independent variables in performance models and generally focus on a few select variables purported as the crux of their research contributions while ignoring implications of other variables. Approaches reviewed did not consider performance implications of virtualization hypervisor type (XEN, KVM, etc.) or disk I/O throughput, and only one approach considered implications of network I/O throughput and VM placement across PMs. Learning approaches which tend towards the use of simplified performance models may fail to capture the cause of improved performance when too many variables have been omitted from models. Other issues not addressed include the use of heterogeneous environments when PMs have varying capabilities, resource contention from interference between application components, and interference from non-application VMs.

Table 2.1. Autonomic Infrastructure Management Comparison

Feature controlled / modeled	**Wood et al.** [31]	**Xu et al.** [21]	**Kousiouris et al.** [32]	**Niehorster et al.** [27]
CPU scheduler credit		X	X	
VM memory allocation	X	X		X
Hypervisor type (KVM/XEN)				
Disk I/O Throughput				
Network I/O Throughput	X			
Location of application VMs			X	
Service composition of VM images				
Scaling # of VMs per application component				X
Application specific parameters				X
SLA/SLO enforcement	X	X		X
Determine optimal configuration			X	
Live VM migration	X			
Live tuning of VM configuration	X	X		X
# of CPU cores	X	X		X
memory	X	X		X
Performance modeling		X	X	X
Multi-tier application support	X			X

Research should establish the most important variables for impacting application performance on IaaS clouds to form a base set of parameters for future performance models. Performance models may be further improved by the development and application of heuristics which capture performance behavior and characteristics specific to IaaS clouds. New performance models should be developed which better capture effects of virtualization, component and VM location, and characteristics of physical resource sharing to support ideal load balancing of disk I/O, network I/O and CPU resources.

Existing gaps in research result from the infancy of cloud computing and the difficulty of performance modeling while resulting from a large problem space(s) with many potential independent variables. Benchmarking application resource utilization is difficult as isolating resource utilization data using public clouds with heterogeneous PMs which may potentially host multiple unrelated applications is difficult. Collecting resource utilization data involves overhead and can skew the accuracy of the data. Use of isolated testing environments such as private IaaS clouds can support testing but private IaaS virtual infrastructure management (VIM) software is still evolving and presently exhibits variable performance, incomplete feature sets, and inconsistent product stability [34]. Virtualization further complicates IaaS research, in that the effects of virtualization are often misunderstood as the underlying implementation of virtualization hypervisors are often misunderstood.

CHAPTER 3

MIGRATION OF SERVICE ORIENTED APPLICATIONS

3.1. INTRODUCTION

Migration of service oriented applications (SOAs) to Infrastructure-as-a-Service (IaaS) clouds involves decomposing applications into an application stack of service-based components. Application stacks may include components such as web server(s), proxy server(s), database(s), file server(s) and other servers/services. Service isolation involves separating components of the application stack so they execute using separate virtual machine (VM) instances. Isolation provides components explicit sandboxes, not shared by other systems. Using hardware virtualization, isolation can be accomplished multiple times for separate components on a single physical server. Previously service isolation using a physical data center required significant server real estate. Hardware virtualization refers to the creation and use of VMs which run on a physical host computer. Recent advances in x86-based virtualization enabled by CPU-specific enhancements to support device simulation have eliminated the need for specialized versions of guest operating systems as required with XEN-based paravirtualization [2]. Full virtualization, where the guest operating system is unaware that it is being virtualized is now possible as hardware is simulated with no direct access to the physical host's hardware. Virtualization provides for resource elasticity where the quantity, location, and size of VM allocations can change dynamically to meet varying system loads, as well as increased agility to add and remove services as an application evolves.

Together service isolation, hardware virtualization, and resource elasticity are key benefits motivating the adoption of IaaS based cloud-computing environments such as Amazon's

Elastic Compute Cloud (EC2). Despite these advantages, cloud-based virtualization and service isolation raise new challenges which must be addressed when migrating SOAs to IaaS clouds. Provisioning variation, the ambiguity over how and where application components hosted by VMs are deployed across physical machines of a cloud, can lead to unpredictable, and even unwanted performance variation [16], [18], [20]. Unwanted multi-tenancy occurs when multiple resource intensive VMs reside on a single physical host computer potentially leading to resource contention and application performance degradation. Virtualization incurs overhead because a VM's memory, CPU, and device operations must be simulated on top of the physical host's operating system.

In this chapter we investigate the following research questions:

RQ-1: How can service oriented applications be migrated to Infrastructure-as-a-Service cloud environments, and what factors must be accounted for while deploying and then scaling applications for optimal throughput?

RQ-2: What is the impact on application performance as a result of provisioning variation? Does multi-tenancy, having multiple application VMs co-located on a single physical node machine, impact performance?

RQ-3: What overheads are incurred while using Kernel-based virtual machines (KVM) for hosting components of a service oriented application?

3.2. RELATED WORK

Rouk identified the challenge of finding optimal image and service composites, a first step in migrating SOAs to IaaS clouds in [35]. Chieu et al. [36] next proposed a simple method to scale applications hosted by VMs by considering the number of active sessions and scaling the number of VMs when the number of sessions exceed particular thresholds. Iqbal et al. [37] using a Eucalyptus-based private cloud developed a set of custom Java-components based on the Typica API which supported auto-scaling of a 2-tier web application consisting of web server and database VMs. Their system automatically scaled resources when system performance fell below a predetermined threshold. Log file heuristics and CPU utilization data were used to determine demand for both static and dynamic web content to predict which system components were most heavily stressed. Appropriate VMs were then launched to remedy resource shortages. Their approach is applicable web applications where the primary content being served is static and/or dynamic web pages. Liu and Wee proposed a dynamic switching architecture for scaling a web server in [38]. Their work was significant in identifying the existence of unique bottlenecks occurring at different points when scaling up web applications to meet greater system loads. In each case fundamental infrastructure changes were required to surpass each bottleneck before scaling further. They identified four web server scaling tiers for their switching architecture including: (1) a m1.small Amazon EC2 VM (consists of: 1.7 GB memory, 32-bit ~2.6 GHz CPU core, 160 GB HDD), (2) a set of load balanced m1.small Amazon EC2 VMs, (3) a c1.xlarge Amazon EC2 VM (consists of: 7GB memory, 64-bit ~2.4 GHz 8 CPU cores, 1690 GB HDD), and (4) the use of DNS level load balancing to balance across multiple c1.xlarge Amazon EC2 VMs. DNS load balancing was required when more than 800 Mbps of network bandwidth was required, the threshold found to exceed an Amazon EC2 c1.xlarge instance. Their work is important because they identified the complexity of scaling a web server by showing that multiple unique bottlenecks occur while scaling infrastructure to meet greater system loads. Wee and Liu further demonstrated a cloud-based client-side load balancer, an alternative to DNS load balancing, which achieves greater throughput than software load balancing [39]. Using Amazon's simple storage service (S3) to host client-side script files with load balancing logic, they demonstrate load balancing against 12 Amazon VMs enabling a total throughput greater than the bandwidth of a single c1.xlarge VM. The investigations above made contributions in investigating approaches to host and scale web sites hosted in cloud environments, but they did not consider issues of hosting and scaling more complex SOAs such as web services and models in IaaS clouds.

Schad et al. [18] demonstrated the unpredictability of Amazon EC2 VM performance, an effect caused by resource contention for physical machine resources and provisioning variation of VMs in the cloud. Using a XEN-based private cloud Rehman et al. [16] tested the effects of resource contention on Hadoop-based MapReduce performance by using IaaS-based cloud VMs to host Hadoop worker nodes. They tested the effect of provisioning variation of three different provisioning schemes of VM-based Hadoop worker nodes and observed performance degradation when too many worker nodes were physically co-located. Zaharia et al. further identified that Hadoop's scheduler can cause severe performance degradation as a result of being unaware of resource contention issues when Hadoop nodes are hosted by Amazon EC2-based VMs [20]. They improved upon Hadoop's scheduler by proposing the Longest Approximate Time to End (LATE) scheduling algorithm and demonstrated how this approach better dealt with virtualization issues when Hadoop nodes were implemented using Amazon EC2-based VMs. Both of these papers identified implications of provisioning variation when migrating Hadoop worker nodes from a physical cluster to an IaaS-based cloud, but implications resulting from provisioning variation of hosting components of SOAs was not addressed.

Camargos et al. investigated different approaches to virtualizing linux servers and computed numerous performance benchmarks for CPU, file and network I/O [3]. Several virtualization schemes including XEN, KVM, VirtualBox, and two container based virtualization approaches OpenVZ and Linux V-Server were evaluated. Their benchmarks targeted different parts of the system including tests of kernel compilation, file transfers, and file compression. Armstrong and Djemame investigated performance of VM image propagation using Nimbus and OpenNebula two different IaaS cloud infrastructure managers [40]. Additionally they benchmarked throughput of both XEN and KVM paravirtualized I/O. Though these works investigated performance issues due to virtualization neither study investigated the virtualization overhead resulting from hosting complete SOAs in IaaS clouds.

3.3. CONTRIBUTIONS

This chapter presents the results of an investigation on deploying two variants of a popular scientific erosion model to an IaaS-based private cloud. The variants enabled us to study application migration for applications with two common resource footprints: a processor bound and an I/O-bound application. Both application variants provided erosion modeling capability as a webservice and were implemented using four separate virtual machines on an IaaS-based private cloud. We extend previous work which investigated effects of provisioning variation for Hadoop worker nodes deployed on IaaS clouds [16], [20] and virtualization studies which largely used common system benchmarks to quantify overhead [3], [40]. Our work also extends prior research by investigating the migration of complete SOAs to IaaS clouds [36]–[39] and makes an important contribution towards understanding the implications of application migration, service isolation and virtualization overhead to further the evolution and adoption of IaaS-based cloud computing.

3.4. EXPERIMENTAL INVESTIGATION

3.4.6. Experimental Setup

For our investigation we deployed two variants of the Revised Universal Soil Loss Equation – Version 2 (RUSLE2), an erosion model as a cloud-based web service to a private IaaS cloud environment. RUSLE2 contains both empirical and process-based science that predicts rill and interrill soil erosion by rainfall and runoff [41]. RUSLE2 was developed primarily to guide conservation planning, inventory erosion rates, and estimate sediment delivery and is the USDA-NRCS agency standard model for sheet and rill erosion modeling used by over 3,000 field offices across the United States. RUSLE2 is a good candidate to prototype SOA migration because its architecture consisting of a web server, relational database, file server, and logging server serves as a surrogate for multi-component SOA with a diverse application stack.

RUSLE2 was originally developed as a Windows-based Microsoft Visual C++ desktop application. To facilitate functioning as a web service a modeling engine known as the RomeShell was added to RUSLE2. The Object Modeling System 3.0 (OMS 3.0) framework [42], [43] using WINE [44] provides middleware to facilitate model to web service inter-operation. OMS was developed by the USDA–ARS in cooperation with Colorado State University and supports component-oriented simulation model development in Java, C/C++ and FORTRAN. OMS provides numerous tools supporting data retrieval, GIS, graphical visualization, statistical analysis and model calibration. The RUSLE2 web service was implemented as a JAX-RS RESTful JSON-based service hosted by Apache Tomcat [45].

A Eucalyptus 2.0 [46] IaaS private cloud was built and hosted by Colorado State University which consisted of 9 SUN X6270 blade servers on the same chassis sharing a private 1 Giga-bit VLAN with dual Intel Xeon X5560-quad core 2.8 GHz CPUs each with 24GB ram and 146GB HDDs. The host operating system was Ubuntu Linux (2.6.35-22) 64-bit server 10.10. VM guests ran Ubuntu Linux (2.6.31-22) 32 and 64-bit server 9.10. 8 blade servers were configured as Eucalyptus node-controllers, and 1 blade server was configured as the Eucalyptus cloud-controller, cluster-controller, walrus server, and storage-controller. Eucalyptus-based managed mode networking was configured using a managed Ethernet switch isolating VMs on their own private VLANs.

QEMU version 0.12.5, a Linux-based PC system emulator, was used to provide VMs. QEMU makes use of the KVM Linux kernel modules (version 2.6.35-22) to achieve full virtualization of the guest operating system. Recent enhancements to Intel/AMD x86-based CPUs provide special CPU-extensions to support full virtualization of guest operating systems without modification. With these extensions device emulation overhead can be reduced to improve performance. One limitation of full virtualization versus XEN-based paravirtualization is that network and disk devices must be fully emulated. XEN-based paravirtualization requires special versions of both the host and guest operating systems with the benefit of near-direct physical device access [3].

3.4.7. Application Components

Table 3.1 describes the four VM image types used to implement the components of RUSLE2's application stack. The Model $\mathcal{M}$ VM hosts the model computation and web services using Apache Tomcat. The Database $\mathcal{D}$ VM hosts the spatial database which resolves latitude and longitude coordinates to assist in parameterizing climate, soil, and management data for RUSLE2. Postgresql was used as a relational database and PostGIS extensions were used to support spatial database functions [47], [48]. The file server $\mathcal{F}$ VM was used by the RUSLE2 model to acquire XML files to parameterize data for model runs. NGINX [49], a lightweight high performance web server provided access to a library of static XML files which were on average ~5KB each. The logging $\mathcal{L}$ VM provided historical tracking of modeling activity. The codebeamer tracking facility was used to log model activity [50]. Codebeamer provides an extensive customizable GUI and reporting facility. A simple JAX-RS RESTful JSON-based web service was developed to encapsulate logging functions to decouple Codebeamer from the RUSLE2 web service and also to provide a logging queue to prevent logging delays from interfering with the RUSLE2 webservice. HAProxy was used to provide round-robin load balancing of $\mathcal{M}$ and $\mathcal{D}$ VMs. HAProxy is a dynamically configurable very fast load balancer which supports proxying both TCP and HTTP socket-based network traffic [51].

Table 3.1. Virtual Machine Types

VM		Description
$\mathcal{M}$	Model	64-bit Ubuntu 9.10 server w/ Apache Tomcat 6.0.20, Wine 1.0.1, RUSLE2, Object Modeling System (OMS 3.0)
$\mathcal{D}$	Database	64-bit Ubuntu 9.10 server w/ Postgresql-8.4, and PostGIS 1.4.0-2. Spatial database consists of soil data (1.7 million shapes, 167 million points), management data (98 shapes, 489k points), and climate data (31k shapes, 3 million points), totaling 4.6 GB for the state of TN and CO
$\mathcal{F}$	File server	64-bit Ubuntu 9.10 server w/ nginx 0.7.62 to serve XML files which parameterize the RUSLE2 model. 57,185 XML files consisting of 305MB.
$\mathcal{L}$	Logger	32-bit Ubuntu 9.10 server with Codebeamer 5.5 running on Tomcat. Custom RESTful JSON-based logging wrapper web service.

3.4.3. Component Deployments

Our application stack of 4 components can be deployed 15 possible ways across 4 physical node computers. Tables 3.2 shows the four stack deployments we tested labeled as P1-P4 and V1-V4 respectively. P1-P4 denotes physical stack deployments where components were deployed on physical machines by installing software directly on the host operating system. V1-V4 denotes virtual stack deployments where components were imaged and then launched as VMs in our private Eucalyptus cloud. Eucalyptus does not provide control where VMs are physically deployed. To test (V1-V4) deployments placeholder VMs were launched and terminated to force the desired VM placements as using Eucalyptus' round-robin VM deployment scheme. We expected the $\mathcal{D}$ and $\mathcal{M}$ components to be the most resource intensive components motivating our interest to test their deployment in isolation on physical nodes (P2/V2 and P4/V4). P1/V1 tested the deployment of all components on a single machine. P1/V1 should benefit from locality of dependent services which should reduce dependence on network I/O with the added cost of greater contention for local disk and CPU resources. P3/V3 tested running each component in isolation, allowing components the greatest freedom to fully utilize local CPU and disk resources, at the expense of greater network I/O requirements.

Table 3.2. Physical (P) and Virtual (V) Stacks Deployment

	Node 1	**Node 2**	**Node 3**	**Node 4**
P1/V1	$\mathcal{M}$ $\mathcal{D}$ $\mathcal{F}$ $\mathcal{L}$			
P2/V2	$\mathcal{M}$	$\mathcal{D}$ $\mathcal{F}$ $\mathcal{L}$		
P3/V3	$\mathcal{M}$	$\mathcal{D}$	$\mathcal{F}$	$\mathcal{L}$
P4/V4	$\mathcal{M}$ $\mathcal{L}$ $\mathcal{F}$	$\mathcal{D}$		

Eucalyptus 2.0 allows custom definitions for VM sizes (small, medium, large) supporting customization of the number of virtual CPUs, memory, and disk size allocations. We tested a variety of VM resource allocations for our application VMs. For some tests we over-allocated the number of virtual CPUs far beyond the number of physical CPUs present on the host machine. For stack deployments with multi-tenancy this increased contention for computational resources.

3.4.4. Testing Infrastructure

The RUSLE2 web service supports individual model runs and ensemble runs which are groups of modeling requests bundled together. To invoke the web service a client sends a JSON object including parameters for management practice, slope length, steepness, latitude, and longitude. Model results are computed and returned as a JSON object. Ensemble runs are processed by dividing the set of modeling requests into individual requests which are resent to the web service, similar to the "map" function of MapReduce. A configurable number of worker threads concurrently executes individual runs of the ensemble, and upon completion results are combined (reduced) into a single JSON response object and returned. A simple program generated randomized ensemble tests of 25, 100, and 1000 runs. Latitude and longitude coordinates were randomly selected within a large bounding box from the state of Tennessee. Slope length, steepness, and the management practice parameters were also randomized. Randomization of latitude and longitude resulted in variable spatial query execution times due to the variable complexity of the polygons coordinates intersected with. To counteract the effect of caching, before each ensemble test was run, all application server components were stopped and restarted and a 25-model run ensemble test was executed to warm up the system. The warm up test was warranted after we observed postgresql performing slowly on initial spatial queries upon startup.

To measure performance, the RUSLE2 model and web service code was instrumented to capture timing data for various operations and returned in the JSON response objects. Custom parsing programs were used to extract timing data from the JSON objects for analysis and graphing. Captured timing data included: "fileIO" the time required to load data files provided by nginx, "model" the time spent shelling to the operating system to execute the model using WINE, "climate/soil query" the time spent executing spatial queries, "logging" the time spent submitting models to the logger, "overhead" representing all operations not specifically timed, and "total" the total time of the web service call from start to finish. "fileIO" was a subset of the “model” time because nginx file I/O occurred simultaneously during model execution.

3.4.5. Application Variants

Our investigation tested two variants of RUSLE2 which we refer to herein as the “d-bound” for the database bound variant and the “m-bound” for the model bound variant. By testing two variants of RUSLE2 we hoped to gain insights on two common types of SOAs, an application bound by the database tier, and an application bound by the middleware (model) tier. For the d-bound version of RUSLE2 two primary spatial queries were modified to perform a join on a nested query, while the m-bound variant was unmodified. This modification significantly increased demand for computational resources from the database. The d-bound variant should require the same resources as the m-bound plus additional processing to compute results of several thousand additional queries making the d-bound application more CPU bound than the m-bound variant.

3.5. EXPERIMENTAL RESULTS

3.5.1. Application Profiling

An application's profile refers to its processing, I/O, and memory requirements which change over time as an application evolves. To assess the application profiles of the RUSLE2 variants we used the V1 stack configuration. An identical 100-model run ensemble test was used to determine the time distribution of model operations as shown in figure 3.1. For the d-bound application the “climate” and “soil” spatial queries consumed about ~77% of “total” execution time, while the “model” spent about ~22%, with remaining time split between “logging” time and “overhead”. “Logging” time was negligible because the logger queued logging requests which then executed independently of model execution. “FileIO” a subset of the “model” time took approximately 3.5% of the overall time. D-bound climate queries were generally fast compared to soil queries. Much of the execution time reported for climate queries we observed was time spent waiting for soil queries to complete. For the m-bound application the model consumed ~91% of the "total" execution time, while spatial queries accounted for about 1%. "Overhead" was just over 8% of the "total" time, while "FileIO" operations, a subset of "model" time, increased to 19%. Performance for the D-bound and M-bound application variants appeared bounded by their respective named components $\mathcal{D}$ and $\mathcal{M}$.

The application variants were next tuned to minimize the 100-model run ensemble test execution time. Virtual resource allocations were determined for CPU cores, memory size, and disk space. Application tuning included determining an optimal number of shared database connections for the database connections pool, and the number of model execution (worker) threads for ensemble runs. For each tuning step we identified ideal parameter configurations by identifying when performance improvements leveled off and appeared as normal variation or when performance actually decreased.

To determine an optimal number of database connections we tested using a $\mathcal{D}$ VM allocated with 6 virtual cores while using 6 worker threads to run models. Figure 3.2 shows the best performance for the d-bound application occurred when using approximately 5 database connections, while the number of database connections did not appear to have a significant impact on the m-bound application. For subsequent tests we used 5 and 8 connections for the d-bound and m-bound applications respectively. According to the postgresql documentation individual connections can utilize at most only 1 CPU core leading to our assignment of 8 connections and 8 cores for the m-bound application.

For the d-bound application with 5 shared connections, we varied the $\mathcal{D}$ VM's number of virtual cores to test the impact on performance as shown in Figure 3.3. The best performance was observed while allocating approximately 6 virtual cores with a slight performance degradation seen when using additional virtual cores. Sharing 16 database connections while increasing the number of D VM virtual cores did not improve performance. When observing the *D* VM's KVM process on the host machine, we observed with virtual CPU allocations (>6), the *D* VM did not utilize more than ~500-600% of the 8-core physical machine's CPU capacity, where 100% represents a fully allocated CPU core. It was unclear if this limitation was caused by postgresql or through the use of KVM.

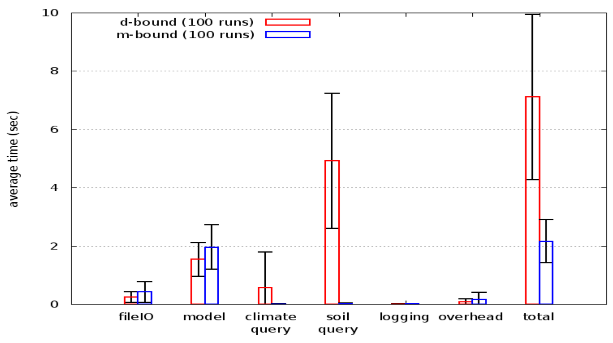


Figure 3.1. RUSLE2 Application Time Footprint

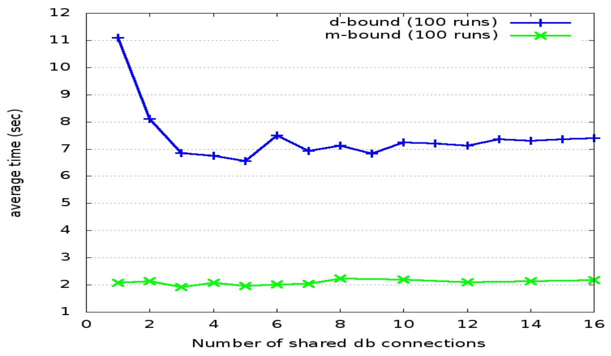


Figure 3.2. V1 stack with variable database connections

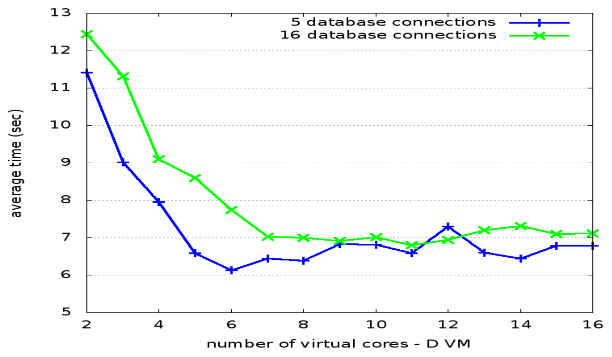


Figure 3.3. V1 stack d-bound with variable 𝒟 VM virtual CPUs

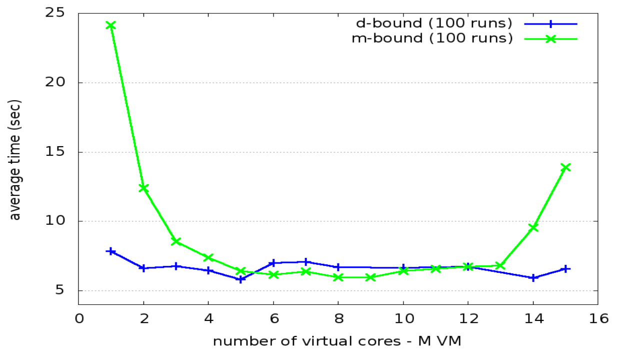


Figure 3.4. V1 stack with variable ℳ VM virtual CPUs

Figure 3.4 shows the average model run time while varying the number of virtual cores allocated to the ℳ VM. Optimal performance was observed using 5 or more virtual cores for the d-bound application and 8 virtual cores for the m-bound application. The m-bound application benefited from additional virtual cores but suffered when cores were over-allocated beyond the number of physical cores on the host. Figure 3.5 shows the 100-model run ensemble time using 16 and 8 shared database connections for the d-bound and m-bound applications respectively. Each worker thread concurrently executed RUSLE2 model runs. Using 6 worker threads appeared to be an optimal number for the d-bound application with similar performance seen using 5 or 7 worker threads. For the m-bound application using at least 6 threads appeared optimal. For the m-bound application, we tested using up to 100 worker threads but did not observe a significant performance difference versus 6 threads.

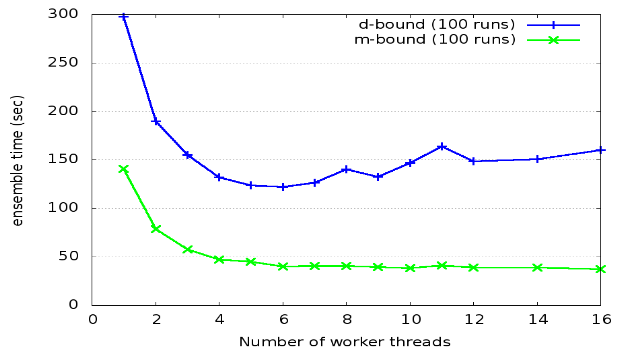


Figure 3.5. V1 stack with variable worker threads

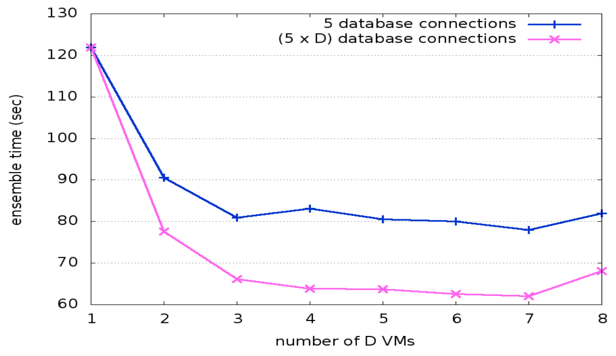


Figure 3.6. D-bound ensemble time with variable $\mathcal{D}$ VMs

Upon completion of application tuning for the V1 provisioning scheme the d-bound application required an average of ~120 seconds to complete a 100-model run ensemble test, and the m-bound application ~32 seconds.

3.5.2. Virtual Resource Scaling

After tuning a V1 deployment of our application variants we next scaled the variants to fully utilize all available resources of our private cloud to obtain optimal performance for 100-model run ensemble tests. Additional $\mathcal{D}$ and $\mathcal{M}$ VMs were allocated for the d-bound and m-bound applications. Figure 3.6 shows the performance of the d-bound application when we allocated multiple $\mathcal{D}$ VMs with each running in isolation on a separate physical machine. For the m-bound application allocating additional $\mathcal{D}$ VMs was not tested because we were unable to fully saturate a single $\mathcal{D}$ VM. We tested the performance using 5 shared database connections and also database connections equal to the number of $\mathcal{D}$ VMs multiplied by 5. Increasing the number of database connections was required to ensure that the tomcat server would have at least one connection to each postgresql database. Scaling the number of $\mathcal{D}$ VMs was shown to result in a favorable performance improvement until approximately 3 to 4 $\mathcal{D}$ VMs. Beyond this performance improvements could not be differentiated as the results appeared similar to variance.

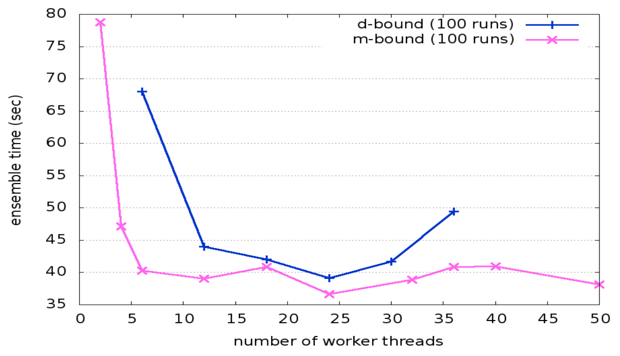


Figure 3.7. Ensemble runtime with variable worker threads

To move past the d-bound application bottleneck the number of worker threads was increased as shown in figure 3.7. For the d-bound application we observed a bottleneck when 40 shared database connections and 24 concurrent worker threads were used. Increasing beyond 24 worker threads appeared to degrade performance. For the m-bound application only 1 $\mathcal{D}$ VM is used for tests in figure 3.7, but a similar performance result is seen when exceeding 24 worker threads. To realize further performance improvements both applications required us to next increase the number of $\mathcal{M}$ VMs.

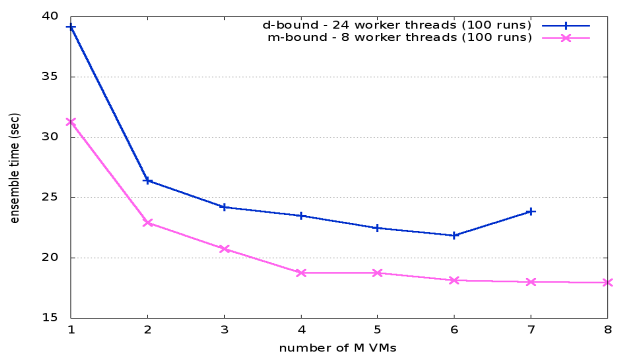


Figure 3.8. Ensemble runtime with variable $\mathcal{M}$ VMs

Figure 3.8 shows the speed improvement realized by scaling the number of $\mathcal{M}$ VMs. While scaling the number of $\mathcal{M}$ VMs, a fixed number of 24 and 8 worker threads were used for the d-bound and m-bound applications respectively. For the d-bound application beyond allocating 3 $\mathcal{M}$ VMs performance gains appeared minimal. At 7 $\mathcal{M}$ VMs we observed slight performance degradation. At the completion of d-bound application scaling the 100-model run ensemble test executed in 21.8 seconds, 5.5x faster than before VM scaling using {8 $\mathcal{D}$, 6 $\mathcal{M}$, 1 $\mathcal{F}$, 1 $\mathcal{L}$} VMs with 24 worker threads and 40 shared database connections per $\mathcal{M}$ VM.

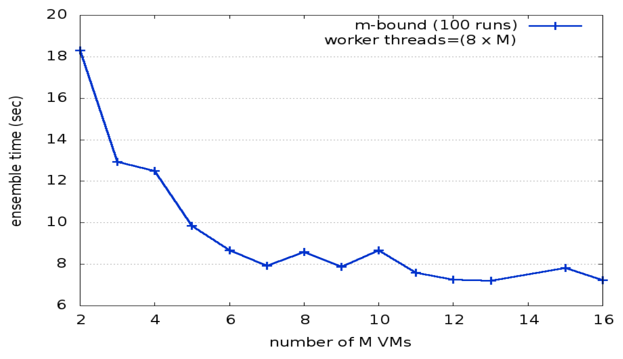


Figure 3.9. M-bound with variable $\mathcal{M}$ VMs and worker threads

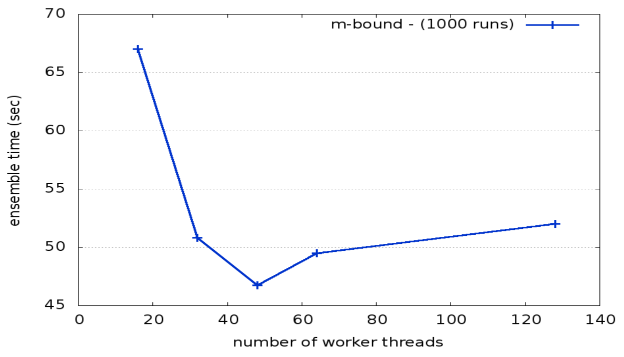


Figure 3.10. M-bound with 16 $\mathcal{M}$ VMs variable worker threads

For the m-bound application a bottleneck was encountered after allocating 4 $\mathcal{M}$ VMs using 8 worker threads and 8 shared database connections. To surpass the bottleneck the number of worker threads was scaled. For each $\mathcal{M}$ VM, an additional 8 worker threads were allocated starting with 8 worker threads for a single $\mathcal{M}$ VM. Figure 3.9 shows the 100-model run ensemble time while scaling to 16 $\mathcal{M}$ VMs with 128 worker threads. The first 8 $\mathcal{M}$ VMs were deployed on separate physical machines. Beyond this we lacked additional physical hosts to run every $\mathcal{M}$ VMs in isolation so multiple $\mathcal{M}$ VMs were deployed on the physical hosts.

A series of 1000-model run ensemble tests were made to assist tuning the optimal number of worker threads for the 16 $\mathcal{M}$ VM deployment shown in figure 3.10. Optimal ensemble test times were observed using 48 worker threads. At the conclusion of m-bound application scaling the 100-model run ensemble test executed in 6.7 seconds, 4.8x faster than before $\mathcal{M}$ VM scaling using {16 $\mathcal{M}$, 1 $\mathcal{D}$, 1 $\mathcal{F}$, 1 $\mathcal{L}$} VMs with 48 worker threads and 8 shared database connections per $\mathcal{M}$ VM.

3.5.3. Provisioning Variation

We tested performance using the physical (P1-P4) and virtual (V1-V4) stack provisioning schemes identified in table II. Timing results for the 100-model run ensemble tests for each stack provisioning for both application variants are shown in Table III. To determine if the stack provisioning schemes performed differently from each other we checked if schemes varied more than 1 standard deviation from each other. For all tests we observed the slowest performance when all application components were co-located on the same physical machine (P1/V1), an expected result. For the virtual tests we observed the best performance when all components ran in physical isolation (V3), also an expected resulted. For the m-bound application we observed slower performance when the $\mathcal{M}$ VM shared physical resources (V1/V4) with other components and for the d-bound when the $\mathcal{D}$ VM shared physical resources (V1/V2). The impact of provisioning variation on application performance appeared dependent on characteristics of the application profile. Best performance required the most computational and I/O intensive components to be run in physical isolation.

Table 3.3. M-BOUND vs. D-BOUND M-Bound vs D-Bound Provisioning Variation

	M-Bound		**D-Bound**	
	Total (sec)	**Rank**	**Total (sec)**	**Rank**
P1/V1	16.15 / 33.65	4	110.21 / 123.61	4
P2/V2	15.89 / 30.99	2	99.08 / 123.43	1 / 3
P3/V3	15.59 / 29.50	1	103.80 / 115.98	2 / 1
P4/V4	16.11 / 33.65	3	104.17 / 116.05	3 / 2

3.5.4. Virtualization Overhead

To investigate the virtualization overhead resulting from using KVM performance the P1 and V1 provisioning schemes were compared by executing 1000-model runs. The V1 d-bound and m-bound applications used 5 and 8 virtual cores respectively for the $\mathcal{M}$ VM. Both applications used 6 virtual cores for the $\mathcal{D}$ VM and 5 virtual cores for the $\mathcal{F}$ and $\mathcal{L}$ VMs. For both physical and virtual deployments the d-bound application used 6 worker threads and 5 shared database connections, while the m-bound application used 8 worker threads and 8 shared database connections.

Table 3.4. P1 vs. V1 KVM Virtualization Overhead

	D-bound			M-bound		
	Virt. O/H	P1 avg (ms)	V1 avg (ms)	Virt. O/H	P1 avg (ms)	V1 avg (ms)
fileIO	319.70%	55.77	234.06	463.54%	56.57	318.79
model	54.50%	968.47	1496.24	100.16%	815.56	1632.46
climate query	-11.41%	691.86	612.95	404.54%	1.28	6.45
soil query	3.25%	4371.20	4513.39	12.04%	11.84	13.26
logging	1360.69%	0.32	4.72	2680.58%	0.35	9.59
overhead	395.14%	14.30	70.81	740.02%	15.54	130.54
total	10.78%	6046.16	6698.10	112.22%	844.56	1792.30

Table IV shows timing of the physical versus virtual stacks. The d-bound application's virtualization overhead for the total system was quite low at just 10.78%, while the m-bound application was 112.22%. When examining the application footprints of the m-bound and d-bound applications, the m-bound application appears more I/O bound with nearly 20% (~319 ms) of the total model execution time spent in "fileIO" versus just 3.5% (~234 ms) for the d-bound application. Similarly "overhead" which consisted primarily of writing logging files was 7.3% (~131 ms) of the total model execution time for the m-bound application, but only 1.1% (~71 ms) for the d-bound application. For the m-bound application I/O operations were not only a greater percentage of the overall application footprint, but the operations themselves took longer to perform. We suspect this result was due to greater contention for CPU and I/O resources because of the higher density of I/O operations for the m-bound application. This effect was barely seen with the P1 provisioning scheme because the physical machines performed direct device I/O and did not experience by additional resource contention from device emulation. The d-bound application was less impacted by I/O virtualization overhead because most of the execution time 77% (~5053 ms) was spent performing CPU-bound nested database queries. Our results demonstrate that application profiles which detail how applications utilize resources (CPU, memory, I/O) are helpful in determining application performance when virtualized.

3.6. CONCLUSIONS

Two variants of the RUSLE2 erosion model serving as surrogates for common SOA architectures were tested to investigate application migration to IaaS clouds. While scaling both application variants, different bottlenecks were encountered based on each variant's application profile. Surmounting these bottlenecks required custom tuning of application parameters and/or virtual resource allocations. Simply increasing the number of VMs did not lead to optimal application throughput. Application scaling required understanding the application profile as well as dependencies among the application components.

Provisioning variation impacted performance based on application profiles. Best performance was observed when the most CPU and I/O intensive components were isolated on separate physical hardware whereas performance degradation occurred when too many resource intensive components were co-located highlighting the importance of considering an application's profile for VM placement across physical machines.

Virtualization overhead varied based on the profile of the application being virtualized. The d-bound application, which was more CPU-bound, appeared less impacted by virtualization overhead (~11% overhead) whereas the m-bound application, which appeared more I/O bound, showed a greater performance degradation due to virtualization (~112% overhead). As file and network device performance varies with different virtualization approaches a future investigation is planned to test other types of virtualization including XEN-based full and para-virtualization to better understand implications of hosting SOAs using different types of virtualization.

In conclusion, application scaling, provisioning variation and virtualization overhead all appear impacted by an application's profile. We have explored how an application's profile relates to interactions between its constituent components and the corresponding implications for migration. Once an application's profile is known, this can guide the efficient deployment of the application to IaaS clouds while accounting for scaling and throughput requirements.

CHAPTER 4

PERFORMANCE MODELING TO SUPPORT SERVICE ORIENTED APPLICATION DEPLOYMENT

4.1. INTRODUCTION

Migration of service oriented applications (SOAs) to Infrastructure-as-a-Service (IaaS) clouds requires applications be decomposed into sets of service-based components known as the application stack. Application stacks consist of components such as web server(s), application server(s), proxy server(s), database(s), file server(s) and other servers/services.

Infrastructure-as-a-Service clouds support better utilization of server infrastructure by enabling multiplexing of resources. Infrastructure supporting specific applications can be scaled based on demand while multiple applications share the physical infrastructure through the use of server virtualization. IaaS clouds consisting of many physical servers with one or more multi-core CPUs can host Virtual Machines (VMs) enabling resource elasticity where the quantity, size, and location of VMs can change dynamically to meet varying system demand.

Many challenges exist when deploying SOAs toIaaS clouds. VM image composition requires application components to be composed across a set of VM images. Resource contention should be minimized by taking advantage of opportunistic placements by collocation of codependent components. Provisioning variation refers to the uncertainty of the physical location of VMs when deployed to IaaS clouds [16]. VM physical location could lead to performance improvements or degradation depending on component resource requirements and interdependencies. Internal resource contention occurs when application VMs are provisioned to the same physical machines (PMs) while competing for the same resources. External resource contention can occur when different applications share physical infrastructure an important issue for public clouds. Virtualization overhead refers to the costs associated with emulating a computer as a software program on a physical host computer. This overhead varies depending on the approaches used to multiplex physical resources among virtual hosts. Virtualization hypervisors vary with respect to their ability to minimize this overhead with some generally responding better to certain resource sharing and simulation scenarios than others [1]–[3], [40]. Resource provisioning refers to the challenge of allocating adequate virtual infrastructure to meet performance requirements while accounting for the challenges of image composition, provisioning variation, resource contention, and virtualization overhead. Research and investigation into approaches supporting autonomic resource provisioning also known as autonomic infrastructure management is an active area of cloud computing research [21], [27], [31], [32].

Service compositions must be determined which map application stacks across VM images. Determining beneficial combinations of components which multiplex resources without causing unwanted resource contention poses a challenge. Component compositions will vary for SOAs as applications have different application stacks of components and resource utilization profiles further complicating determination of ideal VM component deployments.

Using brute force performance testing to determine optimal placements is only feasible for applications with small numbers of components. Bell's number is the number of partitions of a set (k) consisting of (n) members [15]. If we consider an application as a set of (n) components, then the total number of possible component compositions for an application is Bell's number (k). Table 4.1 shows the first few Bell numbers describing the possible number of component compositions. As the number of components increases, the possible number of service compositions grows rapidly making the use of brute force testing to benchmark performance impractical. Web applications such as mashup applications which aggregate many data sources and application programming interfaces (APIs) may have application stacks with a large number of components. Complicating matters further, public IaaS clouds often do not provide the ability to control VM placements making it difficult, if not impossible, to deploy all possible placements. Exclusive reservation of PMs may be available for an additional cost enabling granular control of physical placement of VMs.

Table 4.1. Service Oriented Application Component Counts

Number of Components (n)	**Number of Compositions (B_n)**
3	5
4	15
5	52
6	203
7	877
8	4,140

An exponential generating function to generate Bell numbers is given by the formula:

$$\sum_{n=0}^{\infty} \frac{B_n}{n!} x^n = e^{e^x - 1}.$$

Performance models hold promise as a means to rapidly evaluate a large number of possible service compositions without physically deploying and testing them. Good performance models should be able to predict performance outcomes of VM placement and service compositions allowing brute force testing of the entire configuration space to be avoided. Collecting training data to train performance models should be easier than performing brute force testing of all service compositions. Models should provide the ability to make reasonably accurate performance predictions with reasonable amounts of time spent collecting training data and training models.

This chapter presents results of an exploratory study which investigates building SOA performance models using resource utilization statistics. The utility of using different resource utilization statistics as independent variables for predicting service response time is investigated. Performance models of SOAs deployed to IaaS clouds hold promise to (1) guide application component placement across VM images, and (2) support real-time virtual infrastructure management for IaaS clouds by predicting resource requirements for specific performance goals.

The following research questions are investigated in support of our investigation on Infrastructure-as-a-Service application performance modeling:

RQ-1: (Independent Variables) Which VM and PM resource utilization statistics are most helpful for predicting performance of different application service compositions?

RQ-2: (Profiling Data) How should resource utilization data be treated for use in performance models? Should VM profiling data from multiple VMs be combined or used separately?

RQ-3: (Exploratory Modeling) Comparing multiple linear regression (MLR), multivariate adaptive regression splines (MARS), and an artificial neural network (ANN), which model techniques appear to best predict application performance and service composition performance ranks?

4.2. RELATED WORK

Rouk identified the challenge of creating good virtual machine images which compose together application components for migrating SOAs to IaaS clouds in [35]. Negative performance implications and higher hosting costs may result when ad hoc compositions are used resulting in potential unwanted contention for physical resources. Xu et al. identify two classes of approaches for providing autonomic provisioning and management of virtual infrastructure in [21]: multivariate optimization (performance modeling), and feedback control. Multivariate optimization approaches attempt to support better application performance by modeling the tuning of multiple system variables to predict the best configurations. Feedback control approaches based on process control theory attempt to improve configurations by iteratively making changes and observing outcomes in real time using live systems. Feedback control approaches have been built using reinforcement learning [21], support vector machines (SVMs) [27], ANNs [28], [29], and a fitness function [30]. Performance models have been built using MLR [31], ANNs [21], [32], and SVMs [27]. Hybrid approaches which combine the use of a performance model for model initialization and apply real time feedback control include: [21], [27]–[29].

Multivariate optimization approaches can model far more configurations enabling a much larger portion of the exploration space of system configurations to be considered. Time to collect and analyze model training datasets results in a trade-off between model accuracy vs. availability. Additionally performance models trade-off accuracy vs. complexity. More complex models with larger numbers of independent variables and data samples require more time to build and compute but this investment can lead to better model accuracy.

Feedback control approaches apply control system theory to actively tune resources to meet pre-stated service level agreements (SLAs). Feedback control systems do not determine optimal configurations as they only consider a subset of all possible configurations limited by observations of configurations seen in real time. Feedback control approaches may produce inefficient configurations, particularly upon system initialization. Hybrid approaches combine performance modeling and feedback control to provide better control decisions more rapidly. Hybrid systems use training datasets to initialize performance models to better inform control decisions immediately upon start-up. Control decisions are further improved as the system operates and collects additional data in real time. Hybrid approaches often use simplified performance models trading off accuracy for speed of computation and initialization.

Wood et al. developed Sandpiper, a black-box and gray-box resource manager for VMs [31]. Sandpiper, a feedback control approach, was designed to oversee server partitioning and was not designed specifically for IaaS. Sandpiper detects “Hotspots” when provisioned architecture fails to meet service demand. Sandpiper performs only vertical scaling including increasing available resources to VMs, and VM migration to less busy PMs but does not horizontally scale the number of VMs for load balancing. Sandpiper uses a MLR performance model to predict service time by considering CPU utilization, network bandwidth utilization, page fault rate, memory utilization, request drop rate, and incoming request rate as independent variables. Xu et al. developed a resource learning approach for autonomic infrastructure management [21]. Both application agents and VM agents were used to monitor performance. A state/action table was built to record performance quality changes resulting from control events. Their resource learning approach only considered VM memory allocation, VM CPU cores, and CPU scheduler credit. An ANN model was added to predict reward values to help improve performance upon system initialization when the state/action table was only sparsely populated. Kousiouris et al. benchmarked all possible configurations for different task placements across several VMs running on a single PM [32]. From their observations they developed both a MLR model and an ANN to model performance. Their research was not extended to perform resource control but focused on performance modeling to predict the performance implications of task placements. Kousiouris et al.'s approach used an ANN to model task performance for different VM configurations on a single machine. They contrasted using a ANN model with a MLR model. Model independent variables included: CPU scheduling time, and location of tasks (same CPUs with L1 & L2 cache sharing, adjacent CPUs with L2 cache sharing, and non-adjacent CPUs). Niehorster et al. developed an autonomic resource provisioning system using support vector machines (SVMs) [27]. Their system responds to service demand changes and alters infrastructure configurations to enforce SLAs. They performed both horizontal and vertical scaling of resources and dynamically configured application specific parameters. Niehorster et al.'s performance model primarily considered application specific parameters. The only virtual infrastructure parameters considered in their performance model included # of VMs, VM memory allocation, and VM CPU cores.

4.3. CONTRIBUTIONS

Existing approaches using performance models to support autonomic infrastructure management do not adequately consider performance implications of where application components are physically hosted across VMs. Additionally, existing approaches do not consider disk utilization statistics, and only one approach has considered implications of network I/O throughput [31]. This chapter extends prior work by investigating the utility of using VM and PM resource utilization statistics as predictors for performance models for applications deployed to IaaS clouds. Use of application performance models can support determination of ideal component compositions which maximize performance using minimal resources to support autonomic SOA deployment across VMs. These performance models can also support autonomic IaaS cloud virtual infrastructure management by predicting outcomes of potential configuration changes without physically testing them. To support our investigation we modeled performance of two variants of a scientific erosion model services application. The variants serve as surrogates for common SOAs: an application-server bound SOA and a relational database bound SOA.

4.4. EXPERIMENTAL INVESTIGATION

4.4.1. Experimental Setup

The test infrastructure used to explore SOA migration in [12] was extended to explore our application performance modeling research questions presented in section 1. Two variants of the Revised Universal Soil Loss Equation – Version 2 (RUSLE2), an erosion model, were deployed as a web service and tested using a private IaaS cloud environment. RUSLE2 contains both empirical and process-based science that predicts rill and interrill soil erosion by rainfall and runoff [41]. RUSLE2 was developed primarily to guide conservation planning, inventory erosion rates, and estimate sediment delivery and is the USDA-NRCS agency standard model for sheet and rill erosion modeling used by over 3,000 field offices across the United States. RUSLE2 is a good candidate to prototype SOA performance modeling because its architecture consisting of a web server, relational database, file server, and logging server is analogous to many multi-component SOAs with diverse application component stacks.

RUSLE2 was deployed as a JAX-RS RESTful JSON-based web service hosted by Apache Tomcat [45]. The Object Modeling System 3.0 (OMS 3.0) framework [43], [52] using WINE [44] was used as middleware to support model integration and deployment as a web service. OMS was developed by the USDA–ARS in cooperation with Colorado State University and supports component-oriented simulation model development in Java, C/C++ and FORTRAN.

A Eucalyptus 2.0 [46] IaaS private cloud was built and hosted by Colorado State University consisting of 9 SUN X6270 blade servers on the same chassis sharing a private Gigabit VLAN with dual Intel Xeon X5560-quad core 2.8 GHz CPUs each with 24GB ram and 146GB HDDs. 8 blade servers were configured as Eucalyptus node-controllers, and 1 blade server was configured as the Eucalyptus cloud-controller, cluster-controller, walrus server, and storage-controller. The cloud controller server was supported by Ubuntu Linux (2.6.35-22) 64-bit server 10.10, while node controllers which hosted VMs used CentOS Linux (2.6.18) 64-bit server. Eucalyptus managed mode networking was used to isolate experimental VMs on their own private VLANs. The XEN hypervisor version 3.4.3 supported by QEMU version 0.8.2 was used to provide VMs [1]. Version 3.4.3 of the hypervisor was selected after testing indicated it provided the best performance when compared with other versions of XEN (3.1, 4.0.1, and 4.1).

To facilitate testing, ensemble runs, groups of individual modeling requests bundled together were used. To invoke the web service a client sends a JSON object representing a collection of parameterized model requests with values for management practice, slope length, steepness, latitude, and longitude. Model results are computed and returned using JSON object(s). Ensemble runs are processed by dividing grouped modeling requests into individual requests which are resent to the web service, similar to the "map" function of MapReduce. A configurable number of worker threads concurrently execute individual runs in parallel. Modeling results are then combined (reduced) and returned as a single JSON response object. A test generation program created randomized ensembles. Latitude and longitude coordinates were randomly selected within a bounding box from the U.S. state of Tennessee. Slope length, steepness, and the management practice parameters were also randomized. 20 randomly generated ensemble tests with 100 model runs each were used to test performance of 15 different service compositions. Before executing each 100 model-run ensemble test, a smaller 25 model-run ensemble test was executed to warm up the system. The warm up test was warranted after observing slow spatial query performance from postgresql on startup.

A test script was used to automatically configure service placements and collect VM and PM resource utilization statistics while executing ensemble tests. Cache clearing using the Linux virtual memory `drop_caches` function was used to purge all caches, dentries and inodes before each test was executed to negate training affects resulting from reusing ensemble tests. The validity of this approach was verified by observing CPU, file I/O, and network I/O utilization statistics for the automated tests with and without cache clearing. When caches were not cleared the number of disk sector reads dropped after the system was initially exposed to the test dataset. When caches were force-cleared the system exhibited more disk reads confirming it was forced to reread data each time. Initial experimental observations showed that as the number of records stored in the logging database increased, ensemble test performance declined. To work around performance effects of the growing logs and to eliminate running out of disk space, the Codebeamer logging component was removed and reinstalled after each ensemble test run. Additionally all log files for all application components were purged after each ensemble test. These steps allowed several thousand ensemble tests using all of the required service compositions to be automatically performed without intervention.

4.4.2. Application Components

Table 4.2 describes the four application services (components) used to implement RUSLE2's application stack. The Model M component hosts the model computation and web services using the Apache Tomcat application server. The Database D component hosts the geospatial database which resolves latitude and longitude coordinates to assist in parameterizing climate, soil, and management data for RUSLE2. Postgresql was used as a relational database and PostGIS extensions were used to support geospatial functionality [47], [48]. The file server F component was used by the RUSLE2 model to acquire XML files to parameterize data for model runs. NGINX [49], a lightweight high performance web server provided access to a library of static XML files which were on average ~5KB each. The logging L component provided historical tracking of modeling activity. The codebeamer tracking facility supported by the Derby relational database was used to log model activity [50]. A simple JAX-RS RESTful JSON-based web service was developed to decouple logging requests from the RUSLE2 service calls. This service implemented an independent logging queue to prevent logging delays from interfering with RUSLE2 performance. HAProxy was used to redirect modeling requests from a public IP to potentially one or more backend M VMs. HAProxy is a dynamically configurable very fast load balancer which supports proxying both TCP and HTTP socket-based network traffic [51].

Table 4.2. RUSLE2 Application Components

Component		Description
$\mathcal{M}$	Model	Apache Tomcat 6.0.20, Wine 1.0.1, RUSLE2, Object Modeling System (OMS 3.0)
$\mathcal{D}$	Database	Postgresql-8.4, PostGIS 1.4.0-2 Geospatial database consists of soil data (1.7 million shapes, 167 million points), management data (98 shapes, 489k points), and climate data (31k shapes, 3 million points), totaling 4.6 GB for the state of TN.
$\mathcal{F}$	File server	nginx 0.7.62 Serves XML files which parameterize the RUSLE2 model. 57,185 XML files consisting of 305MB.
$\mathcal{L}$	Logger	Codebeamer 5.5, Apache Tomcat (32-bit) Custom RESTful JSON-based logging wrapper web service. Ia-32libs support operation in 64-bit environment.

4.4.3. Tested Service Compositions

RUSLE2's application stack of 4 components can be deployed 15 possible ways across 4 physical node computers. Tables 4.3 shows the 15 service compositions tested labeled as SC1-SC15. To achieve each of the compositions a single composite VM image was created with all components installed ($\mathcal{M}$, $\mathcal{D}$, $\mathcal{F}$, $\mathcal{L}$). Four PMs were used to host one composite VM each. The testing script automatically enabled/disabled services as needed to achieve all service compositions (SC1-SC15).

Table 4.3. Tested Service Compositions

	VM 1	***VM 2***	***VM 3***	***VM 4***
SC1	$\mathcal{MDFL}$			
SC2	$\mathcal{MDF}$	$\mathcal{L}$		
SC3	$\mathcal{MD}$	$\mathcal{FL}$		
SC4	$\mathcal{MD}$	$\mathcal{F}$	$\mathcal{L}$	
SC5	$\mathcal{M}$	$\mathcal{DFL}$		
SC6	$\mathcal{M}$	$\mathcal{DF}$	$\mathcal{L}$	
SC7	$\mathcal{M}$	$\mathcal{D}$	$\mathcal{F}$	$\mathcal{L}$
SC8	$\mathcal{M}$	$\mathcal{D}$	$\mathcal{FL}$	
SC9	$\mathcal{M}$	$\mathcal{DL}$	$\mathcal{F}$	
SC10	$\mathcal{MF}$	$\mathcal{DL}$		
SC11	$\mathcal{MF}$	$\mathcal{D}$	$\mathcal{L}$	
SC12	$\mathcal{ML}$	$\mathcal{DF}$		
SC13	$\mathcal{ML}$	$\mathcal{D}$	$\mathcal{F}$	
SC14	$\mathcal{MDL}$	$\mathcal{F}$		
SC15	$\mathcal{MLF}$	$\mathcal{D}$		

Every VM ran Ubuntu Linux 9.10 64-bit server and was configured with 8 virtual CPUs, 4 GB memory and 10GB of disk space. Drawbacks to our scripted testing approach include that our composite image had to be large enough to host all components, and for some compositions VM disks contained installed but non-running components. These drawbacks are not expected to be significantly relevant to performance.

4.4.4. Resource Utilization Statistics

Table 4.4 describes the 18 resource utilization statistics collected using an automated profiling script. The profiling script parsed the Linux operating system `/proc/stat`, `/proc/diskstats`, `/proc/net/dev` and `/proc/loadavg` files. Initial resource utilization statistics were captured before execution of each ensemble test. After ensemble tests completed resource utilization statistics were captured and deltas calculated representing the resources expended throughout the duration of the ensemble test's execution. This data was recorded to a series of output files and uploaded to the dedicated blade server performing the testing. The same resource utilization statistics were captured for both VMs and PMs, but 8 statistics were found to have a negligible value for PMs. Resource utilization statistics collected for PMs are designated with "P", and for VMs with "V" in the table. Some statistics collected are likely redundant in that they are different representations of the same system properties. Subtleties in how related statistics are collected and expressed may provide performance modeling benefits and were captured for completeness is this study.

Performance models were built to predict ensemble execution time for different service compositions of RUSLE2. Using estimated average ensemble execution times for service composition rank predictions were made. Accurate performance rank predictions can be used to identify ideal compositions of components to support autonomic component deployment.

Table 4.4. Resource Utilization Statistics

Statistic		Description
P/V	CPU time	CPU time in ms
P/V	cpu usr	CPU time in user mode in ms
P/V	cpu krn	CPU time in kernel mode in ms
P/V	cpu_idle	CPU idle time in ms
P/V	contextsw	Number of context switches
P/V	cpu_io_wait	CPU time waiting for I/O to complete
P/V	cpu_sint_time	CPU time servicing soft interrupts
V	Dsr	Disk sector reads (1 sector = 512 bytes)
V	Dsreads	Number of completed disk reads
V	Drm	Number of adjacent disk reads merged
V	readtime	Time in ms spent reading from disk
V	Dsw	Disk sector writes (1 sector = 512 bytes)
V	dswrites	Number of completed disk writes
V	Dwm	Number of adjacent disk writes merged
V	writetime	Time in ms spent writing to disk
P/V	Nbr	Network bytes sent
P/V	Nbs	Network bytes received
P/V	Loadavg	Avg # of running processes in last 60 sec

4.4.5. Application Variants

Our investigation tested two variants of RUSLE2 which we refer to herein as the "d-bound" for the database bound and the "m-bound" for the model bound application. By testing two variants of RUSLE2 we hoped to gain insight into performance modeling by using two versions of RUSLE2 with different resource utilization profiles. For the d-bound RUSLE2, two primary geospatial queries were modified to perform a join on a nested query (as opposed to a table). The m-bound RUSLE2's geospatial queries used the ordinary table joins. The SC1 "d-bound" deployment required on average 104% more CPU time and 17,962% more disk sector reads (dsr) than the "m-bound" model. This modification significantly increased database CPU time and disk reads. Average ensemble execution time for all service compositions was approximately ~29.3 seconds for the m-bound model, and 4.7x greater at ~137.2 seconds for the d-bound model.

4.5. EXPERIMENTAL RESULTS

Table 4.5 summarizes tests completed for this study totaling approximately 300,000 model runs in 3,000 ensemble tests. The effectiveness of using the resource utilization statistics from table 4.4 as independent variables to predict service composition performance (RQ1) are presented in section 5.1. Section 5.2 discusses experimental results which investigate how to best compose resource utilization statistics for use in performance models (RQ2). Section 5.3 concludes by presenting results of performance model effectiveness for predicting ensemble execution time and service composition performance ranks for different application component compositions (RQ3).

Table 4.5. Summary of Tests

Model	**Trials**	**Ensembles /Trial**	**Service Comps.**	**Model Runs**	**Ens. Runs**
d-bound	2	20	15	60k	600
m-bound	3	20	15	90k	900
m-bound	1	100	15	150k	1,500
Totals	**6**			**300,000**	**3,000**

4.5.1. Independent Variables

This study investigated the utility of resource utilization statistics describing CPU utilization, disk I/O, and network I/O of both VMs and PMs for performance modeling as described in table 4.4. To investigate the predictive strength of each independent variable we performed separate linear regressions for each independent variable to predict ensemble execution time. R^2 is a measure of model quality which describes the percentage of variance explained by the model's independent variable. Adjusted R^2 is reported opposed to multiple R^2 because adjusted R^2 is more conservative as it includes an adjustment which takes into account the number of predictors in the model [53]. Statistics reported in table 4.6 used 20 ensemble runs each for the 15 service compositions for both the "m-bound" and "d-bound" models. Untested statistics are indicated by "n/a". In these cases resource utilization was typically zero. Total resource utilization statistics were calculated by totaling values from VMs and PMs used in the service compositions.

Table 4.6. Independent Variable Strength

Statistic	**Adjusted R^2 "m-bound"**		**Adjusted R^2 "d-bound"**	
	VM	*PM*	*VM*	*PM*
CPUtime	0.7162	-0.0033	0.5096	0.1406
cpuusr	0.7006	-0.0019	0.444	0.04437
Dsr	0.3693	n/a	**0.02613**	n/a
dsreads	0.3129	n/a	**0.02606**	n/a
cpukrn	0.1814	n/a	0.2958	0.2221
dswrites	0.1705	n/a	0.1151	n/a
Dsw	0.1412	n/a	0.02292	n/a
dwm	0.1374	n/a	0.01528	n/a
contextsw	0.0618	-0.001	**0.4592**	0.1775
cpu_io_wait	0.0514	0.086	0.02528	0.05718
writetime	0.0451	n/a	-0.001199	n/a
loadavg	0.0168	0.0132	0.04321	0.004962
cpu_sint_time	0.0112	0.0141	0.02251	0.00003713
readtime	0.0094	n/a	0.02753	n/a
Nbs	0.0042	0.0039	0.01852	0.3385
Nbr	0.0041	n/a	0.01858	0.3368
cpu_idle	0.004	-0.0001	**0.2468**	0.2542
Drm	0.0005	n/a	0.0261	n/a
Total R^2	2.938	0.1109	2.341	1.576

CPU time is shown to predict the most variance for both models. Large differences in R^2 for "d-bound" compared to "m-bound" are shown in bold. For the "d-bound" model *dsr* and *dsreads* were less useful predictors, while *contextsw* and cpu_idle were shown to be better predictors. PM resource utilization statistics were generally found to be less useful as indicated by total R^2 values. No single PM statistic for the "m-bound" model achieved better than

R^2=.086, while PM statistics for the "d-bound" model appeared better but not great with nbs as the strongest predictor at R^2=.3385.

Besides having strong R^2 values, good predictor variables for use in MLRs should have normally distributed data. To test normality of our resource utilization statistics the Shapiro-Wilk normality test was used [54]. 100 ensemble runs were made for each of the 15 service compositions for the "m-bound" model. Combining service composition data together was shown to decrease normality. Normality tests showed an average of 9 resource utilization statistics had normal distributions for individual compositions. When data for compositions was combined only *loadavg*, *cpu_sint_time*, and *cpu_krn* had strong normal distributions for the "m-bound" model and loadavg, *CPUtime*, *cpu_usr*, and *cpu_krn* for the "d-bound" model. Ensemble time appeared to be normally distributed for both applications, but appeared more strongly normally distributed for "d-bound". Histogram plots for CPU time and *dsr* are shown in figure 4.1. *CPUtime* and other related CPU time statistics (*cpu_usr*, *cpu_krn*) were among the strongest predictors of ensemble execution time for both models. *Dsr* was a better predictor for "m-bound" and its distribution appears more normal than for "d-bound". The plots visually confirm results of the Shapiro-Wilk normality tests.

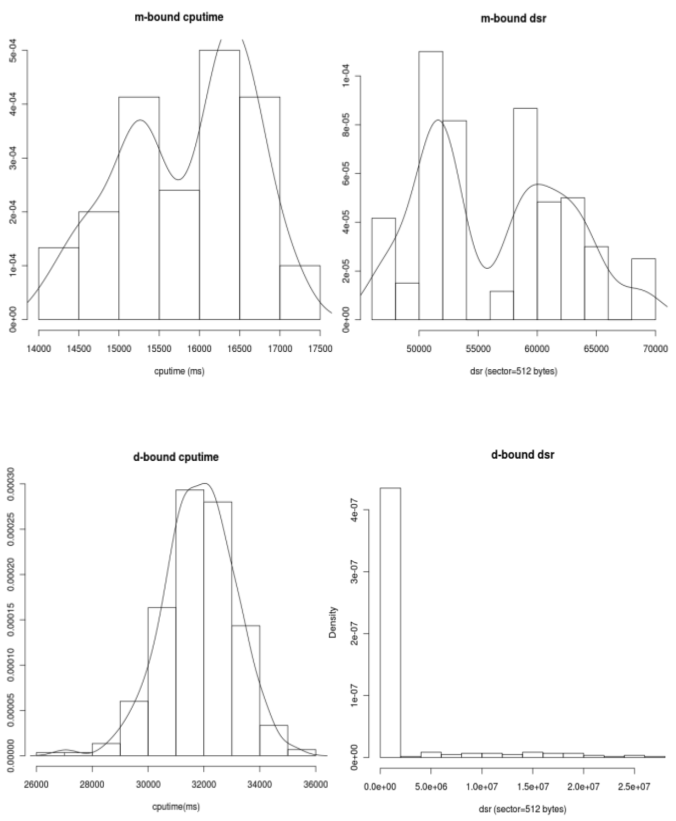


Figure 4.1. CPU time and Disk Sector Read Distribution Plots

4.5.2. Treatment of Resource Utilization Data

The RUSLE2 application's 4 components ($\mathcal{M}$, $\mathcal{D}$, $\mathcal{F}$, $\mathcal{L}$) were distributed across 1 to 4 VMs. Resource utilization statistics were collected at the VM and PM level. Two treatments of the data are possible. Resource utilization statistics can be combined for all VMs and used to model performance: $RU_{data}=RU_{\mathcal{M}}+RU_{\mathcal{D}}+RU_{\mathcal{F}}+RU_{\mathcal{L}}$ or only resource utilization statistics for the VM hosting a particular component can be used to model performance: $RU_{data}=\{RU_{\mathcal{M}};\ RU_{\mathcal{D}};\ RU_{\mathcal{F}};\ RU_{\mathcal{L}};\}$ To test the utility of both data handling approaches 10 MLR models were generated. A separate training and test data set were collected using 20 ensemble runs for each of the 15 service compositions for both the "m-bound" and "d-bound" RUSLE2. Results of the MLR models are summarized in table 4.7.

Table 4.7. Multiple Linear Regression Performance Models

Model	**Data**	**Adj. R^2**	**RMS_{train}**	**RMS_{test}**	**Avg. Rank Error**
d-bound	$RU_{\mathcal{M}}$	.9982	642.78	967.35	.13
d-bound	$RU_{\mathcal{D}}$	.9983	622.24	1248.24	.4
d-bound	$RU_{\mathcal{F}}$	.9984	615.64	606.94	.27
d-bound	$RU_{\mathcal{L}}$	.9983	621.99	978.92	.4
d-bound	$RU_{\mathcal{MDFL}}$	.9107	4532.85	44903.96	1.73
m-bound	$RU_{\mathcal{M}}$	.8733	576.05	759.36	1.47
m-bound	$RU_{\mathcal{D}}$	.67	929.54	971.85	2.13
m-bound	$RU_{\mathcal{F}}$	.7833	775.70	866.18	2
m-bound	$RU_{\mathcal{L}}$	.6247	991.29	42570.5	2.4
m-bound	$RU_{\mathcal{MDFL}}$	.8546	616.98	807.34	1.2

For the models described in table 4.7 VM data (not PM) for all 18 independent variables was used. Adjusted R^2 values describe the variance explained by the models. The root mean squared error (RMS) expresses the differences between the predicted and observed values and serves to provide a measure of model accuracy. A statistically significant model ($p<.05$) will predict 95% of ensemble execution times with less than +/- 2 RMS error from the actual values [55]. RMS_{train} describes error at predicting ensemble times for the training dataset and RMS_{test} describes error at predicting ensemble times using the test dataset. For compositions an average estimate for ensemble execution time was calculated. The estimated average ensemble execution time was used to generate performance rank predictions for each of the 15 service compositions. The average rank error is the average error of actual vs. predicted ranks.

Analysis of model results shows that for the "d-bound" performance model, *CPU_idle* time from individual VMs is an excellent predictor of ensemble execution time. R^2 for *cpu_idle_time* for the $\mathcal{M}$, $\mathcal{D}$, $\mathcal{F}$, $\mathcal{L}$ models is .7716, .7844, .6041, and .4223 respectively but only .2468 when combining VM statistics. This is in contrast to .0223, -.0024, .0271, .1199 and combined .0039 for the "m-bound" model. Further analysis reveals that the "d-bound" model makes 93.6x more disk sector reads than "m-bound" but only requires 2x as much *CPUtime* while having 5.1x more idle *CPUtime*. The "d-bound" model waits while this I/O is occurring making *CPU_idle* time an excellent predictor for ensemble execution time for the "d-bound" application. The number of context switches for the busiest component seems to be a good predictor with $\mathcal{D}$ for "d-bound" at R^2=.4619 and $\mathcal{M}$ for the "m-bound" at R^2=.4786. The strength of using the number of context switches as a predictor of other VMs was less significant.

4.5.3. Performance Models

Combined resource utilization statistics ($RU_{\mathcal{MDFL}}$) were used as training data for 4 modeling approaches: MLR, stepwise multiple linear regression (MLR-step), MARS, and a simple single hidden layer ANN [54]. We investigated both MLR and stepwise MLR. MLR models use every independent variable provided to predict the dependent variable. Stepwise MLR begins by modeling the dependent variable using the complete set of independent variables but after each step adds or drops predictors based on their significance to test various combinations until the best model is found which explains the most variance (R^2). MARS is an adaptive extension of MLR which works by splitting independent variables into multiple basis functions and then fits a linear regression model to those basis functions. Basis functions used by MARS are piecewise linear functions in the form of: f(x)={x-t if x>t, 0 otherwise} and g(x)=(t-x if x<t, 0 otherwise}. Both stepwise MLR and MARS were chosen because they generally provide some small improvement over traditional MLR and were easy to implement in R. ANNs are a very popular statistical modeling technique which excels at handling complex nonlinear relationships in the data. We tested single hidden layer ANN models supported by the R statistical software package to predict ensemble execution times. R's ANNs use the sigmoid function, a bounded logistic function used to introduce nonlinearity in the model. A summary of performance models for the "m-bound" and "d-bound" application are shown in table 4.8.

Table 4.8. Performance Models

Model	Type	Adj. R^2	RMS_{train}	RMS_{test}	Avg. Rank Error
d-bound	MLR	.9107	4532.85	44903.96	1.73
d-bound	MLR-step	.9118	4589.27	43918.55	1.73
d-bound	MARS	.9180	4472.32	45137.28	1.33
d-bound	ANN	n/a	4440.03	44094.03	1.6
m-bound	MLR	.8546	616.98	807.34	1.2
m-bound	MLR-step	.8571	621.41	799.22	1.33
m-bound	MARS	.8718	596.45	825.34	1.86
m-bound	ANN	n/a	595.49	800.71	1.73

R^2 values were not available for the ANN. For both applications, the ANN provided the lowest RMS error for the training dataset but slightly higher RMS error for the test dataset compared with stepwise MLR. For the 8 models RMS_{train} and RMS_{test} values correlated strongly (R^2=.999, p=2.4•10^{-10}, df=6) suggesting that where a model performs well on training data it will likely perform well on test data. There was no relationships between rank error and RMS_{test} (R^2=.02064, p=.734, df=6) suggesting that low error for ensemble time predictions does not guarantee low rank error. All of the models had some error at predicting service composition rank but provided functional predictions as they easily differentiated fast vs. slow service compositions and accurately determined the top 2 or 3 compositions.

4.6. CONCLUSIONS

Modeling performance of component compositions of SOAs deployed to IaaS clouds can help guide component deployment to provide best performance using with minimal virtual resources. Results of our exploratory investigation on performance modeling using resource utilization statistics for two variants of soil erosion model services application include:

(RQ-1) CPU time and other CPU related statistics were the strongest predictors of execution time, while disk and network I/O statistics were less useful. Measured disk and network I/O utilization statistics for our study suffered from non-normality and large variance when data from multiple service compositions were combined together for modeling purposes. CPU idle time and number of context switches were good predictors of execution time when the application's performance was I/O bound. Disk I/O statistics were better predictors when the application was more CPU bound.

(RQ-2) The best treatment of resource utilization statistics for performance modeling, either combining data or using VM data separately, to achieve best model accuracy was dependent on each application's resource utilization profile.

(RQ-3) Advanced modeling techniques such as MARS and ANN provided lower $\text{RMS}_{\text{error}}$ for training and test data sets than MLR but overall all of the modeling approaches tested had similarly performance at minimizing $\text{RMS}_{\text{error}}$. Additionally all models determined the best 2 or 3 service compositions confirming the value of our performance modeling approach for determining ideal component compositions to support IaaS cloud SOA deployment.

CHAPTER 5

PERFORMANCE IMPLICATIONS OF COMPONENT COMPOSITIONS

5.1. INTRODUCTION

Migration of service oriented applications (SOAs) to Infrastructure-as-a-Service (IaaS) clouds involves deploying components of application infrastructure to one or more virtual machine (VM) images. Images are used to instantiate VMs to provide the application's cloud-based infrastructure. Application components consist of infrastructure elements such as web/application servers, proxy servers, NO SQL databases, distributed caches, relational databases, file servers and others.

Service isolation refers to the total separation of application components for hosting using separate VMs. Application VMs are hosted by one or more physical machines (PMs). Service isolation provides application components with their own explicit sandboxes to operate in, each having independent operating system instances. *Hardware virtualization* enables service isolation using separate VMs to host each application component instance. Before virtualization, service isolation using PMs required significant server capacity. Service isolation has been suggested as a best practice for deploying multi-tier applications across VMs. A 2010 Amazon Web Services white paper suggests applications be deployed using service isolation. The white paper instructs the user to "bundle the logical construct of a component into an Amazon Machine Image so that it can be deployed (instantiated) more often" [56]. Service isolation, a 1:1 mapping of application component(s) to VM images is implied. Service isolation enables scalability and supports fault tolerance at the component level. Isolating components may reduce inter-component interference allowing them to run more efficiently. Conversely service isolation adds an abstraction layer above the physical hardware which introduces overhead potentially degrading performance. Deploying all application components using separate VMs may increase network traffic, particularly when VMs are hosted by separate physical machines. Consolidating components together on a single VM guarantees they will not be physically separated when deployed potentially improving performance by reducing network traffic.

Provisioning variation results from the non-determinism of where application VMs are physically hosted in the cloud, often resulting in performance variability [16], [18], [20]. IaaS cloud providers often do not allow users to control where VMs are physically hosted causing this provisioning variation. Clouds consisting of PMs with heterogeneous hardware and hosting a variable number of VMs complicates benchmarking application performance [57].

Service Isolation provides isolation at the guest operating system level as VMs share physical hardware resources and compete for CPU, disk, and network bandwidth. Quantifying VM interference and investigation of approaches to multiplex physical host resources are active areas of research [4], [58]–[63]. Current virtualization technology only guarantees VM memory isolation. VMs reserve a fixed quantity of memory for exclusive use which is not released until VM termination. Processor, network I/O, and disk I/O resources are shared through coordination by the virtualization hypervisor. Popular virtualization hypervisors include kernel-based VMs (KVM), Xen, and the VMware ESX hypervisor. Hypervisors vary with respect to methods used to multiplex resources. Some allow pinning VMs to specific CPU cores to guarantee resource availability though CPU caches are still shared [60]. Developing mechanisms which guarantee fixed quantities of network and disk throughput for VM guests is an open area for research.

This research investigates performance of SOA component deployments to IaaS clouds to better understand implications of component distribution across VMs, VM placement across physical hosts and VM configuration. We seek to better understand factors that impact performance moving towards building performance models to support intelligent methodologies that better load balance resources to improve application performance. We investigate hosting two variants of a non-stochastic multitier application with stable resource utilization characteristics. Resource utilization statistics that we capture from host VMs are then used to investigate performance implications relative to resource use and contention. The following research questions are investigated:

RQ-1: How does resource utilization and application performance vary relative to how application components are deployed? How does provisioning variation, the placement of VMs across physical hosts, impact performance?

RQ-2: Does increasing VM memory allocation change performance? Does the virtual machine hypervisor (Xen vs. KVM) affect performance?

RQ-3: How much overhead results from VM service isolation?

RQ-4: Can VM resource utilization data be used to build models to predict application performance of component deployments?

5.2. RELATED WORK

Rouk identified the challenge of finding ideal service compositions for creating virtual machine images to deploy applications in cloud environments in [35]. Schad et al. [18] demonstrated the unpredictability of Amazon EC2 VM performance caused by contention for physical machine resources and provisioning variation of VMs. Rehman et al. tested the effects of resource contention on Hadoop-based MapReduce performance by using IaaS-based cloud VMs to host worker nodes [16]. They tested provisioning variation of three different deployment schemes of VM-hosted Hadoop worker nodes and observed performance degradation when too many worker nodes were physically co-located. Their work investigated VM deployments not for SOAs, but for MapReduce jobs where all VMs were homogeneous in nature. SOAs with multiple unique components present a more complex challenge for resource provisioning than studied by Rehman et al. Zaharia et al. observed that Hadoop's native scheduler caused severe performance degradation by ignoring resource contention among Hadoop nodes hosted by Amazon EC2 VMs [20]. They proposed the Longest Approximate Time to End (LATE) scheduling algorithm which better addresses performance variations of heterogeneous Amazon EC2 VMs. Their work did not consider hosting of heterogeneous components.

Camargos et al. investigated virtualization hypervisor performance for virtualizing Linux servers with several performance benchmarks for CPU, file and network I/O [3]. Xen, KVM, VirtualBox, and two container based virtualization approaches OpenVZ and Linux V-Server were tested. Different parts of the system were targeted using kernel compilation, file transfers, and file compression benchmarks. Armstrong and Djemame investigated performance of VM launch time using Nimbus and OpenNebula, two IaaS cloud infrastructure managers [40]. Additionally they benchmarked Xen and KVM paravirtual I/O performance. Jayasinghe et al. investigated performance of the RUBBoS n-tier e-commerce system deployed to three different IaaS clouds: Amazon EC2, Emulab, and Open Cirrus [64]. They tested horizontal scaling, changing the number of VMs for each component, and vertical scaling, varying the resource allocations of VMs. They did not investigate consolidating components on VMs but used separate VMs for full service isolation. Matthews et al. developed a VM isolation benchmark to quantify the isolation level of co-located VMs running several conflicting tasks [4]. They tested VMWare, Xen, and OpenVZ hypervisors to quantify isolation. Somani and Chaudhary benchmarked Xen VM performance with co-located VMs running CPU, disk, or network intensive tasks on a single physical host [58]. They benchmarked the Simple Earliest Deadline First (SEDF) I/O credit scheduler vs. the default Xen credit scheduler and investigated physical resource contention for running different co-located tasks, similar to resource contention of co-hosting different components of SOAs. Raj et al. improved hardware level cache management of the Hyper-V hypervisor introducing VM core assignment and cache portioning to reduce inter-VM conflicts from sharing the same hardware caches. These improvements were shown to improve VM isolation [59].

Niehörster et al. developed an autonomic system using support vector machines (SVM) to meet predetermined quality-of-service (QoS) goals. Service specific agents were used to provide horizontal and vertical scaling of virtualization resources hosted by an IaaS Eucalyptus cloud [27]. Their agents scaled # of VMs, memory, and virtual core allocations. Support vector machines determined if resource requirements were adequate for the QoS requirement. They tested their approach by dynamically scaling the number of modeling engines for GROMACS, a molecular dynamics simulation and also for an Apache web application service to meet QoS goals. Sharma et al. investigated implications of physical placement of non-parallel tasks and their resource requirements to build performance model(s) to improve task scheduling and distribution on compute clusters [65]. Similar to Sharma's models to improve task placement, RQ-4 investigates building performance models which could be used to guide component deployments for multitier applications.

Previous studies have investigated a variety of related issues but none have investigated the relationship between application performance and resource utilization (CPU, disk, network) resulting from how components of SOAs are deployed across VMs (isolation vs. consolidation).

5.3. CHAPTER CONTRIBUTIONS

This chapter presents a thorough and detailed investigation on how the deployment of SOA components impacts application performance and resource consumption (CPU, disk, network). This work extends prior research on provisioning variation and heterogeneity of cloud-based resources. Relationships between component and VM placement, resource utilization and application performance are investigated. Additionally we investigate performance and resource utilization changes resulting from: (1) the use of different hypervisors (Xen vs. KVM), and (2) increasing VM memory allocation. Overhead from using separate VMs to host application components is also measured. Relationships between resource utilization and performance are used to develop a multiple linear regression model to predict application performance. Our approach for collecting application resource utilization data to construct performance model(s) can be generalized for any SOA.

5.4. EXPERIMENTAL DESIGN

To support investigation of our research questions we studied the migration of a widely used Windows desktop environmental modeling application deployed to operate as a multi-tier web services application. Section 4.1 describes the application and our test harness. Section 4.2 describes components of the multitier application. Section 4.3 details the configuration of tested component deployments. Section 4.4 concludes by describing our private IaaS cloud and hardware configuration used for this investigation.

5.4.1. Test Application

For our investigation we utilized two variants of the RUSLE2 (Revised Universal Soil Loss Equation — Version 2) soil erosion model [41]. RUSLE2 contains both empirical and process-based science that predicts rill and interrill soil erosion by rainfall and runoff. RUSLE2 was developed to guide conservation planning, inventory erosion rates, and estimate sediment delivery. RUSLE2 is the US Department of Agriculture Natural Resources Conservation Service (USDA-NRCS) agency standard model for sheet and rill erosion modeling used by over 3000 field offices across the United States. RUSLE2 was originally developed as a Windows based Microsoft Visual C++ desktop application and has been extended to provide soil erosion modeling as a REST-based webservice hosted by Apache Tomcat [45]. JSON was the transport protocol for data objects. To facilitate functioning as a web service a command line console was added. RUSLE2 consists of four tiers including an application server, a geospatial relational database, a file server, and a logging server. RUSLE2 is a good multi-component application for our investigation because with four components and 15 possible deployments it is both complex enough to be interesting, yet simple enough that brute force testing is reasonable to accomplish. RUSLE2's architecture is a surrogate for traditional client/server architectures having both an application and relational database. The Object Modeling System 3.0 (OMS3) framework [42] [43] using WINE [44] provided middleware to facilitate interacting with RUSLE2's command line console. OMS3, developed by the USDA-ARS in cooperation with Colorado State University, supports component-oriented simulation model development in Java, C/C++ and FORTRAN.

The RUSLE2 web service supports ensemble runs which are groups of individual model requests bundled together. To invoke the RUSLE2 web service a client sends a JSON object with parameters describing land management practices, slope length, steepness, latitude, and longitude. Model results are returned as JSON objects. Ensemble runs are processed by dividing sets of modeling requests into individual requests which are resent to the web service, similar to the ''map'' function of MapReduce. These requests are distributed to worker nodes using a round robin proxy server. Results from individual runs of the ensemble are ‘‘reduced’’ into a single JSON response object. A test generation program created randomized ensemble tests. Latitude and longitude coordinates were randomly selected within a bounding box from the state of Tennessee. Slope length, steepness, and land management practice parameters were randomized. Random selection of latitude and longitude coordinates led to variable geospatial query execution times because the polygons intersected with varied in complexity. To verify our test generation technique produced test sets with variable complexity we completed 2 runs of 20 randomly generated 100-model run ensemble tests run using the 15 RUSLE2 component deployments and average execution times were calculated. Execution speed (slow/medium/fast) of ensemble tests was preserved across subsequent runs indicating that individual ensembles exhibited a complexity-like characteristic ($R^2 = 0.914$, $df = 18$, $p = 5 \cdot 10^{-11}$).

Our investigation utilized two variants of RUSLE2 referred to as ‘‘d-bound’’ for the database bound variant and ‘‘m-bound’’ for the model bound variant, names based on the component dominating execution time. These application variants represent surrogates for two potentially common scenarios in practice: an application bound by the database tier, and an application bound by the middleware (model) tier. For the ‘‘d-bound’’ RUSLE2 two primary geospatial queries were modified to perform a join on a nested query. The ‘‘m-bound’’ variant was unmodified. The ‘‘d-bound’’ application had a different resource utilization profile than the ‘‘m-bound’’ RUSLE2. On average the ‘‘d-bound’’ application required ~2.45 x more CPU time than the ‘‘m-bound’’ model.

Table 5.1. RUSLE2 Application Components

Component		Description
$\mathcal{M}$	Model	Apache Tomcat 6.0.20, Wine 1.0.1, RUSLE2, Object Modeling System (OMS 3.0)
$\mathcal{D}$	Database	Postgresql-8.4, PostGIS 1.4.0-2 Geospatial database consists of soil data (1.7 million shapes, 167 million points), management data (98 shapes, 489k points), and climate data (31k shapes, 3 million points), totaling 4.6 GB for the state of TN.
$\mathcal{F}$	File server	nginx 0.7.62 Serves XML files which parameterize the RUSLE2 model. 57,185 XML files consisting of 305MB.
$\mathcal{L}$	Logger	Codebeamer 5.5 w/ Derby DB, Tomcat (32-bit) Custom RESTful JSON-based logging wrapper web service. IA-32libs support operation in 64-bit environment.

5.4.2. Application Services

Table 5.1 describes the application components of RUSLE2's application stack. The $\mathcal{M}$ component provides model computation and web services using Apache Tomcat. The $\mathcal{D}$ component implements the geospatial database which resolves latitude and longitude coordinates to assist in providing climate, soil, and management data for RUSLE2 model runs. PostgreSQL with PostGIS extensions were used to support geospatial functionality [47], [48]. The file server $\mathcal{F}$ component provides static XML files to RUSLE2 to parameterize model runs. NGINX [49], a lightweight high performance web server hosted over 57,000 static XML files on average ~5 kB each. The logging $\mathcal{L}$ component provided historical tracking of modeling activity. The Codebeamer tracking facility which provides an extensive customizable GUI and reporting facility was used to log model activity [50]. A simple JAX-RS RESTful JSON-based web service decoupled logging functions from RUSLE2 by providing a logging queue to prevent delays from interfering with model execution. Codebeamer was hosted by the Apache Tomcat web application server and used the Derby filebased relational database. Codebeamer, a 32-bit web application, required the Linux 32-bit compatibility libraries (ia32-libs) to run on 64-bit VMs. A physical server running the HAProxy load balancer provided a proxy service to redirect modeling requests to the VM hosting the modeling engine. HAProxy is a dynamically configurable fast load balancer that supports proxying both TCP and HTTP socket-based network traffic [51].

Table 5.2. Tested Component Deployments

	VM1	VM2	VM3	VM4
SC1	$\mathcal{MDFL}$			
SC2	$\mathcal{MDF}$	$\mathcal{L}$		
SC3	$\mathcal{MD}$	$\mathcal{FL}$		
SC4	$\mathcal{MD}$	$\mathcal{F}$	$\mathcal{L}$	
SC5	$\mathcal{M}$	$\mathcal{DFL}$		
SC6	$\mathcal{M}$	$\mathcal{DF}$	$\mathcal{L}$	
SC7	$\mathcal{M}$	$\mathcal{D}$	$\mathcal{F}$	$\mathcal{L}$
SC8	$\mathcal{M}$	$\mathcal{D}$	$\mathcal{FL}$	
SC9	$\mathcal{M}$	$\mathcal{DL}$	$\mathcal{F}$	
SC10	$\mathcal{MF}$	$\mathcal{DL}$		
SC11	$\mathcal{MF}$	$\mathcal{D}$	$\mathcal{L}$	
SC12	$\mathcal{ML}$	$\mathcal{DF}$		
SC13	$\mathcal{ML}$	$\mathcal{D}$	$\mathcal{F}$	
SC14	$\mathcal{MDL}$	$\mathcal{F}$		
SC15	$\mathcal{MLF}$	$\mathcal{D}$		

5.4.3. Service Configurations

RUSLE2's infrastructure components can be deployed 15 possible ways using 1–4 VMs. Table 5.2 shows the tested service configurations labeled as SC1–SC15. To create the deployments for testing, a composite VM image with all (4) application components installed was used. An automated test script enabled/disabled application components as needed to achieve the configurations. This method allowed automatic configuration of all component deployments using a single VM image. This approach required that the composite disk image was large enough to host all components, and that VMs had installed but non-running components.

For testing SC1–SC15, VMs were deployed with physical isolation. Each VM was hosted by its own exclusive physical host. This simplified the experimental setup and provided a controlled environment using homogeneous physical host machines to support experimentation without interference from external non-application VMs. For provisioning variation testing (RQ-

1) and service isolation testing (RQ-3) physical machines hosted multiple VMs as needed. For all the tests VMs had 8 virtual CPUs, and 10 GB of disk space regardless of the number of components hosted. VMs were configured with either 4 GB or 10 GB memory.

Table 5.3 describes component deployments used to benchmark service isolation overhead (RQ-3). Separate VMs are delineated using brackets. These tests measured performance overhead resulting from the use of separate VMs to isolate application components. Service isolation overhead was measured for the three fastest component deployments: SC2, SC6, and SC11.

Table 5.3. Service Isolation Tests

NC	NODE 1	NODE 2	NODE 3
SC2-SI	[$\mathcal{M}$] [$\mathcal{D}$] [$\mathcal{F}$]	[$\mathcal{L}$]	
SC2	[$\mathcal{M}$ $\mathcal{D}$ $\mathcal{F}$]	[$\mathcal{L}$]	
SC6-SI	[$\mathcal{M}$]	[$\mathcal{D}$ $\mathcal{F}$]	[$\mathcal{L}$]
SC6	[$\mathcal{M}$]	[$\mathcal{D}$] [$\mathcal{F}$]	[$\mathcal{L}$]
SC11-SI	[$\mathcal{M}$] [$\mathcal{F}$]	[$\mathcal{D}$]	[$\mathcal{L}$]
SC11	[$\mathcal{M}$ $\mathcal{F}$]	[$\mathcal{D}$]	[$\mathcal{L}$]

5.4.4. Testing Setup

A Eucalyptus 2.0 IaaS private cloud [46] was built and hosted by Colorado State University consisting of 9 SUN X6270 blade servers sharing a private 1 Giga-bit VLAN. Servers had dual Intel Xeon X5560-quad core 2.8 GHz CPUs, 24 GB RAM, and two 15 000 rpm HDDs of 145 GB and 465 GB capacity respectively. The host operating system was CentOS 5.6 Linux (2.6.18-274) 64-bit server for the Xen hypervisor [1] and Ubuntu Linux 10.10 64-bit server (2.6.35-22) for the KVM hypervisor. VM guests ran Ubuntu Linux (2.6.31-22) 64-bit server 9.10. Eight servers were configured as Eucalyptus node-controllers, and one server was configured as the Eucalyptus cloud-controller, cluster-controller, walrus server, and storage-controller. Eucalyptus managed mode networking using a managed Ethernet switch was used to isolate VMs onto their own private VLANs.

Table 5.4. Hypervisor Performance

Hypervisor	Avg. Time (sec)	Performance
Physical server	15.65	100%
Xen 3.1	25.39	162.24%
Xen 3.4.3	23.35	149.20%
Xen 4.0.1	26.2	167.41%
Xen 4.1.1	27.04	172.78%
Xen 3.4.3 hvm	32.1	205.11%
KVM disk virtio	31.86	203.58%
KVM no virtio	32.39	206.96%
KVM net virtio	35.36	225.94%

Available versions of the Xen and KVM hypervisors were tested to establish which provided the fastest performance using SC1 from Table 5.2. Ten trials of an identical 100-model run ensemble test were executed using the ‘‘m-bound’’ variant of the RUSLE2 application and average ensemble execution times are shown in Table 5.4. Xen 3.4.3 hvm represents the Xen hypervisor running in full virtualization mode using CPU virtualization extensions similar to the KVM hypervisor. Xen 3.4.3 using paravirtualization was shown to provide the best performance and was used for the majority of experimental tests. Our application-based benchmarks of Xen and KVM reflect similar results from previous investigations [3], [40].

The Linux virtual memory drop_caches function was used to clear all caches, dentries and inodes before each ensemble test to negate training effects from repeating identical ensemble tests. This cache-flushing technique was verified by observing CPU, file I/O, and network I/O utilization for the automated tests with and without cache clearing. When caches were not cleared, total disk sector reads decreased after the system was initially exposed to the same ensemble test. When caches were force-cleared for each ensemble run, the system reread data. As the test harness was exercised we observed that Codebeamer’s Derby database grew large resulting in performance degradations. To eliminate decreased performance from log file and database growth our test script deleted log files and removed and reinstalled Codebeamer after each ensemble run. These steps prevented out of disk space errors and allowed uninterrupted testing without intervention.

VM resource utilization statistics were captured using a profiling script to capture CPU time, disk sector reads and writes (disk sector = 512 bytes), and network bytes sent/received. To determine resource utilization of component deployments from all VMs hosting the application were totaled.

5.5. EXPERIMENTAL RESULTS

To investigate our research questions we completed nearly 10,000 ensemble tests totaling ~1,000,000 individual model runs. Tests were conducted using both the ''m-bound'' and ''d-bound'' RUSLE2 model variants. VMs were hosted using either the Xen or KVM hypervisor and were configured with either 4 GB or 10 GB memory, 8 virtual cores, and 10 GB disk space. 15 component placements across VMs were tested, and these VMs were provisioned using physical hosts 45 different ways. Test sets executed 20 ensembles of 100 model runs each to benchmark performance and resource utilization of various configurations. All ensembles had 100 randomly generated model runs. Some test sets repeated the same ensemble test 20 times, while others used a set of 20 different ensemble tests for a total of 2,000 randomly generated model runs per test set. Results for our investigation of RQ-1 are described in Sections 5.1–5.3. Resource utilization characteristics of the component deployments are described in Section 5.1 followed by performance results of the deployments in Section 5.2. Section 5.3 reports on performance effects from provisioning variation, the variability resulting from where application

VMs are physically hosted. Section 5.4 describes how application performance changed whenVM memory was increased from 4 GB to 10 GB, and Section 5.5 reports on the performance differences of the Xen and KVM hypervisors (RQ-2). Section 5.6 presents results from our experiment measuring service isolation overhead (RQ-3). Section 5.7 concludes by presenting our multiple linear regression based performance model which predicts performance of component deployments based on resource utilization statistics (RQ-4).

5.5.1. Component deployment resource utilization

Resource utilization statistics were captured for all component deployments to investigate how they varied across all possible configurations. To validate that component deployments exhibited consistent resource utilization behavior, linear regression was used to compare two separate sets of runs consisting of 20 different 100-model run ensembles using the ‘‘m-bound’’ model with 4 GB Xen VMs. The coefficient of determination R^2 was calculated to determine the proportion of variance accounted for when regressing together the two datasets. Higher values indicate similarity in the datasets. Comparing R^2 resource utilization for CPU time (R^2 = 0.937904, df = 298), disk sector reads (R^2 = 0.96413, df = 298), and network bytes received/sent (R^2 = 0.99999, df = 298) for repeated tests appeared very similar. Only disk sector writes (R^2 = 0.273696, df = 298) was inconsistent. Network utilization appeared similar for both the ‘‘m-bound’’ and ‘‘d-bound’’ model variants as they communicated the same information. For the ‘‘d-bound’’ model $\mathcal{D}$ performed many more queries but this additional computation was independent of the other components $\mathcal{M}$ $\mathcal{F}$ $\mathcal{L}$.

Application performance and resource utilization varied based on the deployment configuration of application components. Comparing resource utilization among deployments for the ‘‘m-bound’’ model network bytes sent/received varied by ~144%, disk sector writes by ~22%, disk sector reads by ~15% and CPU time by ~6.5% as shown in Table 5.5. Comparing the fastest and slowest deployments the performance variation was ~3.2 s, nearly 14% of the average ensemble execution time for all deployments. Resource utilization differences among deployments of the “d-bound” model was greater than ‘‘m-bound’’ with ˜820% for disk sector reads, ~145% for network bytes sent/received, ~111% for disk sector writes but only ~5.5% for CPU time as shown in Table 5.6. ‘‘D-bound’’ model performance comparing the fastest versus slowest deployments varied by 25.7% (>34 s).

Table 5.5. M-bound deployment performance variation

Parameter	**M-bound**	**Deployment Difference**
Avg. ensemble (sec)	23.4	13.7% (3.2 sec)
Avg. CPU time (sec)	11.7	6.5%
Avg. disk sector reads	57,675	14.8%
Avg. disk sector writes	286,297	21.8%
Avg. network bytes rec'd	9,019,414	144.9%
Avg. network bytes sent	9,037,774	143.7%

Table 5.6. D-bound deployment performance variation

Parameter	**D-bound**	**Deployment Difference**
Avg. ensemble (sec)	133.4	25.7% (34.3 sec)
Avg. CPU time (sec)	27.8	5.5%
Avg. disk sector reads	2,836,144	819.6%
Avg. disk sector writes	246,364	111.1%
Avg. network bytes rec'd	9,269,763	145.0%
Avg. network bytes sent	9,280,216	143.9%

Comparing both applications hosted by 4 GB Xen VMs a ~138% increase in CPU time was observed for the ‘‘d-bound’’ model. Network utilization increased ~3% and disk sector reads for the ‘‘d-bound’’ model where the $\mathcal{M}$ and $\mathcal{D}$ components were collocated increased 24,000% vs. the ‘‘m-bound’’ model, but decreased 87% for deployments where $\mathcal{M}$ and $\mathcal{D}$ were not co-located. On average the Xen ‘‘d-bound’’ model ensemble execution times were 5.7×

‘‘m-bound’’, averaging 133.4s versus 23.4. Network utilization likely increased for the ‘‘d-bound’’ model due to the longer duration of ensemble runs.

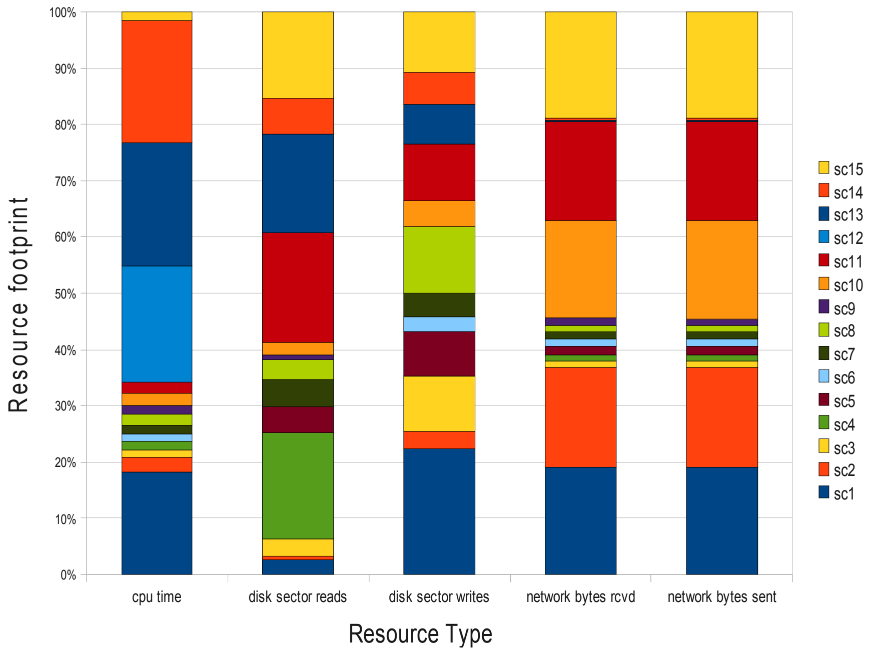


Figure 5.1. Resource Utilization Variation of Component Deployments

Figure 5.1 shows resource utilization variation for component deployments of the ‘‘m-bound’’ model. Resource utilization statistics were totaled from all VMs comprising individual component deployments. The graph shows the absolute value of the deviation from average resource utilization for the component deployments (SC1–SC15). The graph does not express positive/negative deviation from average but the magnitude of deviation. Larger boxes indicate a greater deviation from average resource utilization and smaller boxes indicate performance close to the average. The graph visually depicts the variance of resource utilization for our 15 component deployments.

5.5.2. Component deployment performance

To verify that component deployments performed consistently over time and to verify that we were not simply observing random behavior, two test sets consisting of 20 runs of the same 100-model run ensemble test were performed using all component deployments. The regression plot in Figure 5.2 compares the behavior of the two repeated test sets. Linear regression confirms the consistency of component deployment performance across subsequent test sets ($R^2 = 0.949674$, df = 13, $p = 8.09 \cdot 10^{-10}$). The three ellipses in the graph identify three different performance groups from left to right: fast, medium and slow. Performance consistency of ‘‘d-bound’’ tests was verified using the same technique. The consistency was not as strong due to higher variance of ‘‘d-bound’’ model execution times but was statistically significant ($R^2 = 0.81501$, df = 13, $p = 4.08 \cdot 10^{-6}$).

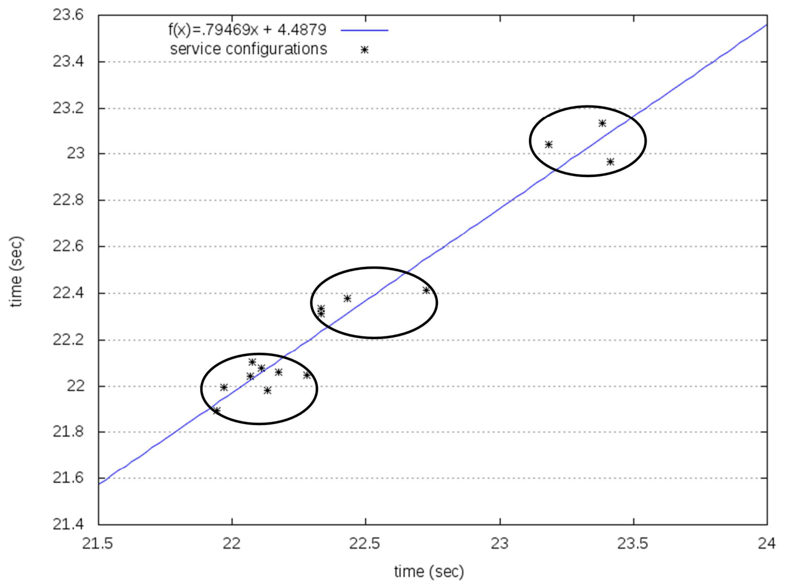


Figure 5.2. 4GB “m-bound” regression plot (XEN)

To simulate a production modeling web service 20 randomized 100-model run ensembles were generated (2000 unique requests) and used to benchmark each of the 15 component deployments. Figure 5.3 shows the performance comparison of the ‘‘m-bound’’ vs. ‘‘d-bound’’ model using the 20 different ensemble tests. Performance differences from average and overall rankings are shown in Table 5.7.

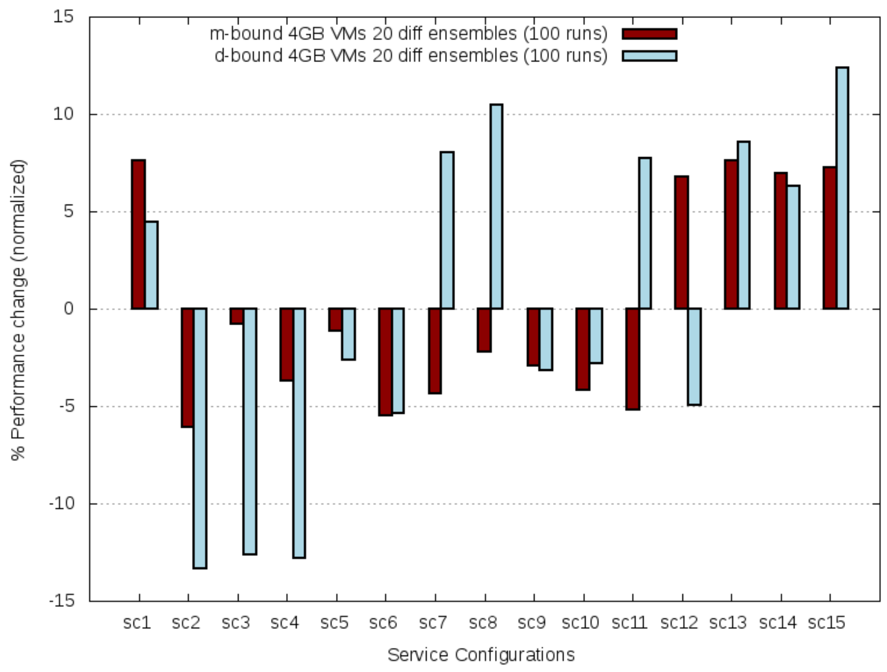


Figure 5.3. Performance Comparison – Randomized Ensembles (XEN)

Table 5.7. Performance Differences – Randomized Ensembles

composition	**m-bound**	**rank**	**d-bound**	**Rank**
SC1	7.59%	14	4.46%	9
SC2	-6.06%	1	-13.35%	1
SC3	-0.80%	10	-12.64%	3
SC4	-3.74%	6	-12.81%	2
SC5	-1.13%	9	-2.64%	8
SC6	-5.50%	2	-5.40%	4
SC7	-4.38%	4	7.98%	12
SC8	-2.21%	8	10.44%	14
SC9	-2.92%	7	-3.16%	6
SC10	-4.21%	5	-2.84%	7
SC11	-5.20%	3	7.72%	11
SC12	6.74%	11	-4.98%	5
SC13	7.63%	15	8.57%	13
SC14	6.97%	12	6.28%	10
SC15	7.22%	13	12.36%	15

We observed performance variation of nearly ~14% for the ‘‘m-bound’’ model and ~26% for the ‘‘d-bound’’ model comparing best-case vs. worse-case deployments. Service compositions for the ‘‘m-bound’’ application with random ensembles can be grouped into three categories of performance: fast {SC2, SC4, SC6, SC7, SC9, SC10, SC11}, medium {SC3, SC5, SC8}, and slow {SC1, SC12, SC13, SC14, SC15}. Compositions with $\mathcal{M}$ and $\mathcal{L}$ components co-located performed slower in all cases averaging 7.25% slower, about 1.7 s. When compositions had $\mathcal{M}$ and $\mathcal{L}$ co-located CPU time increased 14.6%, disk sector writes 18.4%, and network data sent/received about 3% versus compositions where $\mathcal{M}$ and $\mathcal{L}$ were separate. Service isolation (SC7) did not provide the best performance for either model. SC7 was ranked 4th fastest for the ''m-bound'' model and 12th for the ''d-bound'' model. The top three performing deployments for both model variants required only two or three VMs. Prior to testing the authors posited that isolating the application server (SC5), total service isolation (SC7), and isolating the geospatial database isolation (SC15) could be the fastest deployments. None of these deployments were top performers demonstrating our intuition was insufficient. Testing was required to determine the fastest component placements. We observed up to ~26% performance variation comparing component deployments while making no application changes only deployment changes. This variation illustrates the possible consequences for ad hoc component placement.

5.5.3. Provisioning variation testing

IaaS cloud providers often do not allow user-level control of VM placement to physical hosts. The non-determinism of where VMs are hosted results in provisioning variation [16], [18], [20]. In the previous section we identified the best performing application component deployments. We had two primary motivations for provision variation testing. First, to validate if deploying VMs using isolated physical hosts was sufficient to identify the best performing component deployments. For example does one of the deployments (SC11A, SC11B, SC11C) provide fundamentally different performance than SC11? And second, to quantify the average performance change for provisioning variation configurations. Intuition and previous research suggest that hosting multiple VMs on a single PM will reduce performance, but by how much?

Table 5.8. Provisioning Variation VM Tests

	PM 1	**PM 2**		**PM 1**	**PM 2**
SC2A	[ℳ𝒟ℱ] [ℒ]		SC9B	[ℳ] [𝒟ℒ] [ℱ]	
SC3A	[ℳ𝒟] [ℱℒ]		SC9C	[ℳ]	[𝒟ℒ] [ℱ]
SC4A	[ℳ𝒟] [ℱ]	[ℒ]	SC9D	[ℳ] [ℱ]	[𝒟ℒ]
SC4B	[ℳ𝒟] [ℱ] [ℒ]		SC10A	[ℳℱ] [𝒟ℒ]	
SC4C	[ℳ𝒟]	[ℱ] [ℒ]	SC11A	[ℳℱ] [𝒟]	[ℒ]
SC4D	[ℳ𝒟] [ℒ]	[ℱ]	SC11B	[ℳℱ] [𝒟] [ℒ]	
SC5A	[ℳ] [𝒟ℱℒ]		SC11C	[ℳℱ]	[𝒟] [ℒ]
SC6A	[ℳ] [𝒟ℱ]	[ℒ]	SC11D	[ℳℱ] [ℒ]	[𝒟]
SC6B	[ℳ] [𝒟ℱ] [ℒ]		SC12A	[ℳℒ] [𝒟ℱ]	
SC6C	[ℳ]	[𝒟ℱ] [ℒ]	SC13A	[ℳℒ] [𝒟]	[ℱ]
SC6D	[ℳ] [ℒ]	[𝒟ℱ]	SC13B	[ℳℒ] [𝒟] [ℱ]	
SC8A	[ℳ] [𝒟]	[ℱℒ]	SC13C	[ℳℒ]	[𝒟] [ℱ]
SC8B	[ℳ] [𝒟] [ℱℒ]		SC13D	[ℳℒ] [ℱ]	[𝒟]
SC8C	[ℳ]	[𝒟] [ℱℒ]	SC14A	[ℳ𝒟ℒ]	[ℱ]
SC8D	[ℳ] [ℱℒ]	[𝒟]	SC15A	[ℳℒℱ] [𝒟]	
SC9A	[ℳ] [𝒟ℒ]	[ℱ]			

There are 45 provisioning variations of the 15 component deployments described in Table 5.2 and tested in previous sections. Thirty-one of the configurations were tested using the twenty randomized 100-model run ensembles and KVM-based VMs with 4 GB memory allocation. Test configurations are identified by their base service configuration id SC1–SC15 and the letters A–D to identify provisioning variation configurations as described in Table 5.8. There were 14 variations of SC7 which represent the VM-level service isolation variants of component configurations described in Table 5.2. These were not tested because service isolation only adds overhead relative to their equivalents (SC1–SC6, SC8–SC15) as discussed in Section 5.6 for RQ-3. To compare performance, the provision variation deployments from Table 5.8 versus SC1–SC15, we calculated averages for provisioning variation configurations having more than 1 provisioning variation deployment (e.g. SC11A, SC11B, SC11C, SC11D). **Linear regression showed that component deployments performed the same regardless of provisioning variation (R^2 = 0.956701, df = 13, p = $3.03 \cdot 10^{-10}$), <u>though they generally performed slower.</u>** Performance differences observed appeared to result from hosting multiple

VMs on physical hosts. On average performance for provision variation configurations was 2.5% slower. Configurations with 2 VMs averaged 3.05% slower, with 3 VMs 2.33% slower. 6 of 31 configurations exhibited small performance gains: SC4A, SC6A, SC8C, SC11A, SC11C, and SC12A. Provisioning variation configurations which separated physical hosting of the $\mathcal{M}$ and $\mathcal{L}$ components provided an average improvement of 0.39% (10 configurations) whereas those which combined hosting of $\mathcal{M}$ and $\mathcal{L}$ were on average 3.93% slower. Performance differences for provisioning variation configurations are shown in Figure 5.4.

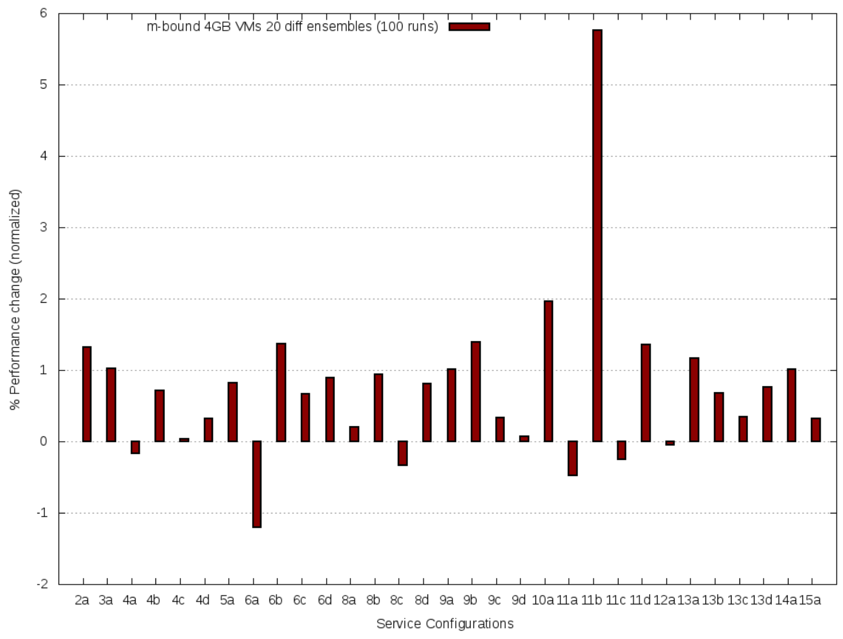


Figure 5.4. Provisioning Variation Performance Differences vs. Physical Isolation (KVM)

5.5.4. Increasing VM memory

In [12], the RUSLE2 model was used to investigate SOA scaling with components deployed on isolated VMs. VMs hosting the $\mathcal{M}$, $\mathcal{F}$, and $\mathcal{L}$ components were allocated 2 GB memory, and the $\mathcal{D}$ component VM was allocated 4 GB. To avoid performance degradation due to memory contention VM memory was increased to 10 GB, the total amount provided using individual VMs. Intuitively increasing VM memory should provide either a performance improvement or no change of performance. 20 runs of an identical 100-model run ensemble were repeated for the SC1–SC15 component deployments using 10 GB VMs. Figure 5.5 shows performance changes resulting from increasing VM memory allocation from 4 GB to 10 GB for both the ‘‘m-bound’’ and ‘‘d-bound’’ applications.

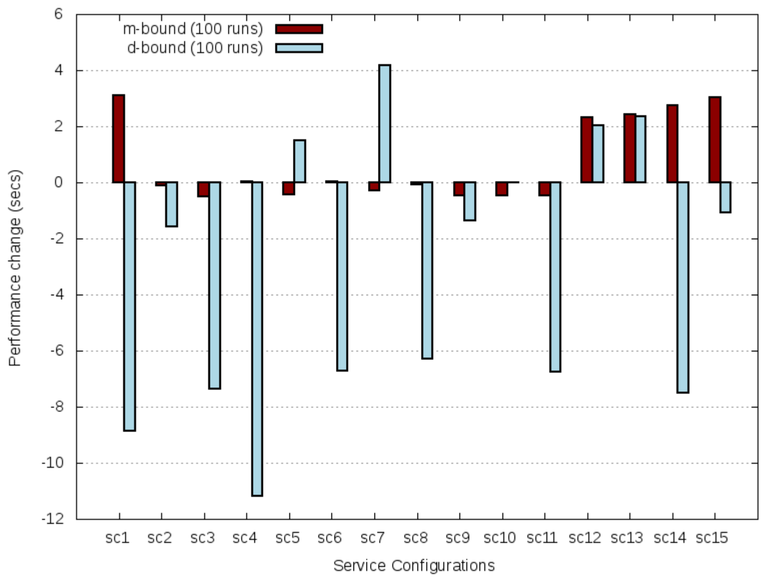


Figure 5.5. 10 GB VM Performance Changes (seconds)

For the ‘‘m-bound’’ application using 10 GB VMs reduced average ensemble performance 0.727 s (~3.24%) versus using VMs with 4 GB. SC11 provided had the best performance, 6.7% faster than average component deployment performance for 10 GB VM ‘‘m-bound’’ ensemble tests. This was half a second faster than with 4 GB VMs. SC1, total service combination, performed the slowest, 8.9% slower than average, 3.1 s longer than with 4 GB VMs. For the ‘‘m-bound’’ application only component deployments which combined $\mathcal{M}$ and $\mathcal{L}$ on the same VM experienced performance degradation. Both $\mathcal{M}$ and $\mathcal{L}$ used the Apache Tomcat web application server, but $\mathcal{L}$ used a 32-bit version for hosting Codebeamer and required the ia32 Linux 32-bit compatibility libraries to run on a 64-bit VM. The performance degradations may have resulted from virtualization of the ia32 library as 32-bit Linux can only natively address up to 4 GB RAM.

The ‘‘d-bound’’ application using 10 GB VMs performed on average 3.24s (2.46%) faster than when using 4 GB VMs. Additional VM memory improved database query performance. SC4 performed best at 12.5% faster or about 11.2 s faster. SC7, total service isolation, performed the slowest at 12.6% slower than average, equaling about 4.2s longer than tests with 4 GB VMs. To verify that these results were not specific to repeated runs of an identical 100-model run ensemble using the Xen hypervisor, we also tested increasing VM memory allocation using 20 different ensembles and the KVM hypervisor. Results were similar for both cases. The ‘‘m-bound’’ model’s 15 component deployments performed on average 342ms slower (−1.13%) with 10 GB VMs and the ‘‘d-bound’’ model performed 3.24s (2.46%) faster on average. Our results demonstrate that increasing VM memory allocation may result in unexpected performance changes in some cases exceeding +/−10%. For VM memory allocation, depending on the application, more may not always be better.

5.5.5. Xen vs. KVM

To compare performance differences between the Xen and KVM hypervisors we ran test sets using 20 different ensemble runs using 4 GB VMs and the ‘‘m-bound’’ application. Tests were repeated using both Xen and KVM hypervisors, random ensembles and the ‘‘d-bound’’ model. On average KVM ensemble performance was ~29% slower than Xen for the ‘‘m-bound’’ model, but ~1% faster for the ‘‘d-bound’’ model. The ‘‘d-bound’’ model was more CPU bound enabling performance improvement compared with Xen. The ‘‘m-bound’’ model had a higher proportion of I/O relative to CPU use and performed faster using Xen. ‘‘D-bound’’ ensemble tests using KVM required on average 4.35x’s longer than the ‘‘m-bound’’ model, while Xen ‘‘d-bound'' runs were 5.7 x longer than ''m-bound''. The average performance difference between the Xen and KVM hypervisors for running both the ''m-bound'' and ''d-bound'' models using 20 random ensemble tests is shown in Figure 5.6.

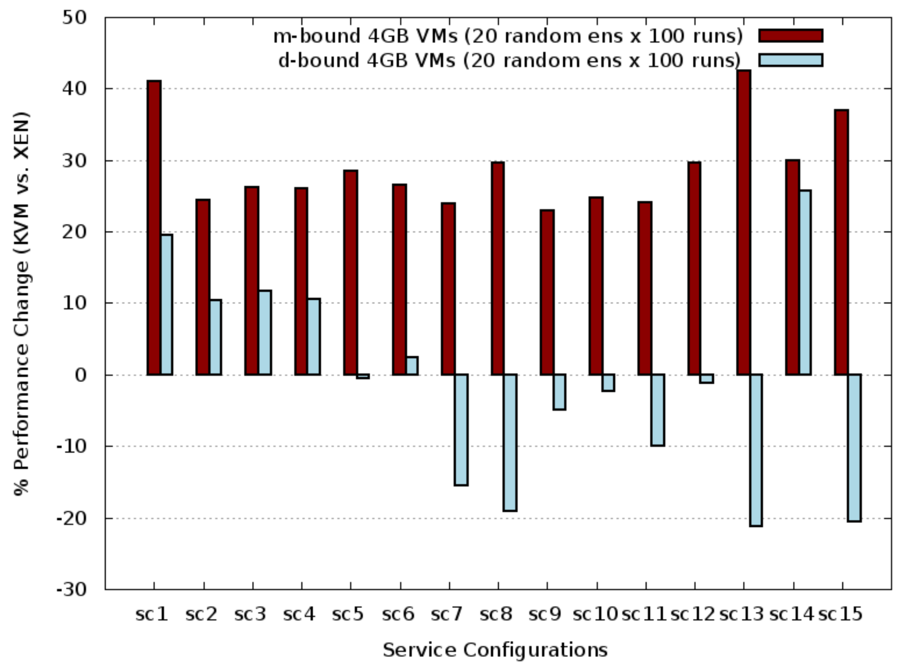


Figure 5.6. XEN vs. KVM Performance Differences, 4 GB VM Different Ensembles

Resource utilization data was collected for the ''m-bound'' model for all component deployments (SC1–SC15) using 20 random 100-model run ensemble tests for the Xen and KVM hypervisors. Resource utilization differences and correlations are summarized in Table 5.9. Resource utilization for Xen and KVM correlated for all statistics. On average KVM used 35% more CPU time than Xen, but nearly an equal number of disk sector reads (98%), but performed far fewer disk sector writes (50%). KVM exhibited 1.8% more network traffic (bytes sent/received) than Xen. Increased CPU time for KVM may result from KVM's full virtualization of devices where devices are entirely emulated by software. Xen I/O uses paravirtual devices which offer more direct device I/O.

Table 5.9. KVM vs. XEN resource utilization – randomized ensembles

Parameter	**KVM Resource Utilization (% of XEN)**	**R^2**	**p**
CPU time (sec)	135.2%	0.787769	.00001
Disk sector reads	97.91%	0.804115	$5.96 \bullet 10^{-6}$
Disk sector writes	50.48%	0.829572	$2.38 \bullet 10^{-6}$
Network bytes rec'd	101.77%	0.999872	$1.09 \bullet 10^{-26}$
Network bytes sent	101.85%	0.999874	$9.61 \bullet 10^{-27}$

We used a simple linear regression to compare Xen and KVM performance of component deployments for the ‘‘m-bound’’ and ‘‘d-bound models. Deployments using 4 GB VMs for the ‘‘m-bound’’ model with random ensembles were shown to perform similarly (R^2 = 0.749912, df = 13, p = 0.00003). Component deployment performance of Xen vs. KVM using the ‘‘d-bound’’ model performance did not correlate. Given KVM’s improved ‘‘d-bound’’ performance relative to Xen, this result was expected. Application performance using the KVM hypervisor appeared to be more sensitive than Xen to disk I/O.

5.5.6. Service isolation overhead

To investigate overhead resulting from the use of separate VMs to host application components the three fastest component deployments for the ‘‘m-bound’’ model were tested. Components were deployed using the SC2, SC6, and SC11 configurations with and without using separate VMs to host individual components.

60 runs using the same 100-model run ensemble, and 3 test sets of 20 different 100-model run ensembles were completed for each configuration. The percentage performance change resulting from service isolation is shown in Figure 5.7. For all but one configuration, service isolation resulted in overhead which degraded performance compared to deployments where multiple components were combined on VMs. The average overhead from service isolation was ~1%. For tests using different ensembles the observed performance degradation for

service isolation deployments was 1.2%, 0.3%, and 2.4% for SC2-SI, SC6-SI and SC11-SI respectively. The same ensemble test performance degradation was 1.1%, ~0.06%, and 1.4%. These results were reproduced using the KVM hypervisor with an average observed performance degradation of 2.4%.

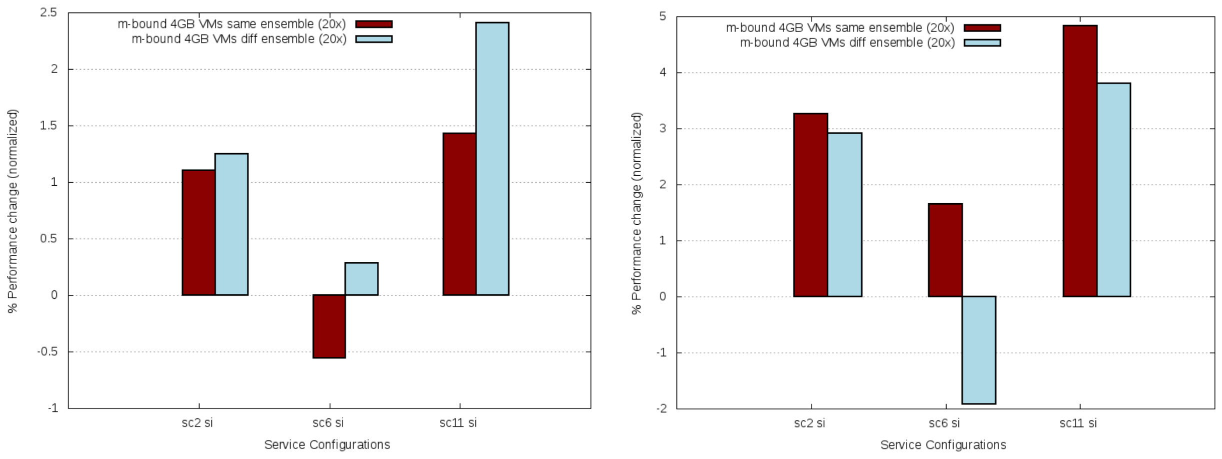


Figure 5.7. Performance Overhead from Service Isolation (XEN left, KVM right)

Although the performance overhead was not large, it is important to consider that using additional VMs incurs higher hosting costs without performance benefits. The isolated nature of our test design using isolated physical hardware, running no other applications, allows us to be certain that observed overhead resulted entirely from VM-level service isolation. This overhead is one of the tradeoffs for easier application tier-scalability with service isolation.

5.5.7. Predictive model

Resource utilization data was collected for CPU time, disk sector reads/writes, and network bytes sent/received as described in Section 5.1. We observed resource utilization variation for each of the deployments tested. Multiple linear regression (MLR) was used to build models to predict component deployment performance using resource utilization data to support investigation of RQ-4.

Multiple linear regression (MLR) is used to model the linear relationship between a dependent variable and one or more independent variables [53]. The dependent variable was ensemble execution time and the independent variables were VM resource utilization statistics including: CPU time, disk sector reads/writes, network bytes sent/received, and the number of virtual machines of the component deployment. The ‘‘R-squared’’ value, also known as the coefficient of determination, explains the explanatory power of the entire model and its independent variables as the proportion of variance accounted for. R-squared values were calculated for each independent variable using single linear regression. Root mean squared deviation (RMSD) was calculated for each variable. The RMSD expresses differences between the predicted and observed values and serves to provide a measure of model accuracy. Ideally 95% of predictions should be less than +/−2 RMSD’s from the actual value.

A MLR model was built using resource utilization variables from the ‘‘m-bound’’ model using Xen 4 GB VMs with 20 different ensemble tests. All of our resource utilization variables together produced a model which accounted for 84% of the variance with a RMSD of only ~676 ms ($R^2 = 0.8416$, RMSD = 664.17 ms). Table 5.10 shows individual R^2 values for the resource utilization statistics used in a simple linear regression model with ensemble execution time to determine how much variance each explained. Additionally the average error (RMSD) is shown. The most predictive parameters were CPU time which positively correlated with ensemble time and explained over 70% of the variance ($R^2 = 0.7171$) and disk sector reads ($R^2 = 0.3714$) with a negative correlation. Disk sector writes had a positive correlation with ensemble performance ($R^2 = 0.1441$). The number of deployment VMs ($R^2 = 0.0444$) and network bytes received/sent were not strong predictors of ensemble performance and explained very little variance.

Table 5.10. Resource Utilization – Predictive Power

Parameter	**R^2**	**RMSD**
CPU time	.7171	887.64
# Disk sector reads	.3714	1323.25
# Disk sector writes	.1441	1544.05
Network bytes recv'd.	.0074	1662.76
Network bytes sent	.0075	1662.68
Number of VMs	.0444	1631.44

Table 5.11. Deployment Performance Rank Predictions

Composition	**Predicted Rank**	**Actual Rank**	**Rank Error**
SC1	12	15	-3
SC2	2	2	0
SC3	7	8	-1
SC4	6	9	-3
SC5	10	4	6
SC6	9	10	-1
SC7	4	5	-1
SC8	8	7	1
SC9	5	6	-1
SC10	3	3	0
SC11	1	1	0
SC12	15	12	3
SC13	14	14	0
SC14	13	13	0
SC15	11	11	0

We applied our MLR performance model to predict performance of component deployments. Resource utilization data used to generate the model was reused to generate ensemble time predictions. Average predicted ensemble execution times were calculated for each component deployment (SC1–SC15) and rank predictions were calculated. Predicted vs. actual performance ranks are shown in Table 5.11. The mean absolute error (MAE) was 462 ms, and estimated ranks were on average +/− 1.33 units from the actual ranks. Eleven predicted ranks for component compositions were off by 1 unit or less from their actual rank, with six exact predictions for SC2, SC10, SC11, SC13, SC14, and SC15. The top three performing deployments were predicted correctly in order. A second set of resource utilization data was collected for the ‘‘m-bound’’ model using 4 GB VMs and 20 random ensembles for SC1–SC15. This data was fed into our MLR performance model and observed MAE was only 324 ms. The average rank error was +/− 2 units. Seven predicted ranks were off by 1 unit or less from their actual rank, with three exact predictions. The top fastest deployment was correctly predicted for the second dataset.

Building models to predict component deployment performance requires careful consideration of resource utilization variables. This initial attempt using multiple linear regression was helpful to identify which independent variables had the greatest impact on deployment performance. Future work to improve performance prediction should investigate using additional resource utilization statistics as independent variables to improve model accuracy. New variables including CPU statistics, kernel scheduler statistics, and guest/host load averages should be explored. The utility of neural networks, genetic algorithms, and/or support vector machines to improve our model should be investigated extending related research [21], [27]–[29], [32], [37]. These techniques can help improve performance predictions if resource utilization data is not normally distributed.

5.6. CONCLUSIONS

(RQ-1) This research investigated the scope of performance implications which occur based on how components of SOAs are deployed across VMs on a private IaaS cloud. All possible deployments were tested for two variants of the RUSLE2 soil erosion model, a 4-component application. Up to a 14% and 25.7% performance variation was observed for the ‘‘m-bound’’ and ‘‘d-bound’’ RUSLE2 models respectively. Significant resource utilization (CPU, disk, network) variation was observed based on how application components were deployed across VMs. Intuition was insufficient to determine the best performing deployments. Ad hoc worst case scenario component placements significantly degraded application performance demonstrating consequences for ignoring component composition. Component deployment using total service isolation did not provide the fastest performance for our application. Provisioning variation did not change the fundamental performance of component deployments but did produce overhead of ~ 2%-3% when two or more VMs resided on the same physical host.

(RQ-2) Increasing VM memory allocation did not guarantee application performance improvements. Increasing VM memory to improve performance appears useful only if memory is the application's performance bottleneck. The KVM hypervisor performed 29% slower than Xen when application performance was bound by disk I/O but slightly faster ~1% when the application was CPU bound. KVM resource utilization correlated with Xen but CPU time was 35% greater when KVM was used to perform the same work.

(RQ-3) Service isolation, the practice of using separate VMs to host individual application components resulted in performance overhead up to 2.4%. Though overhead may be small, the hosting costs for additional VMs should be balanced with the need to granularly scale application components. Deploying an application using total service isolation will always result in the highest possible hosting costs in terms of the # of VMs.

(RQ-4) Resource utilization statistics were helpful for building performance models to predict performance of component deployments. Using just six resource utilization variables our multiple linear regression model accounted for 84% of the variance in predicting performance of component deployments and accurately predicted the top performing component deployments.

Providing VM/application level resource load balancing and using compact application deployments holds promise for improving application performance while lowering application hosting costs. To support load balancing and cloud infrastructure management, performance models should be investigated further as they hold promise to help guide intelligent application deployment and resource management for IaaS clouds.

CHAPTER 6

THE VIRTUAL MACHINE SCALER

6.1. INTRODUCTION

The advent of modern multi-core CPUs, allow today's compute servers to process many tasks in parallel. These multi-core processors can provide increased performance for environmental modeling when: (1) model source code is architected to perform parallel computations to execute using multiple cores, or (2) multiple distinct related or unrelated model runs can be computed in parallel. Service based computing involves hosting a computational engine to perform complex domain-specific calculations sometimes referred to as business logic or middleware. Services operate using web based TCP ports and are therefore referred to as ***web services***. Deployment of environmental models as web services involves migration of the model computation from the user's client computers to run in a centralized modern datacenter to reap the benefits of faster hardware and server scalability. Users submit model service requests by describing the model parameterization including required inputs for the model run. Many service requests can be processed in parallel. Updating model code is made easier with a centralized deployment model. We refer to deployment of scientific models as web services as ***modeling-as-a-service***.

Infrastructure-as-a-Service (IaaS) is a type of web service. Specifically, IaaS provides compute infrastructure, on demand, as a service, to end users. IaaS is one of the fundamental service types enabled by ***cloud computing***. IaaS clouds allow ***service oriented applications*** (SOAs) to have elastic infrastructure where resource allocations can be scaled up or down in real time to meet application demand. IaaS clouds provide ideal server infrastructure for providing modeling-as-a-service. One key challenge of providing scientific modeling-as-a-service on demand to end users is the requirement to provide both good modeling ***performance*** and service ***availability***. Availability is the notion that the modeling service is always available to perspective users. If a large spike in demand for a scientific model occurs, the modeling service should not reject new requests, but continue to accept and process them in a timely manner.

As the number of CPU cores has increased in modern servers, the overall idle time has increased. Today dual and quad processor servers can support 40+ individual processing cores. These cores frequently employ ***hyper-threading***. Hyper-threads provide two execution threads per CPU core, each of which with its own processor architectural state. These hyper-threads, referred to as logical processors, can be individually halted, interrupted or directed to execute a specified program, independent from the other logical processor. Logical processors share execution resources, allowing one processor to borrow resources from the other if stalled waiting for I/O. The benefit seen from CPU hyper-threading depends on the application's balance of computation vs. I/O, but is often 30% or better. Hyper-threading enables fast execution of many more model runs in parallel, enabling higher throughput than single threaded CPUs.

It has become difficult for a single operating system instance to fully utilize so many cores. To support service scalability, modern multi-core servers support server ***virtualization***. Virtualization allows a single physical server to host many ***virtual machines*** (VMs). VMs are implemented using a software program known as the ***hypervisor*** which supports sharing the physical computer's processor(s), network and disk resources. Each virtual machine has its own operating system instance providing isolation. Server virtualization provides partitioning of server resources with the overall goal of increasing utilization. Increasing server utilization supports datacenter consolidation as redundant idle servers can be removed saving physical space and electricity. Popular virtualization hypervisors include kernel-based VMs (KVM), Xen, and the VMware ESX hypervisor. [1]–[3].

Historically the components of multi-tier SOAs were deployed on one or more physical servers as in figure 6.1. In a cloud based setting, SOAs are now deployed across a set of virtual machine images (figure 6.2) instead of being consolidated on a single physical server. Multiple VM instances can be provisioned for each application tier using these virtual machine images enabling the application infrastructure to scale based on service demand.

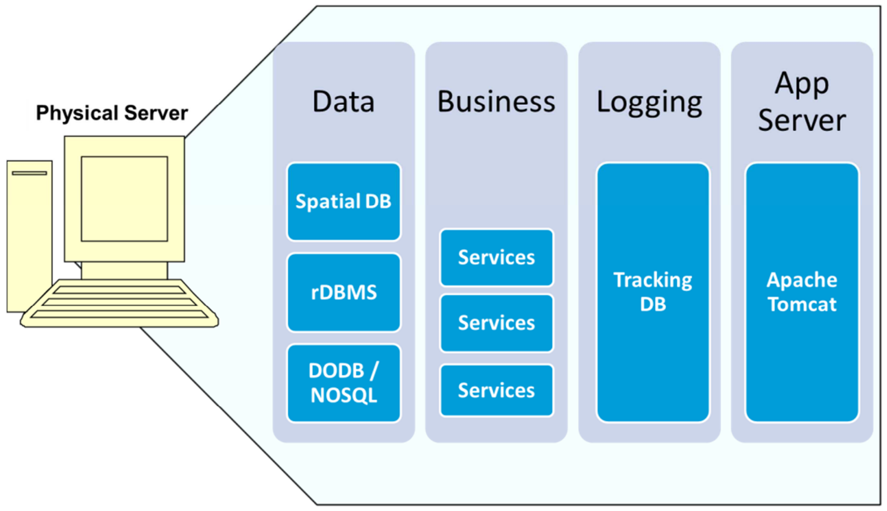


Figure 6.1. Traditional Service Oriented Application Deployment

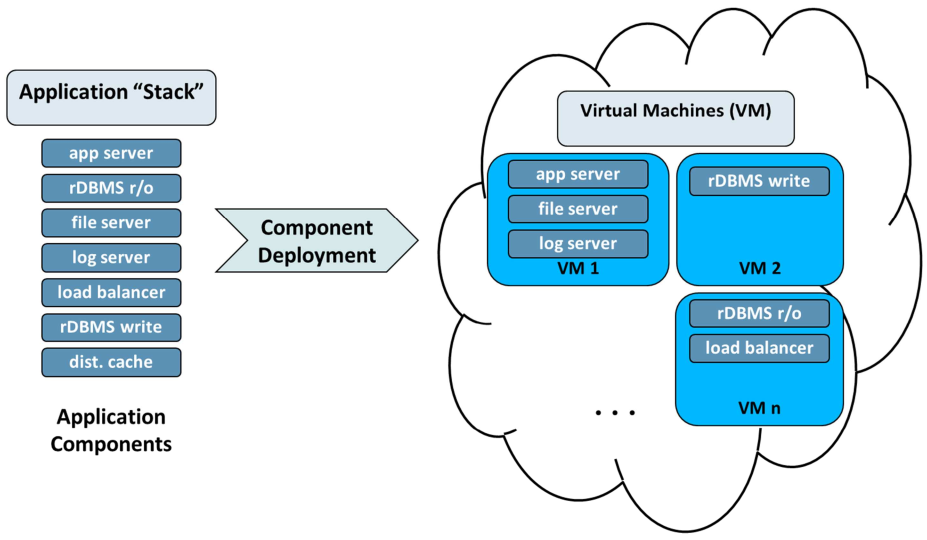


Figure 6.2. IaaS Cloud Service Oriented Application Deployment

Environmental models can be deployed as web services to IaaS clouds using Amazon's public IaaS cloud Elastic Compute Cloud application programming interface (***EC2-API***). The EC2-API enables programmatic management of IaaS clouds enabling dynamic management of the server infrastructure used to host model services [66]. The EC2 API is supported to varying degrees by most open source private clouds including: Apache CloudStack [67], Eucalyptus [46], OpenNebula [68], and OpenStack [69].

To support environmental modeling using IaaS Cloud application scaling and infrastructure management, we have developed the Virtual Machine Scaler (VM-Scaler), a REST/JSON-based web services application. VM-Scaler supports IaaS cloud infrastructure management for environment modeling as part of the US Department of Agriculture and Colorado State University's Cloud Services Innovation Platform (CSIP) [70]. VM-Scaler's support of model infrastructure scalability for CSIP has been evaluated using the Revised Universal Soil Loss Equation – Version 2 (RUSLE2) [41], and the Wind Erosion Prediction System (WEPS) [71] as described in Lloyd [72]. RUSLE2 and WEPS are the US Department of

Agriculture–Natural Resource Conservation Service standard models for soil erosion used by over 3,000 county level field offices across the United States. RUSLE2 and WEPS are used to provide soil erosion modeling services to end users. RUSLE2 contains both empirical and process-based science that predicts rill and interrill soil erosion by rainfall and runoff. RUSLE2 was developed primarily to guide natural resources conservation planning, inventory erosion rates, and estimate sediment delivery. The Wind Erosion Prediction System (WEPS) is a daily simulation model which outputs average soil loss and deposition values for selected areas and periods of time to predict soil erosion due to wind. WEPS consists of seven sub-models including: weather, crop growth, decomposition, hydrology, soil, erosion, and tillage.

CSIP provides a common Java-based framework for providing REST/JSON based modeling-as-a-service to end users. VM-Scaler harnesses the Amazon EC2 API to support scaling server infrastructure and management of the underlying clouds to enable modeling-as-a-service [66]. VM-Scaler currently supports Amazon EC2, and Eucalyptus versions 3.1 and 3.3. VM-Scaler provides cloud control while abstracting the underlying IaaS cloud and can be extended to support any EC2 compatible cloud (figure 6.3). VM-Scaler provides a platform for supporting scalable environmental model services and supports: (1) profiling resource requirements of modeling workloads, (2) experimentation with hot spot detection schemes, (3) investigation of VM management/placement approaches, and (4) development of custom model request scheduling/proxy services. The remainder of this chapter describes features provided by VM-Scaler which help support environmental modeling services using public, private, and hybrid IaaS cloud resources.

6.2. THE VIRTUAL MACHINE SCALER

VM-Scaler is a REST/JSON Java-based web application installed to any web application container such as Apache Tomcat or Glassfish. VM-Scaler can be hosted by a virtual machine (VM) or physical machine (PM) having network connectivity to the managed cloud. Upon initialization VM-Scaler probes the host cloud and collects metadata including location and state information for all PMs (private clouds only) and VMs. An object model is constructed in memory to represent the state of the cloud. The Eucalyptus implementation also determines the Eucalyptus round-robin VM launch sequence to identify which node is expected to receive the next VM launch request. VM-Scaler service requests are formulated using JSON objects. Service specific JSON objects are used to describe meta-data for the requested operations. VM-Scaler is easily extensible to support new services as needed. Table 6.1 describes existing VM-Scaler services.

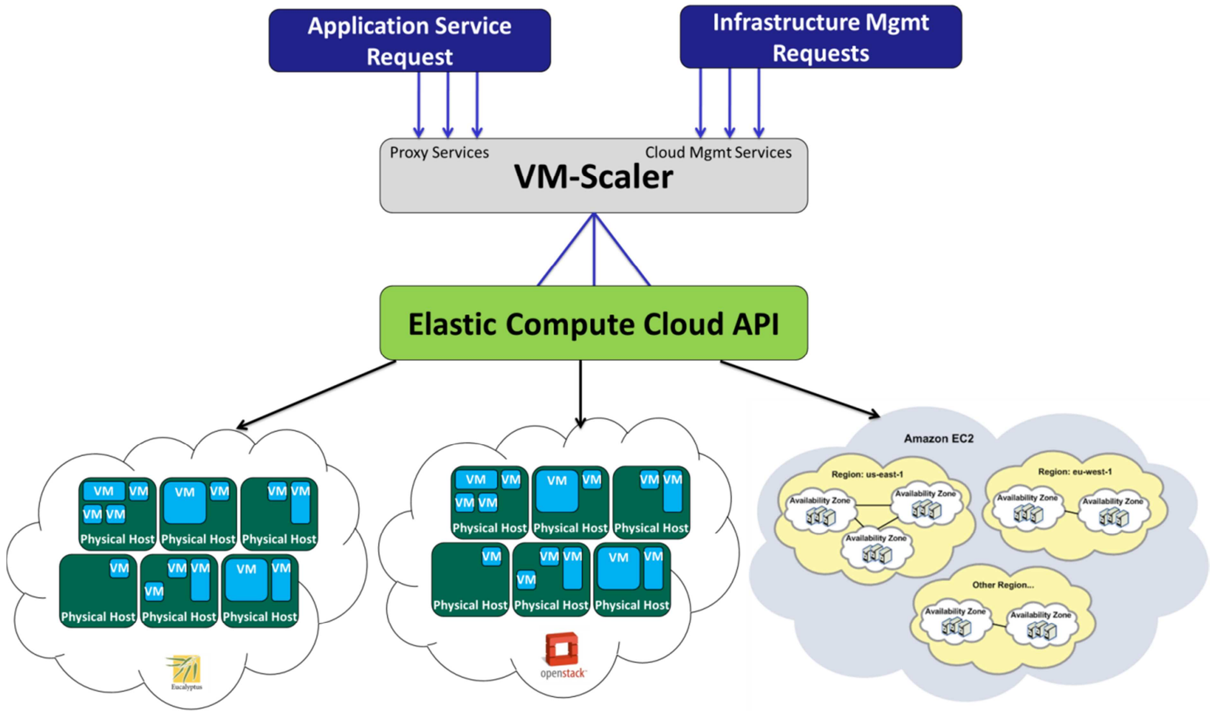
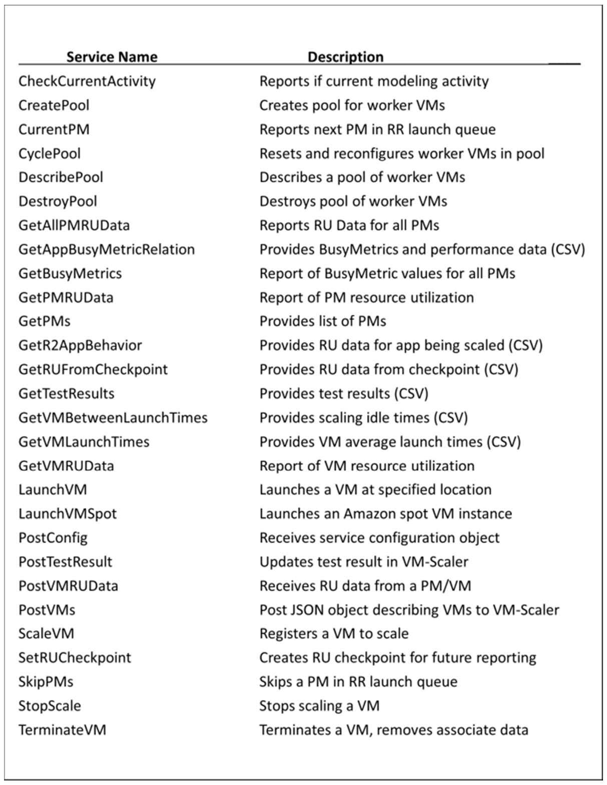


Figure 6.3. VM-Scaler Cloud Abstraction

Table 6.1. VM-Scaler Services



Service Name	Description
CheckCurrentActivity	Reports if current modeling activity
CreatePool	Creates pool for worker VMs
CurrentPM	Reports next PM in RR launch queue
CyclePool	Resets and reconfigures worker VMs in pool
DescribePool	Describes a pool of worker VMs
DestroyPool	Destroys pool of worker VMs
GetAllPMRUData	Reports RU Data for all PMs
GetAppBusyMetricRelation	Provides BusyMetrics and performance data (CSV)
GetBusyMetrics	Report of BusyMetric values for all PMs
GetPMRUData	Report of PM resource utilization
GetPMs	Provides list of PMs
GetR2AppBehavior	Provides RU data for app being scaled (CSV)
GetRUFromCheckpoint	Provides RU data from checkpoint (CSV)
GetTestResults	Provides test results (CSV)
GetVMBetweenLaunchTimes	Provides scaling idle times (CSV)
GetVMLaunchTimes	Provides VM average launch times (CSV)
GetVMRUData	Report of VM resource utilization
LaunchVM	Launches a VM at specified location
LaunchVMSpot	Launches an Amazon spot VM instance
PostConfig	Receives service configuration object
PostTestResult	Updates test result in VM-Scaler
PostVMRUData	Receives RU data from a PM/VM
PostVMs	Post JSON object describing VMs to VM-Scaler
ScaleVM	Registers a VM to scale
SetRUCheckpoint	Creates RU checkpoint for future reporting
SkipPMs	Skips a PM in RR launch queue
StopScale	Stops scaling a VM
TerminateVM	Terminates a VM, removes associate data

6.2.1. Resource Utilization Data Collection

An agent is installed to all infrastructure VMs and PMs (if accessible) to send resource utilization (RU) data to VM-Scaler at fixed intervals. The default interval is 15-seconds. The RU data collection agent is extensible and presently collects resource utilization statistics for (18) parameters as described in table 6.2. RU data is used to calculate resource use for: (1) the last 15-second interval, (2) the previous one minute average, and (3) a historical lifetime average. One minute averages are presently used to perform hot spot detection. (see section 2.4)

6.2.2. Model Workload Resource Utilization Check-pointing

VM-Scaler supports resource utilization checkpoint for environmental model workloads. Resource utilization check-pointing can be used to obtain the total resource utilization profile for a modelling workload. RU profiles can help determine the required machine resources to accomplish similar modeling workloads. RU profiles help quantify the heft or weight of modelling workloads in terms of the resource requirements needed to execute. RU profiles quantify resource usage for all eighteen resource utilization statistics described in table 6.2. Understanding the total CPU time, disk, and network I/O required for a batch of modelling is particularly useful if looking to schedule many similar model runs as is the case for calibration or monte carlo simulations.

Table 6.2. Resource Utilization Statistics

Statistic		Description
P/V	CPU time	CPU time in ms
P/V	cpu usr	CPU time in user mode in ms
P/V	cpu krn	CPU time in kernel mode in ms
P/V	cpu_idle	CPU idle time in ms
P/V	Contextsw	Number of context switches
P/V	cpu_io_wait	CPU time waiting for I/O to complete
P/V	cpu_sint_time	CPU time servicing soft interrupts
V	Dsr	Disk sector reads (1 sector = 512 bytes)
V	Dsreads	Number of completed disk reads
V	Drm	Number of adjacent disk reads merged
V	Readtime	Time in ms spent reading from disk
V	Dsw	Disk sector writes (1 sector = 512 bytes)
V	Dswrites	Number of completed disk writes
V	Dwm	Number of adjacent disk writes merged
V	Writetime	Time in ms spent writing to disk
P/V	Nbr	Network bytes sent
P/V	Nbs	Network bytes received
P/V	Loadavg	Avg # of running processes in last 60 sec

6.2.3. Scaling Tasks

VM-Scaler provides horizontal scaling of application infrastructure by increasing the allocated number of VMs to service a particular tier of a multi-tier SOA. When application hot spots are detected one or more VMs can be launched in parallel in response. A service request is issued to describe the requested scaling task using a JSON object. The JSON object identifies the base VM, the initial VM which provides implementation of a particular application tier. The JSON object includes a VM-type meta-data tag to identify VMs launched to support the tier. An image-id identifies which virtual machine image to launch in response to hot spots. The VM-size attribute specifies the size and type for new VMs, for example m1.xlarge, m2.4xlarge. An access key is included and also the host zone/region and the VM security group are identified.

Three additional configurable scaling parameters include: `min_time_to_scale_again`, `min_time_to_scale_after_failure`, and `max_VM_launch_time`. `Min_time_to_scale_again` provides a time buffer before scaling again, allowing time to consider the impact of recent resource additions. This parameter helps to eliminate the ping-pong effect described in [7] and is equivalent to Amazon Scaling Group cool-down periods [73]. `Max_VM_launch_time` provides a maximum time limit before terminating launches that appear to have stalled. This supports handling launch failures by reissuing stalled launch requests. `Min_time_to_scale_after_failure` provides an alternate wait time when VM launch failures occur.

6.2.4. Hot Spot Detection

VM-Scaler supports both resource utilization threshold and application performance model-based hot spot detection. Threshold based scaling is triggered when resource utilization variables exceed configured thresholds. This application agnostic approach is reactive to current system conditions and supports experimentation because the hot spot detection scheme can remain constant while VM scheduling algorithms or the application being tested are changed. By default scaling thresholds can be specified for one-minute averages of: maximum CPU time, minimum CPU idle time, maximum number of context switches, and maximum load average. Application performance model hot spot detection uses trends in resource utilization to predict average model execution time. Predictions are made for average model execution time for 1, 2, and 3 time steps in the future where a time step is 15 seconds. Scaling thresholds trigger hot spot detection when future predicted model execution time exceeds set values.

6.2.5. Least-Busy VM Placement

For Eucalyptus private IaaS clouds, VM-Scaler supports controlling the placement of new VM's to specific physical hosts. New VM launches can specify a specific host, use the default host provided by Eucalyptus round-robin, or harness VM-Scaler's Least-Busy VM placement algorithm. VM placement to the Least-Busy physical machine is based on using our BusyMetric which aggregates total host resource utilization to determine the best candidates for hosting new VM's by quantifying host busyness [72]. The busy metric double weights CPU time for environmental modelling since most models are CPU-bound in nature. Disk sector reads/writes, network bytes received/sent and host occupancy are also included in the Busy Metric calculation. Busyness is quantified relative to the observed maximum system value for each resource utilization measure. Maximums are determined through stress testing.

For example:

$$cputime_n = \frac{cputime_{obs\_1sec}}{cputime_{\max\_1sec}} \qquad (1)$$

Our Busy-Metric is expressed as:

$$\frac{(2 \cdot cputime_n) + dsr_n + dsw_n + nbr_n + nbs_n + (\frac{2 \cdot Hosted\_VMs}{PM_{cores}})}{7} \qquad (2)$$

Each additional VM hosted linearly increases the value of the Busy-Metric by:

$$e^{(\ln PM_{cores} - 1.2528)} \qquad (3)$$

The Busy-Metric provides an approach to rank available capacity of physical host machines. Our goal has been to develop a general metric which supports new VM placements based on quantifying the total shared load of private cloud host machines. Many variations of our busy metric are possible by using unique resource variable weights based on specific resource requirements of different environmental models.

6.2.6. Model Request Job Scheduling

VM-Scaler supports using the Busy-Metric described in section 2.4 to perform model run scheduling/proxy services. Incoming model requests can be routed to the Least-Busy VM. This provides an alternative to both round-robin load balancing and least-connection load balancing. Round-robin load balancing is supported using the HAProxy load balancer [51], by evenly distributing model requests to the pool of modelling-engine VMs. Least-connection load balancing supported by HAProxy, distributes model requests by evenly balancing the number of active concurrent sessions at each modelling engine VM at any given time. VM-Scaler's Least-busy load balancing routes incoming model requests to run on the modelling engine VM with the most available resources as quantified using the Busy-Metric. Least-Busy job scheduling is a black-box job scheduler which does consider details of the model parameterization to perform the scheduling. Future work plans to investigate the development of white-box job schedulers which harness model parameterization details of incoming model requests to predict model resource and execution time requirements before execution. Harnessing predictions should improve model execution scheduling and reduce model execution times by minimizing server idle time during model workload execution which occurs between scheduled jobs that presently goes to waste.

6.2.7. VM Pools

VM-Scaler supports VM pools to support recycling VMs in cases when the ***launch latency*** time is high. For environmental modelling, VM launch latency is the time required to launch and initialize a new modelling engine VM before it is ready to perform model computations. Launch latency time varies based on the type and size of the VM image, as well as the host cloud and its underlying hardware. For example, on Amazon EC2, using faster VM types such as c3.xlarge generally enables more rapid VM launch and initialization times compared to slower instance types such as m1.large. Similarly, on a private cloud, VM instance types assigned more computational and memory resources typically initialize more rapidly. Launch latency will vary by cloud and the specifics of the VM being provisioned and should be benchmarked to establish baseline time requirements for dynamic scaling.

When dynamically scaling the modelling tier of an SOA it may be necessary to rapidly increase the number of worker VMs in response model demand. To support dynamic scaling when new VMs cannot be launched fast enough, VMs can be prelaunched and reserved for later use using VM-Scaler ***VM pools***. Prelaunched VMs are referred to as spare VMs. A key cost / performance trade-off concerns identifying the number of spare VMs to allocate versus the supported magnitude of model service demand spikes. When too many spare VMs are provisioned hosting costs are high, but scalability performance is excellent. Conversely when too few spare VMs are provisioned the model service may become slow, unavailable, or **crash** in response to demand spikes.

VM pools help support the use of Amazon public cloud spot instances for environmental modeling. Amazon Spot instances are low-cost VMs which are billed at a fluctuating auction price rather than the standard going rate. Prices may be as low as 1/8 to 1/9 the cost of full dedicated instances with the caveat being these instances may terminate at any instant when the bid price is exceeded due to heavy demand. Amazon spot instances have very long launch latency times since two separate Amazon EC2 operations are required to provision a VM. An initial call places a spot market bid, and if the bid is successful a VM launch operation occurs. For CSIP, launch latency times of 4-5 minutes per VM is not unusual. With such long launch latency times dynamic scaling using amazon spot instances is generally not practical without prelaunching VMs.

VM pools also support reusing VM instances for dynamic scaling in lieu of Amazon's billing model. Amazon bills hourly for VM usage. Even if a VM is only needed for 1 minute, the user is charged for an entire hour. For dynamic scaling it is useful then to recapture idle VMs for the duration of the billing cycle in case there is a future opportunity for use.

6.3. SUMMARY AND CONCLUSIONS

By harnessing infrastructure-as-a-service cloud computing, server infrastructure supporting environmental model services can dynamically scale based on user demand to deliver: (1) high availability, (2) high throughput (requests/second) and (3) fast model execution times.

In this chapter we have presented the VM-Scaler, a cloud agnostic autonomic resource manager which supports infrastructure management for multi-tier service oriented applications. VM-Scaler supports dynamic infrastructure scaling for both new and legacy environmental models supporting their deployment as web-based model services enabling model computation "as-a-service", on demand, for users. VM-Scaler supports model service scalability on both private and public clouds by providing key features including: resource utilization data collection, model workload resource utilization check-pointing, dynamic scaling tasks, hot spot detection, Least-Busy VM placement, Job Scheduling, and VM pools. VM-Scaler supports many features in-gratis which are typically pay-for-use infrastructure services in public clouds including resource utilization data collection and dynamic scaling. The utility of VM-Scaler has been demonstrated in support of the USDA’s Cloud Services Innovation Platform (CSIP) and development remains ongoing.

CHAPTER 7

IMPROVING VM PLACEMENTS TO MITIGATE RESOURCE CONTENTION AND HETEROGENEITY

7.1. INTRODUCTION

Supporting scientific modeling computational services, introduces resource management challenges. These challenges must be addressed when deploying model services to Infrastructure-as-a-Service (IaaS) clouds. Given that scientific models are often computationally bound, dynamic scaling of server infrastructure is required to address model service demand spikes. For example a legacy Fortran process based scientific model whose code is primarily sequential might fully occupy a single CPU core for 10 minutes for an individual model run. It is often not practical to refactor the legacy model code due to resource or model design limitations to obtain significant performance improvements [74]. If an insufficient number of CPU cores are provisioned frequent context switching can interrupt model service execution and greatly degrading performance. Hosting model services using IaaS clouds enables server resources to be easily provisioned reducing computation interruptions from CPU context switching. Before the advent of cloud computing's support of resource elasticity through server virtualization, hosting scientific model services required extensive allocations of physical hardware.

IaaS cloud resource management challenges for scientific model services can be broken down into three primary concerns: (1) Determining WHEN infrastructure should be provisioned? (2) Determining WHAT infrastructure should be provisioned? and (3) Determining WHERE infrastructure should be provisioned?

WHEN server infrastructure should be provisioned to address service demand is informed by hotspot detection [6]. Determining when to scale-up resources is complicated by the latency of virtual machine (VM) launches. In some cases, the time required to provision and launch new infrastructure exceeds the duration of demand spikes [7]. By predicting future demand, server infrastructure can be pre-provisioned. Load prediction can be difficult particularly for applications with stochastic load behavior. Care must be exercised as poor predictions can result in overprovisioning and higher hosting costs, or underprovisioning and poor performance.

WHAT server infrastructure should be provisioned concerns the size and type (vertical scaling) and quantity (horizontal scaling) of VM allocations. Vertical scaling involves modifying resource allocations of existing VMs. Altering VM resource allocations including CPU core, memory, disk, and network bandwidth may alleviate poor performance. When vertical scaling is unavailable, or insufficient to address service demand, horizontal scaling can be used. Additional service capacity is provisioned by launching new VMs and the service workload is balanced across the expanded pool of VMs. A key challenge lies in determining how many VMs should be provisioned, and with what resource allocations?

WHERE server resources should be provisioned is abstracted by Infrastructure-as-a-Service (IaaS) clouds. Representing VMs as tuples and using them to pack physical machines (PMs) can be thought of as an example of the multidimensional bin-packing problem that has been shown to be NP-hard [14]. Consequently in practice simplified heuristic based approaches to VM placement are typically used. Two types of VM placement schedulers common to private IaaS clouds include *greedy* and *round-robin*. *Greedy* allocation deploys all VMs to a single PM first. When the host's resources are exhausted another PM is selected and the process is repeated. Greedy allocation packs resources tightly, enabling maximum energy savings without regard to VM/application performance. *Round-robin* placement distributes VMs to each PM in succession, balancing the VM hosting load across the cluster. Round-robin placement typically provides better VM performance by reducing resource contention at the expense of higher energy requirements. Using round-robin placement, all PMs in the cluster receive a portion of the VM hosting load, eliminating potential for idle machines to operate in power saving modes [19].

In a public IaaS cloud setting the user is abstracted from explicit control of VM host location (WHERE) and how specific VM types are implemented (WHAT).

WHERE VMs are provisioned in a public cloud is not only uncontrollable, but difficult to discern as well [16]. End user determination of VM location and co-location remains an open challenge. Previous efforts using heuristics to infer VM co-residency and launching probe VMs for exploration are both expensive and only partially effective at determining VM locations [17]. Resource contention from VM multi-tenancy has been shown to degrade performance and is of concern for cloud-based scientific modeling [16], [18].

WHAT hardware is used to implement public cloud VM types has been shown to vary, as VM-types (e.g. Amazon EC2 m1.large) have been shown to use different hardware implementations over time [10], [11]. Public clouds abstract VM hardware implementation details enabling data center upgrades to seamlessly enable lower energy costs over time. Prior research has demonstrated performance variance resulting from heterogeneous implementations of a single VM type as great as 30% for a web application. The magnitude of this performance variation is significant and deems attention for scientific modeling in the cloud.

7.1.1. Research Questions

The following six research questions are investigated:

Private IaaS :

RQ-1: What performance implications result from VM placement location when dynamically scaling scientific model services? How important is VM placement for scaling in response to increasing service demand?

RQ-2: How do resource costs (# of VMs) vary when dynamically scaling model service applications as a result of VM placement location?

RQ-3: How important is VM placement location when scaling with different VM sizes (# of CPU cores)? Does the granularity of provisioned resources change the importance of making good VM placements?

RQ-4: How do performance and resource cost implications of VM placement location vary under different cluster load scenarios? Is VM location more important when there is more/less contention for resources?

Public IaaS :

RQ-5: Currently, how prevalent is public cloud VM-type implementation heterogeneity? What are the performance implications resulting from VM-type implementation heterogeneity for model service hosting?

RQ-6: What are the performance implications of hosting scientific modeling workloads on worker VMs having high *cpuSteal* measurements? How effective is using *cpuSteal* to identify worker VMs with high resource contention due to multi-tenancy (e.g. noisy neighbors)?

7.1.2. Research Contributions

In this chapter we propose multiple techniques to improve public and private IaaS cloud infrastructure management for hosting scientific modeling services. Through empirical evaluation we quantify the benefits of these techniques by demonstrating model performance improvements and reduced hosting costs. Our results are generalizable and of interest to *any* practitioner hosting service oriented applications in the cloud.

The primary contributions described in this chapter include:

Private IaaS :

1. We present the Least-Busy VM placement/job scheduler (Section 4.2). The Least-Busy scheduler harnesses our Busy-Metric to quantify the aggregate load (CPU/disk/ network) on each physical host in a private cloud (Section 4.1). Our Busy-Metric supports VM placement/job scheduling decisions to hosts with the most available capacity.
2. We conduct an empirical evaluation to investigate the implications of VM placement on model service performance when scaling infrastructure to meet dynamically increasing service demand. We quantify service performance and hosting cost (# of VMs) implications of round-robin and Least-Busy VM placement (Section 6). Additionally implications of host VM size (# of CPU cores) and effects of shared cluster load for VM placement for dynamic scaling are studied (Section 7).

Public IaaS:

3. We investigate model service performance implications of public cloud VM-type heterogeneity for hosting scientific model services. We quantify model service performance variance to evaluate the importance of the trail-and-better approach [10] for

hosting scientific modeling workloads (Section 8).

4. We investigate the utility of the *cpuSteal* CPU metric at detecting resource contention in a public cloud. We propose the "*CpuSteal* Noisy Neighbor Detection Method" (*NN-Detect*) to identify worker VMs with significant resource contention from noisy neighbors (Section 5.3). We quantify model service performance degradation of worker VMs with noisy neighbors to demonstrate potential for performance improvements if *NN-Detect* is used to prune noisy VMs from worker pools (Section 9).
5. We demonstrate our Least-Busy approach to distribute model service requests across worker VMs in a public cloud. We demonstrate performance improvements vs. least-connection load balancing (Section 7.5).

7.2. BACKGROUND AND RELATED WORK

7.2.1. Private Cloud VM-Placement

Amazon's public cloud implements the Elastic Compute Cloud (EC2) application programming interface (API) enabling programmatic control of resource elasticity. The EC2 API is supported by many open source cloud VIMs. Scientific model services applications can harness the EC2 API to enable scalability using private and/or public cloud resources. Private clouds can provide base application infrastructure with demand bursts serviced using public cloud resources (hybrid cloud). Private clouds providing implementations of the EC2 API include: Apache CloudStack [67], Eucalyptus [46], OpenNebula [68], and OpenStack [69].

All private IaaS clouds provide similar mechanisms for provisioning VMs on demand. Eucalyptus supports greedy and round robin VM placement schemes [46]. VM deployment can be localized to specific clusters or subnets using security groups and availability zones. Apache CloudStack provides "fill first" VM placement, equivalent to greedy allocation, and "disperse" mode, equivalent to round-robin [67]. OpenStack provides two primary VM schedulers known as fill-first and spread-first. Fill-first, equivalent to greedy placement, packs VMs tightly onto PMs. Spread-first distributes VMs across PMs in round-robin fashion, but schedules VMs on PMs having the highest number of available CPU cores and memory first. OpenStack supports filters which enable VMs to be co-located or separated as desired to achieve specific VM deployments. OpenNebula provides both a "packing" policy, equivalent to greedy placement, and a "striping" policy equivalent to round-robin [68], [75]. Additionally, custom "rank" expressions are supported which calculate hosting preference scores for each PM. When a VM launch request is received, the PM with the highest score is delegated as host. Scores are recalculated for each VM launch request. Eight system variables can be used in custom rank expressions, none of which include resource load parameters describing CPU, disk or network utilization. Supported variables include: hostname, total CPUs, free CPUs, used CPUs, total memory, free memory, used memory, and hypervisor type.

Of the stock VM schedulers offered by private IaaS cloud software, none support load aware VM placement across physical hosts. Only capacity parameters such as # of CPUs, available memory and disk space are considered to ensure VM allocations have sufficient resources to run. To better support dynamic scaling of scientific model services applications, VM schedulers should consider resource utilization across physical resources to improve application performance and cluster load balancing.

7.2.2. Dynamic Scaling

Previous research on dynamic scaling in the cloud has investigated WHEN to scale including work on autonomic control approaches and hotspot detection schemes [21], [27]–[29], [31]. These and other efforts additionally focus on WHAT to scale in terms of vertical and horizontal scaling [26], [64]. Investigations on WHERE to scale have largely focused on task/service placement [30], [32] or supporting VM live migration for load balancing [31], [76], [77] or energy savings via VM consolidation across physical hosts [14], [31], [77]–[79].

Kousiouris et al. benchmarked all possible configurations for different task placements across several VMs running on a single PM [32]. Their approach did not consider VM scalability, but focused on modeling to predict performance of task placements on already provisioned VMs. In [30], Bonvin et al. proposed a virtual economy which models the economic fitness of web application component deployments across server infrastructure. Server agents implement the economic model on each node to ensure fault tolerance and adherence to SLAs. Bonvin's approach allocated web server components, not VMs, at the application level (e.g. PaaS). Scaling up an application was supported by adding hosting capacity or migrating existing components to "more economical" servers.

Wood et al. developed Sandpiper, a black-box and gray-box resource manager for VMs [15]. Sandpiper provides hotspot detection to determine when to vertically scale VM resources or perform live migration to alternate hosts. Sandpiper's VM-scheduling and management algorithms were designed to oversee VM migration and server partitioning. Horizontal scaling for dynamic scaling was not supported. Andreolini et al. proposed VM management algorithms which support determining WHEN and WHERE to perform VM live migration [78]. Their algorithms harness VM load profiles to detect hotspots and PM load profiles to determine candidate hosts. Andreolini's algorithms were only by simulation and their approach did not consider dynamic scaling or placement of new VMs.

Beloglazoc and Buyya proposed adaptive heuristics to support live migration of VMs to achieve power savings while adhering to SLAs [14]. They evaluated their approach using simulation but did not consider dynamic scaling or placement of new VMs. Roytman et al. proposed algorithms to consolidate VMs to achieve power savings while minimizing performance losses in [79]. Their approach reduced performance degradation as much as 52% compared with existing power saving consolidation algorithms but was limited to placement of single core VMs. The authors mention for their algorithms to schedule VMs which share CPU cores, new approaches to characterize resource contention are necessary.

Xaio et al. developed a skewness metric, an aggregate measure of VM resource utilization. Skewness reflects how balanced VM placements are across cloud PMs [77]. They combined their skewness metric with hot spot detection to perform VM live migration to achieve better load balancing and to consolidate workloads onto fewer servers for energy savings when possible. Mishra and Sahoo identified problems with the use of aggregate resource utilization metrics for VM placement and proposed a series of different metrics to address orthogonal problems [76]. They did not evaluate their approach and admit their heuristics may not be efficient to implement in practice. Of the reviewed methods, none (1) specifically address VM placement for dynamic scaling in a private cloud, or (2) evaluate implications for hosting dynamically increasing scientific model service workloads.

7.2.3. Scientific Modeling on Public Clouds

Ostermann et al. provided an early assessment of public clouds for scientific modeling in [80]. They assessed the ability of 1st generation Amazon EC2 VMs (e.g. m1.*-c1.*) to host HPC-based scientific applications. They identified that EC2 performance, particularly network latency, required an order of magnitude improvement to be practical and suggested that scientific applications should be tuned for operation in virtualized environments. Other efforts highlight the same challenges regarding EC2 performance and network latency for scientific HPC applications [81]–[83].

Schad et al. [18] demonstrated the unpredictability of Amazon EC2 VM performance caused by contention for physical machine resources and provisioning variation of VMs. Using a Xen-based private cloud Rehman et al. tested the effects of resource contention on Hadoop-based MapReduce performance by using IaaS-based cloud VMs to host worker nodes [16]. They investigated provisioning variation of different deployment schemes of cloud-hosted Hadoop worker nodes and observed performance degradation when too many worker nodes were physically co-located.

Farley et al. demonstrated that Amazon EC2 instance types had heterogeneous hardware implementations in [11]. Their investigation focused on the m1.small instance type and demonstrated potential for cost savings by discarding VMs with lesser performant implementations. Ou et al. extended their work by demonstrating that heterogeneous implementations impact several Amazon and Rackspace VM types [10]. They found that the m1.large EC2 instance had four different hardware implementations (variant CPU types) and different Xen CPU sharing configurations. They demonstrated ~20% performance variation on operating system benchmarks for m1.large VM implementations. They provided a "trail-and-better" approach where VM instances upon launch are benchmarked, and lower performing implementations terminated and relaunched. They demonstrated cost savings through better performance when on demand EC2 instances are used for 10 hours or more.

Providing infrastructure elasticity for service oriented applications by launching new VMs when resource deficits are first detected is challenging. In [7], Kejariwal reports on scaling techniques at Netflix used in Amazon EC2. Some Netflix application components required pre-provisioning up to thirty minutes in advance due to long application initialization times. Kejariwal describes techniques used at Netflix to profile historical service demand to predict future load requirements. Load prediction is required to support prelaunching resources in advance to enable ample initialization time. Determining if scientific model services exhibit predictable usage patterns to support infrastructure preprovisioning is likely a domain specific question, and an area for future research.

7.3. THE VIRTUAL MACHINES SCALER

To investigate infrastructure management techniques and support hosting of scientific modeling web services we developed the Virtual Machine (VM) Scaler, a REST/JSON-based web services application [84]. VM-Scaler harnesses the Amazon EC2 API to support application scaling and cloud management and currently supports Amazon's public elastic compute cloud (EC2), and Eucalyptus 3.x clouds. VM-Scaler provides cloud control while abstracting the underlying IaaS cloud and is extensible to any EC2 compatible cloud. VM-Scaler provides a platform for conducting IaaS cloud research by supporting experimentation with hotspot detection schemes, VM management/placement, and job scheduling/ proxy services.

Upon initialization VM-Scaler probes the host cloud and collects metadata including location and state information for all PMs and VMs. An agent installed on all VMs/PMs sends resource utilization statistics to VM-Scaler at fixed intervals. Collected resource utilization statistics are described in [13], [85]. The development of VM-Scaler extends and enables further our previous work investigating the use of resource utilization statistics for guiding cloud application deployment.

VM-Scaler supports horizontal scaling of application infrastructure by provisioning VMs when application hotspots are detected. One or more VMs can be launched in parallel in response to application demand. To initiate scaling, a service request is sent to VM-Scaler to begin monitoring a specific application tier. VM-Scaler monitors the tier and launches additional VMs when hotspots are detected. VM-Scaler handles launch failures, automatically reconfigures the proxy server, and provides application specific configuration before adding new VMs to a tier's working set. Tier-based scaling in VM-Scaler is conceptually similar to Amazon auto-scaling groups [73].

VM-Scaler supports both resource utilization threshold and application performance model-based approaches to hotspot detection. To evaluate For Least-Busy VM placement resource utilization threshold hotspot detection is used. Scaling is triggered when preconfigured thresholds are exceeded for specific resource utilization variables. This approach is reactive to current system conditions and is application agnostic. This eliminates bias and supports experimentation because the hotspot detection approach remains constant while evaluating different VM scheduling algorithms using different model service applications.

Three configurable timing parameters are provided to support autonomic scaling: `min_time_to_scale_again`, `min_time_to_scale_after_failure`, and `max_VM_launch_ time`. `Min_time_to_scale_again` provides a time buffer before scaling again, allowing time to consider the impact of recent resource additions. This parameter helps to eliminate the ping-pong effect described in [7] and is equivalent to Amazon Scaling Group cool-down periods [73]. `Max_VM_launch_time` provides a maximum time limit before terminating launches that appear to have stalled. This supports handling launch failures by reissuing stalled launch requests. `Min_time_to_scale_after_ failure` provides an alternate wait time to improve scaling responsiveness when VM launch failures occur.

VM-Scaler supports multiple VM placement schemes on Eucalyptus private clouds. These include Least-Busy VM placement, Eucalyptus native placement (round-robin or greedy), and fixed VM placement to a specific host.

7.4. PRIVATE IAAS CLOUD HOSTING

7.4.1. Busy-Metric

The Busy-Metric ranks resource utilization by calculating total CPU time (*cputime*), disk sector reads (*dsr*), disk sector writes (*dsw*), network bytes sent (*nbs*), and network bytes received (*nbr*) for all VMs and PMs. Each resource utilization parameter is normalized to 1 by dividing by the observed maximums of the physical hardware. CPU time is double weighted to assign more importance to free CPU capacity.

A VM capacity parameter is included to prevent too many VMs from being allocated to a single host. Busy-Metric scores of the physical host increase linearly for each additional VM hosted at a rate described using equation 3. The rate increases faster for hosts with fewer CPU cores. Incorporating this parameter enables Busy-Metric to favor hosts having the fewest guest VMs. When PMs host fewer guests the degree of hypervisor level context switching required to multiplex resources is reduced. This practice should help reduce virtualization overhead.

Agents installed on all VMs and PMs are configured to send VM-Scaler resource utilization data every 15 seconds. One second averages using the last minute of data samples were used to calculate the Busy-Metric. Observed values for each parameter are divided by approximate one second maximum capacities of the physical hardware determined through testing.

For example:

$$cputime_n = \frac{cputime_{obs\_1sec}}{cputime_{\max\_1sec}} \quad (1)$$

Our Busy-Metric is expressed as:

$$\frac{(2 \cdot cputime_n) + dsr_n + dsw_n + nbr_n + nbs_n + (\frac{2 \cdot Hosted\_VMs}{PM_{cores}})}{7} \quad (2)$$

Each additional VM hosted linearly increases the value of the Busy-Metric by:

$$e^{(\ln PM_{cores} - 1.2528)} \quad (3)$$

The Busy-Metric provides an approach to rank available capacity of physical host machines. Our goal has been to develop a general metric to support VM scheduling based on the total shared load on PMs. **Many Busy-Metric variations are possible. Our goal has not been to develop the perfect metric, but to investigate implications of VM placement for dynamic scaling.**

Algorithm 1 Sequential VM Launch

```
 1:if (hotspot and current_time >
   min_time_to_scale_again) or (recent_failure and
   current_time > min_time_to_scale_after_failure)
   then
 2:   PM ← Least-BusyPM{ All_PMs }
 3:   Launch(VM on PM)
 4:   while VM is launching do
 5:     if current_time > max_VM_launch_time then
 6:       recent_failure ← true
 7:       exit
 8:     end if
 9:   end while
10:   perform_application_specific_config(VM)
11:end if
```

7.4.2. Least-Busy VM Placement

Using our Busy-Metric, we implemented a VM scheduler using Eucalyptus which places new VMs on the least busy physical hosts (Algorithm 1). When a VM launch request is received

Busy-Metric values are calculated for all physical hosts. New VMs are launched on hosts having the lowest resource utilization rankings.

Algorithm 2 Parallel VM Launch

```
Unused_PMs ← { All_PMs }
while All VMs are not placed do
  PM ← Least-BusyPM{ Unused_PMs }
  BM_PM ← Busy-Metric(PM)
  if (BM_PREV != null) then
    /* Schedule NON-First VM */
    if ((BM_PM - BM_PREV) > MIN_DIST_NEXTBUSY
     AND BM_PREV < DOUBLE_SCHEDULE_MAX) then
      /* Distance too far, schedule PM again */
      AddToLaunchQueue(PM_PREV, VM)
      /* Forget prev PM - don't reschedule */
      BM_PREV ← null
    else
      /* Next PM not busy, schedule there */
      AddToLaunchQueue(PM, VM)
     Unused_PMs ← Unused_PMs – PM
     BM_PREV ← BM_PM
     PM_PREV ← PM
    end if
  else
  /* Schedule First VM */
    AddToLaunchQueue(PM, VM)
    Unused_PMs ← Unused_PMs – PM
    BM_PREV ← BM_PM
    PM_PREV ← PM
  end if
end while
```

To support launching multiple VMs in parallel we developed a parallel VM launch algorithm (Algorithm 2), to spread VM launches accordingly based on PM Busy-Metric scores. Two distance thresholds are used to double schedule launches on a single PM or spread them across multiple Least-Busy PMs. Launching more VMs in parallel than the available number of PMs is not presently supported. If the distance between the Least-Busy host and the second Least-Busy host exceeds the MIN_DIST_NEXTBUSY threshold then two VMs are launched on the Least-Busy host. No more than two VMs will be launched in parallel on a single PM for a given scaling task. If a PM's Busy-Metric exceeds the DOUBLE_SCHEDULE_MAX threshold then the PM is considered too busy to support launching more than 1 VM and doubling scheduling is avoided. Our parallel launch algorithm respects that launching multiple VMs on a single host can produce undesired load spikes, but this is acceptable if the next Least-Busy PM is sufficiently busier than the Least-Busy PM.

Eucalyptus version 3.x does not natively support launching VMs on a specific host. VM launches are supported using either round-robin (spread-first) or greedy (fill-first) launch. To circumvent this limitation a workaround approach was employed to achieve specific worker VM placements based on round-robin placement without modifying Eucalyptus. The effectiveness of our workaround is demonstrated by our evaluation of Least-Busy VM placement for dynamic scaling discussed in sections 6 and 7.

7.5. PUBLIC IAAS CLOUD HOSTING

7.5.1. VM Type Implementation Heterogeneity

Previous research has demonstrated that hardware implementations of public cloud VM types change over time [10], [11]. Additionally, several hardware implementations of the same VM type may be offered at the same time each with different performance characteristics. When hosting scientific modeling workloads on public clouds we are interested in understanding the implications of VM type implementation heterogeneity.

VM-Scaler provides VM pools, collections of VMs of the same machine image type (e.g. AMI). Pools support prelaunching VMs to address launch latency for dynamic scaling and VM reuse when modeling workloads do not exceed the minimum billing cycle time increment. For Amazon EC2, instance time is billed hourly. It is advantageous to retain VMs for the full billing cycle time increment to maximize opportunities for potential reuse.

To investigate implications of VM type heterogeneity, VM-Scaler provides type enforcement capabilities equivalent to the trail-and-better approach for VM pools. VM pool creation supports a "forceCpuType" attribute which when specified forces matching of the backing CPU type for member VMs in a pool. This CPU type enforcement incurs the expense of launching and terminating unmatching instances. In Amazon EC2, discarded VMs are billed for 1-hour of usage. We harness CPU type enforcement feature to investigate type heterogeneity implications for model service performance and describe our results in section 8.

7.5.2. Identifying Resource Contention with *cpuSteal*

Resource contention in a public cloud can lead to performance variability and degradation in a shared hosting environment [16], [18]. *CpuSteal* registers processor ticks when a VM's CPU core is ready to execute but the physical host CPU core is busy performing other work. The core may be unavailable because the hypervisor (e.g. Xen dom0) is executing native instructions or user mode instructions for other VMs. High *cpuSteal* time can be a symptom of over provisioning of the physical servers hosting VMs.

On the Amazon EC2 public cloud which uses a variant of the Xen hypervisor, we observe a number of factors which produce *CpuSteal* time. These include:

1. Processors are shared by too many VMs, and those VMs are busy.
2. The hypervisor kernel (Xen dom0) is occupying the CPU.
3. The VM's CPU time share allocation is less than 100% for one or more cores, though 100% is needed to execute a CPU intensive workload.

In the case of 3, we observe high *cpuSteal* time when executing workloads on Amazon EC2 VMs which under allocate CPU cores. A specific example is the m1.small and m3.medium VMs. In spring of 2014, we observed that the m3.medium VM type is allocated approximately 60% of a single core of the 10-core Xeon E5-2670 v2 CPU at 2.5 GHz. Because of this underallocation, all workloads executing at 100% on m3.medium VMs exhibit high *cpuSteal* because they must burst and use unallocated CPU time to reach 100%. These burst cycles are granted only if they are available, otherwise *cpuSteal* ticks are registered. *CpuSteal* is the only CPU metric specifically related to virtualization.

7.5.3. *CpuSteal* Noisy Neighbor Detection Method

We investigate the utility of *cpuSteal* as a means to detect resource contention from "noisy neighbors". Noisy neighbors are busy co-located VMs, which compete for similar resources that can adversely impact performance. We propose the following "CpuSteal Noisy Neighbor Detection method" (*NN-Detect*):

Step 1. Execute processor intensive workload across pool of worker VMs.

Step 2. Capture total *cpuSteal* for each worker VM for the workload.

Step 3. Calculate VM average *cpuSteal* for the workload ($cpuSteal_{\text{avg}}$).

To determine if a worker VM has noisy neighbors $cpuSteal_{\text{VM}}$ should be at least 2 x $cpuSteal_{\text{avg}}$. Additionally a workload specific minimum *cpuSteal* threshold is required. This threshold should be determined by benchmarking representative workloads and observing *cpuSteal*. The minimum number of *cpuSteal* ticks to identify worker VMs with noisy neighbors will depend on characteristics of the computational workload (how CPU bound is it?) and its duration. We describe the evaluation of *NN-Detect* using the WEPS model as the computational workload in section 9.

7.6. PERFORMANCE IMPLICATIONS OF VM PLACEMENT FOR DYNAMIC SCALING

7.6.1. Experimental Setup

To evaluate dynamic scaling for scientific model services in support of RQ-1 and RQ-2 presented in section 1, we harness two environmental model services implemented within the Cloud Services Innovation Platform (CSIP) [70], [86]. Both model services represent diverse applications with varying computational requirements. CSIP has been developed by Colorado State University with the US Department of Agriculture (USDA) to provide environmental modeling web services. CSIP provides a common Java-based framework which supports REST/JSON based service development. CSIP services are deployed using the Apache Tomcat web container [45].

We investigate dynamic scaling for two environmental model web services: the Revised Universal Soil Loss Equation – Version 2 (RUSLE2) [41], and the Wind Erosion Prediction System (WEPS) [71]. RUSLE2 and WEPS are the US Department of Agriculture–Natural Resource Conservation Service standard models for soil erosion used by over 3,000 county level field offices across the United States. RUSLE2 and WEPS are used within CSIP to provide soil erosion modeling services to end users. RUSLE2 was developed primarily to guide natural resources conservation planning, inventory erosion rates, and estimate sediment delivery. The Wind Erosion Prediction System (WEPS) is a daily simulation model which outputs average soil loss and deposition values for selected areas and times to predict soil erosion due to wind.

RUSLE2 was originally developed as a Windows-based Microsoft Visual C++ desktop application. WEPS was originally developed as a desktop Windows application using Fortran95 and Java. RUSLE2 and WEPS are deployed as REST/JSON based web services hosted using Tomcat. RUSLE2 and WEPS are good candidates to prototype scientific model services scaling. Their legacy model implementations are analogous to many legacy scientific models which might utilize IaaS cloud computing as a means to provide scalable model services. Both models consist of a multi-tier architecture including a web application server, geospatial relational database, file server, and logging server.

Table 7.1. Rusle2/WEPS Application Components

Component		RUSLE2	WEPS
$\mathcal{M}$	Model	Apache Tomcat 6.0.20, Wine 1.0.1, RUSLE2, OMS3 [43] [44]	Apache Tomcat 6.0.20, WEPS
$\mathcal{D}$	Database	Postgresql-8.4, PostGIS 1.4, soils data (1.7 million shapes), management data (98k shapes), climate data (31k shapes), 4.6 GB total for Tennessee	Postgresql-8.4, PostGIS 1.4, soils data (4.3 million shapes), climate/wind data (850 shapes), 17GB total, western US data.
$\mathcal{F}$	File server	nginx 0.7.62 file server, 57k XML files (305MB), parameterizes RUSLE2 model runs.	nginx 0.7.62 file server, 291k files (1.4 GB), parameterizes WEPS model runs.
$\mathcal{L}$	Logger	Codebeamer 5.5, Apache Tomcat (32-bit), Ia-32libs	Redis 2.2.12 distributed cache server

RUSLE2 and WEPS model service components are described in Table 7.1. To load balance model service requests we used HAProxy, a high performance load balancer [51], to redirect modeling requests across the active pool of $\mathcal{M}$ worker VMs. HAProxy, installed on a PM, provides public service endpoints.

7.6.2. Hardware Configuration

We conducted scaling tests using a Eucalyptus 3.1.2 IaaS private cloud deployed across nine SUN X6270 blade servers interconnected by a Giga-bit VLAN. Each blade server had dual Intel Xeon X5560-quad core 2.8 GHz CPUs, 24GB ram, and dual 15000rpm HDDs. The host operating system was Ubuntu 12.04 Linux (3.2.0-29) 64-bit server. The Xen hypervisor version 4.1.2 provided VMs in paravirtual mode. VM guests ran Ubuntu Linux 9.10 (2.6.31) 64-bit server. Six blade servers were used as Eucalyptus node-controllers to host VMs. One blade server hosted the Eucalyptus cloud-controller, cluster-controller, walrus server, and storage-controller services. Eucalyptus managed mode networking was used to support network isolation of VMs using private VLANs. A separate blade server generated the modeling work load. Another blade server acted as a client for file transfers to create background network activity for shared load testing.

Random test generation was used to generate 10,000 unique RUSLE2 and 1,000 WEPS test cases. RUSLE2 tests used geospatial data from the state of Tennessee. WEPS tests accessed data primarily from Kansas and Colorado where soil erosion due to wind is a large environmental concern. For scaling tests, individual WEPS model runs were terminated after 10 minutes. This was necessary because some randomly generated WEPS runs required more than 30 minutes to execute.

7.6.3. Test Configurations

To investigate model service performance implications for RQ-1 and RQ-2 we used a fixed simulated shared cluster load. We generated an artificial load across the six PMs used to host VMs. To investigate RQ-1 and RQ-2, tests were performed using the "medium load" shared cluster load described in Table 7.5. The table shows shared load characteristics and initial corresponding Busy-Metric scores for the physical hosts. Our goal was to simulate potential public cloud load conditions where users compete for server resources. Custom scripts generated load activity. CPU load was created for a specified number of cores by performing continuous math computations. Disk load was created by continuously reading, writing, or copying a text file. To force the system to continuously reread the file, cache clearing was performed. To create network load a VM image file was constantly transferred to/from a non-cloud blade server. Sftp's "-l" flag was used to control the transfer bandwidth.

Table 7.2 describes VM size, modeling request rates, and request rate increments for RQ-1 and RQ-2 scaling tests. An exponential distribution based random sleep function spaced individual model requests to simulate a Poisson distribution. This supported random request arrival while achieving the desired constant request rate.

Table 7.2. VM Scaling Tests for RQ-1 and RQ-2

VM size	Mem(MB)/ Disk(GB)	RUSLE2 Req Rate	WEPS Req Rate	Increment RUSLE2	Increment WEPS
2-core	1024 / 3	.5-16/sec	.1-1/sec	.25/15s	.025/15s
4-core	2048 / 3	1-16/sec	.1-1/sec	.5/30s	.05/30s
8-core	4096 / 3	2-16/sec	.1-1/sec	1/min	.1/min

Scaling thresholds for hotspot detection were increased linearly for 2-core, 4-core, and 8-core VM tests. The intent was to use scaling thresholds relative to the number of VM CPU cores. Figure 7.1 provides a quartile plot of RUSLE2 vs. WEPS execution times. WEPS model runs consume nearly 100% of a CPU core for their duration averaging from 80-100 seconds compared to a few seconds for RUSLE2 runs. RUSLE2 features very short model execution time, more I/O, and context switching. Both workloads exhibit different resource use behavior providing for two fairly different test systems.

For RUSLE2, hotspot detection was performed by monitoring resource utilization of the initial worker VM as individual model execution times were short (2s average) and homogenous. Load was evenly balanced across worker VMs using HAproxy round-robin load balancing [51]. This approach was insufficient for WEPS, as model runs had twice the variance and were much longer in duration. For WEPS hotspot detection we calculated average CPU time, CPU idle time, and number of context switches for the entire pool of worker VMs and launched additional worker VMs when averages exceeded the scaling thresholds.

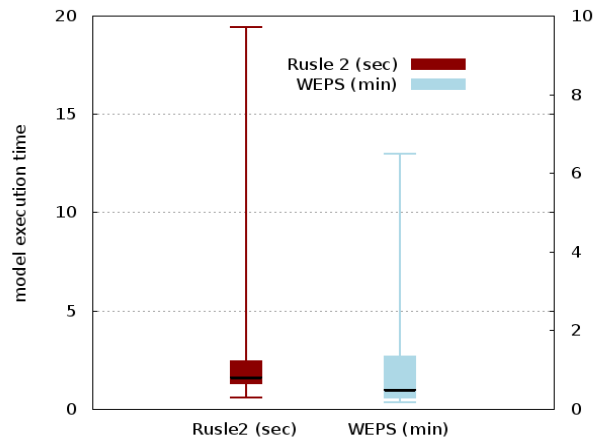


Figure 7.1. RUSLE2 vs. WEPS Model Execution Time Quartile Box Plot

7.6.4. Experimental Results

Sixty RUSLE2 and sixty WEPS scaling test sets were completed to support investigation of RQ-1 and RQ-2 from section 1. Scaling tests were conducted twice, once using Least-Busy VM placement and again using round-robin VM placement. Individual test sets included ~6500 RUSLE2 model runs and 300 WEPS model runs. A total of over 800,000 model runs were completed. For 2-core VM WEPS testing, sequential VM launches were insufficient to achieve good results. To add resources more rapidly three 2-core VMs were launched in parallel.

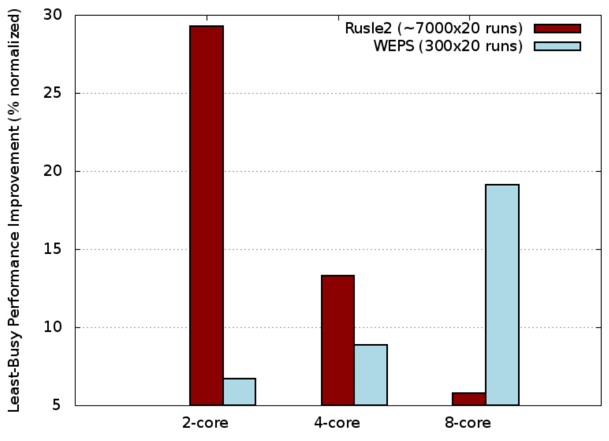


Figure 7.2. Least-Busy VM Placement (RQ-1 & RQ-2):
Percentage Normalized Performance Improvement

Table 7.3 shows statistical significance from t-tests calculated to determine if performance means for scaling performance with Least-Busy were different from round-robin. Normalized performance improvements of Least-Busy VM placement relative to round-robin are shown in figure 7.2. RUSLE2 model service performance improved by 16% on average using Least-Busy VM placement, while WEPS performance improved 12%. Least-Busy demonstrated its largest differential performance for RUSLE2 with 2-core VM tests (29.3% faster, 3.2 hrs. cputime savings/scaling test), and WEPS for 8-core tests (19.1% faster, 1.6 hrs. cputime savings/scaling test). Least-Busy VM placement enabled better model performance for all tests.

Table 7.3. Scaling Test Results for RQ-1 and RQ-2

VM size	RUSLE2	WEPS

2-core VMs	*lb<rr*
	p=.014
	df=18.2
		lb<rr
		p=.162
		n.s. [1]

4-core VMs	*lb<rr*
	p= 0.065
	df = 22.7
		lb<rr
		p= .035
		df = 24.65

8-core VMs	*lb<rr*
	p=.017
	df=24.5
		lb<rr
		p=.00003
		df=33.796

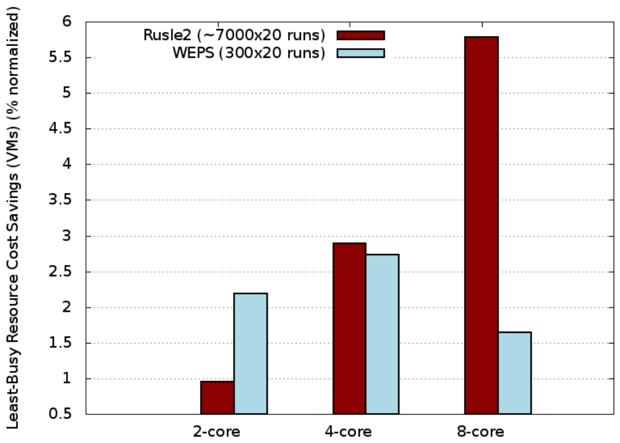


Figure 7.3. Least-Busy VM Placement (RQ-1 & RQ-2):
Percentage Normalized Cost Savings (# VMs)

Figure 7.3 shows the normalized resource cost savings in percentage of VM allocations.

Least-Busy VM placement supported execution of the modeling workload with fewer resources. RUSLE2 required on average 3.2% fewer VMs and WEPS 2.2%. The most economical deployment used 2-core VMs for RUSLE2 (28.92 avg. cores) and 4-core VMs for WEPS (30.56 avg. cores). **Least-Busy VM placement enabled hosting modeling workloads with fewer VMs and total CPU cores. Less physical server capacity was required for hosting while faster modeling performance was achieved.**

7.7. IMPLICATIONS OF VM SIZE AND SHARED CLUSTER LOAD FOR DYNAMIC SCALING

7.7.1. Experimental Setup

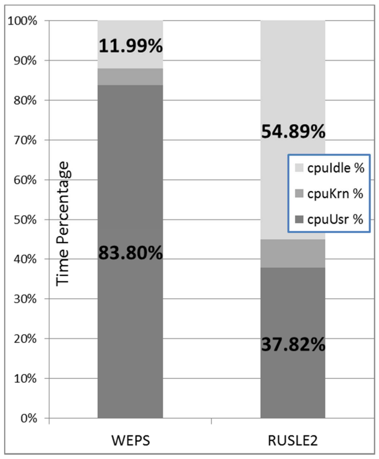


Figure 7.4. CPU Utilization WEPS and RUSLE2 Model Services

Table 7.4. Scaling Tests for RQ-3 and RQ-4

VM size	no-load	medium-load	high-load
2-core VMs	20 scale tests	20 scale tests	--
4-core VMs	20 scale tests	20 scale tests	20 scale tests
8-core VMs	20 scale tests	20 scale tests	20 scale tests

To investigate RQ-3 and RQ-4 from section 1, dynamic scaling tests were conducted using the RUSLE2 and WEPS model services described in section 6.1. CPU utilization for WEPS and RUSLE2 model workloads is depicted in figure 7.4. WEPS model runs are largely

CPU bound occupying approximately 84% of the CPU during execution, while RUSLE2 runs only occupy approximately 38%. To investigate implications of VM placement relative to VM size (RQ-3) all scaling tests were repeated using 2-core, 4-core, and 8-core VMs under three different shared cluster load scenarios. Performance implications of shared cluster load (RQ-4) for dynamic scaling are investigated using the three shared load scenarios described in table 7.5. These shared load scenarios are referred to as no-load (nl), medium-load (ml), and high-load (hl). Initial cluster Busy-Metric values under test loads are provided in the table.

Table 7.5. Shared Cluster Load

Cloud Node		R2	WEPS	CPU	Disk	Network	Busy-Metric
high-load (hl)	PM-1	ℳ	𝒟	6 cores@25%			.241
	PM-2	𝒟 ℒ	ℒ	12 cores@25%			.475
	PM-3	ℱ				↑ @100%	.424
	PM-4		ℳ ℱ	3 cores@25%		↓ @100%	.249
	PM-5				R/W@100%		.264
	PM-6			6 cores@25%	W@ 100%		.571
med-load (ml)	PM-1	ℳ	𝒟	2 cores@25%			.083
	PM-2	𝒟 ℒ	ℒ	4 cores@25%		↑ @20%	.285
	PM-3	ℱ		6 cores@25%			.240
	PM-4		ℳ ℱ	5 cores@25%		↓ @20%	.240
	PM-5			2 cores@25%			.082
	PM-6			4 cores@25%			.156
no-load (nl)	PM-1	ℳ	𝒟				.038
	PM-2	𝒟 ℒ	ℒ				.074
	PM-3	ℱ					.038
	PM-4		ℳ ℱ				.003
	PM-5						.003
	PM-6						.003

7.7.2. Test configurations

Table 7.4 describes tests completed to investigate RQ-3 and RQ-4. All tests were repeated using Least-Busy VM placement and round-robin VM placement. 160 test sets of 6,500 RUSLE2 model runs and 160 test sets of 300 WEPS model runs were performed with the increasing request rates described previously in table 7.2. A total of over 2,000,000 individual model service requests were performed. 2-core VM high-load tests were not performed because under medium-load our cluster struggled to cope.

7.7.3. Experimental Results

Table 7.6 shows statistical significance from t-tests comparing model service performance of Least-Busy vs. round-robin VM placement for RUSLE2. Table 7.7 shows t-test results for the WEPS model. Normalized performance improvements of Least-Busy VM placement compared with round-robin are shown in figure 7.5. Using normalized performance enables the graph to depict performance for both RUSLE2 and WEPS.

Table 7.6. RUSLE2 Scaling Performance (RQ-3) and (RQ-4)

VM size	no-load	medium-load	high-load
2-core VMs	*lb>rr* p=0.032 df=26.397	*lb<rr* p=.014 df=18.2	--
4-core VMs	*n.s.*	*lb<rr* p= 0.065 df = 22.7	*lb<rr* p=.006 df=8.1
8-core VMs	*lb<rr* p=.016 df=32.9	*lb<rr* p=.017 df=24.5	*lb<rr* p=.011 df=28.8

Table 7.7. WEPS Scaling Performance (RQ-3) and (RQ-4)

VM size	no-load	medium-load	high-load
2-core VMs	*n.s.*[1]	*n.s.* [1]	--
4-core VMs	*lb<rr* p=.006 df=15.087	*lb<rr* p= .035 df = 24.65	*n.s.*
8-core VMs	*n.s.*	*lb<rr* p=.00003 df=33.796	*n.s.*

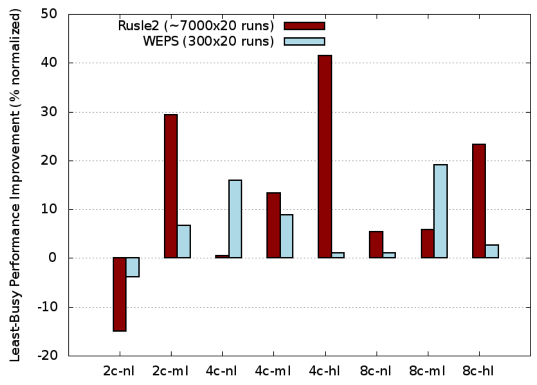


Figure 7.5. Least-Busy VM Placement (RQ-3 & RQ-4):
Percentage Normalized Performance Improvement.

Averaging across all tests, RUSLE2 model service response time improved 13% using Least-Busy VM placement. The average WEPS performance improvement was 5.3%. Least-Busy exhibited its fastest differential performance gain for RUSLE2 on 4-core high-load tests (41.5% faster, 4.9 hrs. cputime savings), and WEPS for 8-core medium-load tests (19.1% faster, 1.6 hrs. cputime savings). Least-Busy VM placement enabled better model service performance for all but 2-core no-load tests. With no shared cluster load launching VMs using round-robin placement is initially very fast. For 2-core scaling tests, launching many more VMs combined with overhead from our Eucalyptus VM placement work around led to slower Least-Busy performance.

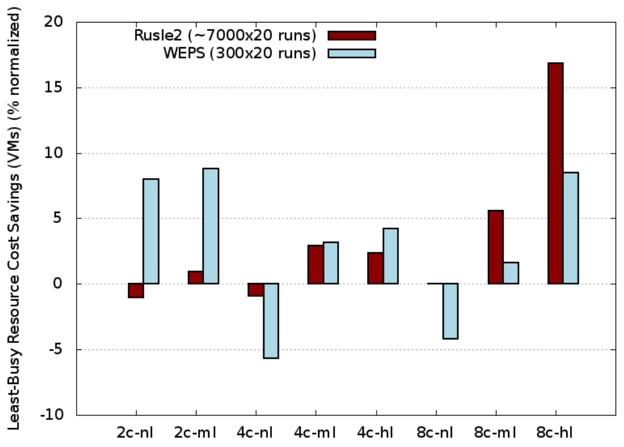


Figure 7.6. . Least-Busy VM Placement (RQ-3 & RQ-4):
Percentage Normalized Cost Savings (# VMs)

Figure 7.6 shows the normalized resource cost savings in percentage of VM allocations. Least-Busy VM placement supported execution of the modeling workload with fewer resources. RUSLE2 required on average 3.4% fewer VMs and WEPS 1.6%. The most economical deployment used 2-core VMs for RUSLE2 and 4-core VMs for WEPS. Least-Busy VM placement led to lower cost deployments in most cases. Hosting our modeling workload using Least-Busy VM placement required fewer total VMs on average than round-robin when comparing normalized VM launch counts across all tests for RUSLE2 (p=.01846 , df=320) and WEPS (p=.03786, df=250).

Loosening the static RU scaling thresholds may be necessary for better WEPS performance. For 4-core no-load WEPS tests we doubled the `min_cpu_idle` threshold from 140 to 280 and observed WEPS performance improvements of 14.4% for Least-Busy and 28.1% for round-robin. Least-Busy provided 15.9% better performance using the 140 `min_cpu_idle` scaling threshold vs. round-robin. Both approaches provided near identical performance at the 280 `min_cpu_idle` scaling threshold. Overall stress from hosting the WEPS workload was higher than RUSLE2 resulting in 15% slower average VM launch times reducing responsiveness of dynamic scaling. **This illustrates a tradeoff between workload intensity and scaling responsiveness in small cluster settings.**

For both models, smaller VMs were able to execute the workload using fewer total CPU cores, though performance suffered somewhat. RUSLE2 2-core VMs serviced the workload while occupying 37% fewer CPU cores than 8-core VMs, and WEPS 4-core VMs serviced the workload with 17% fewer CPU cores freeing these resources for other tasks.

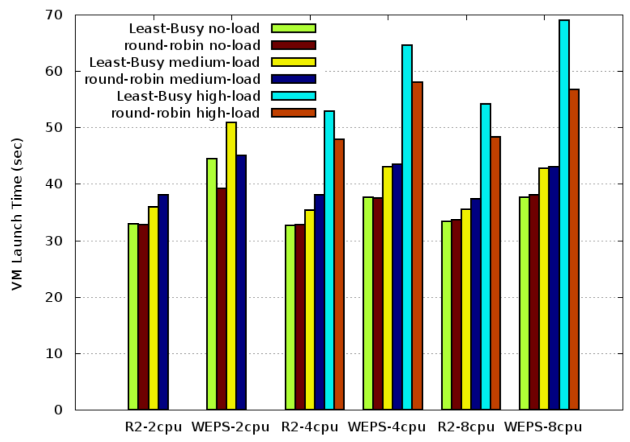


Figure 7.7. VM Launch Times (seconds) (RQ-3 & RQ-4): Least-Busy vs. Round-Robin VM Placement

7.7.4. VM Launch Performance

Average VM launch times are shown in figure 7.7. Our results show that under no-load, VM launch times appear similar for Least-Busy and round-robin. 2-core WEPS tests launched three VMs in parallel. This produced longer VM launch times in cases where the Least-Busy PM launched multiple VMs simultaneously. For 4-core and 8-core high-load tests Least-Busy exhibited higher average VM launch times than round-robin but supported faster average model execution times. Least-Busy prioritizes CPU utilization 2 to 1 vs. disk/network I/O. Least-Busy selected hosts with more CPU capacity but simulated I/O on these hosts slowed VM launches. Once launches were complete, these PMs with less CPU contention supported faster model execution times.

7.7.5. Busy-Metric Testing on Amazon EC-2

We harnessed our Busy-Metric it to perform WEPS model request job scheduling on Amazon EC2. We executed 12 sets of 1,000 WEPS runs each where VM-Scaler provided proxy services to schedule incoming modeling requests across a pool of 76 2-core m2.xlarge worker VMs. This provided an application agnostic approach to job scheduling similar to HAProxy load balancing [51]. For evaluation we compared our Least-Busy job scheduling performance with HAProxy least connection load balancing using the exact same configuration. With least connection load balancing HAProxy monitors the number of active sessions and distributes new sessions evenly. We observed that Least-Busy job scheduling reduced WEPS average model execution time 1.6% vs. HAProxy least connection. Statistical significance of this performance improvement is confirmed by t-test (df=.049, df=22.99).

7.7.6. Analysis

Supporting dynamic scaling for our modeling service workloads using VMs with fewer cores proved challenging as many more VMs had to be rapidly launched on our small cluster stressing system disk and network resources. For WEPS 2-core tests launching VMs in parallel increased overhead and degraded performance illustrating tradeoffs between cluster size, launch overhead, and application performance. For 2-core medium-load tests, nearly continuous sequential VM launches were required for RUSLE2 to cope with demand. WEPS model execution times nearly doubled with 20% of runs timing out after 10 minutes. Parallel VM launches helped our WEPS 2-core VM configuration achieve performance similar to 4-core configurations.

Our results demonstrate that VM placement location was most important when: (1) scaling under stress either from launching many VMs, or (2) when coping with high cluster load. We observed VM launch latency increase with shared cluster load making it more difficult to provide resource elasticity on demand.

7.8. PERFORMANCE IMPLICATIONS OF VM-TYPE HETEROGENEITY

7.8.1. Experimental Setup

To investigate performance implications of VM-type heterogeneity for model service performance (RQ-5), we launched 50 VMs of each type to test for the presence of type implementation heterogeneity. When heterogeneity was detected, we launched 50 more VMs for a total of 100. In [10], VM type heterogeneity is described for m1.small, m1.large, and m1.xlarge across all Amazon EC2 east subregions. We extended previous work by testing type heterogeneity for 1st generation VM types (m1.medium, m1.large, m1.xlarge, c1.medium, c1.xlarge), 2nd generation types (m2.xlarge, m2.2xlarge, and m2.4xlarge) and 3rd generation types (c3.large, c3.xlarge c3.2xlarge, m3.large). Tests were performed in May and July of 2014 using two Amazon regions: us-east-1c and us-east-1d. Type heterogeneity was determined by inspecting Linux's /proc/cpuinfo. We then investigated model service performance implications for the two most heterogeneous types detected. WEPS and RUSLE2 model service performance tests were conducted using pools of 5 VMs for each type variant. We repeated 10 trials of 100 WEPS runs and 660 RUSLE2 runs using these type-fixed pools to determine average model service execution time.

We did not investigate VM type heterogeneity for very small VMs which do not receive 100% of a full CPU core allocation such as the bursting VMs (e.g. t1.small) and 1-core VMs (e.g. m1.small, m3.medium). These VM types provide insufficient throughput for good model service performance.

7.8.2. Experimental Results

VM-type heterogeneity observations made on Amazon EC2 are reported in table 7.8. VM types with no heterogeneity are not shown in the table. This does not imply that type heterogeneity is not possible for these types now or in the future: it simply means for our tests, none was detected. Our tests revealed implementation type heterogeneity for 1st and 2nd generation VMs. We are not expressly interested in quantifying exact heterogeneity amounts as these will change. As CPUs continue to evolve even 3rd generation EC2 VMs will likely be implemented using more core dense and power thrifty CPUs in the future.

Table 7.8. Amazon VM Type Heterogeneity

VM type	Region	Backing CPU	Backing CPU
m1.medium	us-east-1c	Intel E5-2650 v0 8c,95w,96%	Intel Xeon E5645 6c,80w,4%
m2.xlarge	us-east-1c	Intel Xeon X5550 4c, 95w, 48%	Intel Xeon E5-2665 v0 8c, 115w, 42%
m1.large	us-east-1d	Intel Xeon E5-2650 v0 8c,95w,74%	Intel Xeon E5-2651 v2 12c,105w,19%
m1.large	us-east-1d	Intel Xeon E5645 6c,80w,7%	--
m2.xlarge	us-east-1d	Intel Xeon E5-2665 v0 8c, 115w,78%	Intel Xeon X5550 4c, 95w, 22%

Compared to Ou et al.'s results [10], we observed that many CPU types reported in 2011 and 2012 have been replaced with lower power, core-dense CPUs, marking a trend towards their adoption. For example, the m1.xlarge VM is now implemented using 12-core Intel Xeon E5-2651s. Using high core density CPUs should help reduce resource contention while enabling server real estate to expand. Previous m1.large VMs implemented using AMD Opteron's consumed as much as 42.5 watts per core, where today m1.large Intel Xeon E5-2651 v2 implementations require only 8.75 watts per core. This amounts to just ~20% the previous power requirement.

We observed substantial type heterogeneity for the m1.large and m2.xlarge VMs implemented in two EC2 subregions. Model service performance implications are shown in figure 7.8. For m1.large VMs, model performance was slower using the Intel Xeon E5-2650-v0 backed implementation. Average model service response time for RUSLE2 was 108%, and 109% for WEPS versus the Intel Xeon E5645 backed m1.large. For Intel Xeon E5-2665-v0 backed m2.xlarge VMs, model service performance was 114% for RUSLE2 and 104% for WEPS respectively versus the Intel Xeon X5550. **Ironically, newer CPUs provided lower performance than their legacy counterparts in both cases.**

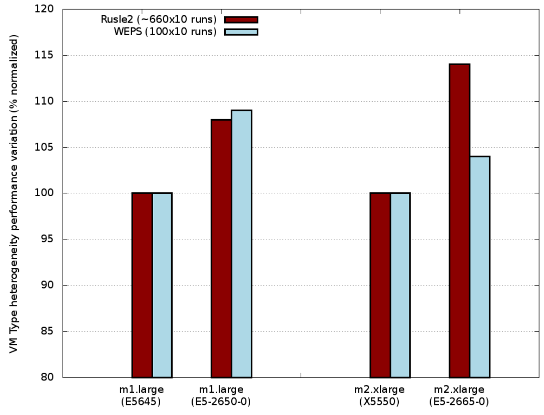


Figure 7.8. VM Type Heterogeneity Performance Variation

7.9. DETECTING RESOURCE CONTENTION WITH *CPUSTEAL*

7.9.1. Experimental Setup

To investigate the utility of *cpuSteal* for detecting resource contention from VM multi-tenancy (RQ-6) we proposed our "*CpuSteal* Noisy Neighbor Detection Method" (*NN-Detect*) in section 5.3. To evaluate *NN-Detect* we used the WEPS model as our test workload since it is more CPU bound than RUSLE2 (figure 7.4).

For **step-1**, we first tested for the presence of *cpuSteal* by launching 50 VMs for each VM type listed in table 7.9. Exceptions included: 8-core c1.xlarge (25 VMs), 1-core m1.medium and 1-core m3.medium (60 VMs). We repeated 4 test sets of approximately 1,000 WEPS runs. Model runs were distributed evenly using HAProxy round-robin load balancing across all available CPU cores of the worker VM pool. Each CPU core received approximately 20 model runs to execute.

Table 7.9. Amazon EC2 *CpuSteal* Analysis

VM type	Backing CPU	Average R^2 linear reg.	Average *cpuSteal* per core	% with Noisy Neighbors
us-east-1c				
c3.large-2c	E5-2680v2/10c	.1753	2.35	0%
m3.large-2c	E5-2670v2/10c	-	1.58	0%
m1.large-2c	E5-2650v0/8c	.5568	7.62	12%
m2.xlarge-2c	X5550/4c	.4490	310.25	18%
m1.xlarge-4c	E5-2651v2/12c	.9431	7.25	4%
m3.medium-1c	E5-2670v2/10c	.0646	17683.2[1]	n/a
c1.xlarge-8c	E5-2651v2/12c	.3658	1.86	0%
us-east-1d				
m1.medium-1c	E5-2650v0/8c	.4545	6.2	10%
m2.xlarge-2c	E5-2665v0/8c	.0911	3.14	0%

The question we sought to answer was, *is there a pattern to cpuSteal behavior across worker VMs over time?* We collected individual VM *cpuSteal* ticks for the 4 test sets (**step-2**). We then took VM *cpuSteal* values from each test set and used linear regression to test if one set's outcome could predict *cpuSteal* for future test sets. The averaged R^2 values from these comparisons appear in table 7.9. R^2 is a measure of prediction quality that describes the percentage of variance explained by the regression. The important discovery here is that a fair degree of the variance is explained for some VM types. **We observe that previous VM *cpuSteal* behavior is useful at predicting future *cpuSteal* in the public cloud.** This relationship was observed for 4 VM types (m1.large, m2.xlarge, m1.xlarge, m1.medium) at ($R^2 > .44$), and as high as R^2=.94 for m1.xlarge.

Some important observations here include: (1) a complete set of tests for one VM type required up to 5 hours to complete. Throughout this time, trends in *cpuSteal* remained consistent to produce our statistically significant model service performance differences described in table 7.9. And, (2) in most cases when *cpuSteal* could not be predicted over time, it was because it was nonexistent: essentially in these cases *cpuSteal* values were too low to be discerned from low-level noise!

We next calculated average *cpuSteal* for the pool of VMs and used the average to determine a threshold level at which to classify a worker VM as having noisy neighbors (**step-3**). For WEPS, in addition to needing at least 2x average *cpuSteal*, worker VMs required a minimum threshold of approximately 12 *cpuSteal* ticks per CPU core. Below this threshold level the amount of *cpuSteal* becomes too low to distinguish noisy neighbors. We sorted *cpuSteal* values for all worker VMs and checked for VMs exceeding the thresholds. When no worker VMs exceeded these thresholds for a given type (e.g. c3.large), *cpuSteal* was not useful at detecting potential performance degradation. In these cases there simply was not enough multi-tenancy present. Table 7.9 shows the "% Noisy Neighbors" identified with by *NN-Detect*. Cases without multi-tenancy primarily occurred on VMs hosted by new highly dense 10 and 12-core CPUs. (e.g. c3.large, m3.large, m3.medium, c1.xlarge). This observation is intuitive. Modern 10 and 12 core Intel Xeon CPUs all feature hyperthreading and new servers often have dual or quad CPUs per server. By increasing the number of available hyperthreads in a dual-CPU server from 16 (X5550) to 48 (E5-2651) the incidence of multi-tenancy should be cut to ¼ the previous amount. Such advances in hardware help mitigate public cloud resource contention until future usage increases to consume this additional capacity.

7.9.2. Experimental Results

We compared the performance of high *cpuSteal* worker VMs with low *cpuSteal* VMs using small 5-VM pools. We executed 10 sets of 100 WEPS runs, and 10 sets of 660 RUSLE2 runs on both a high, and low *cpuSteal* VM pool. Normalized performance degradation from worker pools with high *cpuSteal* is shown in table 7.10. Performance of the low *cpuSteal* VM pool is normalized to 100%. Across 4 VM-types we observe model service performance degradation up to 18% for WEPS and 25% for RUSLE2 using m1.large VMs. Statistically significant (p > .05) performance differences are observed for m1.large, m2.xlarge, and m1.medium VMs. The average performance degradation for both RUSLE2 and WEPS was 9%.

Table 7.10. EC2 Noisy Neighbor Model Service Performance Degradation

VM type	Region	WEPS	RUSLE2

m1.large
E5-2650v0/8c
	us-east-1c	117.68%
		df=9.866
		$p=6.847\cdot10^{-8}$
			125.42%
			df=9.003
			p=.016

m2.xlarge
X5550/4c
	us-east-1c	107.3%
		df=19.159
		p=.05232
			102.76%
			df=25.34
			$p=1.73\cdot10^{-11}$

c1.xlarge
E5-2651v2/12c
	us-east-1c	100.73%
		df=9.54
		p=.1456
			102.91%
			n.s.

m1.medium E5-2650v0/8c	us-east-1d	111.6% df=13.459 $p=6.25\cdot10^{-8}$	104.32% df=9.196 $p=1.173\cdot10^{-5}$

7.10. CONCLUSIONS

Optimizing performance of scientific model services hosted on private and public clouds requires awareness of modeling resource requirements and careful management of cloud-based virtual infrastructure.

To improve model services hosting in private clouds we developed the Least-Busy VM placement scheduler. Least-Busy uses a flexible Busy-Metric to determine CPU, disk, and network I/O utilization of all physical hosts and virtual guests in a private cloud. The Least-Busy VM scheduler harnesses this metric to make load-aware VM placements. We demonstrate performance improvements up to 41% (RUSLE2) and 19% (WEPS) using our load-aware Least-Busy VM placement scheduler to host dynamically increasing model service workloads. (**RQ-**

1). Using our Least-Busy VM placement scheduler enables hosting these workloads with up to 17% fewer VMs for RUSLE2 and 9% fewer for WEPS (**RQ-2**).

Hosting dynamically increasing workloads using large VMs (e.g. 8 CPU cores) is not resource efficient (**RQ-3**). Our results demonstrate hosting RUSLE2 workloads using 2-core VMs required 37% fewer CPU cores, and hosting WEPS workloads with 4-core VMs required 33% fewer CPU cores than with 8-core VMs. Careful VM placement is needed when hosting workloads with a high shared cluster load (**RQ-4**). In these settings Least-Busy VM placement provided performance gains up to 41% for RUSLE2 and ~3% for WEPS while utilizing fewer CPU cores to host modeling workloads.

For public cloud hosting of model services, the trial-and-better approach can improve model service performance and lower hosting costs. Using Amazon EC2 we achieve up to 14% WEPS model performance improvement (m2.xlarge) and 9% performance improvement (m1.large) (**RQ-5**) by harnessing VM-type heterogeneity.

To address public cloud resource contention for model service hosting we provide the *CpuSteal* Noisy Neighbor Detection Method (NN-Detect). Using NN-Detect we demonstrate how trends in VM *cpuSteal* measurements can be harnessed to identify worker VMs with high resource contention from noisy neighbors. Using NN-Detect we identify worker VMs with model service performance degradation up to 18% for WEPS and 25% for RUSLE2 (**RQ-6**).

Abstraction of physical hardware using IaaS clouds leads to the simplistic view that resources are homogenous and scaling can infinitely provide linear increases in performance. Our results demonstrate how careful VM placement in private clouds, and trail and better evaluation of VMs in public clouds, mitigate resource contention and address hardware heterogeneity to deliver performance improvements. Our results contribute key infrastructure management techniques which help improve scientific model service performance while reducing hosting costs in both public and private clouds.

CHAPTER 8

HARNESSING RESOURCE UTILIZATION MODELS FOR COST EFFECTIVE INFRASTRUCTURE ALTERNATIVES

8.1. INTRODUCTION

Deploying service oriented applications (SOAs) to Infrastructure-as-a-Service (IaaS) clouds requires selection of both the *type* and *quantity* of VMs adequate for workload hosting. Public IaaS clouds offer a wide array of VM appliance types featuring different hardware configurations. These VM appliance types provide fixed allocations of CPU cores, system memory, hard disk capacity and type (spindle vs. solid state), and network throughput allocation. By focusing on providing a limited number of VM types, cloud providers can leverage economies of scale to improve performance and availability of VM types in hardware procurement and management. Given an ever increasing number of VM types offered by public cloud vendors, it is becoming increasingly difficult to make informed choices for SOA deployment. For example, at the time of this writing Amazon EC2 and HP Helion offer 34 and 11 predefined VM types respectively, each with different CPU, memory, disk, and network bandwidth allocations available for different costs.

Quantifying the performance expectations of cloud resources is difficult. Amazon EC2 and HP Helion's clouds use qualitative "compute units" to describe relative processing capabilities of VMs. Amazon EC2 describes VM performance using elastic compute units (ECUs), where one ECU is stated to provide the equivalent CPU capacity of a 1.0-1.2 GHz 2007 AMD Opteron or Intel Xeon processor. An HP Cloud Compute Unit (CCU) is advertised to be roughly equivalent to the minimum power of 2/13th of one logical core (a hardware hyperthread) of an Intel 2.6 GHz 2012 Xeon CPU. Recently, Amazon has stopped directly marketing ECUs for its 3rd generation VM-types, though ECUs are still listed in the management console interface [8]. Additionally, Amazon employs approximate categories of expected network throughput. They include: very low, low (250 Mbps), moderate (500 Mbps), high (1000 Mbps), and 10 Gigabit.

Not only do cloud vendors offer a diverse array of VM-types, investigations have shown that VM types are often implemented using heterogeneous hardware resulting in performance variance [10], [11]. Ou et al. identified no less than five hardware implementations of the m1.large Amazon VM-type in 2011, with performance variance up to 28% [10]. Ou also observed the use of different CPU time sharing allotments to implement the m1.large VM type. In some cases, Amazon EC2 multi-core VMs were found to not receive 100% allotments of every core. Using CPU benchmarking techniques we confirmed this phenomenon by observing variant CPU core allocations of the 4-core m1.xlarge backed by the Intel Xeon E5-2650 v0 @ 2.0 GHz. Maximum observed CPU utilization could not be made to exceed 100%, 100%, 95%, and 75% CPU for each respective core.

Beyond VM type heterogeneity challenges, previous research has demonstrated how resource contention from multi-tenancy on VM hosts results in SOA performance variance and degradation [32], [40], [64], [80]. Provisioning variation, the uncertainty of the physical location of VMs across physical hosts, also has been shown to contribute to application performance variance and degradation [13], [16].

In summary, determining the best VM type for SOA hosting is complicated by: (1) a plethora of vendor provided VM-types, (2) vague qualitative descriptions of VM capabilities, (3) heterogeneous vendor hardware and hypervisor configurations, and (4) performance variance from resource contention and provisioning variation across shared hardware. Given these challenges, a practitioner's effectiveness at employing ***only*** intuition to make architectural choices which balance performance and cost tradeoffs for SOA deployment is increasingly in doubt.

8.1.1. Workload Cost Prediction Methodology

Making informed choices regarding VM deployments for SOA hosting requires both *(1) characterization of the workloads* and *(2) benchmarking the performance capabilities* of available VM types. In this paper, we present a workload cost prediction methodology that harnesses **both** to support determination of infrastructure requirements for achieving equivalent performance for SOA workloads.

To develop our approach we focus on hosting workloads consisting of a large number of individual service requests. We focus on achieving equivalent total execution time for the entire workload, irrespective of individual service request execution times. This enables the use of low cost, low throughput VMs to achieve total workload performance equivalent to using high throughput VMs.

Our approach supports determining the *type* and *quantity* of VMs to achieve equivalent workload performance. We consider SOA hosting using VM pools consisting of a single VM type. We do not investigate using pools with mixed VM types. We believe the utility of mixing VM types would be only to achieve some degree of vertical scaling in a public cloud. Vertical scaling is useful, for example, when the optimal infrastructure deployment of 4-core VMs for a given workload is determined to be 22 cores. With vertical scaling this workload could be hosted using five 4-core VMs, and one 2-core VM of similar processing speed.

Identifying multiple potential infrastructure configurations that achieve equivalent performance enables practitioners to select the most economical VM type for SOA workload hosting. Infrastructure costs can be calculated for each VM type by multiplying fixed or spot market prices by the predicted quantity of VMs to derive monetary costs. These cost predictions can then be compared to determine the most cost effective virtual infrastructure to provide the required quality of service for SOA workloads.

Unlike related work in cost optimization for cloud workloads we do not assume that application workloads are identical [87]–[89]. We profile representative SOA workloads and build predictive resource utilization models. These models convert resource requirements from a selected base VM type to alternate VM types needed to achieve equivalent performance. We investigate: (1) the efficacy of our methodology at resource profile prediction for different VM appliance types, (2) the ability of our methodology to determine the required number of VMs to achieve equivalent performance using different VM appliance types, and (3) the efficacy of our methodology at determining the most economical cost predictions using different VM types.

We focus our analysis on service oriented applications where individual service requests are executed independently in parallel. Our evaluation involves analysis of several environmental science applications deployed as web services. We do not investigate SOA workloads where individual service requests are largely parallel, though we suspect our approach is still applicable as long as representative workloads are used to train our predictive models.

We first considered cloud application performance modeling using resource utilization statistics in [85]. We harnessed this approach to predict performance of various component compositions across VMs in [13], [33]. These efforts have demonstrated how intuition is insufficient to determine the best performing componentVM compositions, and quantified the resulting performance variance of ad hoc deployments. We then built the VM-Scaler application to more easily facilitate resource utilization profiling of large scale application deployments in both private and public cloud settings [84]. The workload cost prediction methodology presented in this paper builds on all of our previous work.

8.1.2. Research Questions

This paper investigates the following research questions:

RQ-1: [equivalent performance] How can equivalent SOA workload performance be achieved across different virtual machine types by harnessing resource utilization profiles of SOA workloads?

RQ-2: [profile prediction] How effectively can we predict individual workload profile resource utilization variables across VM types? Specifically, how well can we predict: CPU-user-time, CPU-kernel-time, CPU-idle-time, and CPU-IO-wait-time?

RQ-3: [profile scaling] When scaling up the number of VMs, how can we account for changes in the SOA workload resource utilization profile variables? Specifically, what changes occur and, how do we accommodate them for: CPU-user-time, CPU-kernel-time, CPU-idle-time, and CPU-IO-wait-time?

8.1.3. Research Contributions

In this paper we present our workload cost prediction methodology to predict hosting costs of SOA workloads harnessing resource utilization models. **Our methodology provides infrastructure configuration alternatives that provide equivalent performance allowing the**

most economical infrastructure to be chosen. Our methodology supports: (1) characterization of workload requirements, (2) predicting the required number of VMs of a given type required to host workloads, while (3) ensuring equivalent performance is achieved.

To support development of our workload cost prediction methodology we additionally contribute:

1. A novel resource checkpointing scheme that supports profiling SOA workload resource utilization for jobs executing across VM pools.
2. A research application of Ou et. al's trial-and-better approach [10] to normalize VM pools to ensure every VM has an identical backing CPU to support SOA workload profiling in a multitenant public cloud.

Our resource utilization checkpointing scheme supports profiling application resource utilization across VM pools. This enables us to quantify the composite resource utilization for the 19 resource utilization variables described in table 8.1. We synchronize collection of resource utilization data to the nearest second to ensure profiles reflect resource use for only the SOA workload being benchmarked. We know of no similar effort, which collects the breadth of resource utilization statistics with one second synchronization for workloads executing across large pools of VMs.

Our research represents a novel application of trial-and-better approach to homogenize public cloud infrastructure for supporting experimentation. **We argue that all public cloud research should use trial-and-better to reduce heterogeneity of tested resources.** Trial-and-better provides an important tool to reduce variance of measurement and testing in public clouds.

8.1.4. Chapter Organization

The remainder of this paper is organized as follows: Section 2 provides an overview of related research for approaches that support cost optimization for cloud-based infrastructure and workload deployment. Section 3 describes our workload cost prediction methodology to achieve equivalent SOA workload performance using different VM appliance types. This section addresses RQ-1 and discusses our use of Linux CPU time accounting principles for our workload cost prediction methodology. In Section 4 we describe our use of environmental science SOAs to validate our methodology. We introduce VM-Scaler and the concept of SOA workload resource utilization checkpointing. Hardware and test configurations are discussed. Section 5 presents results of our evaluation and analysis of our workload cost prediction methodology in support of RQ-2 and RQ-3. Section 6 summarizes our findings and contributions, and Section 7 discusses future work.

8.2. BACKGROUND AND RELATED WORK

Research on cloud economics and application hosting costs can be broken down into efforts focused on demand based pricing models (spot markets), and investigations on the cost implications of infrastructure management and scaling approaches.

Amazon introduced spot virtual machine instances as a method to sell unused datacenter capacity in late 2009. Spot instances enable bidding for spare public cloud capacity by granting resources to users whose bids exceed current spot prices. When demand spikes, user VMs whose bid price falls below the current market price are terminated instantly, freeing capacity for higher bidders. Spot instances are ideal for executing computational workloads for scientific modeling where the time of execution is less important than completing the workloads at minimum cost. Spot instances were harnessed to conduct our research.

A number of efforts have investigated spot instance pricing and similar demand based pricing mechanisms [88]–[91]. These efforts employed modeling to predict or set prices. Yi et al. investigated the use of job checkpointing as a mechanism to reduce job costs executed using spot instances [88]. Their approach was limited to supporting jobs with fixed execution times and was evaluated by simulation using spot price histories. Andrzejak et al. developed a model which supports users by providing bid suggestions while considering resource availability, reliability, performance, and resource costs [89]. Their approach was limited to compute intensive, embarrassingly parallel jobs whose computation is easily divided.

Other efforts primarily have focused on infrastructure management to minimize hosting costs [10], [11], [87], [92]–[94]. In [95], Galante and E. de Bona provide a survey of recent research on cloud computing elasticity. They identify 28 works which consider elasticity for infrastructure, platform, and application hosting. Of these only one study [87], focused on cost optimization of application hosting and scaling.

In [87] Sharma et al. describe Kingfisher, a management system supporting cost-aware application hosting and scaling for IaaS clouds. Kingfisher determines the most cost effective approach to transition existing application infrastructures to target infrastructures to meet service level agreements (SLAs). Transitions considered include vertical and horizontal scaling, as well as VM live migration. Kingfisher was evaluated using Amazon’s public cloud and a local private XEN-based cloud. Kingfisher assumes that each VM can service a fixed volume of incoming requests and that all requests require the same resources to process.

In [92], Leitner et al. developed an SLA-aware client side request scheduler which minimizes “aggregate” hosting costs by balancing both price and SLA requirements. They evaluated their approach by simulation using workload archival data to test how their scheduler responds. They compared the aggregate costs of their algorithms with: (1) the minimum infrastructure (1 VM for all requests), (2) the maximum infrastructure (1 VM for each request) and (3) a bin-packing approach which fully packs existing resources before allocating additional VMs. Their approach provided the lowest aggregate costs but their bin packing approach did not address infrastructure launch latency.

Simarro et al. provide a cost aware VM-placement scheduler which seeks to reduce infrastructure costs by provisioning VMs across cloud data centers having the lowest infrastructure prices [93]. Their schedulers use price forecasts to predict pricing trends to support the most economical infrastructure placements. Their approach reduced infrastructure costs but did not address network latency and performance issues resulting when application infrastructure is simultaneously provisioned across different data centers.

In [94] Villegas et al. provide a performance and cost analysis of provisioning and job scheduling policies in the cloud. They assessed policies from recent literature for their analysis using two private clouds and Amazon EC2. They found that statically provisioned virtual infrastructure delivered better performance, but was up to 5Xs more costly. Conversely dynamically provisioned infrastructure provided lower hosting costs but with performance caveats resulting from infrastructure launch latency similar to [7]. This key cost versus performance tradeoff for infrastructure provisioning highlights the need for good hot spot detection and load prediction techniques [6].

Farley et al. demonstrated that Amazon EC2 instance types had heterogeneous hardware implementations in [11]. Their investigation focused on the m1.small instance type and demonstrated potential for cost savings by discarding VMs with less performant implementations. Ou et al. extended their work by demonstrating that several Amazon and Rackspace VM types exhibit heterogeneous implementations [10]. They identified four different implementations of the m1.large VM on Amazon EC2 with varying performance. Performance variations were attributed to the use of different backing CPUs and XEN scheduler configurations. They harnessed this heterogeneity by developing a "trial-and-better" approach to test new instances and discard poor performing instances. The authors demonstrated cost savings for long running jobs as a result of faster job execution. For our work we adopt Ou's "trial-and-better" approach to improve homogeneity of VM profiling.

Previous research investigating cost implications of IaaS clouds has focused on spot market analysis [90], [91], pricing/bid support [88], [89], cost-aware VM scheduling [87], [93], [94], and job placement schemes [92], [94]. For the surveyed approaches workloads were assumed to be heterogeneous. None of the approaches specifically support diverse workloads with varying resource requirements (e.g. CPU and I/O) [87]–[89]. Conversely, we provide a workload cost prediction methodology which harnesses SOA workload profiles and VM benchmarking to capture the unique resource requirements of diverse workloads. Our methodology provides equivalent workload performance using different VM types and supports cost savings by identifying infrastructure alternatives.

8.3. RESOURCE UTILIZATION MODELS FOR COST PREDICTION

Our resource utilization based approach for SOA workload cost prediction focuses on achieving *equivalent performance* for diverse SOA workloads. For the purposes of our evaluation in section 5, we consider equivalent performance to require executions of the same workload to complete within +/- 2 seconds of the observable wall clock time. Our SOA workloads represent discrete sets of individual service requests which are executed in parallel across virtual infrastructure. We are not concerned with response time of individual service requests, but rather the total workload execution time. In fact, we expect individual requests to perform slower on VM-types having slower CPU clock speeds.

For SOA workloads, we observe service execution times for individual service requests can range from being normally distributed to exponentially distributed. These exponentially distributed workloads tend to waste computational resources due to imbalances in the execution times of service requests running in parallel. In these cases most of the workload completes while the fully provisioned virtual infrastructure is mostly idle as a few remaining runs complete. In these scenarios, it is ideal to schedule long running service requests first to optimize costs when service request duration can be anticipated in advance and if workloads are submitted in a batch. We note this behavior and the potential optimization for service requests scheduling within a workload.

8.3.1. Workload Equivalent Performance

Given SOA workloads, we predict the workload resource utilization requirements for pools of distinct virtual machine types. For example, we have 3 pools: one consisting of c3.xlarge VMs, another m1.xlarge, and a third c1.medium. Our methodology supports determining the required number of virtual machines to provide equivalent workload performance using these different VM pools.

Our technique harnesses Linux CPU time accounting principles to account for all available time across the pool of VMs servicing the workload. Workload wall clock time can be determined by summing all CPU resource utilization variables across the VM pool and dividing by the total number of CPU cores.

$$\text{Workload}_{\text{time}} = \frac{cpuUsr_T + cpuKrn_T + cpuIdle_T + cpuIoWait_T + cpuIntSrvc_T + cpuSftIntSrvc_T + cpuNice_T + cpuSteal_T}{VM_{cores}} \quad (1)$$

Table 8.1. Resource utilization variables tracked by VM-Scaler

RU variable	Description
cpuUsr	CPU time in user mode
cpuKrn	CPU time in kernel mode
cpuIdle	CPU idle time
cpuIoWait	CPU time waiting for I/O to complete
cpuIntSrvc	CPU time servicing interupts
cpuSftIntSrvc	CPU time servicing soft interrupts
cpuNice	CPU time executing prioritized processes (user mode)
cpuSteal	CPU ticks lost to other virtualized guests
contextsw	Number of context switches
dsr	Disk sector reads (1 sector = 512 bytes)
dsw	Disk sector writes (1 sector = 512 bytes)
nbs	Network bytes sent
nbr	Network bytes received
dsreads	Number of completed disk reads
drm	Number of adjacent disk reads merged
readtime	Time in ms spent reading from disk
dswrites	Number of completed disk writes
dwm	Number of adjacent disk writes merged
writetime	Time in ms spent writing to disk
loadavg	Avg # of running processes in last 60 sec

Eight resource utilization variables contribute to the observed wall clock time. These eight variables described in table 8.1 include: *cpuUsr*, *cpuKrn*, *cpuIdle*, *cpuIoWait*, *cpuIntSrvc*, *cpuSftIntSrvc*, *cpuNice*, and *cpuSteal*. We use resource utilization checkpointing, a feature of our VM-Scaler cloud infrastructure management application described in section 4.2, to capture the workload resource utilization. A checkpoint is created at the start of workload execution and used to determine the total resource utilization when the workload concludes. Resource utilization sensors on the VMs send resource utilization data to VM-Scaler. All VMs run Linux's Network Time Protocol daemon (ntpd) to synchronize clock times. VM-Scaler collects this resource utilization data and ensures the samples are synchronized at the beginning and end of workload execution.

Of the eight resource utilization variables, *cpuUsr* and *cpuIdle* account for the majority of the time. For our SOA workload evaluation described in section 5, approximately 93.22% of c3.xlarge SOA execution time is accounted for by *cpuUsr* or *cpuIdle*. *CpuUsr* represents the total amount of computation required by the workload. Through extensive testing, we observe that *cpuUsr* time remains generally the same regardless of the number VMs used to host the workload. Introducing additional VMs into the VM pool adds to the total overhead from background Linux processes and the resource utilization sensor. This overhead is relatively constant and can easily be accounted for. *CpuIdle* represents the unused time where CPU cores have been provisioned but remain idle. Workloads exhibiting high *CpuIdle* time demonstrate parallel execution inefficiencies. This indicates significant resource waste in the service implementation. Applications concerned about cloud hosting costs should be architected to decrease *CpuIdle* time.

CpuKrn is the time a workload spends executing kernel mode instructions. When executing SOA workloads across VMs, we found the ratio of time spent in kernel mode is similar, with slightly more *CpuKrn* time occurring on VMs with slower I/O. For our evaluation in section 5 *CpuKrn* is the third greatest contributor to workload execution time at approximately 5.72%. *CpuIntSrvc* and *CpuSftIntSrvc* represent time spent servicing system interrupts and is generally small. *CpuNice* is time spent executing prioritized processes in user mode. This is rare, and only occurs when SOAs employ process prioritization in an attempt to gain a larger share of the CPU.

CpuSteal is an important, though unusable metric. *CpuSteal* registers processor ticks when a virtual CPU core is ready to execute, but the physical core is busy and unavailable. The CPU may be unavailable because the hypervisor is executing native instructions (e.g. XEN Dom0) or other co-located VMs are currently “stealing” the CPU. The difficultly with this measure is that ticks are only registered when execution should occur, but is unable to. These ticks, unfortunately, do not adequately account for the missing time. When workloads exhibit high *CpuSteal* time, Linux CPU time accounting does not work. On the VM there is essentially “missing time”, which is the gap between accounted for time and actual time. There are a number of factors which cause *CpuSteal* time to occur. These include:

1. Processors are shared by too many VMs, and those VMs are busy
2. The hypervisor is occupying the CPU
3. The VM’s CPU core time share allocation is less than 100%, though 100% is needed for a CPU intensive workload

In the case of 3, we observe high *CpuSteal* time when executing workloads on Amazon EC2 VMs which under allocate CPU cores as described earlier in section 1. A specific example of this is the m1.small [10] and m3.medium VMs. In spring of 2014, we observed that the m3.medium VM type is only allocated 1 core of a 10-core 2.5 GHz Xeon E5-2670 v2 CPU with an approximate 60% timeshare. The m3.medium is advertised to provide 3 ECUs. Because of this significant CPU under allocation, all workloads executing on m3.medium VMs exhibit high *CpuSteal* time making time accounting inaccurate. If the degree of *cpuSteal* in these scenarios remains relatively constant, it should be possible to buffer time calculations to compensate for the missing clock ticks.

8.3.2. Workload Cost Prediction Methodology

The steps of our workload cost prediction methodology for cost calculation are outlined in table 8.2.

Table 8.2. Workload Cost Prediction Methodology

Step	Task
0	Train RU models: $M_{VMtype1}$, .. $M_{VMtype-j}$
1	Profile workload: $RU_{w(VM-base)} \leftarrow (W)$ on n x VM_{base} n=base #VMs
2	Convert: $RU_{w(VM-base)} \rightarrow (M_{all}) \rightarrow RU_w\{n \text{ x } VM_{type1}, .. n \text{ x } VM_{type-j}\}$, n=base #VMs, j=number VM types
3	Scale profiles: $RU_w\{n \text{ x } VM_{type1}, .. n \text{ x } VM_{type-j}\}$, n=n to n+x n=base #VMs, x = scale up #VMs
4	Select profile: $perf(VM_{base})=\{perf(n \text{ x } VM_{type1}),.. perf(n \text{ x } VM_{type-j})\}$ n=#VMs w/ equivalent performance
5	Minimize cost: Select $\min\{cost(VM_{type1}), .. cost(VM_{type-j})\}$

Step 0 – train resource utilization models

In this initialization step, VM-type specific resource utilization models are trained using SOA representative workloads. SOA workload training data is collected for each VM type being considered. Infrastructure configurations use the same number of CPU cores, though not necessarily the same number of VMs for each VM type. For example, to collect training data when the VM_{base} is `4 x c1.xlarge` Amazon VMs (32 total cores), `8 x 4-core m1.xlarge` VMs (32 total cores) and `16 x 2-core m1.large` VMs (32 total cores) should be used.

If there is a domain related set of SOAs then a single set of resource utilization (RU) models (Mall) may be trained. This increases the range of resource utilization scenarios the models are exposed to and offers the potential to predict resource requirements for new models with similar performance requirements. The evaluation of our methodology described in section 5 takes this approach.

RU models are trained using stepwise multiple linear regression. One model is trained for each VM type being considered. Each model converts RU data from VMbase to one of the alternate VM types: VMtype(i). We trained RU models using the R statistical package.

Step 1 – profile workload resource utilization

A base VM-type, VM_{base}, is used to support our prediction methodology. Representative samples of SOA workloads are profiled using VM_{base}. For our workloads (*W*) we collect the total resource requirements (RU_{w}) across the set of (*n*) VMs, where *n* is the number of VM_{base} VMs provisioned.

Step 2 – convert resource utilization profile

The workload resource utilization profile $RU_{w(VM\text{-}base)}$ is converted using VM-type specific resource utilization models (M_{all}). Multiple linear regression is used to train (M_{all}) models. These performance models (M_{all}) generate "predicted" resource utilization profiles, ($RU_{w(VM\text{-}type(1..j))}$), for *n* VMs for each possible VM type ($_{1..j}$). These profiles ($RU_{w(VM\text{-}type(1..j))}$) represent the resource utilization to execute the workload (*W*) with *n* VMs. However, we know based on the VM's performance rating (e.g. ECUs, CCUs, GHz, etc.) of VM_{base} vs. each possible VM type ($_{1..j}$) that *n* VMs are likely insufficient for equivalent performance. We address scaling from *n* to *n+x* VMs in step 3.

To simplify the cost prediction methodology, it is best to select VM_{base} to be either a very fast or slow offering so that all other VM types can be scaled in the same direction for equivalent performance. The number of VMs (*n*) must be scaled up (or down) depending on the VM type of VM_{base}.

Step 3 – scale resource utilization profile

To identify infrastructure configurations that provide equivalent workload performance to

$RU_{w(VM\text{-}base)}$, we scale resource utilization profiles $RU_w\{n \times VM_{type1}, .. n \times VM_{type\text{-}j}\}$ from *n* to *n+x* VMs, where *x* is the maximum quantity of VMs over *n* required for equivalent performance. For our investigations described in sections 4 & 5, we observed *x* values of approximately 2 to 4. We suspect *x* will be larger in situations where the performance difference between VM_{base} and the slowest/fastest alternate VM-type is large.

To scale our resource utilization profiles $RU_w\{VM_{type1}, .. VM_{type\text{-}j}\}$ from *n* to *n+x* VMs, we address how individual profile variables change when more VM resources are added to execute the workload. This is RQ-3 from section 1. We investigate two different approaches: Resource Scaling Approach 1 (RS-1) and Resource Scaling Approach 2 (RS-2). **For scaling CPU-bound SOA workloads, effort is focused on scaling up *cpuUsr* and *cpuKrn* time.** For RS-1, we only scale *cpuUsr* and *cpuKrn* because they account for most of the system time (98.94%). If scaling workloads are I/O bound, it becomes important to address scaling of *cpuIoWait*. For RS-2, we additionally include *cpuIoWait* scaling. These approaches exhibit an effort vs. accuracy tradeoff. More accuracy can generally be obtained with greater effort. From a research perspective, we investigate how much accuracy is required (RQ-3).

RS-1: Application Agnostic

Resource Scaling Approach 1 (RS-1) is agnostic to the SOA being scaled. For RS-1 idle SOA VMs are benchmarked in isolation to determine their idle resource consumption. Idle VMs have an active resource utilization sensor sending data to VM-Scaler and typical background Linux server processes. The observed *cpuUsr* time represents the overhead for adding VMs to the pool. Each VM type is tested separately to benchmark background task resource consumption. The average number of background *cpuUsr* ticks per second is then determined.

This background overhead rate is used to scale *cpuUsr* based on workload duration for Step 3. For RS-1 the remaining parameters are converted though not scaled up in the profile: *cpuKrn*, *cpuIoWait*, and *cpuSftIntSrvc*. These parameters account for only a small fraction of the total system time, and represent primarily background activity not directly related to the SOA workload. Table 8.3 shows RS-1 scaling of *cpuUsr* with *cpuKrn* conversion but no scaling for the WEPS SOA (described in 4.1) with a c3.xlarge VM_{base} and $VM_{type(i)}$ of m1.xlarge.

RS-2: APPLICATION AWARE HEURISTIC

Resource Scaling Approach 2 (RS-2) attempts to address application specific characteristics of how resource utilization profiles change when resources are scaled up. A set of scaling runs is conducted using sample workloads for each SOA scaling from *n* to *n+x*. The average percentage change as a result of scaling up by 1 VM for each resource utilization parameter is calculated for *cpuUsr*, *cpuKrn*, and *cpuIoWait*. This average percentage change for each variable is then used to scale the application specific profiles to account for how each application's resource utilization changes relative to the application agnostic approach.

Step 4 – select resource utilization profile

Once SOA workload resource utilization profiles have been converted to alternate VM types (step 2), and scaled up or down (step 3), the final step is to select the profile which provides equivalent SOA performance. An illustration of the selection problem appears in table 8.3. The first row represents converted profile output from step 2. We harness equation 1 to solve for *cpuIdle* time. With only 5 VMs *cpuIdle* is negative! With the specified "wall-time goal" for equivalent performance, there is not enough physical time to execute the workload. Each additional VM increases the total number of available clock ticks. **However, it is**

insufficient to select the first line where *cpuIdle* is not negative. To achieve equivalent performance for SOA workloads there has to be **extra *cpuIdle* time** to account for overhead, context switching, I/O, etc.

We need an approach which estimates when enough *cpuIdle* time is available to provide equivalent performance to VM_{base}. We describe two alternative profile selection approaches: Profile Scaling Approach 1 (PS-1) and Profile Scaling Approach 2 (PS-2) to estimate the required *cpuIdle* time for equivalent performance.

Table 8.3. Scaling Profiles RS-1 & RS-2

RS-1 (WEPS c3.xlarge → m1.xlarge)

VMs / cores	**wall time-goal**	**available clock ticks**	***cpuUsr***	***cpuKrn***	***cpuIdle***
5 / 20	96.774s	193548	219561	10642	-38536
6 / 24	96.774s	232258	220622	10642	-888
7 / 28	96.774s	270967	221684	10642	36760
8 / 32	96.774s	309677	222745	10642	74409
9 / 36	96.774s	348386	223807	10642	112057
10 /40	96.774s	387096	224868	10642	149705

RS-2 (WEPS c3.xlarge → m1.xlarge)

VMs / cores	**wall time-goal**	**available clock ticks**	***cpuUsr***	***cpuKrn***	***cpuIoWait***	***cpuIdle***
5 / 20	96.774s	193548	219561	10642	1867	-38536
6 / 24	96.774s	232258	221822	10856	2005	-2440
7 / 28	96.774s	270967	224107	11074	2153	33619
8 / 32	96.774s	309677	226416	11297	2312	69638
9 / 36	96.774s	348386	228748	11524	2483	105618
10 /40	96.774s	387096	231104	11755	2667	141556

PS-1: Application Agnostic

Profile Selection approach 1 (PS-1) is agnostic to the SOA being scaled. For PS-1 we convert the *cpuIdle* time from *n* x VM_{base} to *n* x $VM_{type\text{-}j}$. We know there must be more than *n* x $VM_{type\text{-}j}$ *cpuIdle* time after scaling to achieve equivalent performance. We also expect more *cpuIdle* to be required than simply the converted *cpuIdle* time value for *n* VMs. We need to know cpuIdle time with $n + x$ VMs. For PS-1 we derived a simple linear function to determine a percentage to increase *cpuIdle* time for each additional VM. Our equation is derived by calculating the average observed % growth in *cpuIdle* time for all SOAs when scaling up with m1.xlarge 2 ECU VMs. We then assumed 0% growth for the VM_{base} of c3.xlarge (3.5 ECUs), and linear growth based on the VM's number of ECUs to derive the linear scaling equation:

$$cpuIdle\%_{growth} = -6.5715\, ECUs + 23 \qquad (2)$$

Our equation expresses percentage growth as a number from 1 to 100, and supports increasing *cpuIdle* time faster for slower VM deployments. From SOA workload testing we observe that slower VMs require more *cpuIdle* to achieve equivalent performance. This approach to scale *cpuIdle* for profile selection is application agnostic. We take advantage of ECUs already being a normalized measure of CPU performance. If ECUs were unavailable a similar approach using CPU clock speed could be derived though we would need to compensate for generational improvements in CPU performance. For example a 2012 Intel Xeon CPU at 2.5 GHz is somewhat faster than a 2010 Xeon at the same clock rate. Table 8.3 shows PS-1 selection as the dark grey row. PS-1 and PS-2 identify the same row in the scaling profile example.

PS-2: APPLICATION AWARE HEURISTIC

Our second profile selection approach (PS-2) attempts to address application specific characteristics relating to *cpuIdle* time when infrastructure is scaled up. We convert the *cpuIdle* time from VM_{base} to $VM_{type\text{-}j}$. After conversion, we scale the required *cpuIdle* time for selection using the SOA specific average percentage change in *cpuIdle* from application scaling test observations. This approach does not assume all SOA *cpuIdle* requirements scale the same, but applies an application specific scaling factor to support determination of required *cpuIdle* time. Table 8.3 shows PS-2 selection as the dark grey row.

Step 5 – minimize cost

Once profile selection has identified the number of VMs for equivalent performance using alternate VM types, we can then calculate infrastructure costs. Cost can be determined by multiplying the required number of VMs by fixed or spot market prices to determine deployment costs. The lowest priced infrastructure can be selected for SOA workload execution while maintaining equivalent performance.

8.4. EXPERIMENTAL INVESTIGATION

8.4.1. Environmental Modeling Services

To evaluate our workload cost prediction methodology and investigate the research questions presented in section 1, we harness six environmental science SOAs from the Cloud Services Innovation Platform (CSIP) [70], [86]. These six SOAs represent a diverse array of applications with varying computational requirements and architectures. CSIP has been developed by Colorado State University with the US Department of Agriculture (USDA) to provide environmental modeling services. CSIP provides a common Java-based framework for

REST/JSON based service development. CSIP services are deployed using the Apache Tomcat web container [45]. Our six SOAs include: the Revised Universal Soil Loss Equation – Version 2 (RUSLE2) [41], the Wind Erosion Prediction System (WEPS) [71], two versions of the Soil Water Assessment Tool for modeling interactive channel degradation (SWAT-DEG) [96], [97], the Comprehensive Flow Analysis LOAD ESTimator (CFA-LOADEST) [98], [99], and the Comprehensive Flow Analysis Load Duration Curve (CFA-LDC) [100].

RUSLE2 and WEPS are the USDA–Natural Resource Conservation Service standard models for soil erosion used by over 3,000 county level field offices. RUSLE2 (Windows/MS Visual C++) contains empirical and process-based science that predicts rill and interrill soil erosion by rainfall and runoff. The Wind Erosion Prediction System (WEPS) is a daily simulation model which outputs average soil loss and deposition values to predict soil erosion due to wind. WEPS (Linux/Java/Fortran) consists of seven sub models for weather, crop growth, decomposition, hydrology, soil, erosion, and tillage. $\mathcal{M}$, $\mathcal{D}$, $\mathcal{F}$, and $\mathcal{L}$ components used by RUSLE2 and WEPS are described in table 8.4. All other tested SOA workloads used only $\mathcal{M}$ and $\mathcal{L}$ components. Resource profiling occurred only on $\mathcal{M}$ VMs. One VM was statically allocated for $\mathcal{D}$, $\mathcal{F}$, and $\mathcal{L}$ components.

Table 8.4. Rusle2/WEPS SOA Components

Component		RUSLE2	WEPS
$\mathcal{M}$	Model	Apache Tomcat 6, Wine, OMS3 [43], [52]	Apache Tomcat 6
$\mathcal{D}$	Database	Postgresql-8.4, PostGIS 1.4: soils (4.3m shapes), mgmt (98k shapes), climate (31k shapes), 4.6 GB total (Tennessee)	Postgresql-8.4, PostGIS 1.4, soils (4.3m shapes), climate/wind (850 shapes), 17 GB total (western US data)
$\mathcal{F}$	File server	nginx file server, 57k XML files (305MB), parameterizes RUSLE2 model runs.	nginx file server, 291k files (1.4 GB), parameterizes WEPS model runs.
$\mathcal{L}$	Logger	Redis - distributed cache server	Redis - distributed cache server

Two variants of SWAT-DEG (Fortran/Linux) were used. A deterministic version simulates stream down-cutting and widening while also outputting a flow duration curve and cumulative stream power. A stochastic version supports Monte Carlo model calibration for model uncertainty encountered within nature for river restoration/rehabilitation projects. SWAT-DEG stochastic invokes SWAT-DEG deterministic repeatedly to perform calibration runs and performs Map-Reduce operations. Individual runs are distributed to $\mathcal{M}$ worker VMs to perform local computations which are later reduced. The reduce phase was largely sequential, resulting in a heavy parallel computation phase followed by a largely sequential reduction phase.

CFA-LOADEST (Windows/FORTRAN) estimates the amount of constituent loads in streams and rivers given a time series of stream flows and constituent concentrations. Estimation of constituent loads occurs in two steps, the calibration procedure and the estimation procedure based on statistical methods. CFA-LDC (java) graphs Weibull plotting position ranks of river flows on a scale of percent exceedance. Graphing flow values in this way allows for a quick visualization of the variability of flow for different flow regimes.

8.4.2. The Virtual Machine (VM) Scaler

To facilitate performance profiling of virtual infrastructures for hosting SOA workloads we developed the Virtual Machine (VM) Scaler, a REST/JSON-based web services application [84]. VM-Scaler harnesses the Amazon EC2 API to support application profiling and cloud infrastructure management and currently supports Amazon's public cloud (EC2) and private clouds running Eucalyptus 3.x. VM-Scaler provides cloud control while abstracting the underlying IaaS cloud and can be extended to support any EC2 compatible virtual infrastructure manager. Key features are provided to support workload management and IaaS cloud research. Features include: hotspot detection, dynamic scaling, VM management and placement, job scheduling and proxy services, VM workload profiling, and VM worker pools.

Upon initialization VM-Scaler probes the host cloud and collects metadata including location and state information for all VMs and physical hosts (private IaaS only). An agent installed on each VM sends resource utilization data to VM-Scaler at fixed intervals. Collected resource utilization variables are described previously in table 8.1. Application and load balancer configuration is performed automatically as needed to support workload execution and profiling tasks. VM-Scaler builds on previous research investigating the use of resource utilization variables for guiding cloud application deployment [13], [85].

VM-Scaler supports group management of VMs using a construct known as a *"VM pool"*. Common operations can be applied to pools in parallel to support flushing memory caches, restarting the web container, checkpointing resource utilization and running scripts. Pools support reuse of VMs for multiple workloads as VMs can be returned to the pool after job assignment. For Amazon's public cloud, VMs are billed for a minimum of one hour. This coarse-grained billing cycle makes it advantageous to retain VMs for at least one hour for potential reuse. Pools maintain a minimum number of members and can be instructed to spawn new VMs in anticipation of future demand to help alleviate VM launch latency.

8.4.3. Resource Utilization Checkpointing

VM-Scaler supports collection of resource utilization data across the pool of worker VMs providing SOA workload execution. A simple script installed on each VM sends VM-Scaler resource utilization data at preconfigured intervals. VM-scaler's checkpoint service is called to mark the start time for workload execution. Upon job completion, the total resource utilization for all VMs involved in the workload execution is computed from the checkpoint. VMs make use of the network time protocol daemon to synchronize RU data collection to the nearest second. Resource utilization checkpoints allow for a composite view of the total resource consumption of an SOA workload. This novel feature helps characterize diverse SOA workloads whose execution is distributed across an array of VMs. Composite resource utilization profiles can be harnessed to examine SOA workload characteristics, resource use efficiency, perform analysis, and to build models to support infrastructure and cost prediction.

8.4.4. Hardware Configuration

We develop and evaluate our methodology to achieve equivalent SOA workload performance using different VM types using Amazon's public elastic compute cloud (EC2). Amazon offers a diverse array of VM types, as well as spot instances which enabled this research to be conducted at a low cost in a public cloud environment with real world multi-tenancy challenges. VM types used in the evaluation of our workload cost prediction methodology are described in table 8.5. We selected VM_{base} to be the c3.xlarge. This third generation VM from Amazon provides 4 cores at 3.5 ECUs per core. The c3 VMs are known as "compute" optimized instances, as they are configured with better CPUs but less memory and disk storage capacity. Third generation VMs are all equipped with solid state storage disks, though most are smaller in capacity than previous first and second generation spindle disks. For our investigation we benchmark all SOA workloads using a pool of 5 x c3.xlarge VMs. Using our workload cost prediction methodology we investigate what is required to achieve equivalent SOA performance using m1.xlarge, c1.medium, m2.xlarge, and m3.xlarge VMs.

8.4.5. Test Configurations

We train our VM-type resource utilization models ($M_{VMtype1}$, .. $M_{VMtype-j}$) using workloads from six CSIP applications as described in table 8.6. Distinct training workloads were used to train models, while other unique workloads were used for validation. These models support conversion of resource utilization profiles from one VM-type to another. We train models to convert *cpuUsr*, *cpuKrn*, *cpuIdle*, and *cpuIoWait* resource utilization between VM types. We could also construct models to convert *cpuIntSrvc*, *cspuSftIntSrvc*, *cpuNice*, and *cpuSteal*. However, for our SOA workloads, these resource utilization variables are shown to have very little impact on total wall clock time.

Table 8.5. Equivalent Performance Investigation VM Types, Networking, and Backing CPUs

VM type	**CPU cores**	**ECUs/core**	**RAM**	**Disk**	**Cost/hr.**
c3.xlarge	4	3.5	7.5 GB	2x40 GB SSD	30¢
m1.xlarge	4	2	15 GB	4x420 GB	48¢
c1.medium	2	2.5	1.7 GB	1x350 GB	14¢
m2.xlarge	2	3.25	17.1 GB	1x420 GB	41¢
m3.xlarge	4	3.25	15 GB	2x40 GB SSD	45¢

VM type	**Network I/O**	**Backing CPU**
c3.xlarge	High-1000 Mbps	Intel Xeon E5-2680 v2 @ 2.8 GHz
m1.xlarge	Moderate-500 Mbps	Intel Xeon E5-2650 v0 @ 2.0 GHz
c1.medium	Moderate-500 Mbps	Intel Xeon E5-2650 v0 @ 2.0 GHz
m2.xlarge	Moderate-500Mbps	Intel Xeon E5-2665 v0 @ 2.4 GHz
m3.xlarge	High-1000 Mbps	Intel Xeon E5-2670 v2 @ 2.5 GHz

Table 8.6. SOA Workloads

CSIP SOA	**Test cases per workload**	**# training workloads**	**Avg. duration 5 x c3.xlarge**
WEPS	100	10	96.6 s
RUSLE2	800	10	104.6 s
SwatDeg-Stoc	10 users x 150 sims	10	133.6 s
SwatDeg-Det	500	10	13.5 s
CFA-LOADEST	500	10	99.6 s
CFA-LDC	500	10	103.7 s

In section 3.1, we discussed the challenges *cpuSteal* presents in accounting for wall clock time. We have chosen to avoid these challenges by selecting SOA workloads and VM type configurations which exhibit very low *cpuSteal* time. It should be noted that it **was not** difficult to avoid these *cpuSteal* challenges for this work. Accounting for *cpuSteal* time may be possible by investigating the use of offset values to account for missing clock ticks in the presence of relatively constant *cpuSteal*.

Figure 8.1 illustrates the resource utilization of our CSIP SOA workloads on 5 x c3.xlarge 4-core VMs. 25% *cpuUsr* is equivalent to exercising one core at 100% for the duration of the SOA workload. The figure demonstrates that these six workloads are primarily CPU bound but vary widely as to how effectively they exercise available cores. WEPS and SWATDEG-deterministic were most effective at using available cores. RUSLE2 and SWATDEG-stochastic appear to continuously exercise from 1 to 2 CPU cores. CFA-LOADEST and CFA-LDC appear to utilize less than one CPU core. The balance between *cpuUsr* time and *cpuIdle* time illustrates how well a given workload performs computations in parallel. Adding increasingly more resources to largely sequential workloads provides little performance benefit as described by Amdahl's Law. VM-Scaler's resource utilization checkpointing supports profiling the parallel efficiency of SOA workloads. Figure 8.1 illustrates the range of efficiencies we observed for our 6 modeling SOAs.

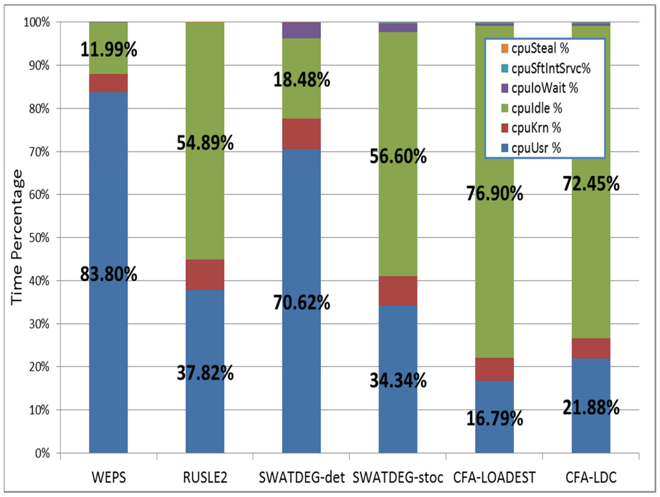


Figure 8.1. CSIP SOA Workload Resource Utilization

This figure shows the diversity of resource utilization of the SOA workloads used to evaluate our workload cost prediction methodology.

Between individual training and validation SOA workloads, all application services were stopped, caches cleared, and services restarted. The Linux virtual memory drop_caches function was used to clear caches, dentries, and inodes. Clearing caches served to negate training effects resulting from reusing test cases.

8.5. EXPERIMENTAL RESULTS

8.5.1. Resource Utilization Profile Prediction

Training resource utilization models which convert SOA workload profiles between VM types requires execution of SOA training workloads. We executed these workloads using isolated dedicated $\mathcal{M}$ VMs. Resource utilization checkpointing enabled profiling data to be collected with minimum overhead. The effectiveness of our methodology is confirmed by the high statistical predictably of key resource utilization variables using linear regression. **A linear regression of *cpuUsr* for m1.xlarge vs. c3.xlarge provides R^2 of .9924 when trained with our 6 CSIP SOAs.** This relationship is shown in figure 8.2. Clusters of data can be seen in groups which represents our distinct SOA workloads.

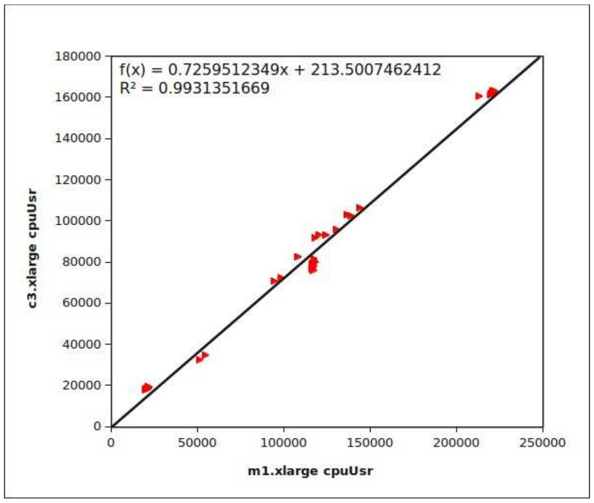


Figure 8.2. CpuUsr c3.xlarge → m1.xlarge linear regression:
This figure shows how linear regression nicely fits cpuUsr data explaining most variance observed in our data demonstrated by the high R2 value.

Using linear regression we tested if the same approach was viable to predict *cpuKrn*, *cpuIdle*, and *cpuIoWait*, the most important variables which account for wall clock time. We observed good, though lower R^2 values for these predictions. To refine our predictions we then applied stepwise multiple linear regression (MLR). We fed stepwise-MLR every available resource utilization variable from table 8.1 to construct MLR models for *cpuUsr*, *cpuKrn*, *cpuIdle*, and *cpuIoWait*. Stepwise MLR begins by modeling the dependent variable using the complete set of independent variables and iterates by dropping the least powerful predictor based on significance for each step. This enables testing various combinations until the best fit model which explains the most variance (R^2) is found. The resulting MLR models had either 7 or 8 independent variables. The independent variables having the highest significance and use for these models besides the variable being predicted include (in decreasing order): *dsw*, *cpuCtxtSw*, *dskWrts*, *cpuSteal*, and *cpuKrn*. R^2 values for our resource utilization variable conversion models are shown in table 8.7.

Table 8.7. Linear Regression Models for VM-type Resource Variable Conversion

RU variable	**adjusted R^2 m1.xlarge LR**	**adjusted R^2 m1.xlarge MLR**	**adjusted R^2 c1.medium MLR**
cpuUsr	.9924	.9993	.9983
cpuKrn	.9464	.989	.9784
cpuIdle	.7103	.9674	.9498
cpuIoWait	.9205	.9584	.9725
	adjusted R^2 m2.xlarge MLR		**adjusted R^2 m3.xlarge MLR**
cpuUsr	.9987		.9992
cpuKrn	.967		.9831
cpuIdle	.9235		.9554
cpuIoWait	.9472		.9831

To test the effectiveness of combining six different SOAs into a single MLR model to convert resource utilization variables across VM-types we inspected regression residual plots. A regression residual plot for the *cpuUsr* c3.xlarge $\rightarrow$ m1.xlarge model is shown in Figure 8.3. Good residual plots show points randomly dispersed around the horizontal X axis. This indicates linear regression is appropriate for the data; otherwise, a non-linear model is more appropriate. Figure 8.3 shows a nice random distribution of predictions. We do note on the tails of our residual plot *cpuUsr* is more often under or over predicted. This effect causes poor *cpuIdle* predictions for SWATDEG-det discussed later in section 5.3. This behavior suggests creating separate resource utilization prediction models for different workload types. SWATDEG-det workloads were only 1/10 as long in duration as the majority of our SOAs explaining reduced quality in model output.

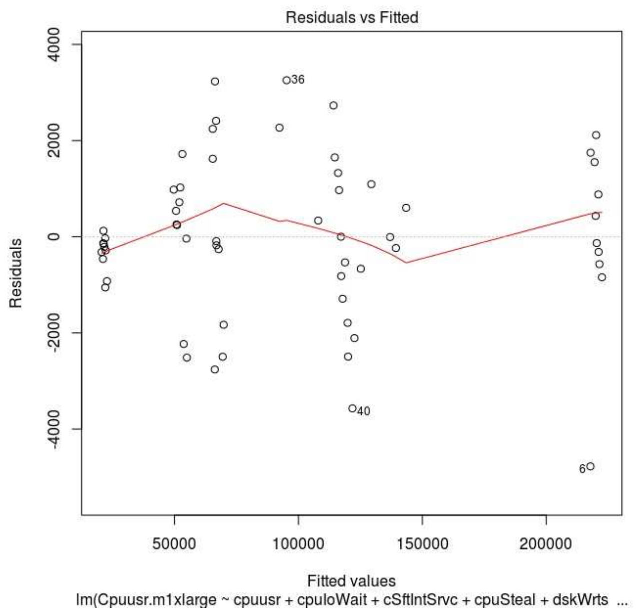


Figure 8.3. CpuUsr c3.xlarge → m1.xlarge residuals plot:
This figure shows the residual plot of our *cpuUsr* linear regression model. Predictions are randomly distributed around the X-axis indicating that linear regression is appropriate for dataset

Using linear and multiple regression we achieve significantly positive results at resource variable conversion across VM-types enabling us to harness this approach for SOA workload VM-type profile prediction **(RQ-2)**.

8.5.2. Resource Utilization Profile Scaling

In section 3.2 we proposed our workload cost prediction methodology. After profiles are converted, we must scale up the profiles to determine the required number of VMs for an alternative type to achieve equivalent performance. We have designed two methods to support resource scaling referred to as RS-1 (application agnostic) and RS-2 (application aware heuristic). We evaluate their effectiveness by scaling 2 validation workloads for each of our 6 SOAs. We predict the required number of VMs to host our workloads with performance equivalent to 5 x c3.xlarge VMs using m1.xlarge, c1.medium, m2.xlarge, and m3.xlarge VM pools.

We discovered that our CFA-LOADEST and CFA-LDC workloads did not scale properly. When additional VMs were added to their VM pools, total workload execution time either remained the same or increased! This explains why these SOAs exhibit very high *cpuIdle* time as shown in figure 8.1. Consequently, our workload cost prediction methodology predicted equivalent performance with 5 or 6 alternate typed VMs. This was an accurate prediction because indeed there was no improvement in workload execution time when the VM pools were scaled. We used CFA-LOADEST and CFA-LDC resource utilization data for training our RU ($M_{VMtype1}$, .. $M_{VMtype-j}$) models for RQ-2, but not for validation.

For our evaluations, we assume equivalent SOA workload performance as the ability to execute the workload within ± 2 seconds total wall clock time of the alternate infrastructure.

For RS-1 we profiled idle CSIP $\mathcal{M}$ VMs in isolation to determine their background CPU usage. We observed the *cpuUsr* overhead per wall clock second and added the relative amount to *cpuUsr* to scale SOA workload profiles in an application agnostic way. For RS-1, we do not scale *cpuKrn*, *cpuIoWait*, and *cpuSftIntSrvc*. We use VM-type converted values but do not scale them further. We make 64 evaluations, 8 each for scaling with m1.xlarge, c1.medium, m2.xlarge, and m3.xlarge profiles, with one evaluation each with PS-1 and PS-2. **RS-1 supported VM prediction with a mean absolute error of .391 VMs per prediction.** RS-1 led to scaling profiles that produced 20 under predictions and only 4 over predictions. Of the prediction errors only 1 had a prediction error of 2 VMs. All other predictions were off by only 1 VM.

For RS-2 we conducted two workload scaling tests for each SOA and averaged the percentage increase for *cpuUsr*, *cpuKrn*, and *cpuIoWait* resulting from scaling the number of $\mathcal{M}$

VMs. To generate scaled profiles we increase these resource utilization variables by this percentage for each VM added. For 2-core VMs we always scale using 2 VMs at a time since our VM_{base} c3.xlarge has 4 cores. For RS-2 we made the same 64 evaluations, 8 for scaling with m1.xlarge, c1.medium, m2.xlarge, and m3.xlarge, twice with PS-1 and PS-2 respectively. **RS-2 supported VM prediction with a mean absolute error of .328 VMs per prediction.** RS-2 led to scaling profiles that produced 17 under predictions and only 4 over predictions. All prediction errors were off by one VM only.

RS-1 and RS-2 represent heuristic-based approaches to scaling the resource utilization profile and provide potential solutions to (**RQ-3**). RS-1 has the advantage of being SOA agnostic and very simple to implement. SOA specific scaling data is **not** required to scale resource profiles. However, predictions were about ~17% less accurate.

8.5.3. Profile Selection for Equivalent Performance

In addition to scaling converted resource utilization profiles, determining equivalent infrastructure performance requires a method to select the resource utilization profile from the set of scaled profiles. It is not sufficient to simply select the first profile that has non-negative *cpuIdle* time. A healthy surplus of *cpuIdle* time is necessary for most SOAs to achieve equivalent performance. In section 3.2, we proposed two heuristic based approaches to resource utilization profile selection for equivalent performance: PS-1 (application agnostic) and PS-2 (application aware).

For SWATDEG-det m1.xlarge evaluations, we observed that our multiple linear regression model over predicted *cpuIdle* time. We believe this prediction error occurred because the average SWATDEG-det workload execution time was only 1/10th of the other SOAs. This caused our regression model to over predict *cpuIdle* time which prevented profile selection. In this case, to correct the SWATDEG-det *cpuIdle* prediction error we used raw c3.xlarge *cpuIdle* values for profile selection.

PS-1 uses a simple linear equation to scale *cpuIdle* time as the VM pool is scaled. The initial *cpuIdle* value is taken from the VM-type resource utilization conversion. Equation 2 (section 3.2, PS-1) is then used to grow *cpuIdle* for each additional VM added. The output value represents the *cpuIdle* threshold for profile selection. Linux CPU time accounting principles are used to calculate the available *cpuIdle* time. The first profile which exceeds the threshold is selected to determine the minimum number of VMs required for equivalent performance. **PS-1 supported VM prediction with a mean absolute error of .375 VMs per prediction.** PS-1 led to profile selections that produced 19 under predictions and only 4 over predictions. Of the prediction errors only 1 had a prediction error off by 2 VMs. All other predictions were off by only 1 VM.

For PS-2 we conducted two SOA specific scaling tests and averaged the observed percentage increase in *cpuIdle* time. The initial *cpuIdle* value is taken from the VM-type resource utilization conversion. The required *cpuIdle* time is increased by the SOA specific percentage to establish a threshold for profile selection. The first profile that exceeds the threshold is selected to determine the minimum number of VMs required for equivalent performance. **PS-2 supported VM prediction with a mean absolute error of .344 VMs per prediction.** PS-2 led to profile selections that produced 18 under predictions and only 4 over predictions. All prediction errors were off by one VM only.

PS-1 and PS-2 represent heuristic-based approaches to selecting the correct resource utilization profile which will provide equivalent SOA workload performance and provide potential solutions to (**RQ-1**). PS-1 has the advantage of being SOA agnostic and very simple to implement. SOA specific scaling data is **not** required. Predictions supported by our application agnostic approach PS-1 were ~9% less accurate, which is to be expected.

Table 8.8. Equivalent Infrastructure Predictions – Mean Absolute Error (# VMs)

SOA / VM-type	**PS-1 (RS-1)**	**PS-2 (RS-1)**	**PS-1 (RS-2)**	**PS-2 (RS-2)**
WEPS	.5	.5	.5	.5
RUSLE2	.25	0	.125	.125
SWATDEG-STOC	.75	.5	.5	.625
SWATDEG-DET	.25	.375	.125	.125
m1.xlarge	.375	.25	.25	.25
c1.medium	.875	.625	.5	.625
m2.xlarge	.25	.25	.25	.25
m3.xlarge	.25	.25	.25	.25
Average	.4375	.34375	**.3125**	.34375

In Section 3.2 we proposed three alternatives for resource scaling and profile selection each with increasing implementation costs though offering improved accuracy. Mean absolute error (# VMs) for our SOA infrastructure predictions using our resource scaling and profile selection heuristics is summarized in table 8.8. **The combination of PS-1 and RS-2 together provided the most accurate predictions with a mean absolute error of only .3125 VMs per prediction.** For resource scaling and profile selection, the application agnostic approaches had slightly more error but were easy and fast to implement with no scaling tests required. Our evaluation demonstrates improvement with an application specific approach. We posit that training regression models proposed for RS-3 and PS-3 will provide even greater accuracy in exchange for the effort.

Table 8.9. Hourly SOA Hosting Cost Predictions with Alternative VM-types

SOA	m1.xlarge	c1.medium	m2.xlarge
WEPS	$3.84	$2.24	$2.46
RUSLE2	$3.84	$2.24	$2.46
SWATDEG-Stoc	n/a	$1.96	$2.46
SWATDEG-Det	$3.84	$2.52	$2.87
Total	$11.52	$8.96	$10.25
	m3.xlarge		**Total error**
WEPS	$2.70		-$.76
RUSLE2	$2.70		**$0**
SWATDEG-Stoc	$2.70		**-$.86**
SWATDEG-Det	$2.70		**+$.13**
Total	$10.80		**-$1.49 (3.59%)**

8.5.4. Cost Prediction

We next evaluated our workload cost prediction methodology's ability to predict actual workload costs for infrastructure which provides equivalent performance. For this evaluation we consider one hour of SOA workload execution using m1.xlarge VMs for WEPS, Rusle2, and SwatDeg-det, and one hour of SOA workload execution using c1.medium, m2.xlarge, and m3.xlarge VMs for WEPS, Rusle2, SWATDEG-stoc, and SWATDEG-det. We allocated the number of VMs required to achieve equivalent workload performance relative to VM_{base}=c3.xlarge. We omit m1.xlarge SWATDEG-stoc testing because our models predicted c3.xlarge equivalent performance could not be achieved using the m1.xlarge VM type, and testing verified this outcome! We apply the fixed instance prices from table 8.5. The results of this evaluation appear in table 8.9. These cost predictions are based on our application specific PS-2/RS-2 approach. The total error column represents the cost prediction error. Error results from under predicting the number of VMs required for equivalent SOA performance. A perfect cost prediction methodology accurately predicts hosting costs for alternate VM types with no error. **Our workload cost prediction methodology produces a cost estimate only 3.59% below the actual hosting cost for equivalent performance using alternate VM types.** Our results demonstrate how different VM-types offer a range of economic outcomes for SOA workload hosting (25% cost variance).

8.6. CONCLUSIONS

This paper describes our workload cost prediction methodology to support hosting SOAs using any virtual machine type to provide equivalent performance. Our cost prediction methodology provides cost effective architecture alternatives for diverse SOA workloads. Armed with infrastructure decision support, system analysts are better able to balance cost and performance tradeoffs of SOA deployments.

(RQ-1) Harnessing Linux time accounting principles and VM-type resource predictions, our approach predicts the required infrastructure to achieve equal or better workload performance using any VM type. **(RQ-2)** Multiple linear regression is shown to support prediction of key resource utilization variables required for Linux time accounting. Strong predictability is found with coefficients of determination of R^2=.9993, .989, .9674, .9585 for *cpuUsr*, *cpuKrn*, *cpuIdle*, and *cpuIOWait* respectively when converting Amazon EC2 VM resource utilization from the c3.xlarge VM-type to m1.xlarge. **(RQ-3)** A series of resource scaling heuristics were tested to scale up resource utilization predictions from n to $n+x$ VMs. Profile selection heuristics were then tested to support determining infrastructure required to provide equivalent or better performance. The efficacy of these heuristics to predict the required number of VMs to host SOA workloads while providing equivalent performance was shown to be as low as .3125 VMs (PS-1 / RS-2).

We implement a novel resource utilization checkpointing technique which enables capturing composite resource utilization profiles for SOA workloads executed across VM pools. We applied the Trial-and-Better approach [10] to normalize the CPUs backing VMs in our study to reduce resource profile variance from VM implementation heterogeneity. Given these profiles we demonstrate the use of linear regression to convert SOA resource utilization profiles to alternative VM types. We offer heuristics to scale our predicted profiles and support infrastructure decisions for equivalent SOA workload performance. Our workload cost prediction methodology provides mean absolute error as low as .3125 VMs, and hosting cost estimates to within 3.59% of actual.

In closing we predict all of the following will change: (1) VM-types offered by public cloud providers, (2) price for these VMs, and (3) the performance levels they provide. Our workload cost prediction methodology helps demystify the plethora of VM types offered by cloud vendors and supports future changes. Our approach is generalizable to any VM-type and helps to clarify ambiguous performance rankings (e.g. ECUs, CCUs) with a quantitative statistically backed approach which combines both application profiling and VM benchmarking.

8.7. FUTURE WORK

As future work we propose Resource Scaling Approach 3 (RS-3), and Profile Selection Approach 3 (PS-3). Both approaches should provide additional accuracy by training SOA workload specific models beyond the heuristics present in section 3.2

RS-3: Scaling Models

Resource scaling approach (RS-3) involves training a set of models, one each for *cpuUsr*, *cpuKrn*, *cpuIoWait*, and *cpuSftIntSrvc* using resource utilization data collected when scaling infrastructure for SOA workloads. Scaling models incorporate resource utilization parameters and the number of CPU cores as dependent variables. One set of models is required for each VM type. The models can then be trained using multiple linear regressions or an alternate machine learning technique. This approach should provide high accuracy with more testing effort.

PS-3: *CPUIDLE* SCALING MODELS

Our third profile selection approach (PS-3) involves training a set of models with scaling runs to predict how *cpuIdle* time increases as infrastructure is scaled up. These *cpuIdle* models incorporate all resource utilization variables from table 8.1 and the number of CPU cores for scaled deployments as dependent variables. One *cpuIdle* model is required for each VM type. These models can then be trained using multiple linear regressions or an alternate machine learning technique. This approach should provide high accuracy with more testing effort.

An interesting extension for this work involves developing an approach to predict resource requirements (CPU time, disk I/O, etc.) for SOA workloads based on scientific model service parameterization. It is possible to analyze the model parameterizations to characterize the expected duration and computing requirements for service quests before they execute. We have attempted initial trials using the WEPS model and have achieved $R^{2=}$~.5 using multiple linear regression using only a subset of the model parameters. This white box approach to predict workload resource requirements would enable initial workload profiling (Step 1) to be eliminated. Service requests could be analyzed, not run, to predict workload execution costs and deployment infrastructure. Developing this approach requires harnessing domain specific characteristics of service requests and there will likely be limitations to the ability when training models to accurately predict model service behavior.

For resource scaling and profile selection we have only tested 2 of the 3 proposed approaches. An extension to our work involves training regression models using representative

SOA workloads. Resource scaling approach 3 (RS-3) would involve training models to predict how *cpuUsr*, *cpuKrn*, and *cpuIoWait* scale for particular SOAs. Profile selection approach 3 (PS-3) would involve training models to predict when ample *cpuIdle* time is present to achieve equivalent workload performance. Logistic regression may be helpful to train RS-3 models.

Our workload cost prediction methodology has been designed to use homogeneous VM pools where all VMs in the pool are the same type (e.g. m1.xlarge, c3.xlarge, etc.) An extension involves investigating what is necessary to achieve equivalent workload performance using VM pools with multiple VM types. One application for mixed pools involves provisioning partial VMs. If a workload runs optimally using 3.25 x 8-core c1.xlarge VMs, we would *not* provision 4 x c1.xlarge VMs. Instead to save costs, 3 x c1.xlarge VMs, and an additional VM of an alternate type would be provisioned. Another application for mixed pools occurs when insufficient resources are available to provision an entire VM pool using a single type. This scenario may be uncommon in public cloud settings, but quite possible in private clouds. When a particular resource is exhausted we would investigate achieving equivalent workload performance by provisioning the remainder of the pool using one or more alternate VM types.

For step 5 of our workload cost prediction methodology, we currently apply fixed values of spot market prices to estimate infrastructure costs when spot instances are used. This approach assumes that spot pricing remains stable for the duration of the SOA workload. Depending on the workload duration and spot market price volatility fixing spot prices may be sufficient for good cost estimation. However, future work could investigate application of spot market price prediction models such as those discussed in [91], to further improve spot instance infrastructure cost estimates.

Future research should investigate tradeoffs of resource utilization model training when estimating costs for multiple SOA workloads. Grouping vs. separating workload profiling data for training models should be investigated. Which scenarios are best for combining diverse and dissimilar SOA workload profiles? And what limitations are encountered?

As future work, VM-Scaler can be extended to automate our workload cost prediction methodology. Extensions would include: implementation of application profiling, integration of R based resource utilization regression models [54], and implementation of resource scaling and profile selection heuristics.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1. CONCLUSIONS

This dissertation has investigated key resource management challenges encountered when migrating hosting of service oriented applications (SOAs) to Infrastructure-as-a-Service (IaaS) clouds. While much has been made about the advantages of IaaS cloud computing, this dissertation identifies significant resource management challenges that must be addressed for promises to come to fruition. Key IaaS cloud resource management challenges can be broken down into three overarching questions. We must address: WHEN to provision infrastructure, WHAT infrastructure to provision, and WHERE infrastructure should be provisioned to ensure service availability and responsiveness. This dissertation describes methodologies for improving both application availability and responsiveness while minimizing hosting costs in terms of required infrastructure (# of VMs) and monetary commitments ($).

Each chapter of this dissertation addresses one or more unique resource management challenges. This dissertation has sought to concretely demystify cloud computing to address performance variance from heterogeneous resource implementation, VM placement uncertainty, provisioning variation, and the ubiquity of vague resource descriptions. When taken together, this dissertation provides a comprehensive study of these challenges that constitutes a seminal work supporting practitioners in deploying SOAs to IaaS clouds.

The following assertions are made regarding the overarching dissertation research questions (DRQs) presented earlier in Section (1.2). Detailed discussion can be found in each question’s respective chapter.

DRQ-1: [Chapter 3] Application migration to IaaS clouds must address component composition to reduce resource contention for shared resources of physical host machines. Scaling up infrastructure to address increasing service demand requires tuning application infrastructure parameters (e.g. number of database connections, number of worker threads) in addition to the quantity and size of VMs.

DRQ-2: [Chapter 4] The best resource utilization variables- that predict performance of applications deployed to IaaS clouds- vary based on the specific resource profiles of applications. The best predictors for CPU bound applications include: CPU user time, CPU kernel time, and the number of completed disk writes. Additionally, the number of context switches, disk sector reads, and completed disk reads support performance predictions for I/O bound applications. Step-wise multiple linear regression and artificial neural networks are preferred techniques for training performance models given application resource utilization data.

DRQ-3: [Chapter 5] Component composition across VMs impacts an application’s resource utilization profile by creating or alleviating resource contention. (See figure 5.1) Optimal performance requires mediation of resource contention. Intuition alone is insufficient to predict the best deployments when applications consist of more than 3 components. The degree of performance variance resulting from component composition depends upon each application’s unique profile characteristics. Performance variance of at least 15-25% is expected.

DRQ-4: [Chapter 7] When dynamically scaling infrastructure to address demand spikes, careful VM placement to the least busy physical hosts in a private cloud improves application performance by at least 10-15% compared to round robin VM placement. The degree of the performance improvement is impacted by the magnitude of the shared load across physical host machines as well as the size of the cloud (# of physical host machines). Temporary increases in resource contention occur when multiple VMs are launched simultaneously on the same physical host from spikes in disk I/O operations required to provision resources. Careful VM placement supports dynamic scaling for SOA hosting using fewer VMs and CPU cores, freeing capacity for other uses.

DRQ-5: [Chapter 7] Trends in CPU Steal clock ticks support detection of resource contention and noisy neighbor VMs in public cloud settings. Performance loss from noisy neighbor interference is reproducible across virtual infrastructure in public cloud settings for several hours. We describe a 3-step process to identify noisy neighbors in Section 7.5.3. SOA workloads executed on VMs with resource contention from noisy neighbors results in performance degradation of at least 10-25%.

DRQ-6: [Chapter 8] Section 8.3 describes a workload cost prediction methodology to support identification of infrastructure alternatives capable of providing equivalent performance based on Linux time accounting principles and resource utilization modeling techniques. Infrastructure predictions are achieved with an average mean absolute error of only .3125 VMs for hosting 4 different SOA workloads using 4 different VM types on Amazon EC2. (Section 8.5) This low mean absolute error supports cost predictions ($) to within 3.59% of actual.

9.2. CONTRIBUTIONS

This dissertation makes three primary contributions to advance Infrastructure-as-a-Service (IaaS) cloud resource management to support application migration to clouds. A more detailed discussion of the research contributions of this dissertation appears in chapter 1, section

3.

In Chapter 4, we introduce resource utilization modeling to support performance prediction of SOA workloads deployed across IaaS cloud infrastructure. Harnessing SOA resource utilization data to make predictions has proven very valuable for: (1) finding the best component compositions for initial SOA deployment (Chapter 5), and (2) determining infrastructure alternatives which provide equivalent performance to support service availability and responsiveness while minimizing hosting costs (Chapter 8).

In Chapter 7, we provide the Least-Busy VM scheduler that improves resource elasticity, service availability, and responsiveness in private cloud settings. Least-Busy supports VM placement to the least busy physical machines in a cluster. Least-Busy helps improve SOA responsiveness and availability when dynamically scaling the number of VM instances. Additionally we demonstrated the utility of our Least-Busy approach to perform job scheduling in both public and private cloud settings. (Section 7.7.5)

In Chapter 8, we present our workload cost prediction methodology. Our methodology harnesses Linux time accounting principles and resource utilization models to provide infrastructure alternatives which provide equivalent application performance for hosting SOAs using various virtual machine types on public and private IaaS clouds. Given infrastructure alternatives, cost comparisons can easily be made to determine the most cost effective hosting infrastructure.

9.3. FUTURE WORK

9.3.1. White Box Resource Utilization Prediction

An extension to work described in Chapters 4 and 5 involves harnessing scientific model parameterization variables to predict and model service execution time and individual resource utilization profile variables. A preliminary investigation of this concept was made using the WEPS model. Multiple linear regressions were performed using key model parameters from WEPS modeling requests. These regressions predicted model execution time while explaining a fair amount of variance (R^2=~.5). This technique if developed further should also support prediction of individual resource utilization variables (e.g. *cpuUsr*, *cpuKrn*, *cpuIdle*, *cpuIoWait*, etc.). This enables the entire application resource utilization profile to be predicted without running actually running any SOA workloads. This would support application of our workload cost prediction methodology (Chapter 8) to best determine how to deploy workloads without performing initial profiling runs (Step 1).

Another use of white box prediction includes development of a job placement scheduler that harnesses service execution time and resource requirement predictions. Two optimizations are possible of job placement decisions. Job placement can optimize for minimum hosting footprint of SOA workloads (# of VMs), or maximum throughput given a fixed number of resources. Such a job scheduler could optimize job placement of long running service requests by not scheduling requests to worker VMs which are already busy, *and are predicted to remain so*. Job requests which require specific types of resources (e.g. network I/O, CPU, or disk I/O) could be scheduled to worker VMs having availability of these specific resources.

In Chapter 7, we demonstrated how job placement to Least Busy worker VMs can provide job scheduling which improves upon haproxy least connection load balancing [51]. We expect our proposed white box prediction extensions will improve job placement and management of available resources even further.

9.3.2. Public Cloud Resource Contention Study

In Chapter 7, we presented our *cpuSteal* based Noisy Neighbor Detection Method (NN-Detect). As future work we propose an extended empirical evaluation of this methodology. An expanded investigation should be performed to further study resource contention in public cloud settings. This investigation should span multiple months, cloud availability regions (e.g. us-east-1a, us-east-1c in Amazon EC2), and VM types. How resource contention varies throughout the day, days of the week, months of the year, by region, and VM type should be studied. This investigation would seek to answer the following questions:

- Are there any patterns to public cloud resource contention which might be harnessed to improve infrastructure management to support performance improvement and/or reduced hosting costs for SOAs?
- Do patterns extend across cloud regions? VM types? public cloud providers?

This proposed effort has the potential to identify and contribute towards the development of new mechanisms to locate resource contention and noisy neighbors to further enhance NN-Detect.

9.3.3. Workload Cost Prediction Methodology

In Chapter 8, we describe our workload cost prediction methodology. Many extensions to this work are possible including:

- VM-Scaler support

VM-Scaler could be extended to automate application of the workload cost prediction methodology. Support for profiling representative workloads to train a set of resource utilization prediction models for *cpuUsr*, *cpuIdle*, *cpuKrn*, and *cpuIoWait* for Step 0 would be included.

Building the initial set of predictive models using representative SOA workloads is likely the most complex step. Additionally the profile conversion (Step 1) and resource scaling heuristics (Step 2) and profile selection heuristics (Step 3) are likely easily automated once predictive models are built in Step 0. Automation of cost predictions involves the simple application of current market prices to infrastructure predictions (Step 4).

- Predictive Models to Support Resource Scaling and Profile Selection

Future work could implement predictive models to better support resource scaling (Step 2) and profile selection (Step 3). These predictive models are SOA specific, but it is expected that they will provide greater accuracy.

- Workload cost prediction using mixed resources

Our workload cost prediction methodology presented in Chapter 8 predicts required resources using VM pools of a single VM type. Future work should extend our approach to support using VM pools with multiple VM types. A compelling use case for mixed VM pools occurs when the optimal infrastructure for hosting an SOA splits a large multiple core VM. For example if an optimal SOA hosting infrastructure requires 3.25 x 8-core c1.xlarge VMs, provisioning 4 x c1.xlarge VMs (32 cores) is not necessary and will likely involve additional expense. The recommended infrastructure using a mixed pool might consist of 3 x c1.xlarge VMs (24 cores) and 1 x c1.medium VM (2 cores).

- Integration and development of spot market price models to enhance cost predictions

Presently cost predictions for our workload cost prediction methodology apply fixed market prices to determine infrastructure prices. An extension involves using spot market pricing models which harness historical pricing data to support prediction of future market prices. These models should provide better long term estimations of infrastructure cost.

BIBLIOGRAPHY

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” *ACM SIGOPS Operating Systems Review*, vol. 37. p. 164, 2003.

[2] A. Kivity, U. Lublin, A. Liguori, Y. Kamay, and D. Laor, “kvm: the Linux virtual machine monitor,” *Proc. Linux Symp.*, vol. 1, pp. 225–230, 2007.

[3] F. Camargos, G. Girard, and B. Ligneris, “Virtualization of Linux Servers,” in *2008 Linux Symposium*, 2008, pp. 63–76.

[4] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, “Quantifying the Performance Isolation Properties of Virtualization Systems,” in *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS 2007)*, 2007, p. 6.

[5] “Cloud Harmony - Provider Directory.” [Online]. Available: https://cloudharmony.com/cloudsquare/cloud-compute.

[6] P. Saripalli, G. V. R. Kiran, R. R. Shankar, H. Narware, and N. Bindal, “Load prediction and hot spot detection models for autonomic cloud computing,” in *Proceedings - 2011 4th IEEE International Conference on Utility and Cloud Computing, UCC 2011*, 2011, pp. 397–402.

[7] A. Kejariwal, “Techniques for optimizing cloud footprint,” in *1st IEEE International Conference on Cloud Engineering (IC2E 2013)*, 2013, pp. 258–268.

[8] “Amazon EC2 Instance Comparison.” [Online]. Available: http://www.ec2instances.info. [Accessed: 05-Feb-2014].

[9] E. Fenton, N. Pfleeger, and S. Lawrence, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. PWS Publishing Company, 1997, p. 638.

[10] Z. Ou, H. Zhuang, A. Lukyanenko, J. K. Nurminen, P. Hui, V. Mazalov, and A. Yla-Jaaski, “Is the Same Instance Type Created Equal? Exploiting Heterogeneity of Public Clouds,” *IEEE Trans. Cloud Comput.*, vol. 1, pp. 201–214, 2013.

[11] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, “More for your money: Exploiting Performance Heterogeneity in Public Clouds,” in *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC ’12*, 2012, pp. 1–14.

[12] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. W. Rojas, “Migration of multi-tier applications to infrastructure-as-a-service clouds: An investigation using kernel-based virtual machines,” *Proc. - 2011 12th IEEE/ACM Int. Conf. Grid Comput. Grid 2011*, pp. 137–144, 2011.

[13] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. W. Rojas, “Performance implications of multi-tier application deployments on Infrastructure-as-a-Service clouds: Towards performance modeling,” *Future Generation Computer Systems*, 2013.

[14] A. Beloglazov and R. Buyya, “Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in Cloud data centers,” in *Concurrency Computation Practice and Experience*, 2012, vol. 24, pp. 1397–1420.

[15] W. Chen, E. Deng, R. Du, R. Stanley, and C. Yan, “Crossing and Nesting of Matching and Paritions,” *Trans. Am. Math. Soc.*, vol. 359, no. 4, pp. 1555–1575, 2007.

[16] M. S. Rehman and M. F. Sakr, “Initial findings for provisioning variation in cloud computing,” in *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010*, 2010, pp. 473–479.

[17] T. Ristenpart and E. Tromer, “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds,” *... Conf. Comput. ...*, pp. 199–212, 2009.

[18] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, “Runtime measurements in the cloud: observing, analyzing, and reducing variance,” *Proc. VLDB Endow.*, vol. 3, pp. 460–471, 2010.

[19] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, “Optimal power allocation in server farms,” *Proc. Elev. Int. Jt. Conf. Meas. Model. Comput. Syst. - SIGMETRICS ’09*, p. 157, 2009.

[20] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, “Improving MapReduce Performance in Heterogeneous Environments.,” *OSDI*, pp. 29–42, 2008.

[21] C. Z. Xu, J. Rao, and X. Bu, “URL: A unified reinforcement learning approach for autonomic cloud management,” *J. Parallel Distrib. Comput.*, vol. 72, pp. 95–105, 2012.

[22] B. Addis, D. Ardagna, B. Panicucci, and L. Zhang, “Autonomic management of cloud service centers with availability guarantees,” in *Proceedings - 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD 2010*, 2010, pp. 220–227.

[23] W. Li, J. Tordsson, and E. Elmroth, “Modeling for dynamic cloud scheduling via migration of virtual machines,” in *Proceedings - 2011 3rd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2011*, 2011, pp. 163–171.

[24] R. Maurer, M., Brandic, I., Sakellariou, “Enacting SLAs in Clouds Using Rules,” *Springer Lect. Notes Comput. Sci.*, vol. vol. 6852,, pp. pp. 455–466, 2011.

[25] M. Maurer, I. Brandic, and R. Sakellariou, “Simulating autonomic SLA enactment in clouds using case based reasoning,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, vol. 6481 LNCS, pp. 25–36.

[26] H. N. Van, F. D. Tran, and J. M. Menaud, “Autonomic virtual resource management for service hosting platforms,” in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing, CLOUD 2009*, 2009, pp. 1–8.

[27] O. Niehorster, A. Krieger, J. Simon, and A. Brinkmann, “Autonomic Resource Management with Support Vector Machines,” in *Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on*, 2011, pp. 157–164.

[28] P. Lama and X. Zhou, “Efficient server provisioning with control for end-to-end response time guarantee on multitier clusters,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, pp. 78–86, 2012.

[29] P. Lama and X. Zhou, “Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee,” in *Proceedings - 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2010*, 2010, pp. 151–160.

[30] N. Bonvin, T. G. Papaioannou, and K. Aberer, “Autonomic SLA-driven provisioning for cloud applications,” in *Proceedings - 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011*, 2011, pp. 434–443.

[31] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, “Sandpiper: Black-box and gray-box resource management for virtual machines,” *Comput. Networks*, vol. 53, pp. 2923–2938, 2009.

[32] G. Kousiouris, T. Cucinotta, and T. Varvarigou, “The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks,” *J. Syst. Softw.*, vol. 84, pp. 1270–1291, 2011.

[33] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. W. Rojas, “Service isolation vs. consolidation: Implications for IaaS cloud application deployment,” in *Proceedings of the IEEE International Conference on Cloud Engineering, IC2E 2013*, 2013, pp. 21–30.

[34] P. Sempolinski and D. Thain, “A comparison and critique of Eucalyptus, OpenNebula and Nimbus,” in *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010*, 2010, pp. 417–426.

[35] M. A. Vouk, “Cloud computing - Issues, research and implementations,” in *Proceedings of the International Conference on Information Technology Interfaces, ITI*, 2008, pp. 31–40.

[36] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, “Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment,” *2009 IEEE Int. Conf. E-bus. Eng.*, 2009.

[37] W. Iqbal, M. N. Dailey, and D. Carrera, “SLA-Driven Dynamic Resource Management for Multi-tier Web Applications in a Cloud,” *Clust. Cloud Grid Comput. (CCGrid), 2010 10th IEEE/ACM Int. Conf.*, 2010.

[38] H. Liu and S. Wee, “Web server farm in the cloud: Performance evaluation and dynamic architecture,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009, vol. 5931 LNCS, pp. 369–380.

[39] S. Wee and H. Liu, “Client-side Load Balancer Using Cloud,” in *Symposium on Applied Computing (SAC 2010)*, 2010, pp. 399–405.

[40] D. Armstrong and K. Djemame, “Performance issues in clouds: An evaluation of virtual image propagation and I/O paravirtualization,” *Comput. J.*, vol. 54, pp. 836–849, 2011.

[41] USDA-ARS, “Revised Universal Soil Loss Equation Version 2 (RUSLE2).” .

[42] L. R. Ahuja, J. C. Ascough II, and O. David, “Advances in Geosciences Developing natural resource models using the object modeling system : feasibility and challenges,” *Adv. Geosci.*, pp. 29–36, 2005.

[43] O. David, J. C. Ascough II, G. H. Leavesley, and L. R. Ahuja, “Rethinking modeling framework design: Object Modeling System 3.0,” in *Modelling for Environment’s Sake: Proceedings of the 5th Biennial Conference of the International Environmental Modelling and Software Society, iEMSs 2010*, 2010, vol. 2, pp. 1190–1198.

[44] “WineHQ - Run Windows Application on Linux, BSD, Solaris, and Mac OS X.” .

[45] “Apache Tomcat.” 2011.

[46] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, “The eucalyptus open-source cloud-computing system,” in *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID 2009*, 2009, pp. 124–131.

[47] “PostgreSQL: The world’s most advanced open source database.” 2011.

[48] “PostGIS.” 2011.

[49] “nginx news.” 2011.

[50] “Welcome to CodeBeamer.” 2011.

[51] “HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer.” .

[52] O. David, J. C. Ascough II, W. Lloyd, T. R. Green, K. W. Rojas, G. H. Leavesley, and L. R. Ahuja, “A software engineering perspective on environmental modeling framework design: The Object Modeling System,” *Environ. Model. Softw.*, vol. 39, pp. 201–213, 2013.

[53] R. H. Myers, *Classical and modern regression with applications*, vol. 2nd. 1990, p. 488.

[54] J. Adler, *R In a Nutshell: A Desktop Quick Reference*, First Edit. O’Reilly, 2010.

[55] P. Teetor, *R Cookbook: Proven Recipes for Data Analysis, Statistics, and Graphics*, First Edit. O’Reilly, 2011.

[56] J. Varia, “Architecting for the Cloud : Best Practices,” *Compute*, vol. 1, pp. 1–23, 2010.

[57] M. Schwarzkopf, D. G. Murray, and S. Hand, “The seven deadly sins of cloud computing research,” in *HotCloud’12 Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, 2012, pp. 1–1.

[58] G. Somani and S. Chaudhary, “Application performance isolation in virtualization,” in *CLOUD 2009 - 2009 IEEE International Conference on Cloud Computing*, 2009, pp. 41–48.

[59] H. Raj, R. Nathuji, A. Singh, and P. England, “Resource management for isolation enhanced cloud services,” in *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009, pp. 77–84.

[60] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, “Cuanta : Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines,” *Proc. 2nd ACM Symp. Cloud Comput.*, pp. 22:1–22:14, 2011.

[61] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal, “Pesto: online storage performance management in virtualized datacenters,” in *ACM Symposium on Cloud Computing*, 2011, p. 19.

[62] H. Kang, Y. Chen, J. Wong, and J. Wu, “Enhancement of Xen’s scheduler for MapReduce workloads,” in *20th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC ’11)*, 2011, pp. 251–262.

[63] T. Voith, K. Oberle, and M. Stein, “Quality of service provisioning for distributed data center inter-connectivity enabled by network virtualization,” *Futur. Gener. Comput. Syst.*, vol. 28, pp. 554–562, 2012.

[64] D. Jayasinghe, S. Malkowski, Q. Wang, J. Li, P. Xiong, and C. Pu, “Variations in performance and scalability when migrating n-tier applications to different clouds,” in *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, 2011, pp. 73–80.

[65] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, “Modeling and synthesizing task placement constraints in Google compute clusters,” *Proc. 2nd ACM Symp. Cloud Comput. SOCC 11*, pp. 1–14, 2011.

[66] “Amazon Elastic compute Cloud API Reference.” [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/APIReference. [Accessed: 03-Feb-2014].

[67] “CloudStack Admin Guide.” [Online]. Available: http://incubator.apache.org/cloudstack/docs/en-US/Apache_CloudStack/4.0.1-incubating/html/Admin_Guide. [Accessed: 03-Feb-2014].

[68] “OpenNebula - Flexible Enterprise Cloud Made Simple.” [Online]. Available: http://opennebula.org/documentation/. [Accessed: 03-Feb-2014].

[69] “OpenStack Compute Admininstration Manual-Essex (2012.1).” [Online]. Available: http://docs.openstack.org/essex/openstack-compute/admin/content/index.html. [Accessed: 03-Feb-2014].

[70] W. Lloyd, O. David, J. Lyon, K. W. Rojas, J. C. Ascough II, T. R. Green, and J. Carlson, “The Cloud Services Innovation Platform - Enabling Service-Based Environmental Modeling Using IaaS Cloud Computing,” in *Proceedings iEMSs 2012 International Congress on Environmental Modeling and Software*, 2012, p. 8.

[71] L. Hagen, “A wind erosion prediction system to meet user needs,” *J. Soil Water Conserv.*, vol. 46, no. 2, pp. 105–11, 1991.

[72] W. Lloyd, S. Pallickara, O. David, M. Arabi, and K. W. Rojas, “Dynamic Scaling for Service Oriented Applications: Implications of Virtual Machine Placement on IaaS Clouds,” in *Proceedings of the 2014 IEEE International Conference on Cloud Engineering (IC2E ’14)*, 2014.

[73] “Auto Scaling Concepts - Auto Scaling.” [Online]. Available: http://docs.aws.amazon.com//AutoScaling/latest/DeveloperGuide/AS_Concepts.html. [Accessed: 03-Feb-2014].

[74] P. Saripalli, C. Oldenburg, B. Walters, and N. Radheshyam, “Implementation and usability evaluation of a cloud platform for Scientific Computing as a Service (SCaaS),”

in *Proceedings - 2011 4th IEEE International Conference on Utility and Cloud Computing, UCC 2011*, 2011, pp. 345–354.

[75] I. Llorente, R. Montero, B. Sotomayor, D. Breitgand, A. Maraschini, E. Levy, and B. Rochwerger, "On the Management of Virtual Machines for Cloud Infrastructures," in *Cloud Computing: Principles and Paradigms*, Hoboken, NJ, USA: J Wiley & Sons, Inc., 2011.

[76] M. Mishra and A. Sahoo, "On theory of vm placement: Anomalies in existing methodologies and their mitigation using a novel vector based approach," in *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, 2011, pp. 275–282.

[77] Z. Xiao, W. Song, and Q. Chen, "Dynamic resource allocation using virtual machines for cloud computing environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, pp. 1107–1117, 2013.

[78] M. Andreolini, S. Casolari, M. Colajanni, and M. Messori, "Dynamic Load Management of Virtual Machines in a Cloud Architectures," in *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, 2010, pp. 201–214.

[79] A. Roytman, S. Govindan, J. Liu, A. Kansal, and S. Nath, "Algorithm design for performance aware VM consolidation, Techincal Report.," 2013.

[80] S. Ostermann, A. Iosup, N. Yigitbasim, R. Prodan, T. Fahringer, and D. Eperma, "A Performance Analysis of EC2 Cloud Computing Serices for Scientific Computing," in *Proceedings 1st International Conference on Cloud Computing (CloudComp '09)*, pp. 115–131.

[81] E. Walker, "Benchmarking Amazon EC2 for High-Performance Scientific Computing," *USENIX Login*, pp. 18–23, 2008.

[82] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud," *2010 IEEE Second Int. Conf. Cloud Comput. Technol. Sci.*, pp. 159–168, 2010.

[83] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud versus in-house cluster: Evaluating Amazon cluster compute instances for running MPI applications," *2011 Int. Conf. High Perform. Comput. Networking, Storage Anal.*, pp. 1–10, 2011.

[84] W. Lloyd, O. David, M. Arabi, J. C. Ascough II, T. R. Green, J. Carlson, and K. W. Rojas, "The Virtual Machine (VM) Scaler: An Infrastructure Manager Supporting Environmental Modeling on IaaS Clouds," in *Proceedings iEMSs 2014 International Congress on Environmental Modeling and Software*, p. 8.

[85] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. W. Rojas, "Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds," in *Proceedings - 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing, UCC 2012*, 2012, pp. 73–80.

[86] O. David, W. Lloyd, K. W. Rojas, M. Arabi, F. Geter, J. Carlson, G. H. Leavesley, J. C. Ascough II, and T. R. Green, "Model as a Service (MaaS) using the Cloud Service Innovation Platform (CSIP)," in *Proceedings iEMSs 2014 International Congress on Environmental Modeling and Software*, p. 8.

[87] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proceedings - International Conference on Distributed Computing Systems*, 2011, pp. 559–570.

[88] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud," in *Proceedings - 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD 2010*, 2010, pp. 236–243.

[89] A. Andrzejak, D. Kondo, and S. Yi, "Decision Model for Cloud Computing under SLA Constraints," in *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010, pp. 257–266.

[90] Q. Zhang, Q. Zhu, and R. Boutaba, "Dynamic Resource Allocation for Spot Markets in Cloud Computing Environments," *2011 Fourth IEEE Int. Conf. Util. Cloud Comput.*, pp. 178–185, 2011.

[91] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 spot instance pricing," in *Proceedings - 2011 3rd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2011*, 2011, pp. 304–311.

[92] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar, "Cost-efficient and application SLA-aware client side request scheduling in an infrastructure-as-a-service cloud," in *Proceedings - 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012*, 2012, pp. 213–220.

[93] J. L. L. Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Dynamic placement of virtual machines for cost optimization in multi-cloud environments," in *Proceedings of the 2011 International Conference on High Performance Computing and Simulation, HPCS 2011*, 2011, pp. 1–7.

[94] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *Proceedings - 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*, 2012, pp. 612–619.

[95] G. Galante and L. C. E. De Bona, “A survey on cloud computing elasticity,” in *Proceedings - 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing, UCC 2012*, 2012, pp. 263–270.

[96] P. M. Allen, J. G. Arnold, and W. Skipwith, “Prediction of channel degradation rates in urbanizing watersheds,” *Hydrological Sciences Journal*, vol. 53. pp. 1013–1029, 2008.

[97] J. Ditty, P. Allen, O. David, J. Arnold, M. White, and M. Arabi, “Deployment of SWAT-DEG as a Web Infrastructure Utilization Cloud Computing for Stream Restoration,” in *Proceedings iEMSs 2014 International Congress on Environmental Modeling and Software*, p. 6.

[98] R. L. Runkel, C. G. Crawford, and T. a Cohn, “Load Estimator (LOADEST): A FORTRAN program for estimating constituent loads in streams and rivers. Techniques and Methods Book 4 , Chapter A5. U.S. Geological Survey.,” *World*, p. 69, 2004.

[99] T. Wible, W. Lloyd, O. David, and M. Arabi, “Cyberinfrastructure for Scalable Access to Stream Flow Analysis,” in *Proceedings iEMSs 2014 International Congress on Environmental Modeling and Software2*, p. 6.

[100] B. Cleland, “An Approach for Using Load Duration Curves in the Development of TMDLs,” Washington DC 24060, 2007.