

Migration of Multi-tier Applications to Infrastructure-as-a-Service Clouds: An Investigation Using Kernel-based Virtual Machines

Wes Lloyd^{1,2}, Shrideep Pallickara¹, Olaf David^{1,2},
Jim Lyon², Mazdak Arabi²

¹Department of Computer Science

²Department of Civil and Environmental Engineering
Colorado State University, Fort Collins, USA
wes.lloyd, shrideep.pallickara, olaf.david, jim.lyon,
mazdak.arabi@colostate.edu

Ken Rojas

USDA-Natural Resource Conservation Service
Fort Collins, Colorado USA
Ken.Rojas@ftc.usda.gov

Abstract— To investigate challenges of multi-tier application migration to Infrastructure-as-a-Service (IaaS) clouds we performed an experimental investigation by deploying a processor bound and input-output bound variant of the RUSLE2 erosion model to an IaaS based private cloud. Scaling the applications to achieve optimal system throughput is complex and involves much more than simply increasing the number of allotted virtual machines (VMs). While scaling the application variants a series of bottlenecks were encountered unique to an application's processing, I/O, and memory requirements, herein referred to as an application's profile. To investigate the impact of provisioning variation for hosting multi-tier applications we tested four schemes of VM deployments across the physical nodes of our cloud. Performance degradation was more pronounced when multiple I/O or CPU resource intensive application components were co-located on the same physical hardware. We investigated the virtualization overhead incurred using Kernel-based virtual machines (KVM) by deploying our application variants to both physical and virtual machines. Overhead varied based on the unique characteristics of each application's profile. We observed ~112% overhead for the input/output bound application and just ~10% overhead for the processor bound application. Understanding an application's profile was found to be important for optimal IaaS-based cloud migration and scaling.

Keywords *Cloud Computing; Infrastructure-as-a-Service; Kernel-based virtual machines (KVM); provisioning variation; scalability; virtualization*

I. INTRODUCTION

Migration of multi-tier client/server applications 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 [1]. 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 multi-tier applications 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 [2-4]. 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 paper we investigate the following research questions:

- 1) *How can multi-tier client/server 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?*
- 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?*
- 3) *What overheads are incurred while using Kernel-based virtual machines (KVM) for hosting components of a multi-tier application?*

II. RELATED WORK

Rouk identified the challenge of finding optimal image and service composites, a first step in migrating multi-tier client/server applications to IaaS clouds in [5]. Chieu et al. [6] 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. [7] 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 [8]. 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 [9]. 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 multi-tier applications such as web services and models in IaaS clouds.

Schad et al. [3] 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. [2] 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 [4]. 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 multi-tier client/server applications 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 [10]. 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 [11]. 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 multi-tier applications in IaaS clouds.

III. PAPER CONTRIBUTIONS

This paper 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 [2][4] and virtualization studies which largely used common system benchmarks to quantify overhead [10-11]. Our work also extends prior research by investigating the migration of complete multi-tier applications to IaaS clouds [6-9] 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.

IV. EXPERIMENTAL INVESTIGATION

A. 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 [12]. 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 multi-tier application migration because its architecture consisting of a web server, relational database, file server, and logging server serves as a surrogate for classical multi-tier client/server based applications.

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 [13-14] using WINE [15] 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 [16].

A Eucalyptus 2.0 [17] 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 [10].

B. Application Components

Table I 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 [18-19]. The file server \mathcal{F} VM was used by the RUSLE2 model to acquire XML files to parameterize data for model runs. NGINX [20], 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 [21]. 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 [22].

TABLE I. 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.

C. Component Deployments

Our application stack of 4 components can be deployed 15 possible ways across 4 physical node computers. Tables II 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 II. PHYSICAL (P) AND VIRTUAL (V) STACKS DEPLOYMENTS

	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.

D. 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.

E. 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 multi-tier applications, 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.

V. EXPERIMENTAL RESULTS

A. 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 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 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. 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 \mathcal{D} VM virtual cores did not improve performance. When observing the \mathcal{D} VM's KVM process on the host machine, we observed with virtual CPU allocations (>6), the \mathcal{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.

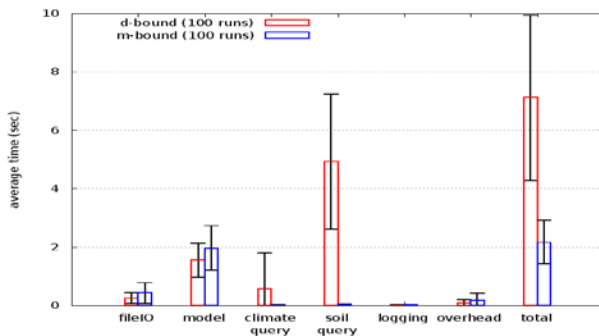


Figure 1. RUSLE2 application time footprint

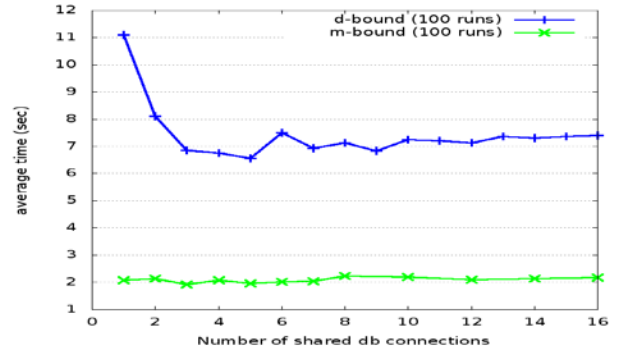


Figure 2. V1 stack with variable database connections

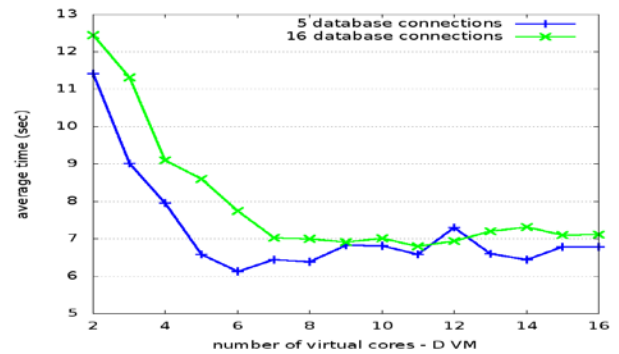


Figure 3. V1 stack d-bound with variable \mathcal{D} VM virtual CPUs

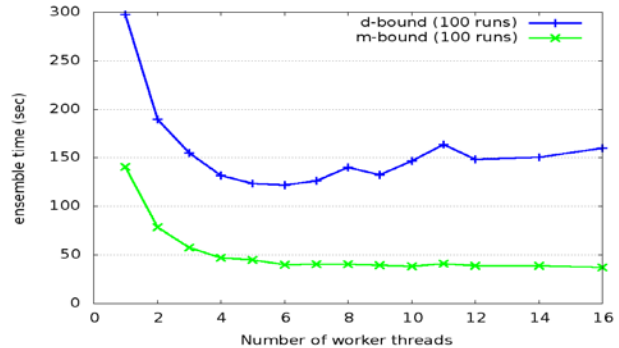


Figure 4. V1 stack with variable \mathcal{M} VM virtual CPUs

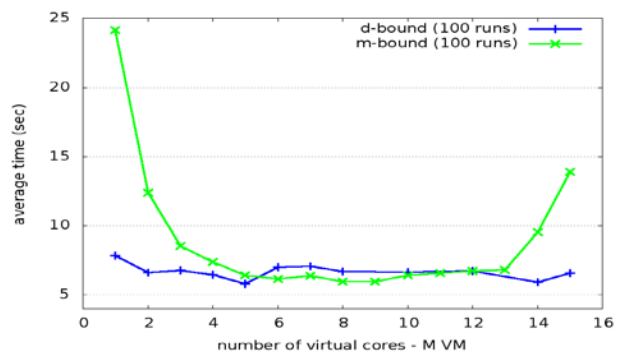


Figure 5. V1 stack with variable worker threads

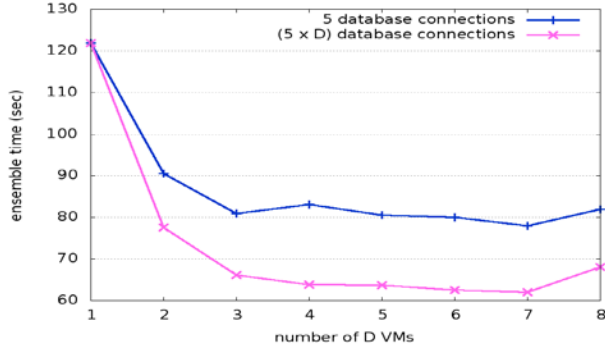


Figure 6. D-bound ensemble time with variable \mathcal{D} VMs

Figure 4 shows the average model run time while varying the number of virtual cores allocated to the \mathcal{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 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.

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.

B. 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 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.

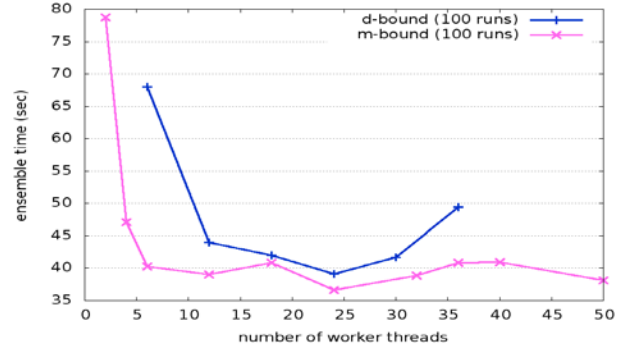


Figure 7. Ensemble runtime with variable worker threads

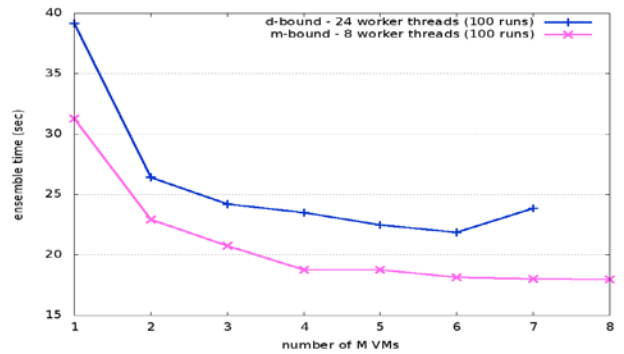


Figure 8. Ensemble runtime with variable \mathcal{M} VMs

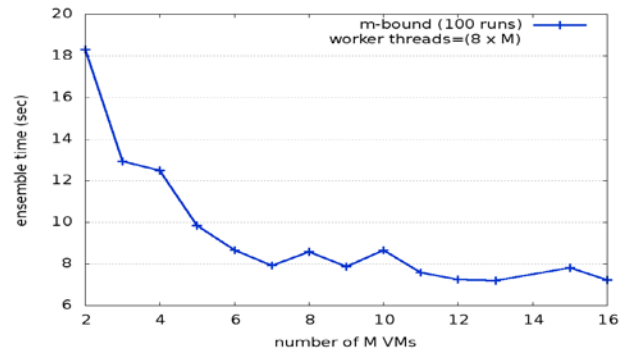


Figure 9. M-bound with variable \mathcal{M} VMs and worker threads

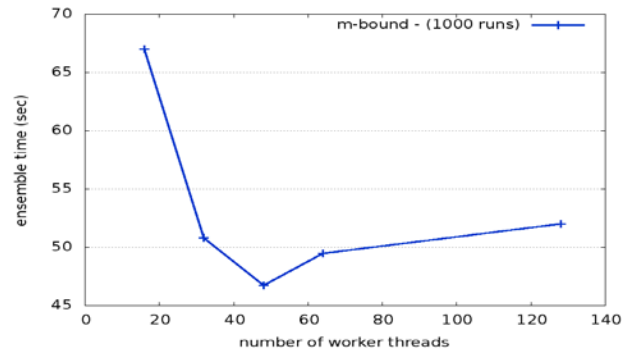


Figure 10. M-bound with 16 \mathcal{M} VMs variable worker threads

To move past the d-bound application bottleneck the number of worker threads was increased as shown in figure 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 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 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.

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 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 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.

C. 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 result. 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 III. 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

D. 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 IV. 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.

VI. CONCLUSIONS

Two variants of the RUSLE2 erosion model serving as surrogates for common multi-tier application 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 multi-tier applications 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.

REFERENCES

[1] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, "kvm: the Linux Virtual Machine Monitor," Proc. 2007 Ottawa Linux

Symposium (OLS 2007), Ottawa, Canada, June 27-30, 2007, pp. 225-230.

[2] M. Rehman, M. Sakr, "Initial Findings for Provisioning Variation in Cloud Computing," Proc. of the IEEE 2nd Intl. Conf. on Cloud Computing Technology and Science (CloudCom '10), Indianapolis, IN, USA, Nov 30 – Dec 3, 2010, pp. 473-479.

[3] J. Schad, J. Dittrich, J. Quiane-Ruiz, "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance," Proc. of 36th Intl. Conf. on Very Large DataBases (VLDB 2010), Singapore, China, Sept 13-17, 2010, pp. 460-471.

[4] M. Zaharia, A. Konwinski, A. Joesph, R. Katz, I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," Proc. 8th USENIX Conf. Operating systems design and implementation (OSDI '08), San Diego, CA, USA, Dec 8-10, 2008, pp. 29-42.

[5] M. Vouk, "Cloud Computing – Issues, Research, and Implementations," Proc. 30th Intl. Conf. Information Technology Interfaces (ITI 2008), Cavtat, Croatia, June 23-26, 2008, pp. 31-40.

[6] T. Chieu, A. Mohindra, A. Karve, A. Segal, "Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment," Proc. IEEE Conf. e-Business Engineering (ICEBE 2009), Macau, China, October 21-23, 2009, pp. 281-286.

[7] W. Iqbal, M. Dailey, D. Carrera, P. Janecek, "SLA-Driven Automatic Bottleneck Detection and Resolution for Read Intensive Multi-tier Applications Hosted on a Cloud," Proc. 5th Intl. Conf. on Advances in Grid and Pervasive Computing (GPC 2010), Hualien, Taiwan, May 10-13, 2010, pp. 37-46.

[8] H. Liu, S. Wee, "Web Server Farm in the Cloud: Performance Evaluation and Dynamic Architecture," Proc. IEEE 1st Intl. Conf. on Cloud Computing Technology and Science (CloudCom '09), Beijing, China, Dec 1-4, 2009, 12p.

[9] S. Wee, H. Liu, "Client-side Load Balancer using Cloud," in Proc. 25th Symposium on Applied Computing (SAC 2010), Sierre, Switzerland, March 22-26, 2010, pp. 399-405.

[10] F. Camargos, G. Girard, B. Ligneris, "Virtualization of Linux servers," Proc. 2008 Linux Symposium, Ottawa, Ontario, Canada, July 23-26, 2008, pp. 63-76.

[11] D. Armstrong, K. Djemame, "Performance Issues In Clouds: An Evaluation of Virtual Image Propagation and I/O Paravirtualization," The Computer Journal, June 2011, vol. 54, iss. 6, pp. 836-849.

[12] United States Department of Agriculture – Agricultural Research Service (USDA-ARS), Revised Universal Soil Loss Equation Version 2 (RUSLE2), http://www.ars.usda.gov/SP2UserFiles/Place/64080510/RUSLE/RUSLE2_Science_Doc.pdf

[13] L. Ahuja, J. Ascough II, and O. David, "Developing natural resource modeling using the object modeling system: feasibility and challenges," Advances in Geosciences, vol. 4, 2005, pp. 29-36.

[14] O. David, J. Ascough II, G. Leavesley, L. Ahuja, "Rethinking modeling framework design: Object Modeling System 3.0," Proc. iEMSs 2010 Intl. Congress on Environmental Modeling and Software, Ottawa, Canada, July 5-8, 2010, 8 p.

[15] WineHQ – Run Windows applications on Linux, BSD, Solaris, and Mac OS X, <http://www.winehq.org/>

[16] Apache Tomcat – Welcome, 2011, <http://tomcat.apache.org/>

[17] D. Nurmi et al., "The Eucalyptus Open-source Cloud-computing System," Proc. IEEE Intl. Symposium on Cluster Computing and the Grid (CCGRID 2009), Shanghai, China, May 18-21, 8p.

[18] PostGIS, 2011, <http://postgis.refractory.net/>

[19] PostgreSQL: The world's most advanced open source database, <http://www.postgresql.org/>

[20] nginx news, 2011, <http://nginx.org/>

[21] Welcome to CodeBeamer, 2011, <https://codebeamer.com/cb/user/>

[22] HAProxy – The Reliable, High Performance TCP/HTTP Load Balancer, <http://haproxy.1wt.eu/>