Scaling VM Deployment in an Open Source Cloud Stack

Kaveh Razavi
VU University Amsterdam
k.razavi@vu.nl

Stefania Costache
VU University Amsterdam
s.v.costache@vu.nl

Andrea Gardiman*
Università di Padova
andrea.gardiman@studenti.unipd.it

Kees Verstoep
VU University Amsterdam
c.verstoep@vu.nl

Thilo Kielmann
VU University Amsterdam
thilo.kielmann@vu.nl

ABSTRACT

Interactive High Performance Computing (HPC) workloads take advantage of the elasticity of clouds to scale their computation based on user demand by dynamically provisioning virtual machines during their runtime. As in this case users require the results of their computation in a short time, the time to start the provisioned virtual instances becomes crucial. In this paper we study the deployment scalability of OpenNebula, an open-source cloud stack, with respect to these workloads. We describe our efforts for tuning the infrastructure's and OpenNebula's configuration as well as solving scalability issues in its implementation. After tuning both infrastructure and cloud stack, the simultaneous deployment of 512 VMs improved by 5.9× on average, from 615 to 104 seconds, and after optimizing the implementation, the deployment time improved by 12× on average, to 53.54 seconds. These results suggest two possible improvement opportunities that can be useful for both cloud developers and scientific users deploying a cloud stack to avoid such scalability issues in the future. First, the tuning process of a cloud stack can be improved through automatic tools that adapt the configuration to the workload and infrastructure characteristics. Second, the code scalability issues can be avoided through a testing infrastructure that supports large scale emulation.

Keywords

Infrastructure-as-a-Service, Scalability

Categories and Subject Descriptors

C.0 [**General**]: System architectures;
C.4 [**Performance of systems**]: Design studies

*Work done while visiting VU University Amsterdam.

ScienceCloud'15, June 15 2015, Portland, OR, USA

DOI: http://dx.doi.org/10.1145/2755644.2755645

1. INTRODUCTION

Interactive HPC workloads become more common, especially with the advent of cloud infrastructures. These workloads involve large amounts of computation that need to be completed in a short amount of time. Observing the results of the computation, scientists submit new batches of computation. Bags of parameter sweep tasks [18], many task computing applications [22], or web portals that process interactive user requests requiring complex data analysis [28] are examples of these workloads. Infrastructure-as-a-Service clouds are a key platform in running these workloads, due to their elastic provisioning model, which can guarantee fast and stable response times.

The promise of elastic computing is instantaneous creation of virtual machines (VMs), according to the needs of an application or web service. In practice, however, VM startup times are in the range of several minutes, along with high variability, depending on the actual system load. Two major factors contribute to the VM startup times: the scalability of the cloud middleware (Amazon EC2, OpenNebula, OpenStack, etc.) and the actual VM boot time, including the transfer of the VM image to the selected compute node. State-of-the-art solutions tackle both problems through a variety of architectural and algorithmic approaches [4, 20, 24]. In this paper, we focus on the first problem, with a different approach from the state of the art. Instead of improving on the design of cloud stacks, we study their existing scalability. As a representative example, we chose OpenNebula [12], a widely-used open source stack. Our choice was motivated by the simplicity of its design and configuration.

Our contribution is a study of potential bottlenecks in the VM deployment time, which can come from the configuration of the infrastructure, of the cloud stack, or its implementation. During our experiments, we focused on understanding and optimizing the problematic steps taken by the cloud stack. Our process was iterative, as some optimizations led us to identify other issues in the infrastructure or other components of the cloud stack. Our results show that major improvements can be obtained in the VM deployment time. By tuning the infrastructure's and OpenNebula's configuration, we improved the VM deployment time by 5.9× on average, from 615 to 104 seconds, and through code optimizations, we could further improve the deployment time by 12× on average, to 53.54 seconds.

We draw two opportunities for improvement that can be useful for both cloud developers and users. We learned that an infrastructure administrator has to follow a long iterative process when tuning the infrastructure and the cloud stack.

Moreover, it is desirable to perform such tuning based on workload characteristics. For example, in our case, to reduce the VM deployment time, we tuned the VM scheduling interval, the VM image storage and the number of parallel commands executed by OpenNebula. This outlines the need for tools to provide an automatic tuning process, with limited involvement from the administrator side. However, even when the infrastructure and the cloud stack configurations are optimized, bottlenecks can appear in the different software components. We learned about such bottlenecks only when performing large-scale deployments of the cloud stack. Nevertheless, this is mostly the case of using the cloud stack in production, and not during its early development. Thus, we argue that better testing tools, which emulate large-scale deployments, could potentially avoid such bottlenecks. These tools could improve the experience of both developers and administrators deploying the cloud stack.

This paper is organized as follows. Section 2 motivates our work and describes the process of deploying VMs in OpenNebula. Section 3 details our experiments, and Section 4 describes the improvement opportunities. We discuss related work in Section 5, and we conclude in Section 6.

2. BACKGROUND

We detail in this section the motivation of our work and give an overview of our investigated cloud stack. We then describe the steps that we followed in our experiments.

2.1 Motivation

Due to their promised elasticity, cloud computing platforms are appealing for scientists wanting to run interactive HPC workloads, which require large amounts of compute resources. These workloads are representative for different scientific computing areas, e.g., bioinformatics, astronomy, or geographic information sciences. Let's take for example, a scientist who wants to simulate small angle scattering (SANS) techniques [7] to classify the shape of a molecule. The normal process in this case involves different runs of the simulation, with the scientist checking the simulation results and changing the input parameters after each run. The choices of parameter ranges with the resolutions of the parameters is commonly referred to as a parameter sweep. A parameter sweep is hence composed of a large number (up to tens of thousands) of tasks, which, if run on the proper number of resources, will produce the results in a small amount of time, e.g., 5 minutes. To run these tasks as fast as possible, the scientists can benefit from a cloud infrastructure, on which a large number of VMs can be "instantly" deployed. A more general example is the case of scientific web portals [28], which receive bursts of requests from users who need interactive response times to understand the effects of their simulation parameters and change them in real time. To cope with the burst in demand, such applications use auto-scaling mechanisms provided by the cloud infrastructure.

Needless to say, in these cases the time to deploy the VMs on the cloud infrastructure has an important role as it can lead to variable and large response time for users. However, in reality, cloud providers overlook the importance of the VM deployment time, which can be in the order of minutes, or even tens of minutes, with a high variability. To show how much time it actually takes to start a batch of VMs, we installed OpenNebula [11], an open source cloud stack, on our private infrastructure using an "out-of-the-box" configuration, i.e., we used the default configuration and we only configured the network and the VM image storage so that it can run successfully on our cluster. We chose OpenNebula due to its simple and flexible design, and also because it is widely used both in industry and in academic research. We considered the scenario of the scientist who runs the SANS parameter sweep; thus, in this case, running the application on a large number of VMs should considerably speed up its execution time. Figure 1 (left) shows the deployment times of 512 VMs, for which the requests were submitted simultaneously to OpenNebula. We noticed that the average deployment time per VM was approximately 10 minutes, while the maximum deployment time (the 100th percentile) was approximately 22 minutes. Thus, in reality the time the user has to wait for the VMs to start running on the infrastructure would be much higher than the makespan of her application.

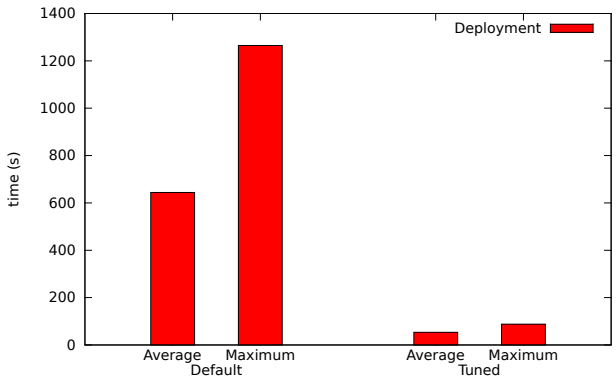


Figure 1: VM deployment performance with the "out-of-the-box" OpenNebula configuration and the tuned one.

This issue motivated us to investigate the real causes of overhead by analyzing the VM deployment workflow in OpenNebula. Our findings consist of multiple optimization rounds over the configuration and implementation of the cloud stack. The results of our improvements can be seen in Figure 1 (right). Although none of our improvements fundamentally change the design of OpenNebula, the VM deployment times reduced with an order of magnitude: the maximum deployment time dropped from 22 minutes to 90 seconds. We describe in more detail the issues we found and the optimizations that we performed in the following sections.

To give a better understanding of our results, we describe next the design of the analyzed cloud stack and our employed methodology.

2.2 OpenNebula

To understand what are the potential scalability overheads when deploying VMs, we analyzed the process performed by OpenNebula. Although other open source cloud stacks exist, like OpenStack [6], Eucalyptus [14], or CloudStack [2], OpenNebula's simple design made it the most suitable for our purposes. The architecture of OpenNebula is modular and extensible. A daemon, which resides on the infrastructure's head node, keeps infrastructure bookkeeping information and manages all operations regarding VMs, users, network, and storage. The daemon uses an XML-RPC server to receive the commands related to infrastruc-

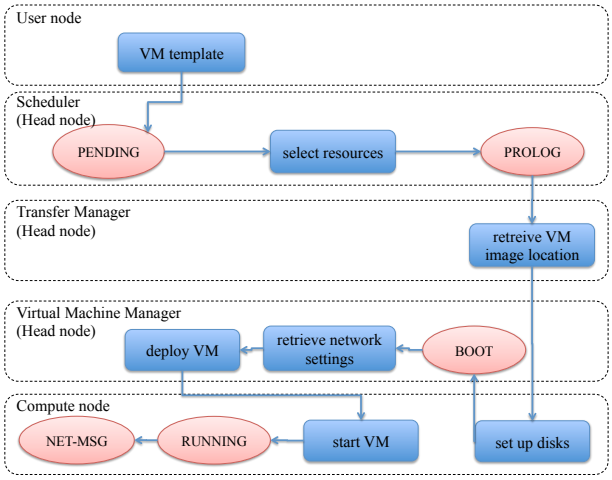


Figure 2: VM deployment process in OpenNebula. The states of a VM are depicted in red and the actions taken by various components in blue.

ture and VM management. Users submit requests to deploy and manage VMs to OpenNebula through either standardized interfaces, like Amazon EC2's API, or through provided CLI and web clients. On the infrastructure's hosts, OpenNebula requires deploying a set of scripts, called remotely by the head node through a set of components named *drivers*. Each driver has a specific function, e.g., to monitor the VMs, to transfer the VM image, or to start and manage the VM on the designated host. The placement of VMs on the hosts is decided by a scheduler, started by default on the head node.

2.2.1 *VM deployment process*

Figure 2 gives an overview of the process required to deploy a VM. The main stages and components involved in this process are summarized in Table 1. With the CLI client, the user has to submit a template to the OpenNebula head node, containing the desired configuration of the VM, e.g., VM disk image, resource and network configuration, scripts to run at boot time, etc. The OpenNebula's scheduler sets the VM in a **PENDING** state and puts it in a queue. The scheduler periodically applies an algorithm to select the hosts on which the pending VMs are placed. The scheduler places multiple VMs on the same host, if enough resources are available. If the scheduler cannot find any suitable host, the VMs are kept in the pending queue.

After a suitable host is selected, the OpenNebula daemon handles the VM's deployment. First, the VM is set in a **PROLOG** state and the VM's disk image is copied or configured on the host. The disk image transfer is managed by a component of the head node daemon, called Transfer Manager (TM). To allow its users to customize the Transfer Manager with different transfer and data storage mechanisms, the design of OpenNebula delegates the execution of the transfer commands to a Transfer Manager driver, which can execute user-defined scripts.

The last part of the boot process happens on the host. The VM is set in a **BOOT** state, network interfaces are set up and the VM is booted by the host's hypervisor. The network setup and VM boot commands are invoked on the hosts by the Virtual Machine Manager (VMM), which is also

Table 1: OpenNebula's components involved in VM deployment.

VM state	OpenNebula Component
Pending	Scheduler
Prolog	TM, TM driver
Boot	VMM, VMM driver

a part of the head node daemon. Like the Transfer Manager, this component is also higly customizable; a Virtual Machine Manager driver can be configured to execute scripts specific to the hypervisor type installed on the hosts. When the VM is started, OpenNebula sets the VM's state to **RUNNING**.

2.2.2 *OpenNebula drivers*

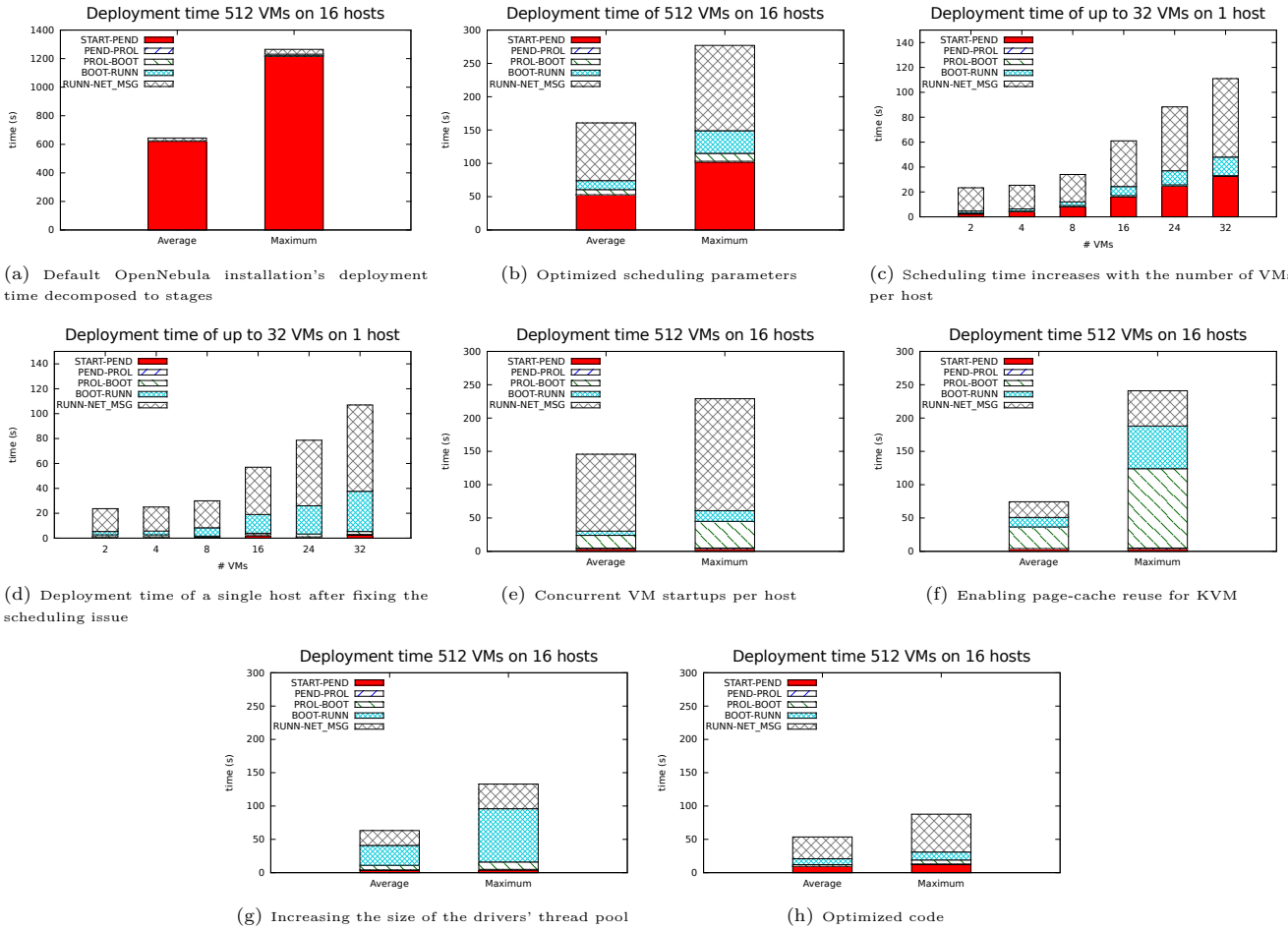
We used two specific drivers of OpenNebula: the QCOW2 driver [10] for the TM, and the KVM driver for the VMM. For a faster deployment, we store the VM disk images in a QCOW2 format accessed through a remote file system. When the user starts a VM, a new copy of the original disk image, initially empty, is created. For read operations, this new image acts as a bypass for the original one: when a data block is read by the application running in the VM, if the block does not already exist in the image, it is fetched from the original image over the network. Write operations are written to the new image, leaving the original remote image unmodified. When a VM disk image needs to be deployed, the TM driver simply invokes the command to create the QCOW2 image on the corresponding host.

2.3 Methodology

We split the deployment in multiple stages to analyze their execution time. We define these stages and measure the time interval for each of them as follows:

- **START-PENDING**: the time interval between the user's submission of the VM and the time at which OpenNebula registers it in the scheduler queue.
- **PENDING-PROLOG**: the time interval required by the scheduler to decide the placement of the VM and send the deployment commands to the head node.
- **PROLOG-BOOT**: the time interval needed by the TM to set up the VM's disk image at the host.
- **BOOT-RUNNING**: the time required by the VMM to start the VM on the host.
- **RUNNING-NET-MSG**: we define this last inteval as the time required by the operating system running in the VM to finish its initialization. We note here that OpenNebula sets the VM in a **RUNNING** state when the operating system of the VM is still initializing. However, the user cannot connect to the VM and start her application at this point. Thus, to measure the exact time when a VM finishes its boot process, we configure each VM to send a network message to the head node once its operating system is initialized. When this message is received, the VM is set in a special state, called **NET-MSG** in Figure 2.

We measured these time intervals in a scenario in which we submit a request to deploy 512 VMs *simultaneously* to OpenNebula. We used the *same* VM disk image for all of the VMs, which fits the scenario of the scientist running an interactive HPC workload. In this case, the VMs running the



(a) Default OpenNebula installation's deployment time decomposed to stages

(b) Optimized scheduling parameters

(c) Scheduling time increases with the number of VMs per host

(d) Deployment time of a single host after fixing the scheduling issue

(e) Concurrent VM startups per host

(f) Enabling page-cache reuse for KVM

(g) Increasing the size of the drivers' thread pool

(h) Optimized code

Figure 3: The process of tuning OpenNebula for large-scale deployments. Note the different scale of the vertical axis on figures a, c, and d.

application often have the same application code installed. During our experiments, we focused on understanding and optimizing the most problematic VM deployment stages. We noticed that different changes in the system's and cloud stack's configuration lead to reducing the time interval of a particular stage, but at the cost of increasing the time of other stages. Thus, we repeated this process iteratively until we eliminated all the potential problems, while avoiding modifications to the architecture of OpenNebula.

3. EXPERIMENTS

We describe the process that we undertook for tuning VM deployment in OpenNebula 4.4.1 (released on 4 Feb, 2014 [16]). Initially, we tried to reduce the VM deployment time by changing the configuration, but after a number of iterations, it was clear that we also need to resolve some of the scalability issues in the OpenNebula's code.

For our experiments, we used 16 compute nodes from DAS4/LU cluster [3] as VM hosts, and the cluster's head node for running the OpenNebula daemons. The nodes are equipped with dual quad-core Intel E5620 CPUs, running at 2.4 GHz, with 24 GB of memory. The nodes are connected using a commodity 1 Gb/s Ethernet switch and a QDR InfiniBand providing a theoretical peak of 32 Gb/s. We used the InfiniBand network for our experiments to avoid the network bottlenecks. In situations where premium networks are not available, low-overhead caching techniques [24, 23] avoid the network bottlenecks associated with VM deployment. The head node runs an off-the-shelf NFS server optimized for read and write requests at QCOW2's cluster size [10], which stores the VM disk images. The guest VMs run Debian 7.4 as their operating system.

3.1 Scheduler's configuration

Figure 3-a shows the time decomposition of the 512 VM deployments that we showed earlier in Section 2.1 in the stages that we discussed in Section 2.2. We notice that the longest stage is scheduling the VMs. Hence, our first step was to change the configuration parameters of the OpenNebula scheduler. We reduced the scheduling interval from 30 seconds to the minimum possible value of 1 second, and we increased the number of VMs dispatched in each interval to 1024, both per host and in total.

Figure 3-b shows the effects of changing these parameters on the deployment time. While this simple modification improves the scheduling time considerably, still a significant portion of the deployment time is spent in the scheduling stage.

To investigate the issue, we removed all the hosts except one from our OpenNebula setup. Figure 3-c shows that

the deployment time increases linearly with the number of VMs *per host*. This turned out to be due to the fact that the OpenNebula scheduler ignored the configuration variable that specifies the number of VMs dispatched per host in each scheduling interval. Fixing the problem, Figure 3-d now shows improved scheduling times compared to Figure 3-c. Scaling back to 16 hosts, as shown in Figure 3-e, 512 VMs now start on average in 145.6 seconds (4.4× improvement), and the slowest one starts in 229 seconds (5.5× improvement).

3.2 Host-side caching

After improving the scheduler, we notice that most of the deployment time belongs to the actual VM starting time which involves the booting of the guest operating system. Note that the operating system reads data blocks on demand from its VM image hosted on the head node. In our scenario of an interactive HPC user, the VM image is the same for all the deployed VMs, and hence it should be cached and reused by all the VMs that are co-running on their host.

Unfortunately, we noticed that the default OpenNebula template for VM deployment uses **DIRECT_IO**, which prohibits the NFS clients to cache the VM image data. As a result, although started from the same disk image, each VM was reading its entire *boot reading set* rather than reusing that of others. To remedy this, we enabled the default caching policy for KVM disks. This reduced the NFS traffic at the head node (storage site) to the expected amount. Figure 3-f shows the effects of caching. We can make two interesting observations: 1) The average deployment is now 74.3 seconds and has improved by almost a factor of two, but the maximum deployment time is now 241 seconds and has degraded by 12 seconds. 2) The stage that takes the longest is now the PROLOG-BOOT followed by BOOT-RUNNING, suggesting that the scalability bottleneck has now shifted to a different location.

3.3 Execution parallelism

According to Table 1, during the BOOT-RUNNING phase the TM driver is active, and during the PROLOG-BOOT phase the VMM driver is active. To improve the performance of these stages, we boosted their concurrency by increasing their number of active threads, from 15 to 128 for the TM driver and from 10 to 32 for the VMM driver respectively. Curiously, increasing the number of VMM driver threads to more than 32 does not improve the performance further. This turned out to be due to a locking issue in the VMM driver which we discuss in Section 3.4.1.

Figure 3-g shows the improvements due to this change. We can see that with a simple modification of the configuration by fixing a simple scheduling issue, the deployment time of 512 VMs is reduced on average from 74.3 to 63.2 seconds, and for the last VM from 241 to 133 seconds.

3.4 OpenNebula's code tuning

Exhausting all the options for optimizing the configuration of OpenNebula, we looked for possible sources of overhead in its source code. Here, we give a brief overview of them before showing the results of our optimizations.

3.4.1 Unnecessary locks in the VMM driver

The VMM driver is the entity that handles VM operations on the target host. For the actual commands, executed with SSH, it uses one of its configured threads. We noticed that there is a lock *per host* that needs to be acquired before processing a new VM. This lock essentially serializes the VM commands assigned to the same host.

This serialization is not protecting any critical state in the OpenNebula core, and is most likely implemented to protect against concurrency problems in the virtualization toolkit that runs on the hosts. In our setting, i.e. libvirt [9] and KVM, this is not an issue, so we can safely remove this lock allowing VMs to be concurrently deployed on each host. Note that libvirt on top of KVM (or Xen) is a commonly used virtualization stack.

3.4.2 Slow vertical parallelism

OpenNebula follows a simple producer/consumer design pattern for communication between each of its components. We roughly describe it here: each component (e.g., VMM driver, transfer manager driver, etc.) has a single thread that takes an action object (i.e., a deployment request) from a queue (i.e., taking a consumer role), does the necessary work on the object, and passes the object on to the next queue (i.e., taking a producer role).

This vertical design synchronizes the actions at the queues between each component, and hence, the action throughput of the system is determined by the slowest component. To remedy this, and given that the actions do not share state between each other, we decided to follow a horizontal threading pattern in which each *action* is handled by one thread throughout its lifetime. This avoids queueing of the action objects at the cost of extra threading, which we offset by using a thread-pool.

3.4.3 Expensive SSH connections

OpenNebula uses SSH connections for executing actions on the VM hosts. For each deployment, there are two actions executed on the host: preparing the VM image (TM driver), and starting the VM (VMM driver). This means that for the deployment of 512 VMs, the head node needs to create 1024 processes that create SSH connections to the hosts in a relatively short amount of time.

This is too expensive and slows down the head node. Instead, we use persistent TCP sockets for communicating the commands to the VM hosts. The security of these connections can easily be ensured using SSL, which we have left for future work.

3.4.4 Serial deployments

For our experiments, we use the **onevm** [17] command provided by the CLI client of OpenNebula. The command receives as arguments a VM deployment template as well as the number of VMs to instantiate from the template. We noticed that **onevm** sends the deployment requests serially using the RPC multicall API of OpenNebula daemon. To avoid slow RPC requests/responses that execute serially, we changed the interface to allow for multiple deployment requests to be sent with a single RPC call.

3.4.5 The effects of OpenNebula's code tuning

Figure 3-h shows the difference in deployment time of 512 VMs after our detailed code optimizations. The deployment time of the VMs has improved on average from 63.2 seconds to 53.54 seconds, and the last VM now starts in 87.8 seconds instead of 133 seconds. At this point, the largest part of the

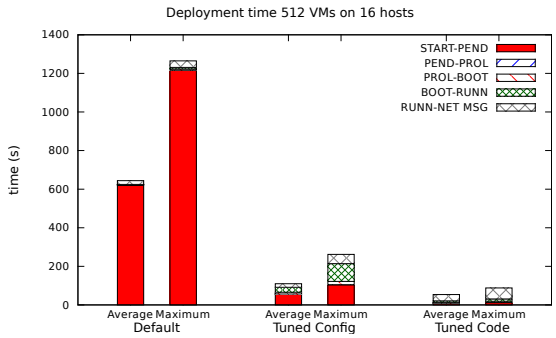


Figure 4: An overview of changes in VM deployment time after tuning the OpenNebula's configuration and fixing bottlenecks in its code. Note how the VM deployment time reduced from an average of 615 seconds to 53.54.

total VM deployment time is the booting of the guest OS that can be further improved as suggested in [24, 26].

3.5 Summary

We described the tuning of OpenNebula for large-scale deployment of VMs. We started first by tuning the configuration parameters, and then looked at the scalability issues of the code. Figure 4 shows the improvements that we achieved by tuning the configuration parameters, and the results of optimizing OpenNebula's code. To summarize, compared to the default installation, optimizing only the configuration improves the average by 5.9× and the maximum case by 4.8×. Optimizing the code, the average improves by 12× and the maximum case by 14.4×.

4. IMPROVEMENT OPPORTUNITIES

We previously discussed our efforts to overcome the scalability issues faced when deploying a large number of VMs on top of a default OpenNebula installation. These efforts revolved around tuning OpenNebula's configuration parameters, tuning the compute hosts, as well as relaxing the locking semantics and parallelizing parts of OpenNebula's code. From these efforts we outline two opportunities for improvement, which can improve the practices of cloud stack development and management. We describe them next.

4.1 Auto-tuning for cloud software stacks

The centralized design of OpenNebula leads to a fairly simple configuration, compared to other existing cloud software stacks, composed of multiple services, each with its own configuration to tune. But even then, we learned from our experience that manual tuning of the configuration parameters, and the infrastructure itself, for large-scale VM deployment is a tedious and time-consuming process. Worse, some of the optimizations can only be applied *after* scalability bottlenecks show up in production. Although popular cloud stacks have their well-established communities and a large amount of information can be found regarding their configurations, one still needs to spend a large amount of time to deal with all their intricacies.

We argue here that the tuning of open source cloud software stacks can be automated to a certain degree through the design of tools that have the in-depth knowledge of configuration parameters and can set these parameters according to the properties of the infrastructure and the intended workload. The administrator involvement in this case would be to pass information to such tools regarding the infrastructure and the workload. For example, the administrator could have some estimates regarding the average number of VM deployments that happen simultaneously, or how many VMs are active on average in the infrastructure. Using this high-level information, the auto-tuning tools would configure the parameters of the cloud software stack to their optimal values. The scheduling period could be decreased, or the maximum number of requests processed in parallel by the cloud stack could be increased. If the number of users of VM images is not expected to grow then the VM images can be fully cached on the hosts reducing network traffic. If VM migration is common then some network bandwidth should be reserved, and so on.

Some tools already exist for tuning the configuration of some cloud stacks [5], but they only focus on user and host management and provide simple parameter configuration, like asking the administrator in what folder the VM images should be stored. These automated management tools can be a basis for developing the more elaborate auto-tuning mechanisms that consider both the details of the infrastructure and the workload.

4.2 Scalability testing suites

Although tuning the cloud software stack and the infrastructure had a large impact on the VM deployment time, in our experiments we also noticed possible improvements in the implementation. These improvements are not fundamental, as they do not change the architecture of the cloud stack, but they are incremental improvements on the scalability of the code. Nevertheless, we showed that these changes improved the performance of VM deployment in our scenario on average by 2× and for the last VM by 3×.

The need for such changes stresses the importance of thorough testing infrastructures, designed to stress each component of the cloud stack at large scale. Because existing open source cloud stacks were developed and tested on small scale testbeds, some issues were overlooked. These issues only appear at scale during large deployments that generate bursts of requests. For example, in our case, we did not notice the poorly designed locking and communication between the components until we deployed a large number of VMs simultaneously and fixed all the configuration issues that delayed the processing of the requests by OpenNebula.

Therefore, we recommend that the cloud stack developers employ scalability testing suites that run, e.g., nightly, to avoid introducing changes that may have adverse effects on scalability. Unfortunately, testing large-scale deployments requires a large infrastructure that may not be available to open source developers. If that is the case, dummy adapters that emulate a cloud infrastructure can be used for running the scalability testing suites.

5. RELATED WORK

We classify the research work related to scalable VM deployment in three main categories, which we discuss next.

5.1 Scalable cloud architectures

Several works focused on the scalability of the cloud stack architecture. Although in our study we chose OpenNebula, which has a centralized design, there are a number of projects that improve the infrastructure services through hi-

erarchical or fully decentralized architectures. OpenStack [6] has a loose component-based design. Different services manage different infrastructure components, e.g., data storage, networking, user authentication, and communicate through scalable message passing queues. The services can run on different nodes to provide scalability and fault tolerance. Eucalyptus [14] and Snooze [4] have a hierarchical design. The hosts are organized in groups managed by local controllers while a central controller monitors the local controllers and delegates requests to them. DVMS [20] is completely decentralized as it relies on a peer to peer overlay for large-scale VM management.

In this paper we do not focus on the architecture optimizations. To the best of our knowledge, we show for the first time how to tune an existing cloud stack for large-scale VM deployment. Tuning a cloud stack, one with an already simple centralized design such as OpenNebula, can be very challenging. Decentralizing the architecture, while having certain advantages, introduces additional complexity that will make tuning more difficult. We are currently looking into scalability problems of VM deployment in OpenStack.

5.2 Scalable transfer/caching of VM images

There is a large body of work that recognizes network and/or storage as the main source of bottlenecks in deploying massive number of VMs due to the transfer of VM images. Techniques for scalable transfer or storage of the VM images can help eliminating these bottlenecks.

Peer-to-peer networking is a common technique for transferring VM images to many compute nodes [1, 15, 27, 31]. In these systems, peers fetch most of the VM image blocks from each other rather than from centralized storage servers, relieving the network bottleneck.

IP multicasting uses the parallelism available in the switch, and has been used previously for scalable delivery of VM image contents to the hosts during VM deployment [29, 30].

To improve VM startup, other systems cache the VM image contents, deduplicated or otherwise, on each host. Liquid [33] and similar systems (e.g., [13, 19, 32]) are designed for scalable VM image distribution. Squirrel's cVolumes [23] persistently cache all the blocks needed for starting different VMs by capturing their boot working sets [24] in a compressed ZFS file system. µVM/Squirrel [26] caches minimal memory snapshots and employs fast VM resume, and resource hot-plugging for instant and scalable VM startup.

Opposed to these works, in this paper, we focus on the scalability of the cloud stack. Our optimized cloud stack can be used orthogonal to these approaches to improve the deployment time of the VMs.

5.3 VM image consolidation

Reducing the VM image size results in faster deployment time due to the reduction of network transfers.

VMPlant [8] is a service for generating custom VM images. The user provides VMPlant with 1) machine requirements (e.g. OS, size of memory and disk, etc.), and 2) a set of configuration scripts and their dependencies in the form of a directed acyclic graph (DAG). VMPlant then generates a new VM image based on previously cached VMIs.

Quinton et al. [21] use package dependency information to install the minimum number of packages for a given application. To minimize the disk size, they estimate the installation size of each package. [25] follows a similar approach, but it uses a fresh VM image for copying the packages, resulting in clean VM images without a need for size estimations.

6. CONCLUSIONS

Infrastructure-as-a-Service clouds are a key platform in running a growing number of interactive HPC applications, which often need large amounts of compute and data storage resources. Given that these workloads require fast access to VMs, cloud providers often overlook the importance of VM deployment time. An important factor in the scalability of VM deployment is the design of the cloud software stack and a correct infrastructure configuration.

In this paper, we described the process that we followed for tuning a popular, open source cloud stack, tuning the infrastructure, and resolving the scalability bottlenecks in the implementation of the cloud stack. We showed that this process plays an important role in improving the scalability of VM deployment, leading to improvements in VM deployment times with an order of magnitude. Starting from a default OpenNebula installation, on which deploying 512 VMs takes 615 seconds on average, our improvements lead to a deployment time of 53.54 seconds. This process led us to identify two main problems. First, the manual tuning of the cloud software stack by an administrator is time-consuming and error-prone. Second, cloud software stack development often does not cover scalability corner cases.

These findings raise the need of automated tuning of the cloud stack to the infrastructure and the workload deployment and management software, as well as testing suites tailored for scalability. In the absence of a large-scale testing infrastructure, large deployment emulation can help identify the majority of the implementation scalability issues.

Acknowledgments

This work is partially funded by the Dutch public-private research community COMMIT/.

7. REFERENCES

[1] Z. Chen, Y. Zhao, X. Miao, Y. Chen, and Q. Wang. Rapid Provisioning of Cloud Infrastructure Leveraging Peer-to-Peer Networks. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, ICDCSW '09, pages 324–329, 2009.

[2] CloudStack. http://cloudstack.apache.org. [Online; accessed 19-02-2015].

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

[4] E. Feller, L. Rilling, and C. Morin. Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12, pages 482–489, 2012.

[5] Fuel. https://wiki.openstack.org/wiki/fuel. [Online; accessed 19-02-2015].

[6] K. Jackson. *OpenStack Cloud Computing Cookbook*. Packt Publishing, 2012.

[7] J. P. Koskinen and L. Holm. Sans: high-throughput retrieval of protein sequences allowing 50%

mismatches. *Journal of Bioinformatics*, 28(18):i438–i443, 2012.

[8] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *2004 ACM/IEEE Conference on Supercomputing*, SC '04, 2004.

[9] libvirt Virtualization API. `http://libvirt.org`. [Online; accessed 19-02-2015].

[10] M. McLoughlin. The QCOW2 Image Format. `http://people.gnome.org/~markmc/qcow-image-format.html`. [Online; accessed 24-01-2014].

[11] D. Milojicic, I. M. Llorente, and R. S. Montero. OpenNebula: A Cloud Management Tool. *IEEE Internet Computing*, 15(2):11–14, 2011.

[12] D. Milojičić, I. Llorente, and R. S. Montero. OpenNebula: A Cloud Management Tool. *IEEE Internet Computing*, 15(2):11–14, 2011.

[13] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu. Going Back and Forth: Efficient Multideployment and Multisnapshotting on Clouds. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC '11)*, pages 147–158, 2011.

[14] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, and L. Y. D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *In the Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 124–131, 2009.

[15] C. M. O'Donnell. Using BitTorrent to distribute virtual machine images for classes. In *Proceedings of the 36th annual ACM SIGUCCS fall conference: moving mountains, blazing trails*, SIGUCCS '08, pages 287–290, 2008.

[16] OpenNebula Release Cycle. `http://opennebula.org/software/release`. [Online; accessed 19-02-2015].

[17] onevm(1) – manages OpenNebula virtual machines. `http://archives.opennebula.org/doc/4.0/cli/onevm.1.html`. [Online; accessed 19-02-2015].

[18] A.-M. Oprescu and T. Kielmann. Bag-of-Tasks Scheduling under Budget Constraints. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CloudCom '10, pages 351–359, 2010.

[19] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual machine image distribution network for cloud data centers. In *29th Conference on Computer Communications*, INFOCOM '10, pages 181–189, 2012.

[20] F. Quesnel, A. Lèbre, and M. Südholt. Cooperative and Reactive Scheduling in Large-Scale Virtualized Platforms with DVMS. *Concurrency and Computation: Practice and Experience*, 2012.

[21] C. Quinton, R. Rouvoy, and L. Duchien. Leveraging feature models to configure virtual appliances. In *2nd International Workshop on Cloud Computing Platforms*, CloudCP '12, 2012.

[22] I. Raicu, Y. Zhao, and I. Foster. Many-task computing for grids and supercomputers. In *1st IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '08, 2008.

[23] K. Razavi, A. Ion, and T. Kielmann. Squirrel: Scatter Hoarding VM Image Contents on IaaS Compute Nodes. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 265–278, 2014.

[24] K. Razavi and T. Kielmann. Scalable Virtual Machine Deployment Using VM Image Caches. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, number 65 in SC '13, 2013.

[25] K. Razavi, L. M. Razorea, and T. Kielmann. Reducing VM Startup Time and Storage Costs by VM Image Content Consolidation. In *1st Workshop on Dependability and Interoperability In Heterogeneous Clouds*, Euro-Par 2013: Parallel Processing Workshops, 2013.

[26] K. Razavi, G. van der Kolk, and T. Kielmann. Prebaked uVMs: Scalable, Instant VM Startup for IaaS Clouds. In *Proceedings of the 35th International Conference on Distributed Computing Systems*, ICDCS '15, 2015.

[27] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein. VMTorrent: scalable P2P virtual machine streaming. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 289–300, 2012.

[28] P. Riteau, M. Hwang, A. Padmanabhan, Y. Gao, Y. Liu, K. Keahey, and S. Wang. A cloud computing approach to on-demand and scalable cybergis analytics. In *Proceedings of the 5th ACM Workshop on Scientific Cloud Computing*, ScienceCloud '14, pages 17–24, New York, NY, USA, 2014. ACM.

[29] M. Schmidt, N. Fallenbeck, M. Smith, and B. Freisleben. Efficient Distribution of Virtual Machines for Cloud Computing. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, PDP '10, pages 567–574, 2010.

[30] B. Sotomayor, K. Keahey, and I. Foster. Combining Batch Execution and Leasing Using Virtual Machines. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 87–96, 2008.

[31] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath. Image Distribution Mechanisms in Large Scale Cloud Providers. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CloudCom '10, pages 112–117, 2010.

[32] Z. Zhang, Z. Li, K. Wu, D. Li, H. Li, Y. Peng, and X. Lu. VMThunder: Fast Provisioning of Large-Scale Virtual Machine Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2014.

[33] X. Zhao, Y. Zhang, Y. Wu, K. Chen, J. Jiang, and K. Li. Liquid: A Scalable Deduplication File System for Virtual Machine Images. *IEEE Transaction on Parallel and Distributed Systems*, 2013.