Abstract Image Management and Universal Image Registration for Cloud and HPC Infrastructures

Javier Diaz, Gregor von Laszewski, Fugang Wang and Geoffrey Fox
Pervasive Technology Institute, Indiana University
2729 E 10th St., Bloomington, IN 47408, U.S.A.
Email: javidiaz@indiana.edu, laszewski@gmail.com, fuwang@indiana.edu and gcf@indiana.edu

***Abstract*—Cloud computing has become an important driver for delivering infrastructure as a service (IaaS) to users with on-demand requests for customized environments and sophisticated software stacks. Within the FutureGrid (FG) project, we offer different IaaS frameworks as well as high performance computing infrastructures by allowing users to explore them as part of the FG testbed. To ease the use of these infrastructures, as part of performance experiments, we have designed an image management framework, which allows us to create user defined software stacks based on abstract image management and uniform image registration. Consequently, users can create their own customized environments very easily. The complex processes of the underlying infrastructures are managed by our sophisticated software tools and services. Besides being able to manage images for IaaS frameworks, we also allow the registration and deployment of images onto bare-metal by the user. This level of functionality is typically not offered in a HPC (high performance computing) infrastructure. However, our approach provides users with the ability to create their own environments changing the paradigm of administrator-controlled dynamic provisioning to *user-controlled dynamic provisioning*, which we also call raining. Thus, users obtain access to a testbed with the ability to manage state-of-the-art software stacks that would otherwise not be supported in typical compute centers. Security is also considered by vetting images before they are registered in a infrastructure. In this paper, we present the design of our image management framework and evaluate two of its major components. This includes the image creation and image registration. Our design and implementation can support the current FG user community interested in such capabilities.**

***Keywords*-Image Management; Image Repository; Dynamic Provisioning; Rain; FutureGrid**

I. Introduction

FutureGrid (FG) [1] is a testbed providing users with grid, cloud, and high performance computing infrastructures. FG employs both virtualized and non-virtualized infrastructures. The testbed is composed of a high-speed network connected to distributed clusters of high-performance computers. This innovative infrastructure can support state-of-the-art research in distributed and parallel computing including grid, cloud computing, as well as HPC. As such, FG offers researchers a flexible reconfigurable testbed to test functionality, performance and interoperability of software systems in a reproducible fashion. Users can customize their environment and place suitable images onto the FG fabric. Therefore, users are not locked into a specific computational environment offered typically by HPC centers. Instead users may choose varieties of *software stacks* which are packaged as part of abstract and reusable images. Such images may provide additional services while exposing platforms, libraries, and tools to users. Users have the ability to select from a variety of preconfigured images to suite their needs, and if these needs cannot be met, users can create their own images and share them with the community.

An important achievement of our image management framework is the ability to support *user-controlled dynamic provisioning* by allowing users to create, deploy, and register the images not only in virtualized, but also in non-virtualized infrastructures. Thus, they have access to bare-metal provisioning. This is a departure of the limited dynamic provisioning provided by typical HPC centers where the administrator governs control about images available for use. To support our more general approach, we have designed and implemented a set of tools expanding upon the traditional dynamic provisioning frameworks.

We use the term *rain* to indicate the process of placing a customized environment onto resources. This is motivated by the observations that the term dynamic provisioning is often not consistently used in the community, and our user-controlled dynamic provisioning drastically enhances the available functionality to integrate bare-metal resources. The process of *raining* goes beyond the services offered by existing scheduling tools due to its higher-level toolset targeting virtualized and non-virtualized resources. We also use the term *rain* to refer to the toolkit that combines a set of tools enabling the process of *raining*. In this context managing various image management workflows for a variety of distinct infrastructures becomes an essential part of the overall components and services to support *rain*.

In this paper, we will focus on a subset of issues related to the process of raining that deal with image management. It addresses every stage of the image management life cycle, from the creation, adaptation, storage, registration, and the instantiation of images into virtualized and non-virtualized resources. Other aspects, such as the FG experiment management framework to conduct scalability experiments using rain are discussed in [2].

The rest of the paper is organized as follows. In Section II, we present a brief background and related work. Next, in Section III, we describe the processes involved in image management. In Section IV we present our design and the tools

used to manage images for virtualized and non-virtualized resources. Section V describes our implementation and Section VI presents performance studies to evaluate characteristics of our tools in the FG testbed. We conclude the paper in Section VII with our findings and provide information about our future activities.

II. Background and Related Work

Image management is a key component in any modern compute infrastructure, regardless if used for virtualized or non-virtualized resources. We distinguish a number of important processes that are integral part of the life-cycle management of images. They include (a) image creation and customization, (b) sharing the images via a repository, (c) registering the image into the infrastructure, and (d) image instantiation (see Figure 1). The problem of targeting not one, but multiple infrastructures amplifies the need for tools supporting these processes. Without them, only the most experienced users will be able to manage them under great investment of time.

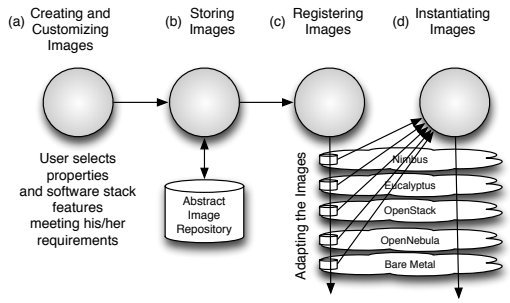


Fig. 1. Process of the image management framework

There are two interplaying approaches to simplify access. The first is the introduction of standards and best practices to interface with the infrastructure. The second is to provide a set of tools that interfaces with these standards and enables exposure of common functionality to the users while hiding the underlying complexities. Standards relevant for our efforts include Open Virtualization Format (OVF) [3], which we plan to integrate in our design. We are also aware and follow developments of efforts such as Open Cloud Computing Interface (OCCI) [4], that are not yet standards, but could provide a uniform access to cloud infrastructures. However, we have decided for now to focus our attention on the de facto standard that is provided by the Amazon cloud API, which is supported by all major cloud frameworks. Of relevance are Nimbus [5], Eucalyptus [6], OpenStack [7], and OpenNebula [8]. In case of HPC services we focused our attention on Moab/xCAT [9], [10] due to its use within the XSEDE project [11]. Alternatives to xCAT includes Chiba City [12], Cobbler [13] or the recent Metal as a Service (MaaS) from Canonical [14]. These infrastructures are supported by various tools on the operating system level and configuration management tools including Kickstart [15], Chef [16], Puppet [17] or Juju [18]. A number of tools have recently been developed that allow the creation of images though a GUI or a Web interface such as SUSE Studio [19] and Easyvmx [20].

One of the issues with such tools is they are limited and bound to a particular infrastructure or have dependencies with a particular operating system. While providing a higher-level abstraction, we strive towards removing such dependencies and offer users a tool that can integrate much more easily with the different infrastructures.

III. Processes

A number of processes need to be coordinated to properly support abstract image management and universal image registration for cloud and HPC infrastructures. We explain in more detail the activities conducted in each of the processes (see Figure 1).

a) Creating and Customizing Images: Advanced users of modern cyber-infrastructure demand creation and customization of images to fit their particular needs. The image creation can be performed using either an interactive or a non-interactive method.

In case of an interactive method, a virtual machine (VM) image file is created as part of a user guided process. This process delivers an image to be booted with an OS media disk attached to it. It starts the installation process as if we were installing a physical machine. This process is achieved using tools provided by the hypervisors to create and boot VMs.

The second method is non-interactive allowing us to automatize the process without user intervention as much as possible. This automation is supported by the tools provided in most of the Linux-based OSes to bootstrap images. They basically install a fresh copy of the OS into a directory. This installation will have the essential packages and binaries needed in a base image and updates can be readily integrated. Examples of these tools are debootstrap in Debian/Ubuntu, yum in CentOS/RedHat and febootstrap in Fedora.

The problem of the first method lies in the need of human interaction that prevents us from automating either the process or integrating it with other software. Moreover, they produce images aimed for a specific purpose, which will be compatible with a particular hypervisor or acting as liveCD. On the other hand, the second method is very flexible and allows us not only to integrate it with other software, but also to establish a clear separation between image creation and customization for a specific infrastructure.

While separating the steps, which are dependent on a specific infrastructure, it becomes possible for the same image to be adapted and to be used in different infrastructures. Typically, this procedure is different for each infrastructure we target and is usually done by users or administrators. In the case of cloud frameworks, users have to select or upload the kernel and ramdisk images to be used, which require strong knowledge of the OS. Hence they must either be experts in the field or they have to spend a considerable amount of time to accomplish this task. Moreover, after customizing the image, it has to be registered in the framework for managing the

deployment of the image onto the selected infrastructure. In bare-metal, this is usually restricted to system administrators while in the cloud frameworks it can be done by any user.

b) Storing Abstract Images: Once we have created an image, we have to store it into a repository. As there are significant differences on how images are managed between IaaS and bare-metal it is necessary to provide an image repository in which we store *abstract images* that get further modified in the registration process.

c) Registering Images: Once an image is created we must register it with the infrastructure we intend to deploy it. Image registration is typically provided in some form by the underlying infrastructure. Nimbus, Eucalyptus, OpenStack, as well as Moab/xCAT provide their own mechanisms for image registration. However, the images need to be adapted to allow utilizing them. Furthermore, we need to provide a significant toolset to expose registration functionality in bare-metal to non-administrators.

d) Instantiating Images: Once the image is registered with the infrastructure, it can be instantiated by the user as part of the deployment framework available within the infrastructure.

IV. Design

Our design targets an end-to-end workflow to support users in creating abstract image management across different infrastructures easily. This includes images for Eucalyptus, Nimbus, OpenStack, OpenNebula, Amazon, and bare-metal.

To summarize the idea behind our design, we prefer users to be able to specify a list of requirements such as an OS, an architecture, software, and libraries in order to generate a personalized abstract image. This image is generic enough that through manipulations it can be adapted for several IaaS or HPC infrastructures with little effort by the users. It will support the management of images for Nimbus [5], Eucalyptus [6], OpenStack [7], and bare-metal HPC infrastructures as they are either already deloyed in FG or going to be deployed as in the case of OpenNebula [8].

It is obvious that such a capability is advantageous to support repeatable performance experiments across a variety of infrastructures. It supports the processes identified in Section III to be able to manage the life cycle of images in a transparent fashion. Within this paper, we focus our attention on the first three processes as we have already described the last one in [21].

Figure 2 shows the architecture of our image management framework, which is capable of supporting the required processes as identified in Figure 1. To support a modular design we have devised a component for each process. This includes an image generation component to create images following user requirements, an image repository component to store, catalog and share images, and an image registration component for preparing, uploading and registering images into specific infrastructures such as HPC or different clouds.

The architecture includes a convenient separation between client and server components for allowing users to easily interact with the hosted services that manage our processes. This design allows users to access to the various processes via a python API, a REST service, a convenient command line shell, as well as a portal interface. The image management server has the task to generate, store, and register the images with the infrastructure. The image management server also interfaces with external services, such as configuration management services to simplify the configuration steps, authentication and authorization, and a service to verify the validity of an image including security checks.

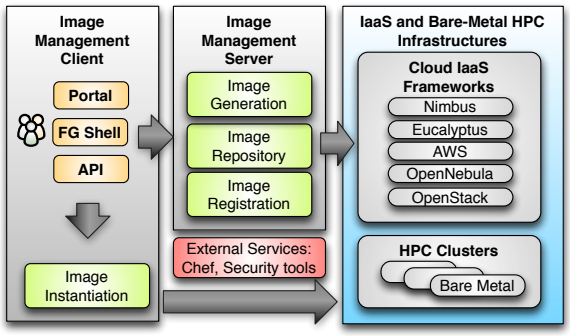


Fig. 2. FutureGrid Image Management Architecture.

One important feature in our design is how we are not simply storing an image but rather focusing on the way an image is created through abstract templating. Thus, it is possible at any time to regenerate an image based on the template describing the software stack and services for a given image. This enables us also to optimize the storage needs for users to manage many images. Instead of storing each image individually, we could just store the template or a pedigree of templates used to generate the images.

To aid storage reduction, our design includes data to assist in measuring usage and performance. This data can be used to purge rarely used images, while they can be recreated on-demand by leveraging the use of templating. Moreover, the use of abstract image templating will allow us to automatically generate images for a variety of hypervisors and hardware platforms on-demand. Autonomous services could be added to reduce the time needed to create images or deploy them in advance. Reusing images among groups of users and the introduction of a cache as part of the image generation will reduce the memory footprint or avoid the generation all together if an image with the same properties is already available.

In the next sections, we will describe in more detail the various components.

A. Image Generation

The image generation provides the first step in our image management process allowing users to create images according to their specifications. As already mentioned, the benefit of our image generation tools and services is that we are not just targeting a single infrastructure type but a range of them.

The process is depicted in Figure 3. Users initiate the

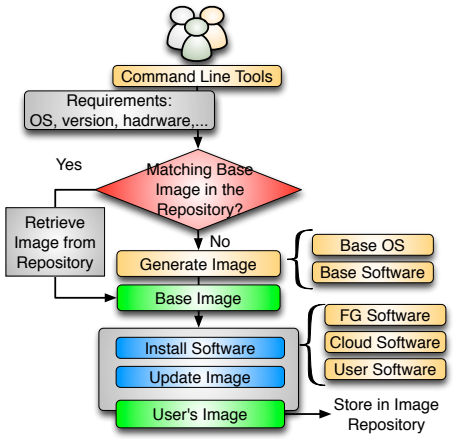


Fig. 3. Image Generation Process.

process by specifying their requirements. These requirements can include the selection of the OS type, version, architecture, software, services, and more. First, the image generation tool searches into the image repository to identify a base image to be cloned, and if there is no good candidate, the base image is created from scratch. Once we have a base image, the image generation tool installs the software required by the user. This software must be in the official OS repositories or in the FG software repository. The later contains software developed by the FG team or other approved software. The installation procedure can be aided by Chef [16]. After updating the image, it is stored in the image repository and becomes available for registration into one of the supported infrastructures. Our tool is general to deal with installation particularities of different OSes and architectures.

One feature of our design is to either create images from scratch or by cloning already created base images we locate in our repository.

In case we create an image from scratch, a single user identifies all specifications and requirements. This image is created using the tools to bootstrap images provided by the different OSes, such as yum for CentOS and deboostrap for Ubuntu. To deal with different OSes and architectures, we use cloud technologies. Consequently, an image is created with all user specified packages inside a VM instantiated on-demand. Therefore, multiple users can create multiple images for different operating systems *concurrently*; obviously, this approach provides us with great flexibility, architecture independence, and high scalability.

We can speed-up the process of generating an image by not starting from scratch but by using an image already stored in the repository. We have tagged such candidate images in the repository as base images. Consequently, modifications include installation or update of the packages that the user requires. Our design can utilize either VMs or a physical machine to chroot into the image to conduct this step.

Advanced features include the automatic upgrade or update of images stored in the repository. The old image can be deleted after the user verifies the validity of the new image.

TABLE I
METADATA INFORMATION ASSOCIATED TO THE IMAGES.

Field Name	Description
imgId	Unique identifier
owner	Image's owner
os*	Operating system
description*	Description of the image
tag*	Image's keywords
vmType*	Virtual machine type
imgType*	Aim of the image
permission*	Access permission to the image
imgStatus*	Status of the image
imgURI	Image location
createdDate	Upload date
lastAccess	Last time the image was accessed
accessCount	# times the image has been accessed
size	Size of the image

$*$ can be modified by users

B. Image Repository

The image repository [22] catalogs and stores images in a unified repository. It offers a common interface for distinguishing image types for different IaaS frameworks but also bare-metal images. This allows us to include a diverse set of images contributed not only by the FG development team but also by the user community that generates such images and wishes to share them. The images are augmented with information about the software stack installed on them including versions, libraries, and available services. This information is maintained in the catalog and can be searched by users and/or other FG services. Users looking for a specific image can discover available images fitting their needs using the catalog interface. In addition, users can also upload customized images, share them among other users, and dynamically provision them. Through these mechanisms we expect our image repository to grow through community contributed images.

Table I lists a subset of metadata associated with images stored in the repository. This includes information about properties of the images, the access permission by users and the usage. Access permissions allow the image owner to determine who has access to the image. The simplest types of sharing include private to owner, shared with the public or shared with a set of people defined by a group/project. Usage information is available as part of the metadata to allow information about usage to be recorded. This includes how many times an image was accessed and by whom.

The image repository is independent from the storage backend. It supports a variety of them and new plugins can be easily created [22].

C. Image Registration

Once the image has been created and stored into the repository, we need to register it into the targeted infrastructure before we can instantiate it. Input parameters are simply the image, the targeted infrastructure and the kernel. The kernel is an optional requirement that allows advance users to select the most appropriate kernel for their experiments. This tool provides a list of available kernels organized by infrastructure. Nevertheless, users may request support for other kernels like

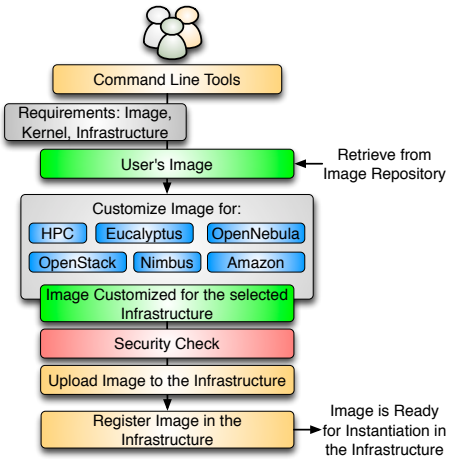


Fig. 4. Image Registration Process.

one customized by them. Registering an image also includes the process of adapting it for the infrastructure. Often we find differences between them requiring us to provide further customizations, security check, the upload of the image to the infrastructure repository, and registering it. The process of adaptation and registration is depicted in Figure 4. These customizations include the configuration of network IP, DNS, file system table, and kernel modules. Additional configuration is performed depending on the targeted deployed infrastructure.

In the HPC infrastructure the images are converted to network bootable images to be provisioned on bare-metal machines. Here, the customization process configures the image, so it can be integrated into the pool of deployable images accessible by the scheduler. In our case this is Moab. Hence, if such an image is specified as part of the job description the scheduler will conduct the provisioning of the image for us. These images are stateless and the system is restored by reverting to a default OS once the running job requiring a customized image is completed.

Images targeted for cloud infrastructures need to be converted into VM disks. These images also need some additional configuration to enable VM's contextualization in the selected cloud. Our plan is to support the main IaaS clouds, namely Eucalyptus, Nimbus, OpenStack, OpenNebula, and Amazon Web Service (AWS). As our tool is extensible, we can also support other cloud frameworks.

Of importance is a security check for images to be registered in the infrastructures. A separate process identifies approved images, which are allowed to be instantiated in FutureGrid. Approval can be achieved either by review or the invocation of tools minimizing and identifying security risks at runtime. Users may need to modify an image to install additional software not available during the image generation process or to configure additional services. Modified images need to go through some additional tests before they can be registered in the infrastructure. To perform these security tests, we plan to create a platform for instantiating the images in a controlled environment such as a VM with limited network access. Hence, we can perform some tests to verify the integrity of the image, detect vulnerabilities and possible malicious software. If the image passes all the tests, it is tagged as *approved*.

To provide authentication and authorization images may interface with the FG account management.

The process of registering an image only needs to be done once per infrastructure. Therefore, after registering an image in a particular infrastructure, it can be used anytime to instantiate as many VMs or in case of HPC as many physical machines as available to meet the users requirements.

V. Implementation

Our implementation uses xCAT [10], Moab [9] and Torque [23] to manage HPC images. Although these tools should, in theory, simplify the management, we found that readily deployable patterns from Moab were not available. Furthermore, we identified hardware and operating system restrictions imposed by xCAT. This motivates us for future development to remove the dependencies of both xCAT and Moab while targeting alternatives providing more suitable free and open-source solutions. The internal cloud to manage concurrent image generation processes is based on OpenNebula. In the IaaS side, we have interfaced with Eucalyptus, OpenStack, and OpenNebula. We target AWS and Nimbus. At this moment only CentOS and Ubuntu are supported, but we plan to extend this support to other OSes.

VI. Evaluation

In this section, we describe our newest performance experiments while focusing our attention on the analysis of the image generator and image registration process. The performance of the image repository was already evaluated earlier and the results are available in [22].

Our performance study uses resources and services deployed on FG. In particular, we used the FG India cluster, which is composed by Intel Xeon X5570 servers with 24GB of memory, a single drive 500GB with 7200RPMm 3Gb/s, and an interconnection network of 1Gb Ethernet.

As part of this study, we configured the image management components as follows:

- The image repository has been configured using MongoDB [24] to store the image metadata. Cumulus [25] is used to store the image files. This configuration was identified to be very good as part of our earlier experiments with the image repository [22].
- To be able to generate images (see Section IV-A) in parallel, we are using OpenNebula. Although we could have used other IaaS frameworks, we chose OpenNebula due to its ease of deployment as documented in [21].
- In order to support the image registration, we provide two independent services: one registers images into a cloud, and another registers them in the HPC infrastructures. The later requires write access to the xCAT image directory and execution permissions to use the xCAT client.

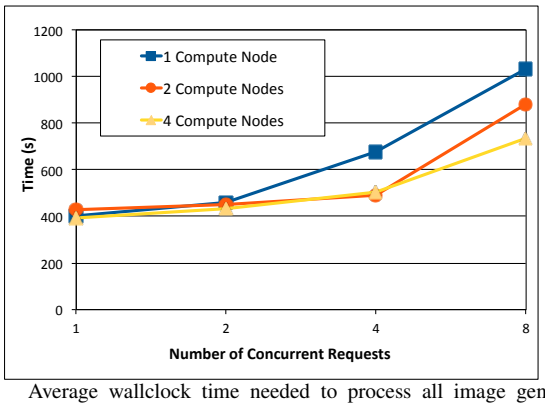


Fig. 5. Average wallclock time needed to process all image generation requests depending on the number of OpenNebula compute nodes. Each graph in the figure represents the number of available OpenNebula compute nodes.

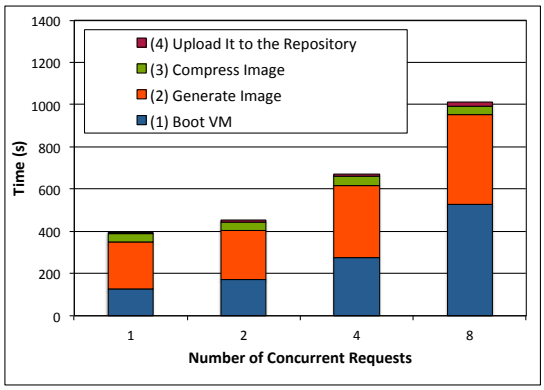
- The image management client has been deployed in the India FutureGrid machine (login node) and is accessible to authorized users.

Next, we describe the tests performed and the results obtained in each case. All tests have been performed three times to obtain average results.

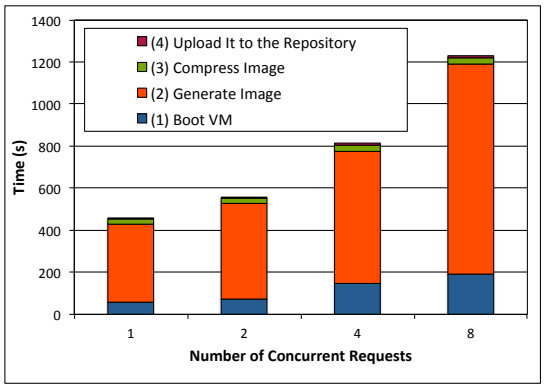
Image Generator: First, we studied the scalability of the image generator tool. We performed experiments varying the number of concurrent image generation requests (from one to eight) to create CentOS images from scratch. As part of these experiments, we increased the number of OpenNebula compute nodes from one to four to highlight the scalability of the service. Figure 5 shows the results of these tests. We observe the overall performance using a single OpenNebula compute node to be good as the minimum average wallclock time to create an image with this setup is around *six minutes* for a single request. When using the same node to handle eight requests, we see an overall time of 16 minutes. To reduce this time, we can increase the number of compute nodes to distribute the workload on other nodes. Finally, we observe a performance degradation when we generate more than two images per compute node. Therefore, this limitation must be considered when deploying our framework in production.

Next, we analyzed the time spent within the image creation process. For these tests we used a single OpenNebula compute node in order to better analyze the behavior of the different steps involved in the image generation process. The results of these tests are shown in Figure 6 (a) for CentOS and Figure 6 (b) for Ubuntu. This includes results for the sub processes to (1) boot the VM, (2) generate the image, (3) compress the image, and (4) upload the image to the repository.

In the Figure 6, we observe the virtualization layer introduces significant overhead in the process (1). This overhead is higher in the case of CentOS images and indicates our CentOS golden image needs to be optimized to speed-up this phase. The time to create the base image and the installation of the software requested by the user is the most time consuming in this process (2) and is worse for Ubuntu. In particular, the overall time used for this phase is up to 69% for CentOS and 83% for Ubuntu. The remaining time for this phase is spent



(a) Generate CentOS Images from scratch



(b) Generate Ubuntu Images from scratch

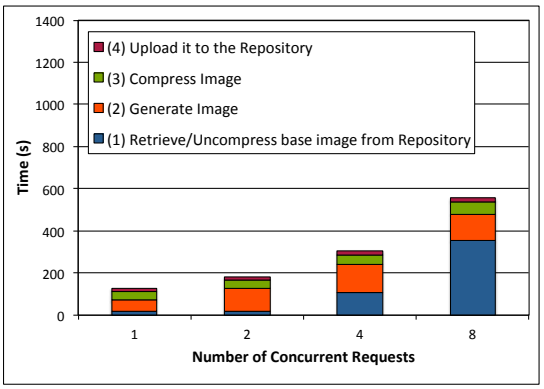
Fig. 6. Average wallclock time needed to process all image generation requests with time associated to the different phases of the process.

on installing the software and packages requested by the user. For this reason, we considered to manage base images that can be used to save time during the image creation process.

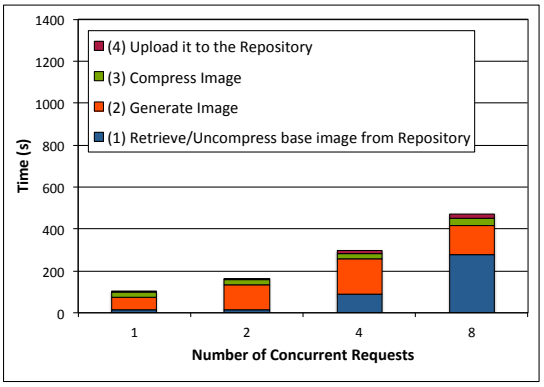
This has been demonstrated and the results are shown in Figure 7. We performed the image generation with help of a caching policy for taking advantage of reusing base images stored in our repository.

According to the results depicted in Figure 7, using a base image as part of the creation process reduces the time needed to complete this step dramatically. In particular, the time to create an image has been reduced from six minutes to less than two minutes for a single request. For the worst case, we reduced the time from 16 minutes to less than nine minutes, where we used one processors handling eight concurrent requests. The reason is twofold: there is no need to create the base image every time, and we do not need to use the virtualization layer since the base image already has the desired OS and only needs to be upgraded with the software requested by the user. Thus, our tool can directly retrieve and uncompress the base image to customize it with users' requirements. Once the image has been customized, it is compressed and uploaded to the image repository.

The results show that the image repository we designed does not cause a bottleneck as the time to upload the images to the repository is negligible (see Figure 6 and 7).

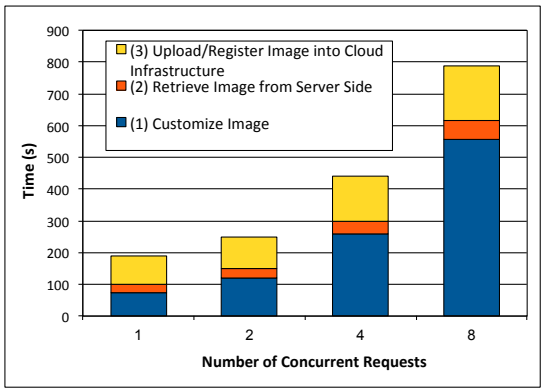


(a) Generate CentOS Images from a base image

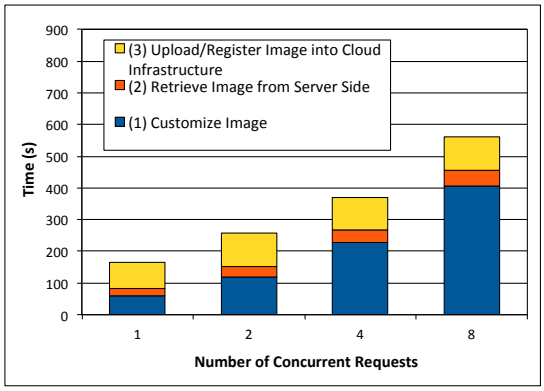


(b) Generate Ubuntu Images from a base image

Fig. 7. Average wallclock time needed to process all image generation requests with time associated to the different phases of the process.



(a) Register Images in OpenStack



(b) Register Images in Eucalyptus

Fig. 8. Average wallclock time needed to register all images in the infrastructure with time associated to the different phases of the process.

Image Registration: Next, we analyzed the behavior of the image registration processes. For that, we registered the same CentOS image in different infrastructures, namely OpenStack (version Cactus configured with KVM hypervisor), Eucalyptus (v2.03 configured with XEN hypervisor), and HPC (Moab v6.0.3, xCAT 2, Torque v2.5.5). For Eucalyptus and OpenStack, we utilized concurrent registrations. In contrast, our service to register images in the HPC infrastructure only processes a single request at a time because it modifies critical parts of our HPC infrastructure. Therefore, at this time it must be performed in an atomic section.

The results of registering images are shown in Figure 8 (a) for OpenStack and Figure 8 (b) for Eucalyptus. The figures use a stacked bar chart to depict the time spent in each phase of this process including (1) customization of the image, (2) retrieval of the image after customization, and (3) the upload and registration of the image to a cloud infrastructure. It is to be noted that (1) is executed in the server side while (3) is executed in the client side. The reason for this is based on our authorization framework, as we need to use the user's credentials to upload and register the image to the cloud infrastructure. Therefore, times associated with (2) represent the time to send the image form the server to the client.

We observe the time needed to customize the image (1) increases with the number of concurrent requests. Part of this activity includes uncompressing the images in the server side to prepare them for being uploaded and registered with the IaaS framework. In our experiment we are concurrently processing all requests in the same machine. This setup has been satisfying our demands on FG and did not result in resource starvation and scalability issues. Additionally, we introduced a parameter to limit the number of concurrent requests that prevents overloading the service.

In our observations, the time to register an image in OpenStack is higher than in Eucalyptus. This is based on two factors. First, in OpenStack, we need to include certain software to allow OpenStack to contextualize the VM during the instantiation time (included in (1)). Second, the process to upload the image to the OpenStack cloud takes longer than in Eucalyptus (2). As part of this process, both frameworks compress and split the image in smaller tgz files that are uploaded to the IaaS server. The difference is that OpenStack uncompresses the image in the server side as part of this process, while Eucalyptus seems to maintain the compressed version. Additionally, we have noticed occasionally OpenStack fails to upload some images when we perform several concurrent requests. Consequently, images get *stuck* as part of the untarring process and can never complete the uncompression. While analyzing this problem further, we suspect it may relate to a scalability issue of the messaging queue system within

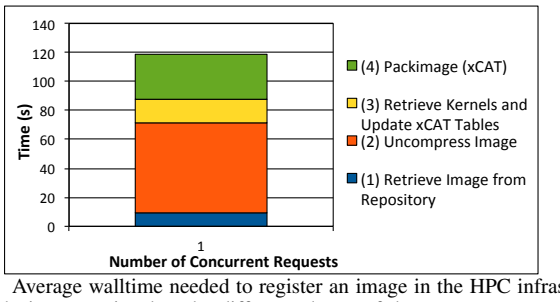


Fig. 9. Average walltime needed to register an image in the HPC infrastructure with time associated to the different phases of the process.

OpenStack. Another observation is that the logfiles generated by OpenStack were not very helpful to debug this problem.

In Figure 9, we show the results obtained from registering images in our HPC infrastructure utilizing Moab/xCAT. We distinguish the following phases (see Section IV-C) : (1) retrieve the image from the repository, (2) uncompress the image, (3) retrieve kernels and update the xCAT tables, and (4) package the image. To have minimal impact on other HPC services, we decided to only process one request at a time. We observe that the overall process only takes about two minutes as no additional software is needed to be installed and everything is executed on the server. The most time consuming parts are uncompressing the image (2) and executing the xCAT packimage command (4). This command creates a tar/cpio image, which will be used to netboot bare-metal machines when users request it. The final step of this process is the registration of the image with Moab and recycling the Moab scheduler. After this step, the image becomes available to users.

VII. Conclusions and Future Work

In this paper, we have presented the FutureGrid *user controlled* image management framework as a revolutionary way to handle images for different infrastructures spanning virtualized and non-virtualized resources. It allows users to register images, created by our software, for Nimbus, Eucalyptus, OpenStack, OpenNebula, as well as bare-metal infrastructures. With our framework, users are able to easily create and manage customized environments within FG. This is achieved by abstracting the details of each underlying infrastructure. Users can with simple tools replicate software stack requirements on the supported IaaS and bare-metal systems.

In our evaluation, we have identified the most time consuming parts of our software. Our results show a linear increase in response to concurrent requests. The image generation tool is able to create images from scratch in only six minutes. When modifying a base image, it allows us to generate images in less than two minutes in many of our use cases. Additionally, we can scale the performance by adding more nodes that are used to generate the images. The image registration tool was able to register images in any infrastructure in less than three minutes. Indirectly we have also seen that our image repository shows excellent behavior in our use cases and introduces only negligible overhead to the overall processes.

We are currently working towards supporting Amazon and Nimbus. Our plan also includes the integration of a messaging queue system and portal interface to allow queuing of user's requests to process them in an asynchronous way. This will introduce a more robust fault tolerant behavior for user access. On-demand resource allocation for supporting peak access is also part of this strategy. Our tools are currently used by a selected number of users on FG.

Acknowledgment

We thank Koji Tanaka, Sharif Islam, Jerome Mitchell, Greg Pike, and the rest of the FG team for their support. This work is supported by the NSF under Grant No. 0910812.

References

[1] "FutureGrid Portal," Webpage. [Online]. Available: http://portal.futuregrid.org
[2] G. von Laszewski, G. C. Fox, F. Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Voeckler, R. J. Figueiredo, J. Fortes, K. Keahey, and E. Delman, "Design of the futuregrid experiment management framework," in *GCE2010 at SC10*, IEEE. New Orleans: IEEE, 2010.
[3] "Open Virtualization Format (OVF)," Webpage. [Online]. Available: http://www.dmtf.org/standards/ovf
[4] "Open Cloud Computing Interface (OCCI)," Webpage. [Online]. Available: http://occi-wg.org/
[5] "Nimbus Project," Webpage. [Online]. Available: http://www.nimbusproject.org
[6] "Open Source Eucalyptus," Webpage. [Online]. Available: http://open.eucalyptus.com/
[7] "OpenStack," Webpage. [Online]. Available: http://openstack.org/
[8] "OpenNebula," Webpage. [Online]. Available: http://www.opennebula.org/
[9] A. Computing, "MOAB Cluster Suit Webpage." Webpage, Last access Feb. 2012. [Online]. Available: http://www.clusterresources.com/products/moab-cluster-suite.php
[10] "xCAT Extreme Cloud Administration Toolkit," Webpage. [Online]. Available: http://xcat.sourceforge.net/
[11] "XSEDE - Extreme Science and Engeneering Environment," 2012. [Online]. Available: http://www.xsede.org
[12] J.-P. Navarro, R. Evard, D. Nurmi, and N. Desai, "Scalable cluster administration " chiba city i approach and lessons learned," in *Proceedings of the IEEE International Conference on Cluster Computing*, ser. CLUSTER '02, 2002, pp. 215–.
[13] "Cobbler Web," Webpage. [Online]. Available: https://fedorahosted.org/cobbler/
[14] "MaaS wiki," Webpage. [Online]. Available: https://wiki.ubuntu.com/ServerTeam/MAAS
[15] "Anaconda/Kickstart definition," Webpage. [Online]. Available: http://fedoraproject.org/wiki/Anaconda/Kickstart
[16] "Chef Wiki," Webpage. [Online]. Available: http://wiki.opscode.com/display/chef/Home
[17] "Puppet Labs," Webpage. [Online]. Available: http://puppetlabs.com/
[18] "Juju wiki," Webpage. [Online]. Available: https://juju.ubuntu.com/
[19] "Suse Studio," Webpage. [Online]. Available: http://susestudio.com/
[20] "EasyVMX," Webpage. [Online]. Available: http://www.easyvmx.com/
[21] G. von Laszewski, J. Diaz, F. Wang, and G. C. Fox, "Towards Cloud Deployments using FutureGrid," Indiana University, Bloomington, IN, FutureGrid Draft Paper, April 2012.
[22] J. Diaz, G. von Laszewski, F. Wang, A. Younge, and G. Fox, "FutureGrid Image Repository: A Generic Catalog and Storage System for Heterogeneous Virtual Machine Images," *Third IEEE International Conference on Coud Computing Technology and Science (CloudCom2011)*, 2011.
[23] "Torque Resource Manager," Webpage. [Online]. Available: http://www.adaptivecomputing.com/products/torque.php
[24] "MongoDB," Webpage. [Online]. Available: http://www.mongodb.org/
[25] J. Bresnahan, K. Keahey, T. Freeman, and D. LaBissoniere, "Cumulus: Open source storage cloud for science," *SC10 Poster*, 2010.