Kangaroo: A Tenant-Centric Software-Defined Cloud Infrastructure

Kaveh Razavi, Ana Ion, Genc Tato, Kyuho Jeong, Renato Figueiredo,
Guillaume Pierre, Thilo Kielmann

Kangaroo: A Tenant-Centric Software-Defined Cloud Infrastructure

Kaveh Razavi*, Ana Ion*, Genc Tato†, Kyuho Jeong‡, Renato Figueiredo‡, Guillaume Pierre†, and Thilo Kielmann*

*VU University Amsterdam, †IRISA / Université de Rennes 1, and ‡University of Florida

***Abstract*—Applications on cloud infrastructures acquire virtual machines (VMs) from providers when necessary. The current interface for acquiring VMs from most providers, however, is too limiting for the tenants, in terms of granularity in which VMs can be acquired (e.g., small, medium, large, etc.), while giving very limited control over their placement. The former leads to VM underutilization, and the latter has performance implications, both translating into higher costs for the tenants. In this work, we leverage nested virtualization and a networking overlay to tackle these problems. We present Kangaroo, an OpenStack-based virtual infrastructure provider, and IPOP$_{\mathrm{sm}}$, a virtual networking switch for communication between nested VMs over different infrastructure VMs. In addition, we design and implement Skippy, the realization of our proposed virtual infrastructure API for programming Kangaroo. Our benchmarks show that through careful mapping of nested VMs to infrastructure VMs, Kangaroo achieves up to an order of magnitude better performance, with only half the cost on Amazon EC2. Further, Kangaroo's unified OpenStack API allows us to migrate an entire application between Amazon EC2 and our local OpenNebula deployment within a few minutes, without any downtime or modification to the application code.**

I. Introduction

Cloud applications are extraordinarily varied, ranging from one-person projects to huge collaborative efforts, and spanning every possible application domain. Many applications start with modest computing resource requirements which vary thereafter according to the success or failure of their authors. However, they all need to define their resources as a combination of instance types selected from a standard list.

Infrastructure providers promote horizontal scaling (the addition or removal of VMs of identical types) as a practical solution for applications which need to vary their resource usage over time. However, although this provides a simple and efficient model for the infrastructure providers, it presents stringent limitations for cloud tenants. First, standard instance types seldom match the exact resource requirements of particular applications. As a consequence, cloud applications rarely utilize all the resources offered by their VMs. Second, cloud platforms offer little support for fine-grained VM placement. However, as we show later in this paper, co-locating compatible workloads in the same physical machine can bring up to an order of magnitude performance improvements.

To address these challenges this paper proposes Kangaroo, a tenant-centric nested infrastructure. Kangaroo executes cloud applications in VMs running within VMs that are allocated from the infrastructure providers. This nested infrastructure enables fine-grained resource allocation and dynamic resizing. When a set of nested VMs outgrow the available capacity of their hosting VM, they can be live-migrated to another possibly new larger VM through a programmable interface. This also allows co-locating complementary workloads in the same machine, which in turn increases resource utilization while improving performance. Finally, Kangaroo runs indifferently within VMs hosted by any cloud provider such as Amazon EC2 and private clouds. It can therefore live-migrate unmodified applications across cloud providers and across different types of virtual machine monitors (VMMs), without disrupting application behavior.

Our evaluations based on microbenchmarks and real-world applications show that Kangaroo's fine-grained resource allocation can significantly reduce resource usage compared to horizontal scaling, while co-locating compatible workloads brings order-of-magnitude performance improvements. Live-migrating workloads across VMs and across cloud providers exhibits acceptable overheads, without imposing any downtime nor application modifications.

The remainder of this paper is organized as follows: Section II discusses the background and our motivation for this work. Sections III and IV respectively present Kangaroo's architecture and implementation. Section V presents evaluations and, after discussing the related work in Section VI, Section VII concludes.

II. Motivation and Background

We describe our motivation for this work by formulating some important tenant requirements that are currently not available in infrastructure providers (Section II-A). We then discuss why it is sometimes difficult or against the interest of the providers to implement these requirements (Section II-B). After establishing our motivation, we take a detailed look at nested virtualization, the driving technology behind Kangaroo that provides the desired tenant requirements (Section II-C).

A. *Tenant requirements*

There are three desired tenant requirements that current providers do not support.

1) Fine-grained VM Allocation: Currently, whenever a tenant needs to scale out, she has the choice between a number of *instance types* that provide the tenant with various amount of resources. These instance types are defined statically, and they reflect resource requirements of various classes of applications. For example, a tenant with a compute-bound workload can acquire a VM from the class of instance types that provide a high number of cores, or a number of GPUs if the workload can run on GPUs.

This "instance type" abstraction creates a static granularity in which resources can be acquired from a provider. From the tenant's point of view, the static nature of this abstraction is limiting: It does not capture the requirements of all possible applications, and for the ones that it does, it is often not an exact match. This results in provisioning of unnecessary resources for the sake of the ones necessary.

As an example, assume that a tenant needs to allocate 10 cores with 2 GB of memory for a certain application. Since this amount of resources is asymmetrical, it is very unlikely that the provider has an instance type with these exact resource requirements. The tenant has two choices: 1) Either allocate one VM from an instance type with the same number of cores, but excessive amount of memory, or 2) a number of VMs from smaller instance types that provide the same number of cores. The former results in over-provisioning of memory and hence waste of money, and the later results in high core-to-core latency, possibly affecting performance. Even with balanced resource requirements, it can easily be that there is no matching instance type.

Since the tenant knows the requirements of her application the best, she should be able to define instance types *dynamically*, and be able to allocate VMs from it.

2) Control over VM placement: A tenant has the best knowledge about how the components of her application communicate with each other. Naturally, the closer the components that communicate with each other are, the better networking performance they observe. This means that the tenants should have control over the mapping of their VMs to the underlying infrastructure. Further, the flexibility of controlling VM placement opens up interesting resource management possibilities that we will discuss in Section IV-C.

Currently, there is no or very limited support for controlled placement of VMs on the infrastructure's physical resources. For example, Amazon EC2 does not provide any support for co-location of most of its instance types. There is some support for placement of cluster compute instance types within a network switch, but these instance types are only targeted towards very specific high performance computing applications that can afford the premium price of these instance types [1]. Google Compute Engine has recently announced some support for migrating VMs between physical resources, but this feature is currently only exposed to the provider (i.e., Google) for maintenance, rather than the tenants [2].

Ideally, the tenant should be *informed* about how her VMs are mapped to physical resources of the provider, and be able to *migrate* them whenever necessary.

3) Unified Provider API: Different infrastructure providers expose different set of APIs for allocating VMs on their resources. While there is substantial effort in standardization of this API [3], [4], it takes a long time for providers to adopt a new standardized API. Abstracting away this API through a library (e.g., libcloud [5], boto [6]) only partially solves this problem; Applications still need to be aware of low-level details of the provider as the libraries cannot abstract away the complete API, and not all providers support the same features.

As a direct result, usually it is tedious to port an application that is written for one provider to another. To make the matter worse, different providers have adopted different virtualization technologies (e.g., Xen, KVM, Hyper-V, etc.) which means that it is typically not possible to run a VM using a certain virtualization technology on another. Thus, even if the API is standardized and adopted by providers, it is still not straightforward to simply migrate an application from one provider to another.

The final requirement is the ability to avoid vendor lock-in by using a *unified API* for allocating VMs, and the possibility for migrating an application without changing its code or any downtime.

B. Infrastructure Provider Conflicts

Implementing the tenants' requirements is not always straightforward for the providers. As we shall see, it may also not be in their interest.

1) Fine-grained VM Allocation: Static instance types allow for simpler scheduling, physical resource provisioning, and pricing. Providing support for fine-grained VM allocation means more engineering effort, and possibly less flexibility in physical resource management that directly translates into higher operational cost.

2) Control over VM placement: Without tenant support for VM migration, the provider has the complete control over the location of all tenants' VMs. It can migrate the VMs at will for resource management purposes such as scaling out/in its physical resources. Supporting tenant-controlled VM placement makes it difficult, if not impossible, to perform these simple resource management tasks.

3) Unified Provider API: Infrastructure providers started appearing before any coordinated standardization effort, and so did the virtualization technologies that the providers adopted. It will be a substantial engineering effort to provide a new set of API while preserving and maintaining the old one. Further, the vendor lock-in helps keeping a stable user base that a standard API threatens.

Given these facts, we propose a tenant-centric architecture that implements the desired high-level tenant requirements. The basic building-block for our architecture is nested virtualization, which we discuss next.

C. Nested Virtualization

Nested virtualization enables the execution of virtual execution units in an already virtualized environment by running a VMM inside a VM. While nested virtualization introduces performance overhead compared to a UNIX process, it can provide better *isolation*, *security* [7], *reproducibility* [8], more *control* over VMs acquired from the infrastructure providers [9], and support for VMM *development* and *migration* [10].

For running the second layer VMs, the user can choose between full virtualization or operating system virtualization (i.e., containers). Each choice come with different possible technologies that can be deployed. We first discuss the non-functional properties that are important for this work, and then compare different technologies according to these properties.

	Hardware support	Stack modification	Performance	Migration
LXC	no	++	++	no
QEMU	no	++	–	yes
HVX	no	++	++	yes
Turtles	yes	–	++	yes
Xen-blanket	yes	+	+	yes

TABLE I: Comparison of different technologies for nested virtualization.

We consider four non-functional properties to be of importance to Kangaroo:

1) Hardware support: Recently, processors have started exposing specific mechanisms to support virtualization (e.g., Extended Page-tables [11] and EPT) and nested virtualization (e.g., nested EPT [10]). While hardware-assisted (nested) virtualization is attractive in terms of delivering close to bare-metal performance, it is not yet available on all infrastructure providers. Hence, a nested virtualization technology that adopts these hardware technologies is limited to providers that offer VMs executing on supported hardware.

2) Stack modification: Different technologies make different assumptions on the extent to which the provider's software stack can be modified to accommodate their requirements. Tenants obviously have a preference for technologies that requires fewer (or no) software modifications, to be able to adopt as many providers as possible.

3) Performance: “All problems in computer science can be solved by another level of indirection”[1], but there is (almost) always an associated trade-off. Here, there is a performance penalty associated with nesting. By adding yet another layer of virtualization, the application performance inside the nested VM degrades. An ideal solution has minimal impact on the application performance.

4) Migration: Virtualization enables migration by abstracting away the underlying hardware. VM (live) migration provides endless possibilities in terms of resource management [12], [13]. The ability of a nested VMM to migrate its VMs is important for a tenant-centric approach to infrastructure resource management.

Table I summarizes virtualization solutions according to these criteria. While HVX, Xen-blanket and Turtles have been explicitly designed for nested scenarios, LXC and QEMU are used in non-nested environments as well. In the rest of this section, we describe these solutions in detail.

1) LXC: Linux Containers [14] are a light-weight operating system-level solution for running multiple isolated Linux systems on the same host. An LXC container does not own an entire OS stack, but it reuses functionalities provided by its host's kernel. Each container runs in a virtual environment characterized by its own CPU, memory, block I/O access and network bandwidth. As no actual VMM is available, resource management, isolation, and security are provided directly by the Linux kernel using cgroups [15], namespaces [16], SELinux [17], and AppArmor [18].

LXCs represent a low-overhead alternative to full-fledged VMs, suitable for nested scenarios. In [19], the authors show that regardless of the type of workload, LXC containers exhibit similar performance to that of native execution. However, as containers share the kernel with their host, usage scenarios are restricted to Linux guests. Although currently live-migration is not included in the LXC framework yet, this feature is already provided by the CRIU project [20] and awaits integration with popular cloud projects such as OpenStack.

2) QEMU: An alternative approach is emulation. QEMU [21] is a popular open-source virtualization solution available in Linux and Windows environments. It implements hardware virtualization, where an application sees an abstract interface which completely hides the physical characteristics of a platform. This gives the VM's OS the illusion it is running on its own hardware. While QEMU supports hardware acceleration via KVM, its emulation capabilities makes it a good candidate for running nested VMs without relying on any hardware support or the first layer VMM. In order to execute guest code, instructions are dynamically interpreted at runtime using a method called binary translation. QEMU however provides portability at the expense of significant performance degradation.

3) HVX: QEMU's dynamic binary translation model is also leveraged by the HVX [22] nested VMM for running unmodified VMWare, KVM and Hyper-V VMs over virtual resources. In single-layer virtualization scenarios, VMMs often utilize the hardware extensions to efficiently execute a VM. However, in a virtual environment, these extensions are no longer available. To preserve performance, HVX translates the nested VM's ring-0 code to enforce security and correctness, and directly executes unprivileged nested VM's ring-3 code.

Designed specifically for nested environments, HVX represents a competitive alternative for container-based solutions. However, this is a proprietary solution that would increase the cost incurred by the end users.

4) Turtles: The Turtles project optimizes nested executions by modifying the first-layer VMM [23]. Turtles is an extension to KVM, which is integrated in Linux, making it a popular virtualization solution. Turtles follows the single-level support for nested virtualization. All the privileged operations generated by the layer two VMM are recursively trapped and forwarded to the lowest one. Normally, this approach would add a significant overhead, but in Turtles' case, it is limited by the use of a series of optimizations at CPU, memory and I/O level such as multi-dimensional page tables and multi-level device assignment. Although KVM is a widely used VMM, the constraint of modifying the first level VMM limits its usability to private data-centers, making this solution unsuitable for scenarios that also target public providers.

5) Xen-blanket: Xen-blanket proposes a tenant-centric approach for efficiently implementing nested virtualization [9]. The second layer VMM is represented by a modified Xen implementation capable of running paravirtualized VMs on top of hardware-assisted VMs. The solution preserves existing Xen functionalities, such as VM migration, but it also enforces the paravirtualized specific limitation of being unable to run unmodified VMs. Xen-blanket obtains good performance for the nested VMs at the expense of requiring specific drivers for each type of first layer VMM.

[1] David Wheeler

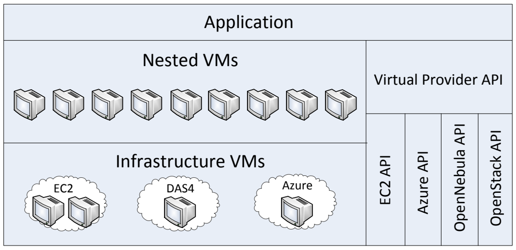

Fig. 1: A nested architecture that virtualizes an infrastructure provider and satisfies the desired tenant requirements not available in infrastructure providers today.

6) Hardware Support for Nested Virtualization: The performance of nested environments can be significantly improved with recent progress in processor architecture from both Intel and AMD. For example, Intel's new Haswell line introduces VMCS shadowing, a feature which allows a nested VMM to access processor virtualization extensions directly, which improves the performance of nested VMs.

While these nested extensions seem to primarily target security and reliability of cloud services, we believe that they can also be leveraged to provide better resource management support for tenants without the associated performance penalty.

D. Discussion

We discussed our motivations for this work in this section. We then looked at nested virtualization, an important building block for Kangaroo. After comparing existing solutions, we decided to use LXCs for our nested infrastructure to preserve performance. LXCs also start up faster than VMs, potentially benefiting soft real-time cloud applications (e.g. autoscalers) that would otherwise require support for scalable VM start up from the providers [24], [25]. However, since LXCs do not have proper support for migration yet, we have used QEMU for our cross-provider migration benchmarks described in Section V-C.

III. Architecture

In this section, we first describe our nested architecture, and then formulate some research questions that come with it.

The tenant requirements discussed in Section II-A rely on basic infrastructure support. We employ nested virtualization as discussed in Section II to enable the possibility of a "virtual infrastructure" that can potentially provide the users this necessary support.

Figure 1 shows our proposed architecture. In this architecture, VMs of different instance types from different infrastructure providers are allocated to back our virtual infrastructure. On top of this virtual infrastructure, we allocate nested VMs from tenant-defined instance types, effectively resolving the first tenant requirement (fine-grained VM allocation).

Further, the tenant is in control of the mapping between nested VMs and infrastructure VMs. She can choose the infrastructure VM for allocating nested VMs, and if necessary migrate nested VMs from one infrastructure VM to another, effectively providing support for the second tenant requirement (control over VM placement).

To manage this virtual infrastructure, we expose a common API to the tenants, and translate it to that of the "real" infrastructure providers. This translation effort is done only once for the virtual infrastructure, and then reused by all tenants through the single virtual infrastructure API. Writing applications against this unified virtual infrastructure API allows tenants to run their applications on different infrastructure providers without any change to the code. Further, since the tenant is in control of the virtualization technology of the nested VMs, she can migrate an entire application across providers without downtime, effectively satisfying the third tenant requirement (unified provider API).

Our nested architecture should implement all requirements discussed in Section II-A. To that end, we need to address a number of research challenges:

- What are the trade-offs in implementing a virtual infrastructure? (Section IV-A)
- What kind of networking primitive is necessary to provide connectivity between nested VMs? (Section IV-B).
- How does a virtual infrastructure API look like? (Section IV-C).
- What are the performance implications of running a virtual infrastructure? (Section V).

In the next section, we describe our implementation of this proposed architecture to address these questions.

IV. Implementation

Our implementation of the proposed architecture in Section III consists of three main components:

1) **Kangaroo** creates a tenant-centric virtual infrastructure on top of VMs allocated from different providers by configuring these VMs as OpenStack compute nodes [26], and creating nested VMs on tenants' requests. We elaborate further on Kangaroo in Section IV-A.
2) **IPOP$_{\mathbf{sm}}$** provides network connectivity between nested VMs, by implementing a virtual switch and an overlay spanning various providers' IP networks. IPOP$_{\mathbf{sm}}$ is a fork of the IPOP project [27] that extends the functionalities of a peer-to-peer IP overlay to a virtual layer-2 switch by forwarding ARP messages in its controller. We discuss the reasons for choosing a switch-based virtual overlay and the high-level details of our implementation in Section IV-B.
3) **Skippy** implements the virtual infrastructure provider API by extending libcloud's OpenStack plugin [28]. We elaborate on this API and the implementation of Skippy in Section IV-C.

A. Kangaroo

Our virtual infrastructure exposes an interface to its tenants for allocating (nested) VMs. This interface is then implemented by the infrastructure cloud middleware. Instead of designing and implementing our own API and middleware, we decided to reuse an existing API (OpenStack compute API [29]) and

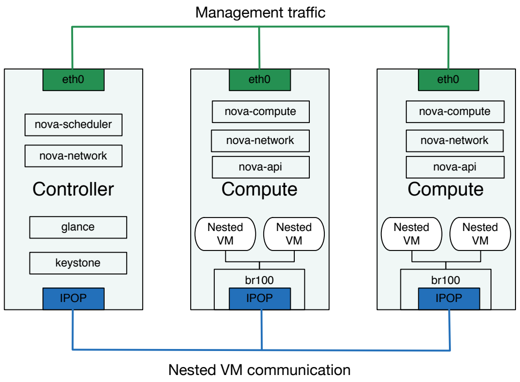

Fig. 2: Kangaroo configures infrastructure VMs as its compute nodes by running OpenStack compute and network services as part of VMs' contextualization. Communication between nested VMs located on different infrastructure VMs is made possible with IPOP$_{\mathbf{sm}}$, discussed in Section IV-B

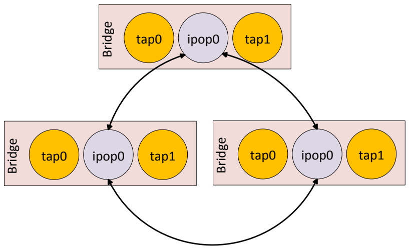

Fig. 3: IPOP$_{\mathbf{sm}}$'s architecture. Each bridge is controlled by one IPOP endpoint (ipop0). Tap interfaces (nested VMs) on the same bridge communicate through their bridge, and through ipop0 to tap interfaces on other bridges.

	VM to VM		LXC to LXC	
	eth0	IPOP$_{\mathbf{sm}}$	same VM	diff VMs
bandwidth(Mb/s)	938,3	69,70	8442,36	19,63
standard deviation	1,18	4,61	79,61	0,54

TABLE II: Comparison of qperf performance between different virtualized environment.

its implementation (OpenStack). The benefits of doing so are twofold: 1) tenant applications that are written against OpenStack are backwards-compatible, and 2) we do not need a new middleware implementation, and we can reuse features from an existing implementation.

Kangaroo is a "configuration" of OpenStack, with a number of scripts to 1) prepare an infrastructure VM to be configured as an OpenStack compute node by running OpenStack's compute and network services on top of that VM, or 2) to prepare an infrastructure VM to be removed from OpenStack when necessary.

Figure 2 shows the high-level architecture of Kangaroo. We use one infrastructure VM hosting an OpenStack controller. This controller can also be configured to run nested VMs if desired. Kangaroo can either be scaled out/in manually or programmatically through Skippy, discussed in Section IV-C. We need a network interface for management traffic between the OpenStack controller and its compute (VM) nodes. In situations where there is no direct connectivity between OpenStack's controller and the compute VMs, we can reuse the same facility that we have developed for nested VM communication, discussed in Section IV-B.

B. IPOP$_{sm}$

We first discuss the requirements for communication among nested VMs, and then we describe IPOP$_{\mathbf{sm}}$, a network overlay implementation that meets these requirements.

Since in Kangaroo the tenant is in control of VM placement, naturally, the nested VMs that communicate the most should be placed as close to each other as possible. Thus, the network overlay should optimize for the cases when nested VMs are co-located (i.e., running on the same infrastructure VM). The second requirement is to provide seamless connectivity between nested VMs when they are not placed on the same infrastructure VM, and when they migrate.

IPOP$_{\mathbf{sm}}$ addresses these requirements without requiring any modifications or privileged access to infrastructure VMs. IPOP$_{\mathbf{sm}}$ supports the ARP layer-2 protocol by intercepting and tunneling packets through a peer-to-peer overlay. IPOP$_{\mathbf{sm}}$ uses Linux bridges for providing connectivity between nested VMs running on the same infrastructure VM, and an ethernet over peer-to-peer overlay to provide connectivity between nested VMs running on different infrastructure VMs. IPOP$_{\mathbf{sm}}$ reuses most of IPOP's functionality to automatically establish, monitor, and maintain peer-to-peer tunnels to provide connectivity between endpoints that are behind NATs — in this case, our nested VMs. As shown in Figure 2, by configuring OpenStack to treat IPOP$_{\mathbf{sm}}$'s interface as the (nested) VMs network, and infrastructure provider interface as the management network, we accomplish seamless integration of IPOP$_{\mathbf{sm}}$ with OpenStack. We now focus on the internal details of IPOP$_{\mathbf{sm}}$.

Figure 3 overviews ARP handling in IPOP$_{\mathbf{sm}}$. Each IPOP$_{\mathbf{sm}}$'s endpoint can be slaved to a Linux bridge that connects multiple interfaces. IPOP$_{\mathbf{sm}}$'s endpoint listens on the ARP traffic over the bridge, and reacts to ARP requests whenever necessary. There are four types of ARP traffic that the IPOP$_{\mathbf{sm}}$'s endpoint needs to deal with:

1) Local ARP request: forward this request to others.
2) Local ARP response: forward this response to the other endpoints, and cache this response for a certain period of time.
3) Remote ARP request: If the information is available in the local cache (i.e., IP is located in the local bridge), send a response for that IP while changing the destination MAC address to that of its own. All traffic for the local (nested VM) interface slaved by the bridge will be received by the endpoint and will be forwarded to the right interface. If the information is not available in the cache, forward the request to the local bridge.
4) Remote ARP response: change the source MAC address of the response and forward it to the bridge. From this point, the endpoint will receive all the local traffic for this IP, and will forward it to the appropriate endpoint.

These ARP handling mechanisms provide the abstraction that all nested VMs are located on the same LAN. Since the endpoint is not in the path for local nested VMs communi-

$add\_compute\_node(provider, type_p)$	Adds an infrastructure VM of $type_p$ from *provider* as a compute node to Kangaroo.
$remove\_compute\_node(compute\_node)$	Removes the specified infrastructure VM from Kangaroo. This call might trigger migration of nested VMs from this infrastructure VM.
$add\_type(core, memory, etc.)$	Adds a VM type with the specified resource requirements to Kangaroo.
$allocate(compute\_node, type)$	Allocates a nested VM from the specified $type$ on the specified $compute\_node$ (i.e. infrastructure VM).
$migrate(vm\_id, compute\_node)$	Migrates the specified nested VM to the specified $compute\_node$.

TABLE III: The extended infrastructure management API implemented by Skippy.

cation, their packets will directly go though the infrastructure VM's memory. This means that the communication between nested VMs hosted on the same infrastructure VM goes through memory, without virtualization overhead, satisfying the first requirement discussed earlier. Further, in cases where the communicating nested VMs are hosted on different infrastructure VMs, they can still communicate with each other through this IP overlay, satisfying the second requirement.

Table II shows the network bandwidth achieved with qperf in different scenarios on two VMs on the DAS4 cluster connected via 1 GbE (details in Section V). Two LXCs running on the same VM achieve up to about one order of magnitude better networking performance compared to two VMs located on different physical hosts. The former is the common case in Kangaroo, whereas the latter is the common case in infrastructure providers today. We have left optimizing nested VM communication over IPOP$_{\mathbf{sm}}$ as future work. We will show in Section V-B1, how this improved networking performance can benefit overall performance of co-located services.

C. Skippy

As discussed earlier, we use OpenStack as our infrastructure middleware. Hence, regardless of the infrastructure provider, the tenant will use this unified API for allocating (nested) VMs. We extend this API to support the management operations on the tenant's virtual infrastructure provider. These operations include scaling out/in the virtual infrastructure, support for creating VM types dynamically (Section II-A1), and control over (nested) VM placement and migration (Section II-A2). This extension effectively provides the tenants with a software-defined virtual infrastructure.

Table III explains the details of our proposed virtual infrastructure management API. Using this API, the tenant can scale up her virtual infrastructure by attaching new infrastructure VMs from different providers as new compute nodes to Kangaroo. New VM types can be created by specifying the required resources such as cores, memory, etc. Using these VM types, the tenant can ask for nested VMs, on the specified compute node. Finally, the tenant can initiate migration of nested VMs from a source infrastructure VM to a destination infrastructure VM, potentially migrating her workload from a certain provider to another.

Skippy is our implementation of this proposed API as a library that cloud applications can use. Skippy is implemented as an extension of libcloud's OpenStack plugin [29]. libcloud provides us with the basis for scaling out/in Kangaroo on different infrastructure providers (the first two calls in Table III), and the OpenStack plugin that Skippy is based upon, provides us with the original OpenStack API (The last three calls in Table III).

D. Summary

We described our implementation of Kangaroo and IPOP$_{\mathbf{sm}}$ that follows the nested architecture discussed in Section III. We also discussed Skippy, our implementation of a software-defined virtual infrastructure provider for scaling out/in Kangaroo, allocation of nested VMs on desired compute nodes (i.e., infrastructure VMs), and migration of nested VMs across compute nodes. Skippy provides the necessary mechanisms for efficient resource management to a higher-level entity with the knowledge of applications' resource requirements (e.g. a Platform-as-a-Service).

V. Evaluation

We evaluate various aspects of our nested infrastructure in this section. In Section V-A, we briefly describe our use-case ConPaaS [30], an open-source Platform-as-a-Service (PaaS), that we have ported to Kangaroo. We show that ConPaaS services on top of Kangaroo can benefit from 1) better resource utilization (Section V-B2), 2) better performance due to co-location, and 3) with minimal interference with other independent workloads (Section V-B1). Further, we show that Kangaroo can live-migrate nested VMs with an acceptable overhead (Section V-C1), and it can live-migrate a real-world application consisting of two ConPaaS services between Amazon EC2 and our local OpenNebula provider within a matter of minutes without any change to the services (Section V-C2).

Our experiments were conducted on two cloud platforms: a private OpenNebula environment available on the DAS4/VU cluster [31] and Amazon EC2. On DAS4 we utilized medium instances equipped with 4 GB of memory and 4 cores running on physical nodes connected via 1 Gb/s Ethernet. For the EC2 experiments, we utilized different instance types, so we specify their flavor for each experiment. On the infrastructure VMs, we are running Ubuntu 14.04, and for the nested VMs, we are running Debian Squeeze. For the experiments in Sections V-B, we have used LXCs for nesting, and for the experiments in Section V-C, we have used QEMU emulation. We are planning to repeat the migration experiments with LXCs, as soon as CRIU [20] is properly integrated in libvirt-lxc [32].

A. ConPaaS and Integration with Kangaroo

ConPaaS is an open-source framework that provides a runtime environment for hosting applications on infrastructure clouds. Addressing both high performance and Web applications, ConPaaS supports a broad range of services: web hosting services (PHP, Java), storage services (MySQL DB, Scalarix NoSQL, XtreemFS POSIX FS), high performance services (MapReduce, TaskFarming, High-Throughput Condor), In addition, the service catalog can be complemented by user defined implementations.

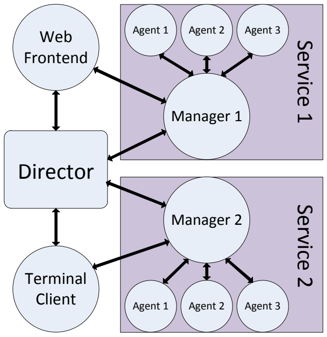

Fig. 4: ConPaaS architecture.

Figure 4 shows the architecture of ConPaaS. Each service consists of a manager VM, and a set of agent VMs. The manager VM performs management tasks for the service such as scaling decisions, and the agent VMs perform the actual work. The agent VMs are scaled out/in by the managers in response to variation in workload. In ConPaaS terms, an application is an orchestration of a number of services. For example, a WordPress application consists of a web service and a database service. The application logic (i.e., the management of service managers), as well as user authentication, and accounting are implemented by the ConPaaS director.

While the careful separation of duties in ConPaaS has resulted in a clean and robust design, resource allocation can become wasteful on a public infrastructure provider since the instance types do not match the requirements of different components. Hence, we believe ConPaaS is a good candidate to benefit from the resource management flexibility that Kangaroo provides.

We have ported ConPaaS to Kangaroo with minimal efforts since ConPaaS supported OpenStack already. We just exposed Skippy's API for adding and removing virtual compute nodes, as well as defining new instance types to ConPaaS director and to ConPaaS managers. We further exposed a mechanism to migrate ConPaaS services and applications using Skippy's migration capabilities. In the rest of this section, we show how ConPaaS benefits from these new mechanisms.

B. Co-location

One of the privileged operations that is possible in a nested platform such as Kangaroo is control over VM placement. Consequently, ConPaaS can implement co-location policies that can maximize resource utilization to reduce cost, while improving performance due to improved networking performance. We show these improvements using a number of ConPaaS services.

1) Performance: We compare the performance of WordPress, a real-world ConPaaS application using VMs acquired from Amazon EC2 and from Kangaroo running on top of a VM that is also acquired from Amazon EC2. WordPress consists of two ConPaaS services: PHP and MySQL. Each of the services require two VMs (four VMs in total).

When running directly on top of Amazon EC2, we chose m1.small for VM allocation, the smallest instance type with sustained performance. At the time of writing this paper, this translates into 0.104 \$ monetary cost per hour (4×0.026 \$). When running on Kangaroo, we chose a single t2.medium to run all nested containers. This translates into 0.052 \$ monetary cost per hour, which is half of the non-nested case.

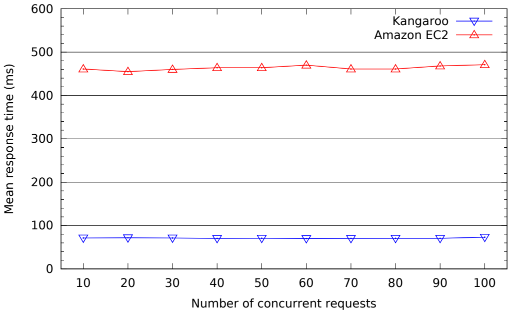

Fig. 5: Mean resp. time of WordPress under an increasing number of concurrent requests.

Figure 5 shows the mean response time of WordPress when increasing the number of concurrent requests using the AB benchmark [33]. The response time of ConPaaS using Kangaroo is about an order of magnitude better than the response time of ConPaaS when using Amazon EC2 VMs directly and remains constant till a hundred concurrent requests. The reason for this improvement is mostly due to co-location of the PHP service and MySQL that avoids a round trip network delay between these services.

In the next section, we look at the resource utilization of WordPress application in both scenarios to better understand why such performance numbers are achievable despite 50% saving in cost.

2) Resource Utilization: We look at the resource utilization of the aforementioned WordPress application. We have measured the amount of idle CPU cycles and memory while executing the same benchmark that we discussed in the previous section.

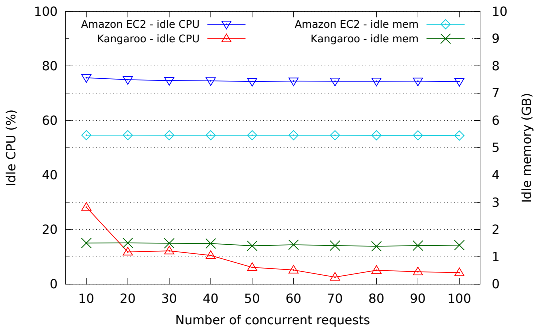

Fig. 6: Idle resources of WordPress under an increasing number of concurrent requests.

Figure 6 shows the results for both CPU and memory. It is clear that the ConPaaS WordPress is wasting resources when using Amazon EC2's VMs. This is because the smallest suitable instance type is not a match for different components of the ConPaaS services. In the case of Kangaroo however, the resources are more efficiently utilized.

In spite of the cost reduction due to more efficient resource utilization, performance is not affected, as the two services target different resources: PHP is a CPU bound service, while MySQL under high load stresses storage and memory. Consolidating services with different resource requirements by means of nesting is directly reflected in the price paid by the end user, opening opportunities for new pricing models for PaaS environments.

3) Disjoint Workloads: Service co-location is not only useful for tightly coupled services, but also in the case of disjoint workloads. We have conducted tests in which a PHP service (CPU bound) is running along with a memcached (memory bound) service, sharing the same first layer VM (t2.medium instance). Both services are tested against heavy load using AB and memslap [34] benchmarks.

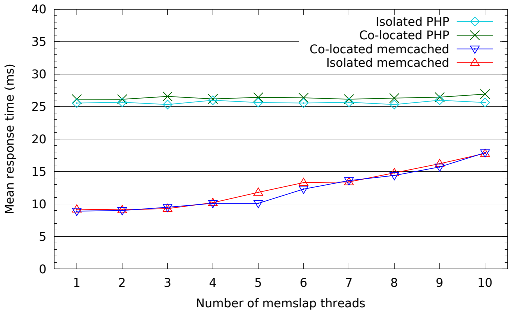

Fig. 7: Benchmarking co-located services with disjoint workloads.

As shown in Figure 7, there is no performance degradation for any of the services as a result of their co-location. The PHP service maintains the mean response time close to 25 ms (for a hundred concurrent requests), even when increasing the concurrency level of the memslap client. At the same time, the linear trend shown by running the memcached service in isolation is not affected by the additional computational load incurred by the PHP service. The mean response time in this benchmark is smaller than the benchmark described in Figure 5 since there is no MySQL service in the loop.

C. Migration

One of the interesting possibilities with Kangaroo is the possibility of migrating tenant's workload between infrastructure VMs without support from the providers. This opens up interesting possibilities in terms of resource management, and cross provider migration. In this section, we first quantify migration times of nested VMs via synthetic microbenchmarks (Section V-C1), and then we explore a scenario where an entire application is live-migrated across providers without any change to the application's code (Section V-C2).

1) Single VM: As described earlier, Kangaroo supports provider-independent live-migration of the nested VMs. We ran a series of microbenchmarks to measure the migration time of a nested VM with 1 GB of memory under different workloads. We used *stress* [35] as a synthetic workload generator inside the nested VM. We varied the amount of consumed memory by *stress* and with and without an I/O bound thread.

Our nested VMs use a 3.2 GB virtual disk. OpenStack offers two options for sharing the disk contents of the migrated VM: 1) block migration (the disk is copied during the migration process), and 2) shared file systems (the images of the guests are accessible for all the compute nodes through a shared file system). We opted for block migration so that the VM is not impacted by the overhead of accessing the image over the network. In addition, OpenStack was configured to utilize qcow2 images, so that the transferred data is limited only to the blocks that were changed during the life time of the nested VM. Finally, we configured Kangaroo to use the QEMU's pre-copy live-migration.

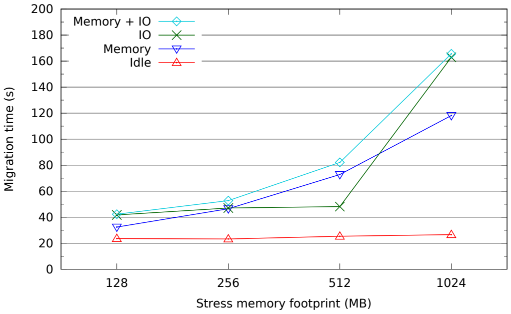

Fig. 8: Migration of a single nested VM under different workloads.

Figure 8 shows the migration time of a nested VM between two VMs running on different DAS4/VU physical hosts. In the case of memory-bound workloads the migration time increases linearly, and peaks in I/O intensive scenarios. In all cases, the nested VM could live-migrate within a few minutes. These numbers can be easily improved by means of more advanced migration facilities such as post-copy migration [13], but they are sufficiently small for our proof-of-concept prototype.

2) Entire Application: One of the important aspects related to migrating an application from one infrastructure provider to another is the downtime encountered by the end users. Currently, the process for performing cross provider migration includes shutting down the application, updating the VM images to make it compatible with the VMM of the new provider, and then starting the application again in the new environment. Along with the downtime incurred by this cold migration process, this approach raises significant technical challenges on the end user side. For example, the end user may need to rewrite parts of her application to make it compatible with the new provider's environment. Kangaroo's unified API, complemented by its support for VM placement, solves the cross provider migration problem and can migrate an entire application from one provider to another without downtime or change to the application code.

We experimented with cross-provider migration by deploying WordPress, a real-world application, and live-migrated it from Amazon EC2 to our local DAS4 OpenNebula provider. The migration was initiated with a single ConPaaS command that identifies the application, and the destination provider.

As mentioned before, WordPress utilizes two of the ConPaaS services, PHP and MySQL. Each service runs with two nested VMs (a manager and an agent) so in total the application used four VMs. The nested VMs were using 1 GB of memory and a single core. Initially the nested guests were running on a single infrastructure VM on Amazon EC2 (t2.medium).

Upon the migration request, ConPaaS started a VM on the OpenNebula deployment on DAS4, and contextualized it as an OpenStack compute node. As soon the VM became online, ConPaaS initiated the nested VMs migration to the new VM.

The total time from the moment the application migration command was issued to the moment when the last nested VM finished its migration was a little under 10 minutes (8m54s, 9m42s, 8m5s for successive executions). Without Kangaroo, migrating a PaaS environment from EC2 to a private OpenStack cloud took about 30 minutes of application downtime and about 45 minutes of total migration time [36], while relying heavily on user intervention for performing all the cold migration steps. Our approach is fully automated and significantly reduces the total migration time.

D. Summary

We evaluated the Kangaroo framework and showed that service co-location and careful management of resources provides up to an order-of-magnitude performance improvement while also reducing the monetary costs due to efficient utilization of infrastructure resources. Further, access to the second-layer VMM enables applications such as ConPaaS to utilize previously unavailable operations like migration. We have successfully live-migrated a WordPress application from Amazon EC2 to our local DAS4 in less than 10 minutes.

VI. Related Work

There has been a number of efforts to create nested infrastructures to complement the properties of current infrastructure providers, which we study in Section VI-A. Containers have also been extensively employed to build better PaaS architectures, which we look at in Section VI-B.

A. Nested Infrastructures

Inception [37] lays the ground for a new infrastructure model where VMs acquired from public providers are used as an infrastructure layer for a second infrastructure middleware. The authors emphasize the reduced constraints on the new (virtual) provider with regard to deploying and running the infrastructure, and the increased flexibility in resource management. Inception proposes a layer-2 network overlay to provide connectivity between the nested VMs. HVX [22] proposes a similar nested model, and uses hSwitch for providing layer-2 connectivity between the nested VMs.

Kangaroo's architecture is similar to both Inception and HVX, but it offers much more: 1) it provides dynamic membership for the infrastructure VMs, making it possible to scale out/in the virtual infrastructure whenever necessary. This has been made possible by IPOP_{sm}'s support for joining/leaving the layer-2 overlay network without any need for (re)configurations, even when infrastructure VMs are behind NAT. 2) Skippy provides a convenient API for programming the virtual infrastructure, paving way for a new era of software-defined infrastructures for the tenants. Further, to the best of our knowledge, this is the first study to extensively evaluate real-world applications on top of these nested infrastructures.

Spilllner et al. analyze the feasibility of using a resource broker in a nested cloud environment in order to optimize resource allocation in relation to the utilization cost [38]. They show that nesting does not degrade the performance of applications considerably, and argue that tenants can resell the extra resources that they do not need to increase their resource utilization and decrease the cost. Kangaroo takes a different approach by allowing the tenants to partition the rented resources according to the needs of their applications.

Nesting has also been employed to improve the security and reliability of infrastructure clouds. CloudVisor [7] builds a thin VMM inside a VM to protect the application workload against breaches in the first level VMM (controlled by the infrastructure provider). RetroVisor [39] replicates user workload inside multiple nested VMs, each running inside a nested but different VMM. In this model, if any of the VMMs is exposed to a vulnerability, it can be seamlessly replaced by another.

B. Container-based platforms

Platform providers have long noticed that the interface exposed by current infrastructure providers does not allow for flexible resource management. Further, handling more than a single infrastructure API results in extra development efforts and makes maintenance difficult. To address these issues, they often rely on containers (e.g., LXC).

Heroku [40] is a platform provider that can be used in order to run web services on top of Amazon EC2. The services run in LXCs, which are managed both by the end user for scaling purposes and by the Heroku engine in order to detect and restart when failed. While similar to Kangaroo, Heroku applications need to be written in certain programming languages, while Kangaroo is an infrastructure provider and agnostic to the type of applications that it runs.

Docker [41] is a deployment tool that eases environment replication using LXCs. A user-space API allows bundling an application and all its dependencies in order to be run in an isolated container on another physical host or VM. Docker is compatible with Kangaroo, and Kangaroo users can use Docker for building the environment for their applications.

Deploying Docker, Heroku, and other similar platform providers (e.g., Cloud Foundry [36], OpenShift [42], etc.) over a cloud infrastructure, and then using these solutions to create services that run in separate containers build up to a nested architecture similar to our solution. Infrastructure providers such as Amazon [43], Microsoft [44], and Google [45] are also beginning to offer native container support over their VMs. However, Kangaroo enables more extensive features such as cross cloud migration, seamless integration of multiple providers, and the ability to allocate resources that match tenants' requirements through tenant-controlled co-location.

VII. Conclusions

Infrastructure clouds revolve around the notion of pay-per-use, where tenants only pay for the resources that they use. In reality however, the providers only allow VM allocation from a fixed number of instance types (e.g., small, medium, large, etc.). This typically creates a mismatch between tenants' requirements and the resource allocation policy, which results in the underutilization of tenant resources. Further, providers do not allow co-location or migration of tenants' VMs that could potentially improve the performance of their applications, and in turn reduce their operation costs.

To address these issues, we designed and implemented a tenant-centric software-defined cloud infrastructure named Kangaroo. Kangaroo is based on running OpenStack on top of VMs acquired from infrastructure providers. According to their application requirements, the tenants can allocate nested VMs from Kangaroo. To provide connectivity between these nested VMs, we designed and implemented $\mathrm{IPOP_{sm}}$, a layer-2 overlay network based on the IPOP project. Through our proposed infrastructure management API, tenants can programmatically scale out/in the size of their virtual infrastructure, define new instance types, and co-locate or migrate their nested VMs within their software-defined virtual infrastructure.

Through deployment of real-world applications on Amazon EC2, we showed that Kangaroo can provide an order of magnitude performance improvements while cutting down the operation cost of the applications by half. We also successfully migrated an entire WordPress application, consisting of several VMs, from Amazon EC2 to our local OpenNebula cloud in under ten minutes using a single command, without downtime, or change to the application code.

Acknowledgments

This work is partially funded by the FP7 Programme of the European Commission in the context of the Contrail project under Grant Agreement FP7-ICT-257438, the HARNESS project under Grant Agreement FP7-ICT-318521, the Dutch public-private research community COMMIT/, and the National Science Foundation under Grants No. 1339737 and 1234983. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors would like to thank Stefania Costache and Ana Oprescu for their valuable feedback on an early draft.

References

[1] "Amazon EC2 pricing: Compute optimized - current generation," http://aws.amazon.com/ec2/pricing, [Online; accessed 26-08-2014].

[2] "Google Compute Engine Live Migration Passes the Test," http://www.rightscale.com/blog/cloud-industry-insights/google-compute-engine-live-migration-passes-test, [Online; accessed 26-08-2014].

[3] "OCCI," http://occi-wg.org/, [Online; accessed 26-08-2014].

[4] "Open Grid Forum," http://ogf.org, [Online; accessed 26-08-2014].

[5] "libcloud," https://libcloud.apache.org/, [Online; accessed 26-08-2014].

[6] "boto," https://boto.readthedocs.org, [Online; accessed 26-08-2014].

[7] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. SOSP*, 2011.

[8] M. Wannous, H. Nakano, and T. Nagai, "Virtualization and Nested Virtualization for Constructing a Reproducible Online Laboratory," in *Proc. EDUCON*, 2012.

[9] D. Williams, H. Jamjoom, and H. Weatherspoon, "The Xen-Blanket: virtualize once, run everywhere," in *Proc. EuroSys*, 2012.

[10] O. Wasserman, "Nested Virtualization: Shadow Turtles," in *KVM forum*, 2013.

[11] S. Yang, "Extending KVM with new Intel Virtualization technology," in *KVM forum*, 2008.

[12] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. NSDI*, 2005.

[13] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proc. VEE*, 2009.

[14] "LXC," https://linuxcontainers.org, [Online; accessed 26-08-2014].

[15] "Control Groups," https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt, [Online; accessed 26-08-2014].

[16] "Linux namespaces," http://lwn.net/Articles/531114, [Online; accessed 26-08-2014].

[17] "SELinux," http://selinuxproject.org, [Online; accessed 26-08-2014].

[18] "The AppArmor security project," http://wiki.apparmor.net/index.php/Main_Page, [Online; accessed 26-08-2014].

[19] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proc. PDP*, 2013.

[20] "CRIU," http://www.criu.org, [Online; accessed 26-08-2014].

[21] "QEMU," http://www.qemu.org, [Online; accessed 26-08-2014].

[22] A. Fishman, M. Rapoport, E. Budilovsky, and I. Eidus, "HVX: Virtualizing the cloud," in *Proc. HotCloud*, 2013.

[23] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The Turtles project: Design and implementation of nested virtualization," in *Proc. OSDI*, 2010.

[24] K. Razavi and T. Kielmann, "Scalable Virtual Machine Deployment Using VM Image Caches," in *Proc. SC*, 2013.

[25] K. Razavi, A. Ion, and T. Kielmann, "Squirrel: Scatter Hoarding VM Image Contents on IaaS Compute Nodes," in *Proc. HPDC*, 2014.

[26] "OpenStack Compute Nodes," http://docs.openstack.org/openstack-ops/content/compute_nodes.html, [Online; accessed 26-08-2014].

[27] "IPOP Project," http://ipop-project.org/, [Online; accessed 26-08-2014].

[28] "OpenStack libcloud plugin," http://libcloud.readthedocs.org/en/latest/compute/drivers/openstack.html, [Online; accessed 26-08-2014].

[29] "OpenStack Compute API," http://docs.openstack.org/api/openstack-compute/2/content/, [Online; accessed 26-08-2014].

[30] G. Pierre and C. Stratan, "ConPaaS: a Platform for Hosting Elastic Cloud Applications," *IEEE Internet Computing*, vol. 16, no. 5, 2012.

[31] "DAS-4 clusters," http://www.cs.vu.nl/das4/clusters.shtml, [Online; accessed 24-01-2014].

[32] "libvirt LXC container driver," http://libvirt.org/drvlxc.html, [Online; accessed 26-08-2014].

[33] "ApacheBench," http://httpd.apache.org/docs/2.2/programs/ab.html, [Online; accessed 26-08-2014].

[34] "Memslap," http://docs.libmemcached.org/bin/memslap.html, [Online; accessed 26-08-2014].

[35] "Stress workload generator," http://people.seas.harvard.edu/~apw/stress/, [Online; accessed 26-08-2014].

[36] "Cloud Foundry Cloud Migration," http://blog.cloudfoundry.org/2014/01/29/migrating-a-cloud-foundry-paas-to-run-on-openstack/, [Online; accessed 26-08-2014].

[37] C. Liu and Y. Mao, "Inception: Towards a nested cloud architecture," in *Proc. HotCloud*, 2013.

[38] J. Spillner, A. Chaichenko, A. Brito, F. Brasileiro, and A. Schill, "Analysis of overhead and profitability in nested cloud environments," in *Proc. LatinCloud*, 2012.

[39] A. Wailly, M. Lacoste, and H. Debar, "RetroVisor: Nested virtualization for multi IaaS VM availability," in *Proc. ComPaS*, 2013.

[40] "Heroku," https://www.heroku.com/, [Online; accessed 26-08-2014].

[41] "Docker," https://www.docker.com/, [Online; accessed 26-08-2014].

[42] "OpenShift," http://www.openshift.com, [Online; accessed 26-08-2014].

[43] "EC2 Container Service (ECS)," https://aws.amazon.com/blogs/aws/cloud-container-management, [Online; accessed 05-01-2015].

[44] "Windows Server containers," http://azure.microsoft.com/blog/2014/10/15/new-windows-server-containers-and-azure-support-for-docker, [Online; accessed 05-01-2015].

[45] "Containers on Google Cloud Platform," https://cloud.google.com/compute/docs/containers, [Online; accessed 05-01-2015].