

**REPORT ON THE COLLABORATIVE RESEARCH:
ENABLING ON-DEMAND SCIENTIFIC WORKFLOWS
ON A FEDERATED CLOUD**

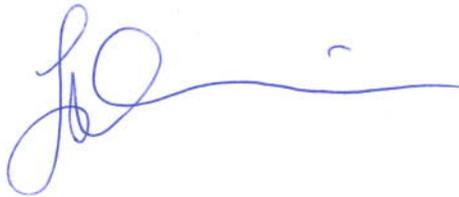
Submitted to

**Korea Institute of Science and Technology
Information**

**245 Daehangno, Yuseon, Daejeon, 305-806,
Republic of Korea**

October 23, 2014

Garbiele Garzoglio

A handwritten signature in blue ink, consisting of a stylized 'G' followed by a long horizontal line.

**Grid and Cloud Service Department
Fermi National Accelerator Laboratory, USA**



Fermi National Accelerator Laboratory
Computing Division P.O. Box 500
Batavia, Illinois 60510

From:
Gabriele Garzoglio
Grid and Cloud Services Department
Fermi National Accelerator Laboratory
Batavia, IL, USA

Oct 17, 2014

To:
Korea Institute of Science and Technology Information
245 Daehangno, Yuseong, Daejeon, 305-806
Republic of Korea

Dear Sir or Madam,

It is with great pleasure that I present to you this report of the Collaborative Research And Development Agreement with KISTI for 2014 (CRADA FRA 2014-0002/ KISTI-C14014). At this point we can state that all the goals presented in the proposal will be successfully completed by the end of the contract (Oct 2014). The final report, the documentation produced, the papers, and the source code developed are attached.

The collaboration focused on a multi-year program of research and development for a federated Cloud computing infrastructure. In the first year, we have demonstrated proof-of-principle integrations of Cloud and Grid systems to run scientific workflows on multiple dynamically allocated resources. In this second year, we have extended the scale and scope of the work, focusing on deployment of on-demand services in support of scientific computation and demonstrating scalability to 1,000 Virtual Machines on the Fermilab private Cloud (FermiCloud) and Amazon Web Services.

We look forward to further collaborative initiatives with KISTI and renew our interest in continuing the joint multi-year program started with this project.

Sincerely,

A handwritten signature in blue ink, appearing to read "Garzoglio", with a long horizontal flourish extending to the right.

Gabriele Garzoglio, Ph.D. (Principal Investigator)
Grid and Cloud Services Department, Head
Scientific Computing Division
Fermi National Accelerator Laboratory

**REPORT ON THE COLLABORATIVE RESEARCH:
ENABLING ON-DEMAND SCIENTIFIC WORKFLOWS ON A
FEDERATED CLOUD**

CRADA FRA 2014-0002 / KISTI-C14014

**STEVEN TIMM, GABRIELE GARZOGLIO,
FERMI NATIONAL ACCELERATOR LABORATORY**

**SEO-YOUNG NOH, HAENG-JIN JANG,
KISTI**

Table of Contents

| | |
|---|---|
| 1. Executive Summary | 2 |
| 2. Introduction | 2 |
| 3. Technical deliverables | 2 |
| 3.1. Virtual Infrastructure Automation and Provisioning | 4 |
| 3.2. Interoperability and Federation of Cloud Resources | 5 |
| 3.3. On-demand Services for Scientific Workflows | 6 |
| 4. Qualitative and quantitative output | 7 |
| 5. Budget Allocation | 8 |
| 6. References | 8 |

1. EXECUTIVE SUMMARY

The Fermilab Grid and Cloud Computing Department and the KISTI Global Science experimental Data hub Center are working on a multi-year Collaborative Research And Development Agreement. With the knowledge developed in the first year on how to provision and manage a federation of virtual machines through Cloud management systems, in this second year, they have enabled scientific workflows of stakeholders to run on multiple cloud resources at the scale of 1,000 concurrent machines. The demonstrations have been in the areas of (a) Virtual Infrastructure Automation and Provisioning, (b) Interoperability and Federation of Cloud Resources, and (c) On-demand Services for Scientific Workflows. This is a matching fund project in which Fermilab and KISTI will contribute equal resources.

2. INTRODUCTION

The Cloud computing paradigm has revolutionized the approach to Information Technology in many sectors of society, from telecommunication to the military to science. In particular for science, national laboratories, computing centers and universities in many countries are adapting their current paradigm on distributed computing, complementing the Grid model of federated resources [1,2] with the Cloud model of on-lease dynamically instantiated resources from private and commercial entities.

For institutions such as KISTI and Fermilab, the focus on Cloud computing is dictated by the need to support ever more effectively individual large communities (e.g. the LHC experiments) together with many medium size ones (e.g. the Intensity Frontier experiments and the STAR experiment). With each community requiring slightly different configurations for their computational environment, the use of dynamically instantiated virtual machines managed through a Cloud layer becomes an attractive solution to enable such diversity. In addition, in a time where computing budgets are unable to follow the growth of demand from the stakeholders, research institutions are compelled to improve their flexibility in the allocation of resources. To address the need for flexibility, the characteristic of the Cloud paradigm to instantiate computing services on-demand on a common pool of computing hosts provides a valuable solution.

KISTI and Fermilab have been collaborating on Cloud computing since 2011. In 2013, this collaboration resulted in a formal Collaborative Research And Development Agreement (CRADA) for a multi-year program of work to offer production-quality on-demand computing services to their scientific stakeholders. The vision is to provide layered services on a federation of Clouds, provisioning execution environment on a high-throughput fabric of resources, with the scientists interacting with the Software as a Service layer.

In the first year of collaboration, we focused on proof-of-principle demonstration of basic capabilities: resource provisioning through a few selected Cloud interfaces (Infrastructure as a Service), techniques for virtual machine federation and compatibility, and high-throughput virtualization to build the “fabric” of the computational infrastructure. In this second year, we expanded the work on provisioning and federation, increasing both scale and diversity of solutions, and we started to build on-demand services on the established fabric, introducing the paradigm of Platform as a Service to assist with the execution of scientific workflows. In the years to come we envision to increase both the diversity of Cloud providers and the scale of utilization, improving our ability to federate resources and deploy ensembles of complex services in support scientific computation.

This report focuses on the deliverables for the second year of the program.

3. TECHNICAL DELIVERABLES

The program of work for the second year of the agreement consisted in demonstrations and studies to run scientific workflows on dynamically provisioned resources at the scale of 1,000 concurrent virtual machines. The work was organized in three major areas: (1) Provisioning; (2) Federation and Compatibility; (3) On-Demand Services.

1. **Virtual Infrastructure Automation and Provisioning** focuses on finding fast and reliable mechanisms to access a large amount of resources on private, community, and commercial cloud providers. The area was organized in 5 major activities:
 - a. Infrastructural scalability to 1,000 VM: demonstrate a mechanism which can transparently extend grid jobs into FermiCloud and other clouds at scale of 1000 or more simultaneous virtual machines
 - b. Scientific workflows scalability to 1,000 VM: run production scientific workflow of at least 1000 simultaneous virtual machines on federated cloud resources via GlideinWMS [4,8] and cloud web

- services API's. This activity demonstrates that workflows for the NOvA experiment can take advantage of Cloud resources for their computational peaks at the scale of 1,000 VM.
- c. **Cost-sensitive provisioning:** Continue development of provisioning algorithms that calculate relative cost of commercial cloud and private cloud provisioning, including the potential for spot pricing. Hao Wu, a PhD student from the Illinois Institute of Technology is focusing his research on this topic. The results of his research this year have been accepted at the MTAGS workshop as a paper jointly authored with KISTI [9].
 - d. **Provisioning of a platform of services:** Investigate deploying complicated ensembles of virtual machines in support of scientific workflows. This year, the work focused on the dynamic deployment of an ensemble of Squid services on Amazon Web Services. Relying on a cache discovery system (Shoal), the ensemble acts as a discoverable platform for web-based data caching. This infrastructure is used in conjunction with the CERN VM File System (CVMFS) to provide scalable access to software applications on Grid and Cloud platforms.
 - e. **Idle VM detection improvements:** we investigated whether it was necessary to add additional functionality to the idle VM detection software developed in the 1st year of the CRADA program, making it publicly available to the OpenNebula ecosystem, in case. We are finalizing the investigation on whether the idle VM detection system is sufficient for the current level of utilization of the Cloud resources or further improvements are necessary for next year. Our aim is to generalize the system to work on other Cloud management systems, in addition to OpenNebula, and for commercial clouds.
2. **Interoperability and Federation of Cloud Resources** consists in finding a set of virtual image formats and application programming interfaces that can be used by all members of a virtual organization across a heterogeneous infrastructure. The area was organized in 4 major activities:
- a. **Authentication / Authorization:** perform comparative investigation of X.509 authentication and authorization used by other cloud software providers and federated cloud taskforces. The investigation was accepted as a paper to the IEEE First International Workshop on Cloud Federation Management [10].
 - b. **VM image portability:** develop automatic virtual machine image format conversion service. This activity resulted in the development of an open source tool and related documentation for porting VM images from an Open Nebula system, such as FermiCloud, to Amazon Web Services. The tool removes configurations specific to the hosting environment, such as disk mount point, converts the resulting image in a format compatible with AWS, and automatically transfers the image in the AWS repository for immediate use.
 - c. **VM distribution:** investigate virtual machine distribution methods and distributed object-based storage. This investigation evaluates scalable global storage technologies shared among hosts in a private Cloud (FermiCloud). Within our scope, a global storage is necessary to speed up the deployment of virtual machines to the hosts and to enable virtual machine live-migration. The current technologies (Red Hat clustering) do not scale well. The investigation focused on the CEPH storage system and gave encouraging results, although deployment and operations are expected to be more stable on the upcoming generation of operating systems (Scientific Linux 7)
 - d. **Cloud interoperability:** expand interoperability studies to cover more commercial and community clouds. This study focused on the documentation and proof-of-principle integration of the Google Compute Engine and Microsoft Azure Cloud with the Fermilab computational environment. The goal of this activity was to expand the range of commercial Cloud providers available to our scientific stakeholders, to avoid vendor lock in and optimize workflow execution.
3. **On-demand Services for Scientific Workflows** aims at finding the most efficient methods for scientific grid and cloud computing middleware to distribute data and execution across the WAN to meet the demand. This area was organized in 4 activities:
- a. **Data movement on-demand:** evaluate strategies for data storage and movement in support of scientific workflows. For virtual machines running at AWS, we evaluated the cost effectiveness of storing data on local storage (S3): taking advantage of the S3 scalability, jobs could store data immediately at the end of the jobs. We compared this with keeping virtual machines waiting on a data transfer queue to move data back to Fermilab storage. In our model, the latter approach was most cost effective. In addition, this activity supported the provisioning a platform of services, discussed in (1), focusing on the deployment of an ensemble web caching servers (Squid), discoverable through a common name server (Shoal).

- b. MPI on Virtual Clusters: continue virtualized MPI computational chemistry workflows with KAIST stakeholders and other interested parties. In the first year of the CRADA we enabled the use of InfiniBand from virtualized nodes, this year we collaborated with Prof. Yousung Jung to try the system for quantum chemistry and quantum mechanical periodic solid computations. The demonstration exposed the need for further optimization of the infrastructure, in order to achieve the expected scalability.
- c. User-oriented best practices: develop best practices for running workflows on public clouds with limited Internet accessibility and significant data bandwidth charges. As the number of communities using Cloud resources for production activities is still relatively small, direct interaction with the users has been the preferred way to discuss best practices on the Cloud. We envision writing documentation on this topic in the next year of our collaboration.
- d. Fabric improvements: do high-bandwidth storage and network fabric research as necessary to support the above workflows. At the scale of 1,000 virtual machines, we did not uncover the need for further tuning of the fabric to support the computational activities.

The following sections describe in more detail these demonstrations and studies.

3.1. Virtual Infrastructure Automation and Provisioning

Infrastructural and scientific workflows scalability to 1,000 VM

We have previously identified the cosmic-ray Monte Carlo simulation of the NOvA experimental detector as a workflow that is ideally suited for commercial cloud computing due to almost no input files, a modest 1GB output file, and being predominantly CPU bound. We have previously run up to 100 simultaneous virtual machines on AWS and up to 150 jobs on FermiCloud (50 VM's at 3 jobs per VM). On Amazon Web Services the limiting factor had been bandwidth to an auxiliary Squid caching server on the site of Fermilab. It was also cumbersome to create a new working image on AWS due to our requirements for use of special kernel modules. The "provisioning of a platform of services" activity, described below, successfully addressed the caching issue. The virtual machine image format conversion tool, described below in the "virtual machine image portability" section, simplified the process of creating or changing the stock virtual machine. On AWS we typically run machine type m3.large, which has the capacity to run two NOvA jobs in parallel and features a 32GB SSD-based Instance store.

To expand FermiCloud to the capacity of running 1000 Virtual Machines the following steps were necessary: 140 Dell PowerEdge 1950 worker nodes (8 CPU cores each, 16GB of RAM) were made available for the purpose of running batch jobs on the cloud. We used a locally routable private network for the virtual machines to access the Fermilab site networks. Those few files that we needed from off-site were accessed via a Squid proxy server. We also deployed OpenNebula 4.8 with X.509 authentication. This implements a much more reliable and fully-featured emulation of Amazon EC2. We used our Bluearc NFS server as the image repository, copying the image from the NFS server to the disk and starting it there.

Our worker node images on FermiCloud start with the "Golden Image", which is a minimal Linux installation, and add the extra software packages needed to run a job at boot time of the image. This is done via a series of contextualization scripts that use a one-time application of the Puppet configuration system. On the public network this takes less than a minute to complete. On the routable private network, where we are using a web proxy to access off-site files via http, it can up to 10 minutes for a single VM and significantly longer if many are initializing in parallel on the same node. For the second phase of this test, we used a virtual machine with these configurations pre-applied, which significantly helped the launch and fill times.

The native command line utilities of OpenNebula 4.8 can fill the 1000-VM cluster in approximately 30 minutes. In production we use GlideinWMS and HTCondor for VM provisioning and management. HTCondor creates a new ssh key-pair for each virtual machine submitted so that it can uniquely identify them. OpenNebula 4.8 has a hard limit of 300 key-pairs that can be stored per user. An error occurs on each VM submission after that, but it is possible to have HTCondor continue after the error and thus fill the cluster to the full capacity of 1000 virtual machines. It is also necessary to aggressively prune the MySQL database otherwise some queries (DescribeInstances) will time out.

After filling the clouds with 1,000 test jobs, we engaged the NOvA users to submit a real workflow that in total consists of 20000 files to process. We processed these files on AWS and FermiCloud simultaneously, reaching the limit of 1000 simultaneously running VMS (1 job/VM) on FermiCloud and 1000 simultaneously running jobs (2 jobs/VM) on AWS.

The Amazon AWS and FermiCloud private network virtual machines have been added to our production job submission servers as part of this work and we will keep them available for further user access in a service that will be known as the “On-Demand Batch Slot Instantiation Service”

Cost-sensitive provisioning

The design goal of a cost-sensitive algorithm is to enable automatic provision of resources for different scientific applications so that the QoS of the scientific application is met and the operational cost is minimized. The main challenge of designing such an algorithm consists in deciding when and where to allocate resources so that the goals are met. In this study [9], we developed a mechanism to automatically train the VM launching overhead reference model [11]. Based on the virtual machine launching overhead reference model, we have developed an overhead-aware-best-fit (OABF) resource allocation algorithm to help the cloud infrastructure reduce the average VM launching time. This OABF algorithm has been implemented for FermiCloud as a plug in of the vcluster system [5] developed by KISTI. The experimental results indicate that the OABF can significantly reduce the VM launching time (reduced VM launch time by 4 minutes on average) when many VMs are launched simultaneously.

Provisioning of a platform of services

In year 1 we launched individual instances of virtual machines. In this year 2, we started to deploy them as an ensemble with the services that they rely on for scalability. This year we focused on web caching [24], which is our most pressing need in running remotely. We used the Shoal system, developed at the University of Victoria, for discovering Squid servers. It consists of three major components:

1. Shoal Agent: it is on each remotely-deployed Squid server and advertises its location to the central server via the RabbitMQ messaging protocol;
2. Shoal Server: it collects the information on the availability and location of the squids ;
3. Shoal Client: it runs on each worker node virtual machine to query the Shoal Server and then modifies the configuration of the virtual machine appropriately with the current list of squid servers

All components of squid, the shoal agent and client are automatically deployed via puppet apply scripts at boot time, using the cloud-init service of AWS.

As part of this investigation we learned about auto-scaling groups, which are available both in AWS and in Google Cloud, and are currently testing a load balancing system that brings multiple squid servers up or down on demand. For the workflows that we currently run on AWS, a single remote squid is sufficient to handle the load. In year 3 we envision deploying submission nodes on demand and storage transfer servers.

3.2. Interoperability and Federation of Cloud Resources

Virtual machine image portability

The current Cloud Management and Virtualization technologies support a number of diverse Virtual Machine formats, not all compatible. VM image conversion is necessary in many cases to instantiate a VM from a Cloud system to another. For our use cases, this happens commonly when transferring a “golden” image (i.e. a VM with stable configuration) from FermiCloud to AWS, for example to run scientific workflows locally and on the commercial platform using the exact same computational environment.

To address this problem, in year 1 we developed step-by-step documentation to manually convert between commonly-used desktop virtualization formats and server-based virtualization and cloud solution, such as OpenNebula, OpenStack, and Amazon EC2 [3]. This year, we developed a tool to automate the process [25].

The tool takes as input a golden image from the FermiCloud image storage; the image does not need to be instantiated for the conversion process to work. It first resizes the image to optimize the space utilization and cost when the VM is uploaded to the AWS image storage, then it removes certain Fermilab-specific configurations, such as disk mount points and network configurations. At this point, the image is imported to the AWS storage.

Amazon Machine Images (AMI) support two types of virtualization: paravirtual (PV) and hardware virtual machine (HVM). In general, paravirtual VMs can run on host hardware that does not have support for hardware-assisted virtualization, and can be fitted with special network and storage drivers to better take advantage of the underlying hardware. Historically, PV VMs had better performance than HVM VMs in many cases, but because of enhancements in HVM virtualization and the availability of Solid State Device (SSD) disk drives, this tends not to be the case any longer. PV formats also tend to be smaller in size than their HVM counterparts. Irrespectively, after the import both VM formats can be directly instantiated at AWS.

The tool takes about an hour for the conversion process to complete, resulting typically in VM sizes around 3 GB for PV images and 12 GB for HVM. Early operational experience has identified ways to cut this time significantly.

In the next year of the program, we aim at extending the automation of this process to additional commercial cloud providers, such as Google Computational Engine and Microsoft Azure.

Virtual machine distribution

We initially planned to evaluate OpenStack's Swift object store, the GlusterFS system, and the Ceph system. All of these are distributed object storage systems. Given CERN's recent success in deploying Ceph as the back-end image storage for their large OpenStack system [22], we decided to focus on evaluating this system. We found it to be a very stable system and easy to operate and maintain. We successfully used Ceph to run virtual machines on network-attached remote block devices and also to import and export full images to and from local VM host disks [23]. We found the Ceph system to be very stable in operation and fault-tolerant against single machine failures. As Red Hat Enterprise Linux 7/Scientific Linux 7 becomes more widespread we expect that it will become much more straightforward to deploy and maintain in production and we intend to continue planning towards that deployment.

Cloud interoperability

In order to increase the diversity of resources in our Federation, we have continued our investigation of Application Program Interfaces from major Commercial Cloud providers. Our goal is to enable the use of different providers to run our scientific workflows. Different providers might be more effective in terms of performance or cost to address the specific needs of our scientific workflows. This strategy also mitigates the risk of vendor lock-in. In year 1 we successfully tested Amazon EC2 [3]. This summer we tested the Google Compute Engine and the Microsoft Azure Cloud. We produced a manual [26] for beginning users to start new virtual machines on the Google Compute Engine using the web GUI or by using a Python script with the associated Python bindings. We also collected information on how to start virtual machines on the Microsoft Azure Cloud. For bulk usage of either of these clouds, support will have to be added to HTCondor and GlideinWMS. HTCondor at one point supported the Google Compute Engine but within the past year the API has changed sufficiently that the interface will have to be totally redone and we have supplied the info to the developers on how to do it. We have also identified the work that will have to be done with the Microsoft Azure cloud to get HTCondor to support that. In the next year of the program, we plan to evaluate the integration of HTCondor with the OpenStack Native Cloud Interface, which is being integrated this year.

3.3. On-demand Services for Scientific Workflows

Data movement on-demand

We have evaluated the cost of two different strategies to transfer the output of 1,000 jobs running on AWS to a remote archive (e.g. the Fermilab Archival facility). The two strategies incur in different costs depending on parameters such as the effective bandwidth of the transfer, the total size of the output, and the hourly cost of virtual machines. The two strategies are described below:

1. When a job finishes its processing phase and is ready to transfer its output, the job initiates a direct transfer to the external archival facility. We assume that multiple jobs finish approximately at the same time and the archival service allows only a limited number of transfers to avoid overload; the archival service queues up the requests that cannot be immediately served. Jobs that are waiting still incur the regular cost of running a VM at AWS. Eventually, all jobs transfer data back to archive.
2. Finished jobs transfer output to S3, the scalable storage service at AWS. Because of its scalability, all transfers can happen approximately at the same time and all VM can terminate afterwards: there is no cost due to idle jobs waiting to transfer data. Storing data in S3, however, has a cost depending on data size and storage usage time. All data is transferred asynchronously to the archive, then the data can be erased.

In our model, depending on the specific values of the parameters, sometimes the cost of strategy (1) is smaller than (2), sometimes vice versa. For example, with 1,000 VM each transferring a 2 GB output, considering a bandwidth to storage of 10 Gbps, strategy (1) costs \$302 and strategy (2) \$275. With the same conditions but with an output size of 1 GB per VM, the costs reverse, with strategy (1) costing \$155 and strategy (2) \$151.

This investigation, did not consider costs reductions on outbound data transfer from AWS through ESNNet (the DOE research network provider), which was established at a later time. This study can be further refined and the strategy reevaluated as the costs of AWS change. It seems clear, however, that using S3 can be cost effective, depending on the circumstances and may be worth implementing for year-3 of our collaboration. Making use of S3 caching, in fact, will require changes to our data handling software.

MPI on Virtual Clusters

In year 1, we enabled the use of InfiniBand interconnectivity from Virtual Machines for node-to-node inter-process communication [13,19]. This year we worked with Prof. Yousung Jung from KAIST to evaluate the deployment with out-of-the-box network settings. The evaluation uncovered that further tuning of the communication layer is necessary to use the infrastructure for heavily parallel jobs. As a benchmark, Prof. Jung used two scientific applications based on quantum chemistry and quantum mechanical periodic solid models, using the Message Passing Interface (MPI) for inter-process communication. When run on a local reference cluster, tuned for efficient use of the InfiniBand interface, the running time of the overall computation was reduced almost linearly by growing the number of allocated processors (e.g. for quantum chemistry calculations, 11,693 seconds for 8 cores in 1 node vs. 5,354 seconds for 16 cores from 2 nodes). On FermiCloud, the same result could not be reproduced as the computational time increased by adding cores from different nodes, exposing that the inter-process communication dominated the computational time. Although standard benchmarks (HPL Linpack) on FermiCloud showed that virtualized InfiniBand through SR-IOV drivers had performance equivalent to bare metal especially for large messages, the experience running scientific code demonstrated that further tuning of the communication layer is necessary to efficiently support generic scientific applications. We note that Infiniband work that was done by our student Tiago Pais and reported in the first year of the CRADA has now been included as part of a larger journal article by I. Sadooghi et al [13], which is a comparative study of launch times and HPC performance between AWS and FermiCloud; this has been submitted to the IEEE Transactions on Cloud Computing.

4. QUALITATIVE AND QUANTITATIVE OUTPUT

The collaboration between Fermilab and KISTI achieved the goals in the statement of work for the second year of the CRADA. Broadly, we have achieved two main results:

1. We have developed techniques and methods to increase the scale of resource provisioning to 1,000 VM and the complexity of on-demand services dynamically deployed in support of scientific computation;
2. We have automated and simplified the formatting and distribution of virtual machines across an increasingly broad federation of diverse Cloud providers.

Quantitative considerations:

This year, the collaboration has worked on 5 papers. It has published a paper at the Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS) [9]; one at the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014) [11]; another at the Cloud Federation Management workshop [10], which is part of the Utility and Cloud Computing 2014 conference in London in Dec 2014. It has also submitted another two to the journal “IEEE transactions on Cloud computing” [12,13], receiving positive feedback at this stage of the process. In summary, the two-years program has produced eight papers to date, adding these five to the three published in the first year [6, 7, 19].

The work from year 2 was presented at four talks [11,14,15,16] and is scheduled at three more after Oct 2014 [10,17,18]. Some of these talks will result in additional papers in the proceedings. In summary by early 2015, the overall 2 years program will have been presented at eleven talks, adding these seven to the three from year 1 [19,20,21].

We also made publicly available all documentation produced [23,24,25,26], including this report. In addition, all the code is available from the code repository of the FermiCloud project or github.com. The code consists of the automated virtual machine image format conversion tool [27] and the infrastructure to provision a platform of web caching services based on Squid and Shoal [29]. This code consists of interpreted scripts and puppet manifests that can be used directly without a binary distribution. A third development [28] consists of a plug-in and modified code for the vcluster system [5], developed at KISTI. This code instruments the process of virtual machine provisioning and implements the cost-sensitive provisioning algorithm described above [9, 11].

Qualitative considerations:

Build relationships with cloud facilities in US and Pacific Rim: we have acquired allocations on Amazon Web Service, Google Computational Engine, and Microsoft Azure and we have started exploring potential fare reductions applicable to research institutions. In particular, AWS has recently agreed to discount the cost for outgoing network traffic through ESNet (our main research network provider) and we are in the process of enabling this feature for our traffic. We are also discussing potential grant funding from AWS and Microsoft to bootstrap the use of Cloud computing for scientific computation at the next scalability level (beyond 1,000 concurrent VM). At the same time, we continued our collaboration with Rackspace, University of Texas at San Antonio, and University of Wisconsin at

Madison, acting as stakeholders in the joint work to integrate the native OpenStack “NOVA” interface with HTCondor. This work will enable the native integration of OpenStack deployment, including University of Notre Dame, in the Federation.

Identify and begin to fix potential interoperability problems in cloud Application Programming Interfaces: We have identified defects in the use of the Google Computational Engine API from HTCondor and reported the problems to the HTCondor and Google teams. In addition, we’ve uncovered defects in the OpenNebula V4 implementation for which we’ve implemented a work around and reported the problem the development teams.

Create a virtual infrastructure able to interoperate with leading commercial and scientific cloud facilities: through the demonstration of running scientific workflows for the NOvA experiment on 1,000 virtual machines deployed on FermiCloud and AWS, we have effectively created a federated virtual facility for our scientific stakeholders. During the course of the past year, three other Fermilab experiments, the Dark Energy Survey, the MicroBooNe experiment, and the mu2e experiment, have also made use of on-site FermiCloud batch resources, easily leveraging the work that was done to enable NOvA.

5. BUDGET ALLOCATION

- A. Fermilab-funded effort: TOTAL INDIRECT COSTS Fermi Research Alliance, LLC (FRA) / Fermilab FY2014 provisional indirect cost rate is currently 70.00% (Salaries – SWF), 15.50% (Travel), and 21.28% (Other Material & Services – M&S) of Modified Total Direct Cost, in accordance with Fermilab's contract with the Fermi Research Alliance, LLC (FRA) and the Department of Energy.

The budgeted amount to contribute \$100,000 of direct costs (equivalent to \$170,000 with indirect cost) was estimated at 7.5 FTE-months. The table below shows effort until the end of September. We estimate for October an effort of at least ¼ FTE-month, thus the budget estimates were met.

| PERSON | ADJUSTED EFFORT, FTE-Months (to SEP 30, 2014) |
|-------------------------|---|
| Bernabeu Altayo, Gerard | 0.76 |
| Garzoglio, Gabriele | 0.45 |
| Kim, Hyun Woo | 3.41 |
| Pregonow, Nicholas | 0.22 |
| Timm, Steven | 2.42 |
| TOTAL | 7.26 |

- B. Obligated funds provided by KISTI as of Oct 2014 - Indicative report

| | |
|--|------------------|
| 3 IIT students for the summer (Xu Yang, Sandeep Palur, Hao Wu) | \$33,540 |
| 1 INFN student for the summer (Alessio Balsini) (labor, housing, transportation) * | \$6,000 |
| 1 consultant for the summer (Kirk Shallcross) | \$32,800 |
| Computing cycles at Amazon Web Services * | \$1,579 |
| Travel * | \$5,233 |
| Miscellaneous (temp. housing, etc) | \$1,088 |
| TOTAL DIRECT COST | \$80,230 |
| INDIRECT COST (15.50% on Travel; 21.28% on other M&S; 70.00% on SWF) | \$16,770 |
| DOE ADMINISTRATIVE FEE (3%) | \$3,000 |
| INDICATIVE TOTAL | \$100,000 |

* Not all obligated money has yet been invoiced by the vendors or paid out. Some expenses (such as travel to KISTI) will be reconciled in November. The complete financial report will be available on December.

6. REFERENCES

- [1] Pordes, R. et al. (2007). *The Open Science Grid*, J. Phys. Conf. Ser. 78, 012057.doi:10.1088/1742-6596/78/1/012057.
- [2] Kranzlmüller, D., J. Marco de Lucas, and P. Öster. "The European Grid Initiative (EGI)." In *Remote Instrumentation and Virtual Laboratories*, pp. 61-66. Springer US, 2010.
- [3] Virtual machine interoperability documentation: <http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=5208>

- [4] Sfiligoi, I., Bradley, D. C., Holzman, B., Mhashilkar, P., Padhi, S. and Wurthwein, F. (2009). *The Pilot Way to Grid Resources Using glideinWMS*, 2009 WRI World Congress on Computer Science and Information Engineering, Vol. 2, pp. 428–432. doi:10.1109/CSIE.2009.950.
- [5] Seo-Young Noh, Steven C. Timm, Haeng-jin Jang: *vcluster: A Framework for Auto Scalable Virtual Cluster System in Heterogeneous Clouds*, in Cluster Computing (2013).
- [6] Hao Wu, Shangping Ren, Gabriele Garzoglio, Steven Timm, Gerard Bernabeu, Hyun Woo Kim, Keith Chadwick, Seo-Young Noh, Haeng-Jin Jang, *Automatic Cloud Bursting Under FermiCloud*, ICPADS CSS workshop, Seoul, 2013, published in IEEE Xplore Digital Library, DOI: 10.1109/ICPADS.2013.121
- [7] S. Timm, K. Chadwick, G. Garzoglio, *Grids, Clouds and Virtualization at Fermilab*, accepted in the Proceedings of the Journal of Physics: Conference Series by IOP Publishing, 2013.
- [8] P. Mhashilkar, A. Tiradani, B. Holzman, K. Larson, I. Sfiligoi, M. Rynge, *Cloud Bursting with Glideinwms: Means to satisfy ever increasing computing needs for Scientific Workflows*, accepted in the Proceedings of the Journal of Physics: Conference Series by IOP Publishing, 2013
- [9] Hao Wu, Shangping Ren, Steven Timm, Gabriele Garzoglio, Seo-Young Noh, *Overhead-Aware-Best-Fit (OABF) Resource Allocation Algorithm for Minimizing VM Launching Overhead*, 7th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS) 2014, Nov 2014, New Orleans, Louisiana, USA
- [10] Hyunwoo Kim, Steve Timm, *X.509 Authentication/Authorization in FermiCloud*, IEEE 1st International Workshop on Cloud Federation Management, Dec 2014, London, UK
- [11] Hao Wu, Shangping Ren, Gabriele Garzoglio, Steve Timm, Gerard Bernabeu, Seo-Young Noh, *Modeling the Virtual Machine Launching Overhead under Fermicloud*, 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014), May, 2014, Chicago, IL, USA
- [12] Hao Wu, Shangping Ren, Gabriele Garzoglio, Steve Timm, Gerard Bernabeu, Keith Chadwick, Seo-Young Noh, *A Reference Model for Virtual Machine Launching Overhead*, submitted in 2014 to the IEEE Transactions on Cloud Computing.
- [13] Sadooghi, Iman; Hernandez Martin, Jesus; Li, Tonglin; Brandstatter, Kevin; Zhao, Yong; Maheshwari, Ketan; Raicu, Ioan; Pais Pitta de Lacerda Ruivo, Tiago; Garzoglio, Gabriele; Timm, Steven, *Understanding the Performance and Potential of Cloud Computing for Scientific Applications*, submitted in 2014 to IEEE Transactions on Cloud Computing TCC-2013-11-0278.R1
- [14] G. Garzoglio, *On-demand Services for the Scientific Program at Fermilab*, International Symposium on Grids and Clouds 2014 (ISGC 2014), March 2014, Taipei, Taiwan
- [15] S. Timm, G. Garzoglio, *FermiCloud On-demand Services: Data-Intensive Computing on Public and Private Clouds*, HEPiX Spring 2014 Workshop, May 2014, Annecy-le-Vieux, France
- [16] S. Timm, G. Garzoglio, *FermiCloud On-demand Services: Data-Intensive Computing on Public and Private Clouds*, Computing Technique Seminar at CERN, May 2014, Geneva, Switzerland
- [17] S. Timm, *Authentication, Authorization, and Federation in OpenNebula with FermiCloud*, OpenNebula Conf 2014, Dec 2014, Berlin, Germany
- [18] S. Timm, et al, *Cloud services for the Fermilab scientific stakeholders*, submitted to Computing in High-Energy Physics 2015 (CHEP15), Apr 2015, Okinawa, Japan
- [19] Tiago Pais Pitta de Lacerda Ruivo, Gerard Bernabeu, Gabriele Garzoglio, Steve Timm, Hyunwoo Kim, Seo-Young Noh, Ioan Raicu, *Exploring Infiniband Hardware Virtualization in OpenNebula towards Efficient High-Performance Computing*, 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014), May, 2014, Chicago, IL, USA
- [20] Timm, S. (2013, Sep 23) *High Throughput and Resilient Fabric Deployments on FermiCloud*, invited talk at ISC Cloud 2013 symposium, Heidelberg, Germany. <https://cd-docdb.fnal.gov:440/cgi-bin/ShowDocument?docid=5202>
- [21] Timm, S. (2013, Sep. 24) *Enabling Scientific Workflows on FermiCloud using OpenNebula*, keynote talk at OpenNebulaConf 2013, Berlin, Germany. <https://cd-docdb.fnal.gov:440/cgi-bin/ShowDocument?docid=5203>
- [22] Daniel van der Ster, Arne Wiebalck, *Building an organic block storage service at CERN with Ceph*, presented at the 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013), pub. Journal of Physics: Conference Series 513 (2014) 042047, doi:10.1088/1742-6596/513/4/042047
- [23] Fermilab DocDB: Xu Yang - Ceph Documentation – <http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=5428>
- [24] Fermilab DocDB: Sandeep Palur - Squid and Shoal Server Documentation – <http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=5428>
- [25] Fermilab DocDB: Automated Image Format Conversion from FermiCloud to AWS - Documentation – <http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=5428>

- [26] Fermilab DocDB: Alessio Balsini work on Google and Microsoft Cloud – <http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=5428>
- [27] Fermilab DocDB: Automated Image Format Conversion from FermiCloud to AWS - Code – <http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=5428>
- [28] Code repository at <https://github.com/philip-wu5/project/tree/fermi> ; Printout at the Fermilab DocDB: Hao Wu - Cost-sensitive Provisioning Algorithm - Code – <http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=5428>
- [29] Fermilab DocDB: Sandeep Palur - Coordinated Workflows with Squid - Code – <http://cd-docdb.fnal.gov/cgi-bin/ShowDocument?docid=5428>

Modeling the Virtual Machine Launching Overhead under Fermicloud

Hao Wu^{*§}, Shangping Ren^{*}, Gabriele Garzoglio[†], Steven Timm[†], Gerard Bernabeu[†], Seo-Young Noh[‡]

Abstract—FermiCloud is a private cloud developed by the Fermi National Accelerator Laboratory for scientific workflows. The *Cloud Bursting* module of the FermiCloud enables the FermiCloud, when more computational resources are needed, to automatically launch virtual machines to available resources such as public clouds. One of the main challenges in developing the cloud bursting module is to decide *when* and *where* to launch a VM so that all resources are most effectively and efficiently utilized and the system performance is optimized.

However, based on FermiCloud’s system operational data, the VM launching overhead is not a constant. It varies with physical resource (CPU, memory, I/O device) utilization at the time when a VM is launched. Hence, to make judicious decisions as to *when* and *where* a VM should be launched, a VM launch overhead reference model is needed. The paper is to develop a VM launch overhead reference model based on operational data we have obtained on FermiCloud and uses the reference model to guide the cloud bursting process.

I. INTRODUCTION

Cloud technology has been benefiting general purpose computing for quite some years. The *pay-on-demand* model brought by cloud computing allows companies to avoid over provision at its early project development stage. As the cloud technology develops, many scientific research institutions have migrated their research from traditional grid and distributed computing platform to the cloud computing environment. These research areas include life science [13], astronomy [15] and earthquake research [10], to name a few.

One successful example of using cloud computing technology is the STAR project on Relativistic Heavy-Ion Collider at the Brookhaven National Laboratory [1], [2]. The STAR project studies the fundamental properties of nuclear matter which only exist in a high-density state called a Quark Gluon Plasma [1]. Because of resource shortage from the local grid service, the STAR team started to collaborate with the Nimbus team at Argonne National Laboratory to migrate its experiment to a computer cloud. The Nimbus tools enable virtual machines in private cloud to be deployed on Amazon EC2.

One of the advantages a computer cloud has over traditional grid computing is that the resource utilization of the underlying

infrastructure can be significantly improved by deploying different tasks on the same physical computer node. In addition, computation power can also be dynamically allocated to tasks when more resources are needed by the tasks. The other benefit of using a computer cloud over a grid is that a cloud has “unlimited” resources – when the private cloud is fully occupied, cloud bursting techniques can temporarily acquire external resources from, for instance, public clouds to fulfill the need.

Fermi National Accelerator Laboratory (Fermilab), as a leading research institution in the high energy physics (HEP) field, started to build a private computer cloud, the FermiCloud, in 2010. The FermiCloud has successfully served the HEP experiments since its establishment. The cloud bursting tool vCluster [8] is developed to automatically allocate resources for scientific workflows from both FermiCloud and public clouds such as Amazon EC2. However, how to dynamically allocate resources for the scientific workflows that reduces the average response time of scientific workflows as well as entire system’s operational cost is a research and also an engineering challenge yet to be overcome.

The resource allocation problem in cloud computing started to draw more attention in the research community in recent years [11], [6]. However, most of the research in the resource allocation area assumes that the virtual machine launching overhead is negligible. As a result of this assumption, neither the launching overhead nor the dependency between the overhead and resource utilization are taken into consideration in designing their resource allocation algorithms. However, our production line operation data indicates that the virtual machine launching overhead can have significant variations.

The VM launching overhead has two aspects, i.e. (1) it consumes system resources while it is launched and (2) it takes time to complete the process of the launch. Both of these types of overhead can impact the system’s performance. More specifically, VM launching consumes a significant amount of CPU and disk IO resources, leading to a high system CPU and disk IO utilizations at the time of launching. If each host computer in the private cloud happens to launch a new virtual machine at the same time, due to high system utilization caused by VM creations, the computer cloud may consider all the hosts fully occupied and decides to perform cloud bursting and create the virtual machine on an external public cloud. Such additional cost is unnecessarily rendered and could be prevented if we have a VM launch overhead reference model.

Furthermore, if a task requires an additional virtual machine in order to complete its work, but the virtual machine takes a much longer time to complete its launching than expected, it is possible that the task has already finished its

^{*}Illinois Institute of Technology, 10 W 31st street, 013, Chicago, IL, USA, {hwu28, ren}@iit.edu. The research is supported in part by NSF under grant number CAREER 0746643 and CNS 1018731.

[†]Fermi National Accelerator Laboratory, Batavia, IL, USA, {garzogli, timm, gerard1}@fnal.gov

[‡]National Institute of Supercomputing and Networking, Korea Institute of Science and Technology Information, Daejeon, Korea, rsyoung@kisti.re.kr

[§]Hao Wu works as an intern in Fermi National Accelerator Laboratory, Batavia, IL, USA

work before the virtual machine is ready for executing the task. This again leads to resource waste and an added cost due to without a VM launching overhead reference model.

In this paper, we are to (1) study the VM launching overhead behavior based on real operational data obtained from FermiCloud; (2) develop a reference model for virtual machine launching overhead from both timing and utilization perspectives; and (3) evaluate the accuracy of the developed reference model.

The rest of the paper is organized as follows: Section II discusses related work. Section III analyzes the virtual machine launching overhead through a large amount of experiments on FermiCloud. Section IV presents a reference model for virtual machine launching overhead. Section V evaluates the accuracy of the proposed model. We conclude the work in section VI.

II. RELATED WORK

Lots of researches have been done on evaluating the cloud performance and modeling cloud. One of the most influencing cloud modeling tool is CloudSim [6] developed by CLOUDS lab from the University of Melbourne. The CloudSim is a java based cloud simulation tool that supports modeling and simulation of large scale cloud computing environments. It provides a cloud modeling that models cloud infrastructure physical machines', and virtual machines' characteristics and behaviors, a cloud market modeling that models the cost of resources, a network modeling that models the network behavior of inter-networking of clouds, a cloud federation modeling that models the communication between clouds, a power consumption modeling that models the power consumptions in the datacenter, and a resource allocation modeling that models virtual machine allocation policies. The CloudSim provides a relative comprehensive modeling tool that covers almost all the basic elements under a cloud environment.

Recently, Huber *et al.*'s work evaluated the virtualization performance and proposed a virtualization overhead model [9]. In their work, they mainly focus on two virtualization platforms, XenServer and VMware ESX. They test the performance downgrades that is brought by the virtualization. They test the CPU, memory, disk IO, and network performance degradations on both XenServer and VMware ESX platforms. Based on the experiments, they categorized the virtualization performance influencing factors into four major categories: virtualization type, hypervisor's architecture, resource management configuration and workload profile. However, Huber's model does not consider virtual machine launching overhead, it only provides the computation overhead that is brought by the virtualization.

Researchers adapted the above cloud models and cloud simulations tools and proposed significant contributions to resource allocation on clouds, such as resources provisioning algorithms from QoS perspective [7], from service providers' profit perspective [14] and from energy consumption perspective [5]. Recently, Mengxia Zhu *et al.* proposed a cost effective scheduling for scientific workflow under cloud environments [11]. Their scheduling algorithm aims to shorten the application's response time and reduce the energy consumption simultaneously by considering the virtual machine launching overhead.

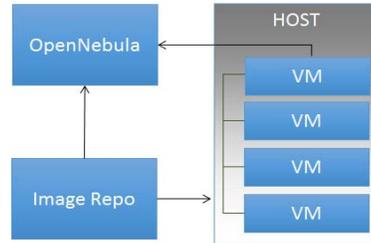


Fig. 1: System Architecture

However, they did not consider the variation of the virtual machine launching overhead. Not only Zhu's work, including CloudSim, few other researchers have taken virtual machine launching overhead variation as a key variable for designing resource allocation algorithms. However, the virtual launching overhead may have a large variation that may cause significant impact on the resource allocation process. In the FermiCloud bursting project, the design of the resource allocation algorithm aims to automatically allocate resources for the scientific workflows that need extra computational resources. If the virtual machine launching overhead is not well modeled and calculated, the system utilization and efficiency may be pulled down dramatically. Furthermore, it may cause resource and energy waste. Hence, we need an accurate mathematical model for the virtual machine launching overhead. The reference model we propose in the paper is drawn from a large amount of experimental observations. The formal analysis of the experiments is discussed in the next section.

III. ACTUAL VM LAUNCHING OVERHEAD ON FERMICLOUD

In this section, we study the patterns of the virtual machine launching overhead based on the virtual machine operations in the FermiCloud production cloud environment.

A. FermiCloud System Configuration

The FermiCloud uses OpenNebula [3], [12] as its cloud platform. As illustrated in Fig. 1, the system has an OpenNebula front-end server that manages the entire cloud infrastructure, an image repository that stores all VM images, and a set of host machines on which VMs are deployed.

The OpenNebula front end server has 16-core Intel(R) Xeon(R) CPU E5640 @ 2.67GHz, 48GB memory. Fifteen homogeneous hosts are used for the experiments. All the fifteen hosts are configured with 8-core Intel(R) Xeon(R) CPU X5355 @ 2.66GHz and 16GB memory. All these machines are connected through high speed Ethernet.

Under OpenNebula [4], the VM launching process consists of four major states. Fig. 2 illustrates the state change during a VM launching process in OpenNebula [4]. In particular, when a user creates a new VM, the VM enters the *pending* state. In the pending state, the cloud scheduler decides where to deploy the VM. Once the VM has been deployed on a specific host, it enters into the *prolog* state in which all VM related files (images in our case) are transferred from the image repository to the host machine. After all the files are copied to the host, the VM enters the *boot* state, during which it is booted from the host. Finally, after the VM is successfully booted, it enters

into the *running* state. Once a VM is in its running state, it is ready to execute tasks.

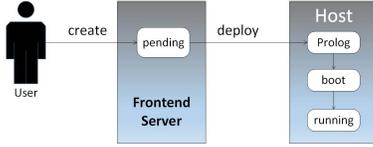


Fig. 2: VM Launching State Diagram[4]

B. Base VM Launching Overhead

We first obtain the baseline utilization overhead of launching a new VM. In order to get the baseline utilization overhead of launching a new VM, we let all the host machines in the private cloud be empty, i.e. have no application being deployed, before launching a VM. Each time, a single VM is launched. All the launched VMs are configured with one virtual core and 2 GB memory. We retrieve the exact virtual machine launch time from each virtual machine’s system log. The experiment is repeated ten times.

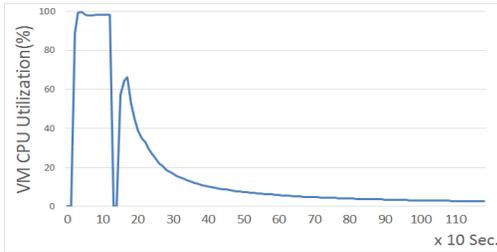


Fig. 3: VM Launching CPU Utilization Overhead

Fig. 3 shows the average system CPU utilization variation in the process of launching a VM. The x-axis represents the time instance of sampling points. The sampling interval is every 10 seconds and it is used for all the other experiments. The y-axis indicates the host machine’s CPU utilization consumed by the process of a single virtual machine. For convenience, throughout the paper, we refer the CPU utilization consumed by a VM on a host machine as the VM’s CPU utilization.

Since the host machine consists of multiple CPU cores, the VM’s CPU utilization represents a single CPU utilization consumption by the VM’s process. If the VM’s CPU utilization exceeds 100%, it means the VM occupies more than one CPU cores.

As shown in Fig. 3, there are two different CPU utilization variation trends. The first part, from time 0 to time 14, is due to the *prolog* procedure, which fully consume a CPU until the image is copied to the host. The second part, from time 15 to time 120, is due to the *booting* procedure. Once the booting procedure starts, it immediately reaches a high CPU utilization and the CPU utilization slowly decreases after the services are started. When VM’s CPU utilization remains close to constant, the VM is considered to be in *running* state. We denote the VM’s CPU utilization variation trends in Fig. 3 as the baseline VM launching CPU overhead and use it as the base for comparisons in following experiments.

C. CPU Utilization Impact

The above experimental data indicates that VM launching overhead causes system CPU utilization to change on an empty machine. In reality, most of the VMs are not launched on empty hosts. Thus, it is interesting to see how the system utilization influences the VM launching process. The following sets of data are obtained to investigate the influence of system utilization on the VM launch overhead.

1) *Different system utilization:* In this experiment, VMs are launched under different system utilizations. Fig. 4 depicts the system CPU utilization change when VMs are launched under different system CPU utilizations. It indicates that every time a new VM is launched, the system utilization is suddenly increased to a high level and remains at the level for a while before it goes down.

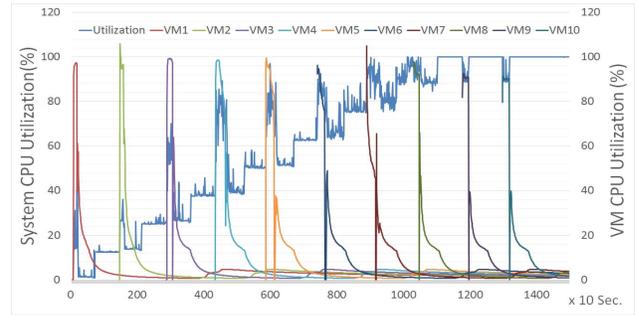


Fig. 4: VM Launching Overhead under Different System Utilization

Fig. 5 shows the booting utilization variations for different VMs. As indicated in the figure, the variations converges to the same value. In fact, the variation of the booting process is at most 10% while the peak utilization of booting a VM has a variation of 40%.

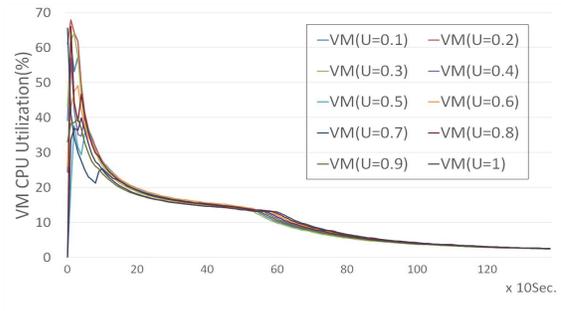


Fig. 5: VM Booting Overhead

To clearly distinguish the VM launching overhead change from the system utilization change, we extract individual VM launching overheads from Fig. 4 and depict the results in Fig. 6. Fig. 6 clearly demonstrates that the time of the VM *prolog* process changes quite significantly when launching VMs under different system utilizations.

Table I summarizes the variations. “Util” column indicates the system’s CPU utilization when a new virtual machine is created; the column of “Prolog” represents the time increases for the prolog process when it is compared with the baseline virtual machine prolog time; the column of “Boot” represents

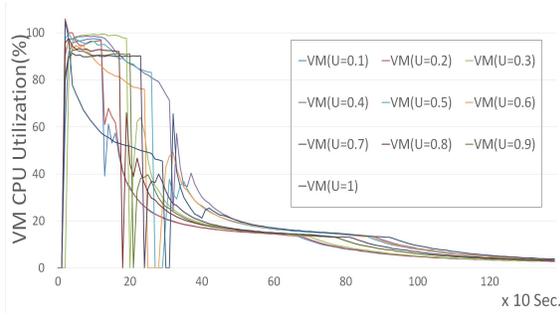


Fig. 6: VM Launching Overhead Comparison

| Util (CPU) | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|------------|-----|------|------|-------|-------|-------|------|------|-------|-------|
| Prolog | 0 | 0 | 0.38 | 0.61 | 0.69 | 0.84 | 1.23 | 1.23 | 1.46 | 1.46 |
| Boot | 0 | 0 | 0 | 0.03 | 0.05 | 0 | 0.12 | 0.03 | 0.01 | 0.03 |
| Peak Util. | 0 | 0.10 | 0.02 | -0.05 | -0.38 | -0.19 | 0.07 | 0.07 | -0.35 | -0.34 |

TABLE I: VM Launching Overhead Comparison

the time increases for the booting process compared with the baseline virtual machine booting time; and the “Peak Util.” column represents the VM’s peak CPU utilization increases for the booting process compared with the baseline virtual machine’s booting CPU peak utilization.

As table I indicates, when the system’s CPU utilization reaches 100%, launching a new VM takes 1.5 times compared with launching a VM on an empty host. However, if the VM booting process is isolated out, surprisingly, all the booting processes take similar amount of time no matter how high the system utilization is as shown in Fig. 5.

2) *Different VM configuration:* For previous experiments, all VMs have the same configuration. To know how the configurations of the VMs may impact the VM launching overhead, we repeat the above experiments but with different VM configurations (2 virtual cores and 4 GB memory). Table II lists the results of the experiments. The incremental times are compared with the baseline VMs from table I.

Intuitively, the VM prolog time will not change much as the same VM image is used for the VMs and the only changes are the number of CPU cores and the size of memory. The results confirm that the prolog times remain the same trends as the baseline VMs. Furthermore, without a surprise, the VM booting processes also take the same amount of time as the single core VMs do. Hence, we can conclude that the VM configurations do not have significant impact on the VM launching overheads.

D. Disk IO Utilization Impact

The baseline experiments show that the VM launching overhead consists of two parts, one is the image transferring and copying process and the other is the VM booting process. From the above CPU utilization experiments, we have learned

| Util (CPU) | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|------------|-------|-------|------|------|-------|------|-------|-------|-------|-------|
| Prolog | 0 | 0 | 0.15 | 0.23 | -0.30 | 0.38 | 0.84 | 1.69 | 1.69 | 1.76 |
| Boot | -0.03 | -0.01 | 0 | 0 | -0.05 | 0 | 0.07 | 0.05 | 0.07 | 0.05 |
| Peak Util. | 0.24 | 0.26 | 0.02 | 0.06 | -0.04 | 0.07 | -0.04 | -0.14 | -0.31 | -0.31 |

TABLE II: VM Launching Overhead Comparison under Different VM Configurations

that the VM booting overhead does not change much when the system utilization changes. However, it is possible that during the image copying process, the overhead may be influenced by the disk IO operations and network traffic. We discuss each below.

1) Launching overhead under different IO utilization:

Since disk IO operations also consume CPU resources. In order to focus on the impact of disk IO utilization variations, we keep the system’s CPU utilization as low as possible. As illustrated in Fig. 7, even when the IO utilization reaches 100%, the CPU utilization still remains at a relatively lower level (less than 20%). Similar to the VM’s CPU utilization, we refer VM’s IO utilization as the host machine’s disk IO utilization consumed by the single VM and system’s IO utilization as the host machine’s total utilization.

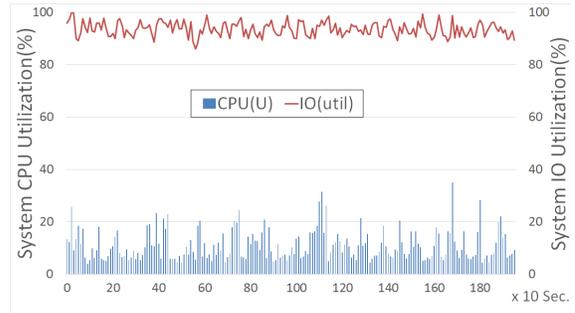


Fig. 7: IO Utilization v.s. CPU Utilization

We use the same VM configuration as used for the baseline experiments. VMs are launched under different system disk IO utilization and the results are shown in Fig. 8. As Fig. 8 indicates the VM launching overhead has large variations when VM starts under different disk IO utilizations. If we isolate the VM booting overhead, as shown in Fig. 9, we can clearly notice that the VM booting overhead under different IO utilizations has significant changes when the disk IO utilization changes. The VM’s booting time is almost doubled when launched under fully IO utilized situation when it is compared to the one launched under an idle host.

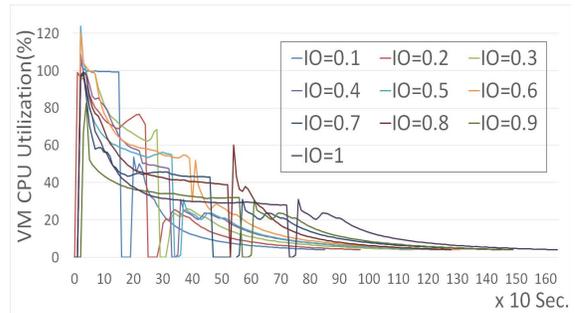


Fig. 8: VM Launch Overhead Comparison Under Different IO Utilization

Table III gives the detailed VM launching overhead increments under different disk IO utilizations in comparison with the baseline overhead. In particular, when compared with the baseline launching overhead, the prolog process takes a much longer time to copy images to the host machine when the host machine’s disk IO utilization is high. When the host machine’s

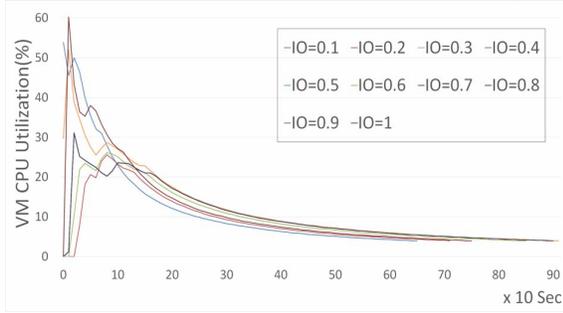


Fig. 9: VM Booting Overhead Comparison Under Different IO Utilization

| Util(io) | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|------------|-------|-------|-------|-------|-------|-------|-------|------|-------|-------|
| Prolog | 0.61 | 1.07 | 1.31 | 1.61 | 1.69 | 2.07 | 2.54 | 3.23 | 3.53 | 4.69 |
| Boot | 0.03 | 0.06 | 0.12 | 0.20 | 0.32 | 0.40 | 0.41 | 0.41 | 0.78 | 1.09 |
| Peak Util. | -0.10 | -0.57 | -0.56 | -0.48 | -0.40 | -0.13 | -0.34 | 0.03 | -0.31 | -0.38 |

TABLE III: VM Launching Overhead Comparison under Different IO Utilization

disk IO utilization reaches 100%, it take almost 5 times to copy an image compared with copying an image to idle host machines. Notice that the VM booting processes also take a longer time (almost twice as much) when the host machine has frequent disk IO operations. An interesting finding is that the peak utilization caused by the booting process is much lower when it is compared with the baseline overhead when the host disk IO utilization is high.

2) *Simultaneous Launching Overhead*: Above experiments give the insight of how host machine’s disk IO operations can impact the VM launching overhead. This set of experiments is to investigate if there are mutual influences on launching overhead among the different VMs when multiple VMs are simultaneously launched to the same host machine. Under the same system disk IO utilization, we launch two and later multiple VMs simultaneously and deploy them on the same host machine under different IO utilization. Fig. 10 depicts the results. The data clearly indicates that the *prolog* time for the VM is prolonged significantly (700%) when more than one VMs are launched at the same time.

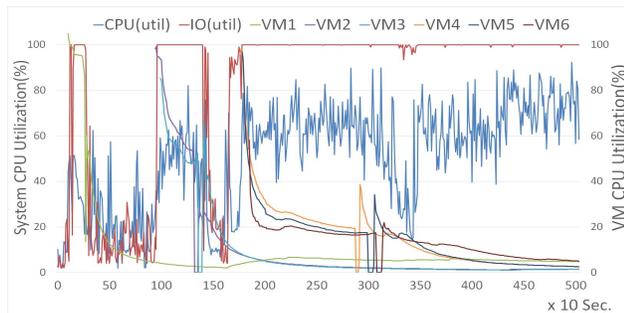


Fig. 10: System Utilization Variation On Simultaneous Launching

Figure 11 illustrates the individual VM launching overhead when multiple VMs are started simultaneously. It is interesting to see that when multiple VMs are launched, the system evenly distributes CPU resources to each VM for the *prolog* processes. The peak utilization of the VM booting process

| No. VMs | 2VMs(U=0) | 3VMs(U=0) | 2VMs(U=1) | 3VMs(U=1) |
|------------|-----------|-----------|-----------|-----------|
| Prolog | 2.15 | 6.92 | 3.07 | 7.84 |
| Boot | 0.21 | 0.80 | 0.70 | 0.88 |
| Peak Util. | -0.20 | -0.36 | -0.40 | 0.08 |

TABLE IV: Simultaneous VM Launching Overhead Comparison under Different IO Utilization

also decreases proportionally to the reduction of the number of simultaneously launched VMs.

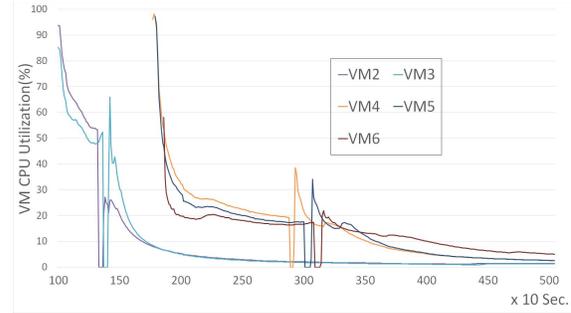


Fig. 11: Simultaneous VM Launching Overhead Comparison

As depicted in Fig. 11, the VM *prolog* process time increases as the system disk IO utilization increases. It is because the *prolog* process not only competes with the other newly launched VMs, it also competes with other VMs that are running. Table IV shows the statistic comparisons of simultaneous VM launching overheads under different system IO utilizations. The first row of the table indicates the number of VMs created simultaneously and the system IO utilization on which these VMs are deployed. As shown in the table, when the system disk IO is idle, simultaneously launching two VMs takes twice the time to copy the images to the host compared with the baseline copying process; and seven times the time to transfer an image to the host when three VMs are launched in the meantime. When the disk IO is fully utilized, the time of copying an image is three times as much for two simultaneous launches and eight times as much for three simultaneous launches compared with the baseline image transferring process. While the booting time for each VM is also increased when the disk IO is fully utilized, the booting time increase is rather much slower compared with the *prolog* process — it is only 1.9 times as much compared to the baseline booting time.

E. Network Traffic Impact

As discussed above, the image copying process may also be influenced by the network traffic. In this section, we discuss the impact of network traffic on the VM launching overhead. We consider two scenarios for the experiments, the impact of downstream and upstream bandwidth utilization on the VM launching overhead, respectively.

1) *Network downstream bandwidth Impact*: We first test the influences when the downstream bandwidth is utilized for other running VMs on the hosts. Intuitively, the downstream bandwidth will affect the image transferring time. If the available spare bandwidth for the newly launched VM is relatively small, the bandwidth then will become the bottleneck for launching VMs. However, after VM images are copied to the host, the booting process will not be affected.

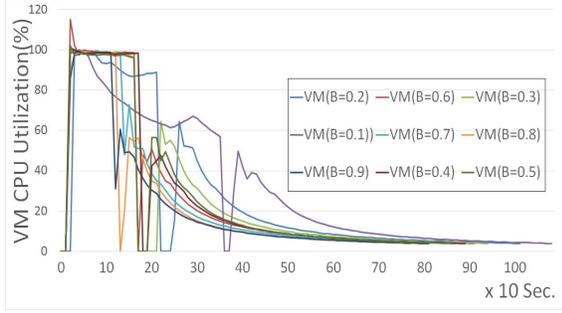


Fig. 12: VM Launching Overhead Comparison Under Different Network Downstream Bandwidth

| Bandwidth (down) | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
|------------------|------|-------|-------|-------|-------|-------|-------|-------|-------|
| Prolog | 2.16 | 1.08 | 0.66 | 0.83 | 0.66 | 0.50 | 0.16 | 0.16 | 0 |
| Boot | 0 | 0.10 | 0.04 | 0.05 | -0.02 | 0.02 | -0.08 | 0 | -0.02 |
| Peak Util. | 0.07 | 0.075 | -0.20 | -0.05 | -0.15 | -0.05 | 0.21 | -0.05 | 0.01 |

TABLE V: VM Launching Overhead Comparison under Downstream Bandwidth

The overall VM launching overhead comparison is dispatched in Fig. 12. It is not difficult to see that when the downstream bandwidth is highly utilized, the prolog process of launching a VM is increased. However, without a surprise, all the VM booting processes take almost the same amount of time as indicated in Fig. 13.

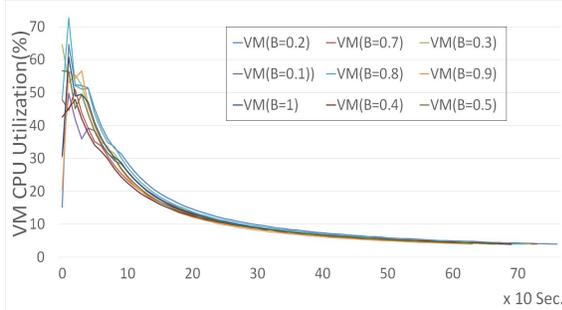


Fig. 13: VM Booting Overhead Comparison Under Different Network Downstream Bandwidth

Furthermore, as shown in table V, even when the bandwidth is 90% utilized, the prolog time is only twice of the baseline prolog time; and if the bandwidth utilization is low, the prolog time decreases quickly and remains at a steady level when the utilization reaches 0.3. The reason for such prolog time variation is that the total network downstream bandwidth is very large compared with the disk IO bandwidth. When the spare bandwidth available for transferring an image becomes larger than the available disk IO bandwidth, the disk IO bandwidth becomes the bottleneck. Hence, the minimum available network downstream bandwidth and disk IO bandwidth decides the image transferring overhead.

2) *Network upstream bandwidth Impact:* Evaluating the impact of upstream bandwidth utilization on the VM launching overhead takes the same steps as for the downstream bandwidth limitations. Intuitively, the upstream bandwidth utilization does not have significant impact on the VM launching process and our data confirms this.

As shown in Fig. 14 and Fig. 15, almost all the VMs' overheads match with each others'. However, there is one

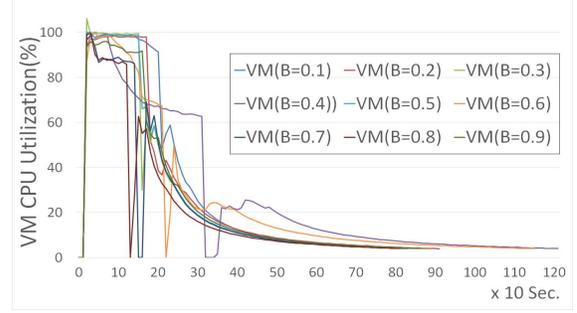


Fig. 14: VM Launching Overhead Comparison Under Different Network Upstream Bandwidth

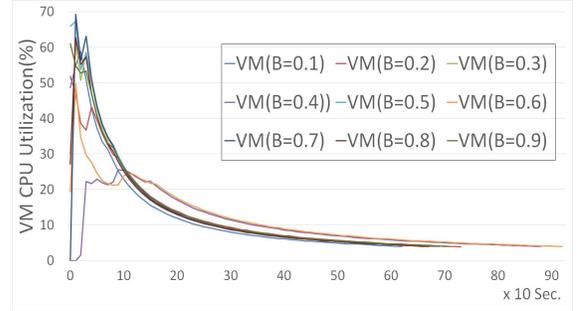


Fig. 15: VM Booting Overhead Comparison Under Different Network Upstream Bandwidth

exception. The VM launched under 40% upstream bandwidth utilization takes an extremely long time for the entire launching process. Our further investigation of the system log indicates that at the time when the VM is launched, the host machine happens to have IO operations for some system critical services.

F. Image Repository Impact

The FermiCloud architecture as shown in Fig. 1 contains an image repository which can also become a bottleneck when large number of VMs are launched simultaneously even when they are deployed on different hosts. In order to evaluate the impact of sudden large number of simultaneous launches on the VM launch overhead, we set up another experiment using the baseline VM configuration. In particular, we launch a VM to a host, and simultaneously launch more VMs to different hosts. Fig. 16 illustrates the overall VM launching overheads when different number of VMs are launched simultaneously. It is obvious that when more VMs are launched simultaneously, the image repository's disk IO/network bandwidth become a bottleneck. In particular, when seven VMs are launched simultaneously, the prolog time increases 3.5 times compared with the baseline prolog time.

Notice that the CPU utilization for the prolog process becomes lower when a VM is launched with more VMs simultaneously. This is because when multiple VMs are launched

| Bandwidth(down) | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
|-----------------|-------|------|-------|-------|------|-------|-------|-------|-------|
| Prolog | 0.61 | 0.38 | 0.23 | 1.53 | 0.23 | 0.76 | 0.23 | 0.07 | 0.31 |
| Boot | -0.10 | 0.05 | -0.01 | 0.27 | 0 | 0.32 | -0.02 | -0.02 | 0.028 |
| Peak Util. | -0.02 | 0.03 | 0.016 | -0.55 | 0.11 | -0.18 | 0.15 | 0.03 | 0.016 |

TABLE VI: VM Launching Overhead Comparison under Upstream Bandwidth

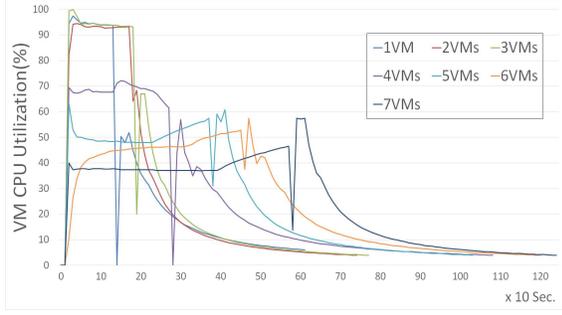


Fig. 16: VM Launching Overhead Under Different Simultaneous Launches on Different Hosts

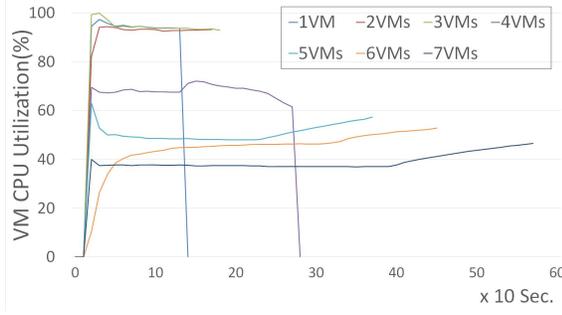


Fig. 17: VM Prolog Overhead Under Different Simultaneous Launches on Different Hosts

together, the disk IO bandwidth/network bandwidth of image repository is evenly distributed among each of them. For each machine, its prolog process does not fully occupy the disk IO utilization, hence the CPU utilization for the prolog process becomes lower.

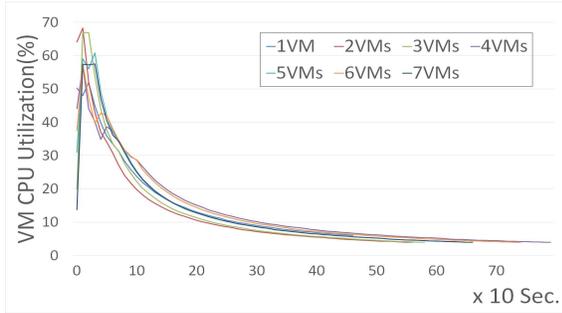


Fig. 18: VM Booting Overhead Under Different Simultaneous Launches on Different Hosts

As shown in Fig. 18 and table VII, the overhead for the VM booting processes remains at the same level as the baseline VM booting overhead.

G. Summary

From these experiments, we can conclude the following:

| Simul. Launches | 1VM | 2VMs | 3VMs | 4VMs | 5VMs | 6VMs | 7VMs |
|-----------------|-------|-------|-------|-------|------|-------|-------|
| Prolog | 0.15 | 0.58 | 0.46 | 1.23 | 1.92 | 2.53 | 3.46 |
| Boot | -0.12 | -0.13 | -0.10 | 0.21 | 0.02 | 0.13 | 0.03 |
| Peak Util. | -0.13 | 0.13 | 0.11 | -0.05 | 0.01 | -0.04 | -0.04 |

TABLE VII: VM Launching Overhead Comparison Under Different Simultaneous Launches on Different Hosts

- VM launching overhead mainly contains two parts: prolog (image copying/transferring) overhead and booting overhead;
- booting overhead is relatively steady, i.e. has less variations, when it is compared to the prolog overhead;
- prolog overhead, on the other hand, has significant variations under different disk IO utilization, network bandwidth utilization on both host machines and image repository; and
- disk IO utilization has significant impact on both prolog overhead and booting overhead.

In the next section, we present a reference model for the VM launching overhead based on the data obtained.

IV. VM LAUNCHING OVERHEAD MODELING

Before we present the reference model for the virtual machine launching overhead in private cloud, we first introduce notations to be used in defining the reference model. In particular, A virtual machine in a private cloud is defined as $v = (f, t, h)$, where f is the image size of the virtual machine, t is the virtual machine launch time and h is the host machine that the virtual machine is to be deployed on. For each host h_i , we denote $V_{h_i} = \{v_1, v_2, \dots, v_n\}$ as the set of virtual machines the host has. In the set V_i , virtual machines are sorted according to their launch time in none decreasing order. The B_n and B_d denote host machine network bandwidth and disk IO bandwidth, respectively; $av_n(h_i, t)$, $av_d(h_i, t)$ and $av_i(t)$ denote the available network bandwidth on host h_i , disk bandwidth on host h_i , and network bandwidth on image repository at time t , respectively.

The proposed reference model contains three different overheads: timing overhead which is the time needed for launching a VM until it is ready to execute tasks; disk IO utilization overhead and CPU utilization overhead. We first model the CPU utilization and disk IO utilization that a single VM consumes on the host machine during the launching process. Then we model the host machine's entire system CPU utilization and disk IO utilization. As discussed in section III, the complete VM launching process mainly consists of two parts: prolog and boot process. We discuss the reference models for these two steps below.

A. Prolog Overhead Model

The prolog overhead we modeled in here also contains three different overheads: timing overhead which is the time needed for transferring an image from image repository to the host machine; disk IO utilization overhead and CPU utilization overhead. Let $AV_{band} = \min\{av_d(h_i, t_i), av_n(h_i, t_i), av_i(t_i)\}$. The image transfer time for virtual machine v_i is defined below:

$$Trans_i = \frac{f_i}{AV_{band} * w * U_s(h_i, t - 1)} \quad (1)$$

where $U_s(h_i, t)$ is the system's CPU utilization that is defined in section IV-E, and w is a constant that represents how much impact that the system's CPU utilization has on the image transferring process.

From the experiments we know that if the disk IO is fully utilized for the image transferring process, the process also fully utilizes one physical core of the host machine. If the disk IO is not fully utilized for the image transferring process, then the CPU utilization is the available disk IO bandwidth proportional to the total IO bandwidth. We first define the base CPU utilization function for image transferring process as follows:

$$U_{tr_base}(i, t) = \frac{1}{1 + e^{-0.5(Trans_i + t_i)(t - t_i)}} - \frac{1}{1 + e^{-0.5(Trans_i + t_i)(t - (Trans_i + t_i))}} \quad (2)$$

The IO utilization consumed by a VM's prolog process is modeled as the IO bandwidth occupied by image transferring process to the total bandwidth. Hence, the IO utilization of transferring an image for virtual machine v_i is modeled as:

$$IO_{tr}(i, t) = \begin{cases} \frac{AV_{band}}{B_a} & t_i \leq t \leq t_i + Trans_i \\ 0 & otherwise \end{cases} \quad (3)$$

Then, the CPU utilization of transferring an image for virtual machine v_i is modeled as:

$$U_{tr}(i, t) = IO_{tr}(i, t) * U_{tr_base}(i, t) \quad (4)$$

B. Booting Overhead Model

The virtual machine booting overhead also refers to the timing overhead and CPU utilization overhead. As once the image is copied to a host, it will not consume any disk IO utilization for the booting process. We consider that there is no disk IO overhead for the virtual machine booting process. The experiments also indicate that the system CPU utilization impact the booting overhead. Hence, we model the CPU utilization overhead for the virtual machine v_i 's booting process as follow:

$$U_b(i, t) = c * \frac{1}{m} e^{-\gamma(1 - IO_s(h_i, t-1))(t - Trans_i)} \quad (5)$$

where c and γ are two constants, m is the number of cores on the host machine and $IO_s(h_i, t)$ represents the system's disk IO utilization at time t . We will formally define the system disk IO utilization in section IV-E.

In OpenNebula, VMs are not immediately ready for use until all the necessary services, such as ssh, are started. As there is no accurate way to tell the actual time when a virtual machine is booted and ready to use unless entering the running virtual machine and check the log, therefore, we base our estimation for the time points on the variation of the virtual machine's CPU utilization consumption. If the virtual machine's CPU utilization consumption remain stable, then we consider the virtual machine is booted and ready to use. We define the time point $t_b(i)$ of a virtual machine v_i is ready to use as:

$$t_b(i) = \max\{t | U'_b(i, t) \leq \epsilon\} \quad (6)$$

where ϵ is the threshold to determine whether the virtual machine's CPU utilization consumption become stable. Then, we can calculate the virtual machine booting time is $t_b(i) - Trans_i$.

C. Virtual Machine Launching Overhead Model

We have formally modeled image transferring overhead and virtual machine booting overhead. Combining the two components together, we derive virtual machine launching overhead functions. In particular, combining equation 4 and equation 5, the virtual machine v_i 's launching CPU utilization function is modeled as:

$$U(i, t) = \begin{cases} U_{tr}(i, t) & t \leq t_{tran} \\ U_b(i, t) & t > t_{tran} \end{cases} \quad (7)$$

Since the virtual machine booting process does not consume any IO utilization, the IO utilization function for virtual machine v_i 's launching process is still equation 3.

The total time needed for launching a virtual machine v_i then can be calculated as image copying time plus virtual machine booting time. It is formally defined as follow:

$$t_{overhead}(i) = t_b(i) - t_i \quad (8)$$

D. Virtual Machine Utilization Consumption Model

The complete virtual machine utilization functions consist of the virtual machine launching overhead utilization functions and the utilization functions after workloads are deployed on the virtual machine. We assume at time $t' \geq t_b(i)$, the virtual machine v_i starts executing tasks; and the CPU and disk IO utilization consumption function of v_i at t' are $U_w(t)$ and $IO_w(t)$, respectively. Then the virtual machine CPU utilization consumption model is defined below:

$$U_c(i, t) = \begin{cases} U_{tr}(i, t) & t \leq t_{tran} \\ U_b(i, t) & t > t_{tran} \\ U_w(i, t) & t \geq t' \end{cases} \quad (9)$$

The virtual machine IO utilization consumption model is defined as:

$$IO_c(i, t) = \begin{cases} IO_{tr}(i, t) & t_i \leq t \leq t_i + Trans_i \\ IO_w(i, t) & t \geq t' \\ 0 & otherwise \end{cases} \quad (10)$$

E. System Utilization Model

We assume that host machines only run virtual machines and all other critical system services consumes a small portion of the system CPU and IO utilization. Then we can calculate the system CPU and disk IO utilization as the summation of the virtual machines' CPU and IO utilization consumptions. The system CPU utilization of host h_i is modeled below:

$$U_s(h_i, t) = \max\{1, \sum_{j=1}^{|V_{h_i}|} \{U_c(j, t)\}\} \quad (11)$$

The system IO utilization of host h_i can be modeled as:

$$IO_s(h_i, t) = \max\{1, \sum_{j=1}^{|V_{h_i}|} \{IO_c(j, t)\}\} \quad (12)$$

V. EVALUATION

We build the reference model for the virtual machine launching overhead from a large amount of real system experimental data. However, we cannot guarantee the accuracy of the model unless we compare the calculated data using the model we built with the real system data and prove the accuracy of the model. In order to measure the accuracy of the proposed reference model, we introduce an evaluation criteria called average utilization difference. We denote N as the total number of sampling points. The average difference is defined as follows:

$$dif = \frac{1}{N} \sum_{i=1}^N |U_r(i) - U_s(i)| \quad (13)$$

where $U_r(i)$ and $U_s(i)$ represents the real data and calculated data at i th sampling point.

Another important criteria needed to be evaluated is the launching time overhead. To check the real time point for the virtual machine that is ready to use, we use the virtual machine system log to check the starting time point of the ssh service. We also calculate the difference between the real VMs' ready time and the calculated ready time to evaluate the accuracy of the proposed model.

We first compare the baseline overhead obtained by calculating the value based on formula 7, and the real data obtained on FermiCloud.

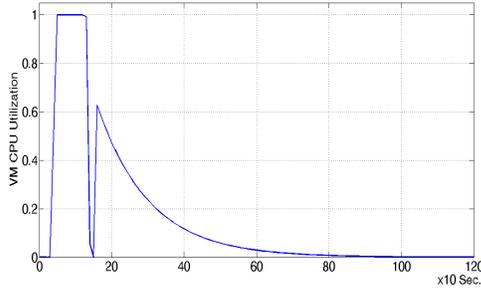


Fig. 19: Baseline VM Launching Overhead using Proposed Model

Fig. 19 draws the CPU utilization during a virtual machine's launching process using the proposed VM launching overhead model. Compare the graph with the utilization variations shown in Fig. 3 as for the baseline virtual machine launching overhead. The calculated data using our proposed model is very close to the real data.

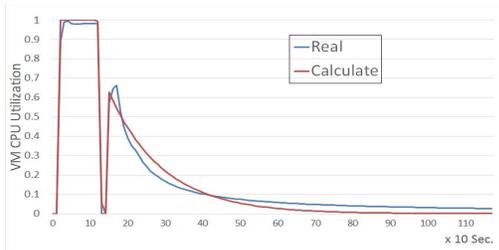


Fig. 20: Baseline VM Launching Overhead Comparison

| | Utilization Difference | Time Difference |
|-----------------|------------------------|-----------------|
| Baseline | 0.0003 | -0.023 |
| 2 Sim. Launches | 0.0446 | -0.051 |
| 3 Sim. Launches | 0.0628 | -0.017 |
| Random Launches | 0.0491 | -0.069 |
| Overall | 0.0392 | -0.040 |

TABLE VIII: Performance of the Proposed Model

If we put two data sets at the same page, as shown in Fig. 20, our model accurately represents the baseline CPU utilization variation. Table VIII gives a more detailed comparison between the real data and calculated data. From the table, we can observe that the difference between the real data and the calculated data using our proposed model is only about 0.03% of the system CPU utilization. The estimated launching overhead calculated by our model is only 2.3% below the actual launching overhead.

We further evaluate when more than one VMs are launched simultaneously. Fig. 21 shows the CPU utilization variation comparison between the real data and calculated data from the reference model, i.e., formula 7, when two virtual machines are launched at the same time. Fig. 21 indicates that the CPU utilization difference between real data and calculated data is 4.46% of the system CPU utilization, launching time overhead is also very close to the actual time, only 6% difference. The detailed analysis is given in table VIII.

We increase the number of simultaneous launches to three VMs. The results are depicted in Fig. 22. The CPU utilization difference between real data and calculated data is 6.28% of the CPU utilization; and the estimated launching time overhead is about 1.7% less than the actual measured value.

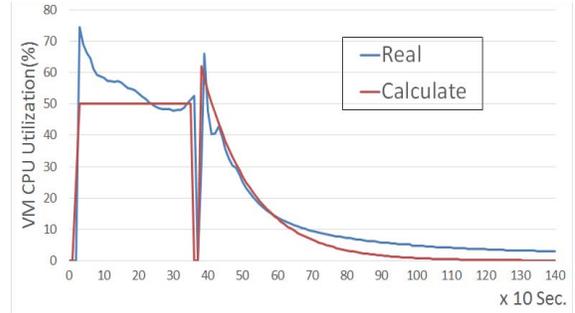


Fig. 21: VM Launching Overhead Comparison with 2 Simultaneous Launches

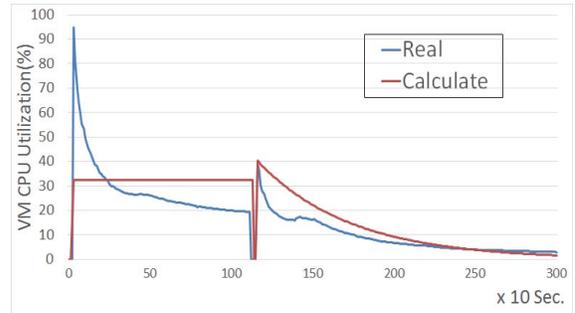


Fig. 22: VM Launching Overhead Comparison with 3 Simultaneous Launches

For the last set of evaluations, we randomly launch multiple

virtual machines under different CPU and IO utilization and at different time instances. We use the reference model to calculate the same scenario for the virtual machine launching process in a real cloud environment. The results are shown in Fig. 23.

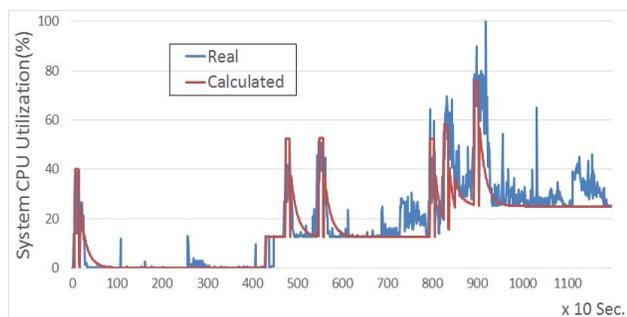


Fig. 23: System Utilization Variation Comparison

As shown in the figure, the two lines are almost merged together. In real environment, there are additional services running on the host machine other than the virtual machines. As a result, they whole system CPU utilization variation is more unpredictable as there are additional CPU utilization consumption in addition to virtual machines. However, the whole system utilization difference between the real data and calculated data is rather small, only 4.91% of the system CPU utilization. When we compare the virtual machine launching time overhead, the difference is still very small, less than 7%.

VI. CONCLUSION

The FermiCloud is a private cloud built by Fermilab for the scientific workflow. The Cloud Bursting project on the FermiCloud enables the FermiCloud, when more computational resources are needed, to automatically launch virtual machines to available resources such as public clouds. One of the main challenges in developing the cloud bursting module is to decide *when* and *where* to launch a VM so that all resources are most effectively utilized and the system performance is optimized. We have found that the VM launching overhead has a very large variation under different system states, i.e. CPU/IO utilizations can have significant impact on cloud bursting strategies. Hence, being able to model accurately the dependency between VM launching overhead and system resource utilization is critical in deciding *when* and *where* a VM should be launched. This paper has studied the VM launching overhead patterns based on data obtained on FermiCloud and presented a VM launching overhead reference model to guide cloud bursting process. To our best knowledge, this is the first reference model for virtual machine launching overhead that incorporates the dynamics and variation during virtual machine launching process. Our next engineering step is to integrate the reference model into the cloud bursting decision algorithms.

It is worth pointing out that during our experiments, we find that virtual machine launching overhead is mainly caused

by the image transferring process. It is not hard to understand that if the image copying/transferring process can be well controlled, the virtual machine launching overhead will become relatively stable and easy to model. As overhead reference model we proposed in this paper consists of two different parts, i.e., prolog overhead and virtual machine booting overhead, the model can easily fit the situation when image transferring process is well managed. We believe that the proposed model is applicable to other private cloud in general.

REFERENCES

- [1] Feature - clouds make way for STAR to shine. <http://www.isgtw.org/feature/isgtw-feature-clouds-make-way-star-shine>.
- [2] Nimbus and cloud computing meet STAR production demands. http://www.hpcwire.com/hpcwire/2009-04-02/nimbus_and_cloud_computing_meet_star_production_demands.html.
- [3] Opennebula. <http://opennebula.org>.
- [4] Opennebula managing virtual machines. http://opennebula.org/documentation/archives:rel3.0:vm_guide_2.
- [5] A. Beloglazov and R. Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 577–578. IEEE, 2010.
- [6] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.
- [7] R. N. Calheiros, R. Ranjan, and R. Buyya. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 295–304. IEEE, 2011.
- [8] G. G. S. T. G. B. H. W. K. K. S.-Y. N. H.-J. J. Hao Wu, Shangping Ren. Automatic cloud bursting under fermicloud. *Workshop on Cloud Services and Systems*, 2013.
- [9] N. Huber, M. von Quast, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *CLOSER*, pages 563–573, 2011.
- [10] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on amazon ec2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. IEEE, 2009.
- [11] Y. Z. Mengxia Zhu, Qishi Wu. A cost-effective scheduling algorithm for scientific workflows in cloud. *Proceedings of 31st IEEE International Performance Computing and Communications Conference*, 2012.
- [12] R. Moreno-Vozmediano, R. Montero, and I. Llorente. IaaS cloud architecture: from virtualized data centers to federated cloud infrastructures. 2012.
- [13] J. Qiu, J. Ekanayake, T. Gunarathne, J. Y. Choi, S.-H. Bae, H. Li, B. Zhang, T.-L. Wu, Y. Ruan, S. Ekanayake, et al. Hybrid cloud and cluster computing paradigms for life science applications. *BMC bioinformatics*, 11(Suppl 12):S3, 2010.
- [14] A. N. Toosi, R. N. Calheiros, R. K. Thulasiram, and R. Buyya. Resource provisioning policies to increase iaaS provider's profit in a federated cloud environment. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 279–287. IEEE, 2011.
- [15] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, and B. Berriman. Experiences using cloud computing for a scientific workflow application. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 15–24. ACM, 2011.

A Reference Model for Virtual Machine Launching Overhead

Hao Wu*, Shangping Ren*, Gabriele Garzoglio[†], Steven Timm[†], Gerard Bernabeu[†], Keith Chadwick[†], Seo-Young Noh[‡]

Abstract—*Cloud bursting* is one of the key research topics in the cloud computing communities. A well designed *cloud bursting* module enables private clouds to automatically launch virtual machines (VMs) to public clouds when more resources are needed. One of the main challenges in developing cloud bursting module is to decide *when* and *where* to launch a VM so that all resources are most effectively and efficiently utilized and the system performance is optimized. However, based on system operational data obtained from the FermiCloud, a private cloud developed by the Fermi National Accelerator Laboratory for scientific workflows, the VM launching overhead is not a constant. It varies with physical resource utilization, such as CPU and I/O device utilizations, at the time when a VM is launched. Hence, to make judicious decisions as to *when* and *where* a VM should be launched, a VM launching overhead reference model is needed. In this paper, we first develop a VM launching overhead reference model based on operational data we have obtained on the FermiCloud. Second, we apply the developed reference model on the FermiCloud and compare calculated VM launching overhead values based on the model with measured overhead values on the FermiCloud. Our empirical results on the FermiCloud indicate that the developed reference model is accurate. We believe, with the guidance of the developed reference model, efficient resource allocation algorithms can be developed for cloud bursting process to minimize the operational cost and resource waste.

Index Terms—VM Launching Overhead, Reference Model, Cloud, FermiCloud, Virtual Machine, VM Launching, VM Startup Time, Launch, Overhead, Model, Predict



1 INTRODUCTION

CLOUD technology has been benefiting general purpose computing for a number of years. The *pay-on-demand* model brought about by cloud computing allows companies to avoid over-provisioning in early stages of project development. Furthermore, comparing to the traditional grid computing, cloud computing can better utilize resources provided by its underlying infrastructure, as it can deploy different tasks on the same physical computer node. In addition, computation power can also be dynamically allocated to tasks when more resources are needed by the tasks. Another benefit of using a cloud over a grid is that a cloud has “unlimited” resources – when a private cloud is fully occupied, cloud bursting techniques can temporarily acquire external resources from public clouds to fulfill the need.

Many scientific research institutions have foreseen the benefits of using computer clouds and have migrated their research platforms from traditional grid and distributed computing platform to the cloud computing environment [1] [12] [17] [19] [10]. Fermi National Accelerator Laboratory (Fermilab), a leading research institution in the

high energy physics (HEP) field, started to build a private infrastructure-as-a-service facility, the FermiCloud, in 2010. The FermiCloud has successfully served the HEP experiments since its establishment. S. Y. Noh *et al.* [20] [16] of KISTI collaboratively developed the *vcluster* cloud management tool for FermiCloud to automatically allocate cloud cycles on FermiCloud and KISTI’s GCloud as well as Amazon AWS. However, how to dynamically allocate resources so that application’s average response time and system’s total operational cost are reduced is a research and engineering challenge yet to be addressed.

Resource allocation problems in cloud computing has drawn more and more attention in research community in recent years [14], [7]. However, most research in the area assume that VM launching overheads with respect to time and resource consumption are negligible. As a result of this assumption, neither the launching overhead nor the dependency between the overhead and resource utilization are taken into consideration in designing resource allocation algorithms. However, our production line operation data (Fig. 1) indicates that the VM launching overhead can have significant variations when it is launched at different time or on different physical machines. Figure 1 depicts the launching time for 227 VMs that have been deployed on FermiCloud over one month period. The VM launching time ranges from few seconds to over one thousand seconds.

In addition to time overhead, VM launching overhead also includes system resources a VM consumes during its launching process. Both of time and utilization overheads can impact the system’s performance. For instance, VM launching process consumes a significant amount of CPU and I/O resources, leads to a high system CPU and I/O

*Illinois Institute of Technology, 10 W 31st street, 013, Chicago, IL, USA, {hwu28, ren}@iit.edu.

[†]Fermi National Accelerator Laboratory, Batavia, IL, USA. {garzogli, timm, gerard1, chadwick}@fnal.gov.

[‡]National Institute of Supercomputing and Networking, Korea Institute of Science and Technology Information, Daejeon, Korea, rsyoun@kisti.re.kr

The research is supported in part by NSF under grant number CAREER 0746643 and CNS 101873, by the U.S. Department of Energy under contract number DE-AC02-07CH11359 and by KISTI under a joint Cooperative Research and Development Agreement CRADA-FRA 2013-0001 / KISTI-C13013.

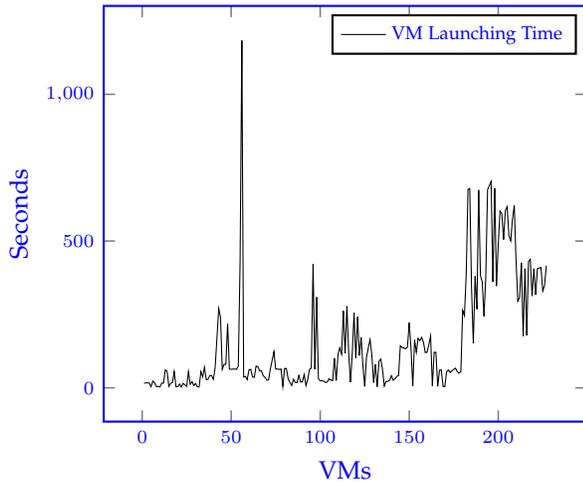


Fig. 1: VM Launching Time on FermiCloud (08/22/2014 - 09/19/2014)

utilization at the time of launching. If each host computer in the private cloud happens to launch a new VM at the same time, due to high system utilization caused by VM creation, the computer cloud may consider all its hosts are fully occupied and decide to perform cloud bursting and create VMs on an external public cloud. Such additional cost of bursting to external public cloud is unnecessarily rendered and can be prevented if we have a VM launching overhead reference model. Furthermore, if a task requires an additional VM in order to complete its work, but the VM takes much longer time to complete its launching than expected, it is possible that the task has already finished its work before the VM is ready for executing the task. This again leads to resource waste and an added cost due to the lack of a VM launching overhead information.

In this paper, we are to 1) analyze the patterns of VM launching process based on large amount of data obtained from real working systems, 2) define a reference model to represent the patterns, and 3) validate the accuracy of the developed reference model with real system operational data. At this initial study stage, we emphasize a methodological point of view rather than definitive numerical results based on accurate parameter values. The main purpose of this study is to show how an analytic model can be developed.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 introduces the FermiCloud and its architecture. Section 4 defines the terms that are used in the paper. Section 5 analyzes the VM launching overhead based on a large set of experiments on the FermiCloud. Section 6 presents a reference model for VM launching overhead. Section 7 evaluates the accuracy of the proposed model. We conclude the work in section 8.

2 RELATED WORK

Researches on modeling and evaluating the cloud's performance started almost at the same time when computer cloud itself emerged. One of the most influential cloud modeling tool is CloudSim [7] developed by the CLOUDS

lab from the University of Melbourne. CloudSim is a Java-based cloud simulation tool that supports modeling and simulation of large scale cloud computing environments. The CloudSim provides a comprehensive modeling tool that covers almost all basic elements under a cloud environment. In particular, it provides an infrastructure modeling to capture the characteristics and behaviors of both VM and physical infrastructure where VMs are deployed. It also provides a cloud market model that models the cost of resources, a network model that models the network behavior of inter-networking of clouds, a cloud federation model that models the communication between clouds, a power consumption model that models the power consumptions in the datacenter, and a resource allocation model that models VM allocation policies.

Recently, Huber *et al.* evaluate the virtualization performance and propose a virtualization overhead model [9]. In their work, they mainly focus on two virtualization platforms, i.e., XenServer and VMware ESX. They test the performance downgrades that are brought by the virtualization. They test the CPU, memory, disk IO, and network performance degradations on both XenServer and VMware ESX platforms. Based on the experiments, they categorize the virtualization performance influencing factors into four major categories: virtualization type, hypervisor's architecture, resource management configuration, and workload profile. However, Huber's model does not consider virtual machine launching overhead, it only provides the computation overhead caused by the virtualization.

Researchers have adapted the above cloud models and cloud simulations tools and made significant contributions to optimize resource allocation process on clouds, such as resources provisioning algorithms from QoS perspective [8], from service providers' profit perspective [18], and from energy consumption perspective [6]. Recently, Mengxia Zhu *et al.* have proposed a cost effective scheduling for scientific workflow under cloud environments [14]. Their scheduling algorithm aims to shorten the application's response time and reduce the energy consumption simultaneously by considering VM launching overhead. Some of the researches, such as Zhu's work [14] and CloudSim [7], have taken VM launching overhead variation as a key variable for designing resource allocation algorithms. However, they treat the VM launching overhead as a constant.

Some of the researchers have observed that VM overhead may have a large variation on public cloud and realized that the variation of VM launching overhead may cause significant impact on the resource allocation process [13] [11]. Hence, significant contributions have been made on reducing the impact of VM launching overhead. For instance, Lagar-Cavilla *et al.* [11] have developed a cloud programming paradigm and a system called SnowFlock that can significantly improve the VM scaling efficiency using fast VM cloning.

We have also observed significant VM launching overhead variations on the FermiCloud's daily operations. In the FermiCloud bursting project, the design of the resource allocation algorithm aims to automatically allocate resources for applications that need extra computational resources. If the VM launching overhead variation is not well modeled and calculated, the system utilization and efficiency may

be pulled down dramatically, causing resource and energy waste. Hence, we need an accurate mathematical model for VM launching overhead. Rather than aiming to reduce VM launching overhead, the goal of this paper is to understand and analyze VM launching process and developed a reference model to predict the overhead during such process.

3 FERMICLOUD

The FermiCloud uses OpenNebula [3], [15] as its cloud infrastructure management tool and uses KVM as its VM management tool. VMs running on the FermiCloud are all paravirtualized. Under OpenNebula [4], the VM launching process consists of four major states. Fig. 2 illustrates the state change during a VM launching process in OpenNebula [4]. In particular, when a user creates a new VM, the VM enters the *pending* state. In the pending state, the cloud scheduler decides where to deploy the VM. Once the VM is deployed on a specific host, it enters into the *prologue* state in which all VM related files (images in our case) are transferred from the image repository to the host machine. After all the files are copied to the host, the VM enters the *boot* state, during which it is booted from the host. Finally, after the VM is successfully booted, it enters into the *running* state. Once a VM is in its running state, it is ready to execute tasks.

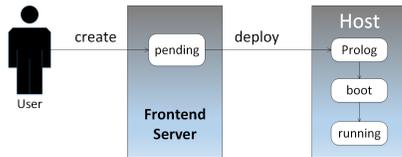


Fig. 2: VM Launching State Diagram[4]

Fig. 3 illustrates the system architecture of FermiCloud. The FermiCloud system has an OpenNebula front-end server that manages the entire cloud infrastructure, an image repository that stores the VM images, and a set of host machines on which VMs are deployed. Both front-end server and VM hosts use the GFS2 clustered shared file system, which is hosted on a fibre-channel connecting SAN with two NexSan SataBeast servers for storage servers. The logical unit used in the study has ten 7200-RPM SATA disk drives in a 9+1 RAID5 configuration for 17TB of usable space. Each SAN controller has 2GB cache memory. Both front-end server and VM hosts are configured with 16-core Intel(R) Xeon(R) E5640 @ 2.67GHz CPU and 48GB memory. All the machines in FermiCloud are installed with Scientific Linux operating system [2].

4 TERMINOLOGY

This section defines the terms used in the paper.

Host CPU utilization: The host CPU utilization is defined as total CPU utilizations consumed by all the processes on one host machine.

Host disk write utilization: The host disk write utilization is defined as the total disk write bandwidth utilizations consumed by all the processes on one host machine.

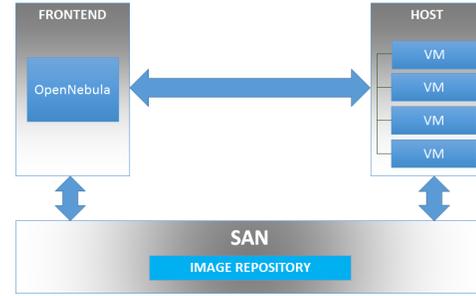


Fig. 3: System Architecture

Host disk read utilization: The host disk read utilization is defined as the total disk read bandwidth utilizations consumed by all the processes on one host machine.

VM CPU utilization: The VM CPU utilization is defined as the CPU utilization consumed by a VM on a single core. For example, in a 16-core CPU machine, the VM CPU utilization is 100% means the VM fully occupies one core. It equals to the consumption of 6.25% host CPU utilization.

VM disk write utilization: The VM disk write utilization is defined as the amount of disk write bandwidth consumed by a VM over the total disk write bandwidth.

VM disk read utilization: The VM disk read utilization is defined as the amount of disk read bandwidth consumed by the VM over the total disk read bandwidth.

Prologue: Prologue is an OpenNebula [4] terminology that indicates the process of copying an image from image repository to host machine. In the paper, the term *prologue* is interchangeable with the term image transmission process.

5 VM LAUNCHING OVERHEAD ON FERMICLOUD

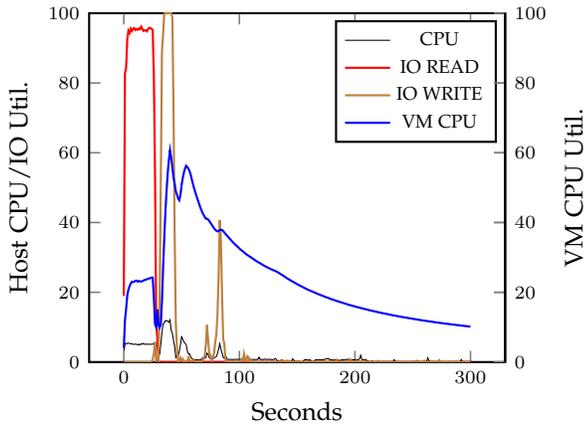
In this section, we study the patterns of VM launching overhead based on the VM operations in the FermiCloud environment.

5.1 VM Preparation

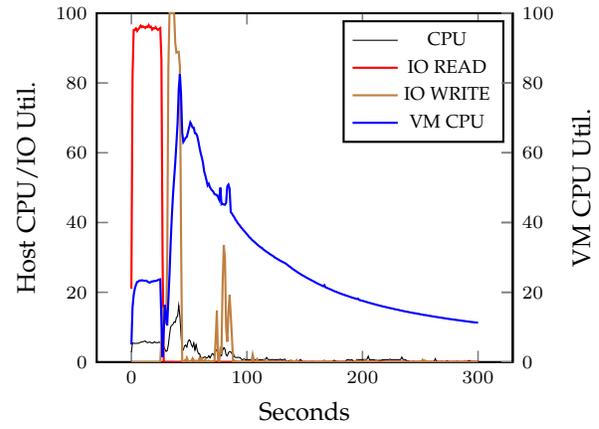
In our experiment tests, we focus on two types of VM instances, i.e., a small instance configured with one virtual CPU core and 2GB memory, and a large instance configured with 16 virtual CPU cores and 32GB memory. There are also two types of VM images tested, i.e., 4.7GB "QEMU Copy On Write 2" (QCOW2) image and 15.6GB raw image. Hence, four different types of VMs are tested during the experiments, i.e., small instance VM with QCOW2 image (SQ), small instance VM with raw image (SR), large instance VM with QCOW2 image (LQ) and large instance VM with raw image (RL).

5.2 Methodology

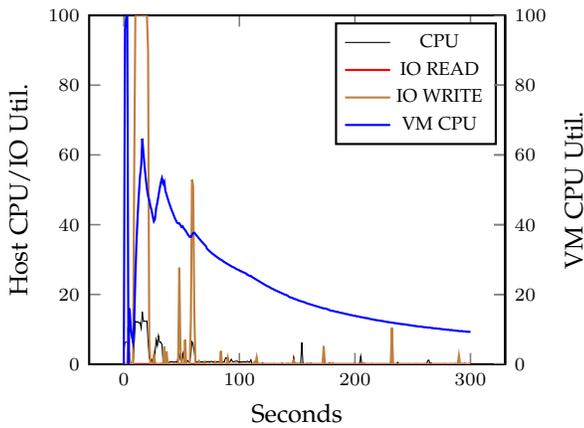
We use *noninvasive* programs, i.e., *iostat* and *sar* to obtain system information. The OpenNebula platform logs the time points when VMs enter the *running* state. In order to minimize the impact of cloud management tools, i.e., OpenNebula in our case, on VM launching process, we count VM start time as the time when a VM is deployed on the host machine. In cloud environment, VMs are not considered to be ready for use until the users can access the



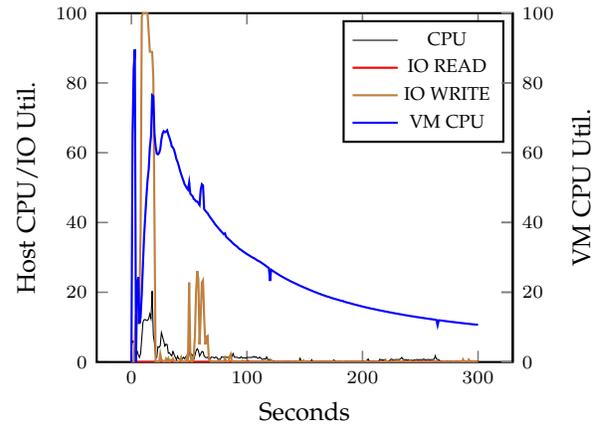
(a) Utilization Variation of Small Instance with Uncached QCOW2 Images



(a) Utilization Variation of Large Instance with Uncached QCOW2 Images



(b) Utilization Variation of Small Instance with Cached QCOW2 Images



(b) Utilization Variation of Large Instance with Cached QCOW2 Images

Fig. 4: Utilization Variation of Small Instance with QCOW2 Images

Fig. 5: Utilization variation of Large Instance with QCOW2 Images

VMs. Hence, we retrieve the start time of SSHD service from VMs' system logs as the times when VMs are actually ready for use.

In the cloud environment, existing running VMs may have significant impact on VM launching overhead. However, from physical machine (VM host machine)'s point of view, all VMs that are running on it are processes. Applications that are running on different VMs are transparent to physical machines. Hence, resources consumed by the VMs and workloads inside the VMs are reflected by the consumption of physical resources. Hence, we mimic the scenario that VM is launched when there are VMs already running in system by manipulating host machine's physical resource usage, i.e. CPU and I/O utilizations.

5.3 Base VM Launching Overhead

We first obtain the baseline overhead of launching a new VM. In order to obtain the baseline utilization overhead of launching a new VM, we let all the host machines in the private cloud be empty, i.e., have no application being deployed on the cloud. Each time, a single VM is launched on an empty host machine. The experiment is repeated twenty times for each type of VM.

Fig. 4(a), Fig 4(b), Fig. 5(a), Fig 5(b), Fig. 6(a), Fig. 6(b), Fig. 7(a) and Fig. 7(b) illustrate the average utilization changes for SQ, LQ, SR and RL VMs, respectively. The blue line depicts the VM CPU utilization, the black line the host CPU utilization, the brown line the host disk write utilization, and the red line the host disk read utilization. Table 1 gives the statistics of the utilization and timing variations of baseline VM launching processes.

5.3.1 Cached and Uncached Images

File cache is a Linux operating system feature that usually happens in file copying process. The VM launching process first copies VM's image from an image repository to the host machine. Once the VM image is copied onto the host machine, it is also cached into host machine's memory. If the VM being launched on the host machine has the same image as the cached one, the VM's image is directly copied from the memory instead of copied from an image repository. Hence, for the four types of tested VMs, we also test the launching overheads when launch them from both cached images and uncached images. For convenience, we use SQ_U, SQ_C, LQ_U, LQ_C, SR_C, LR_U and LR_C to denote the test cases of small instance VM with

TABLE 1: Statistics of Base VM Launching Process

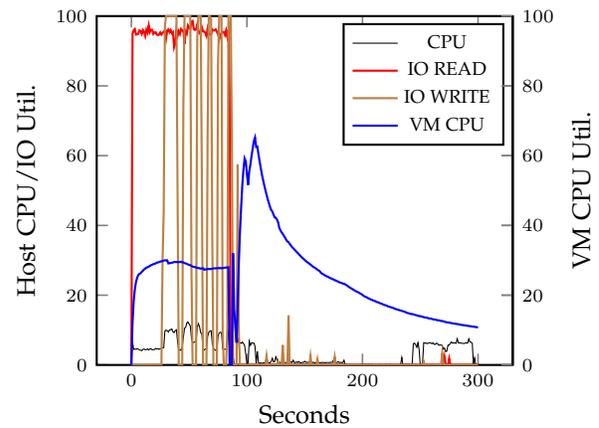
| | SQ_U | SQ_C | LQ_U | LQ_C | SR_U | SR_C | LR_U | LR_C |
|--------------------|-------|-------|-------|-------|--------|-------|--------|-------|
| LAUNCH TIME (Sec.) | 74.10 | 49.40 | 73.56 | 51.30 | 129.80 | 70.80 | 135.44 | 71.50 |
| MAX | 79.00 | 53.00 | 75.00 | 56.00 | 136.00 | 77.00 | 140.00 | 74.00 |
| MIN | 72.00 | 46.00 | 71.00 | 47.00 | 126.00 | 67.00 | 132.00 | 68.00 |
| TRANS. TIME (Sec.) | 28.80 | 4.80 | 28.30 | 4.30 | 87.00 | 19.80 | 89.30 | 19.70 |
| MAX | 31.00 | 5.00 | 30.00 | 5.00 | 90.00 | 21.00 | 91.00 | 21.00 |
| MIN | 28.00 | 4.00 | 27.00 | 4.00 | 88.00 | 19.00 | 88.00 | 19.00 |
| BOOT TIME (Sec.) | 45.30 | 44.60 | 45.33 | 47.00 | 42.80 | 51.00 | 46.11 | 51.80 |
| MAX | 48.00 | 48.00 | 47.00 | 51.00 | 52.00 | 57.00 | 52.00 | 54.00 |
| MIN | 44.00 | 41.00 | 43.00 | 42.00 | 38.00 | 47.00 | 44.00 | 49.00 |
| PEAK Util. (%) | 62.83 | 64.56 | 81.20 | 83.42 | 58.83 | 57.76 | 78.28 | 75.97 |
| MAX | 65.40 | 66.50 | 85.00 | 90.70 | 65.90 | 59.80 | 81.90 | 78.80 |
| MIN | 54.30 | 62.90 | 78.40 | 80.60 | 53.40 | 55.60 | 70.90 | 72.10 |
| TRANS. Util. (%) | 24.55 | 75.89 | 24.97 | 72.72 | 25.91 | 78.91 | 25.00 | 78.40 |
| MAX | 26.11 | 98.50 | 27.36 | 79.53 | 27.56 | 81.98 | 28.13 | 81.25 |
| MIN | 22.13 | 66.13 | 22.68 | 66.25 | 23.77 | 74.78 | 23.53 | 73.96 |

uncached QCOW2 image, small instance VM with cached QCOW2 image, large instance VM with uncached QCOW2 image, large instance VM with cached QCOW2 image, small instance VM with uncached raw image, small instance VM with cached raw image, large instance VM with uncached raw image and large instance VM with cached raw image, respectively. Fig. 4(a), Fig. 5(a), Fig. 6(a) and Fig. 7(a) illustrates the utilizations changes for SQ_U, LQ_U, SR_U, and LR_U, respectively. While Fig. 4(b), Fig. 5(b), Fig. 6(b) and Fig. 7(b) illustrates the utilizations changes for SQ_C, LQ_C, SR_C and LR_C, respectively.

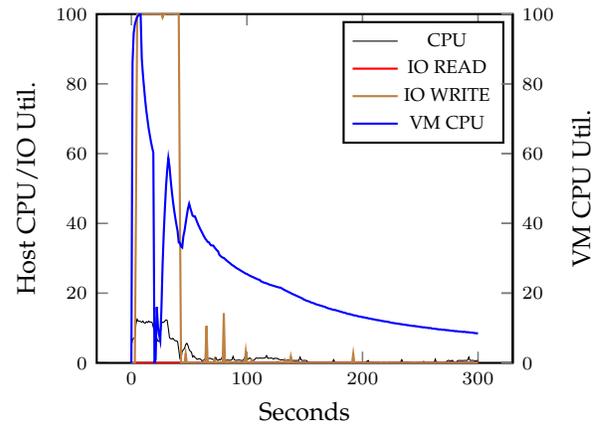
5.3.2 VM Prologue Process

The VM prologue process that copies the VM image from the image repository to the host machine is the first step of the entire VM launching process. As the measured disk read bandwidth of SAN in FermiCloud is 180MB/s, the measured disk write bandwidth of SAN in FermiCloud is 400MB/s. The theoretical time of transferring a 4.7GB QCOW2 image and a 15.6GB raw image from the image repository to the host machine is 26 seconds and 87 seconds, respectively. As indicated in Table 1, the average prologue time of SQ_U and LQ_U are 28.8 seconds and 28.3 seconds, respectively; the average prologue time of SR_U and LR_U are 87 seconds and 89.3 seconds, respectively. The measured prologue times align well with the calculated values. It is also clear from Fig. 4(a), 5(a), 6(a) and 7(a) that during the prologue time, disk read utilization almost constantly reaches 100% of the SAN read bandwidth (as shown by the red lines in the figures). The measured local disk read bandwidth is 500MB/s. When the local disk read bandwidth is fully utilized, the host CPU utilization consumption is around 70% on single core. Hence, the calculated VM CPU utilization during the prologue process is $180/(500/0.7) = 25.2\%$. The calculated value matches the measured average VM CPU utilization, which is around 25%.

Notice from the figures that the disk write activities are not synchronized with the disk read activities. In Linux, when a file is copied, the file is written into dirty pages first, it is then flushed into the hard disk. By Linux default settings, the flushing process happens if the dirty page size reaches 10% of the system active memory or 30 seconds after the content is written into the dirty page. As the 4.7GB QCOW2 image is smaller than the size of 10% of



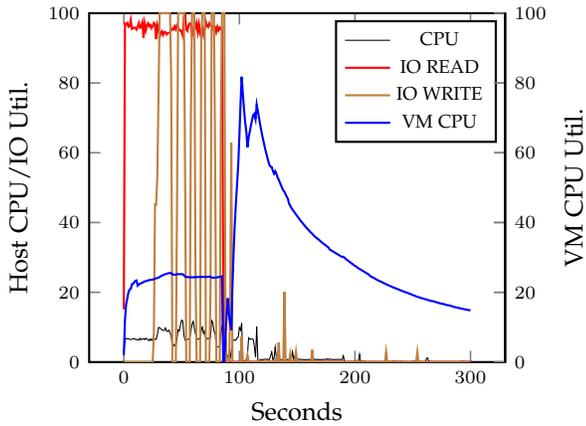
(a) Utilization Variation of Small Instance with Uncached Raw Images



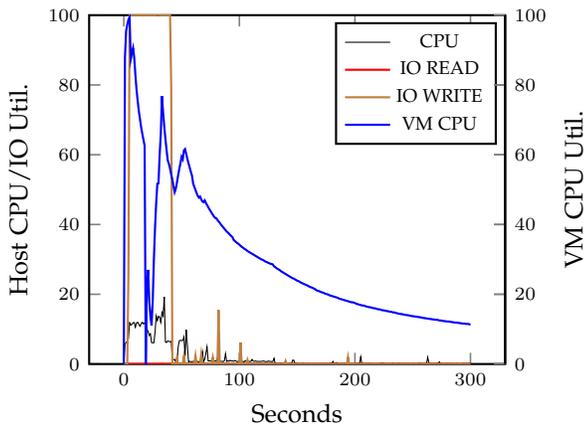
(b) Utilization Variation of Small Instance with Cached Raw Images

Fig. 6: Utilization Variation of Small Instance with Raw Images

system memory (48GB * 10%) and the transmission time of the image is less than 30 seconds. The flush process is activated immediate after the whole image is copied into the dirty page (as illustrated in Fig. 4(a) and Fig. 5(a)). For large images, i.e., 16GB raw images, as illustrated in Fig. 6(a) and Fig. 7(a), the flushing processes are activated



(a) Utilization Variation of Large Instance with Uncached Raw Images



(b) Utilization Variation of Large Instance with Cached Raw Images

Fig. 7: Utilization Variation of Large Instance with Raw Images

after 30 seconds or when the dirty page reaches 10% limits (whenever happens first).

There is an interesting observation from Fig. 6(a) and Fig. 7(a) that the write process is not constantly writing images into the disk. This is because the write speed (400MB/s) is much faster than the read speed (180MB/s), by Linux default, the flushing process is awoken every 5 seconds if such scenario happens.

On the other hand, for cached images, as illustrated in Fig. 4(b), 5(b), 6(b) and 7(b), there are no disk read activities happening during the VM prologue processes. This is because images are read directly from the memory. As the measured cache read speed in FermiCloud host machines are around 1.2 GB/s. The theoretical transmission time for QCOW2 and raw images are 3.9 seconds and 18.5 seconds, respectively. As given in Table 1, the measured transmission times for cached images are consistent with the calculated values. Notice that, when images are read from memory, it takes about 4 seconds for the dirty pages reaches 10% limit. Hence, the flushing process happens after 4 seconds of the VM prologue process, and images are continuously being written into the disk. Since the prologue process fully utilize the memory bandwidth, the VM CPU utilization during the

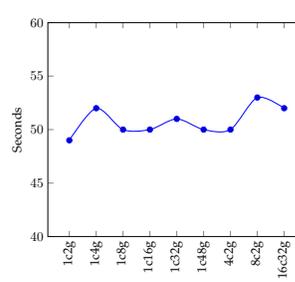


Fig. 8: VM Booting Time Comparison for Different Configurations

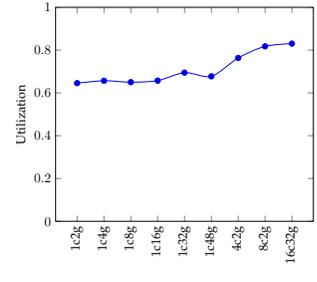


Fig. 9: VM Booting Peak CPU Utilization Comparison for Different Configurations

prologue process for cached images also reaches 100%.

5.3.3 VM Boot Process

After the image is copied into the host machine, the VM then enters the boot process. Once the VM is booted, it can be used by end users. Note that, the boot process starts immediately after the image is copied into cache, not after the entire image is written into the disk. From the figures we can tell that the patterns of VM CPU utilization variation for all test cases are similar. The VM CPU utilization soon reaches a peak after the boot process begins, then VM CPU utilization slowly decreases and remains a low level. Table 1 also shows that the booting times for all cases are almost the same. The only difference among the different test cases is the peak VM CPU utilization. For the small instance VMs, the peak VM CPU utilization is around 60% of single core CPU utilization. However, for the large instance VMs, the peak VM CPU utilization is around 80% of single core CPU utilization.

5.3.4 VM Boot process comparison under different configurations

From the above experiments, we observe that different VM configurations do not affect the VM booting time. However, the configurations change the peak VM CPU utilization during the booting process. In order to understand how the configuration affects the peak VM CPU utilization during the booting process, we perform a more detailed test on varies VM configurations. We first test the VMs with 1 virtual core and change the memory from 2GB to 48 GB. Then we configure the VMs with 2GB memory and change the number of virtual cores from 2 to 16. All VMs are launched with QCOW2 images. The booting times of different VMs are depicted in Fig. 8. As can be seen from the figure, the booting times of VMs with different configurations vary only within a small range. We consider them to have the same booting times. Fig. 9 depicts the peak VM CPU utilization of VMs with different configurations during the booting process. It clearly shows that the memory configuration change does not affect the peak VM CPU utilization. However, the peak VM CPU utilization increases as the number of virtual cores increases.

5.4 CPU Utilization Impact

In this experiment, VMs are launched under different host CPU utilizations. We write a small bash script that con-

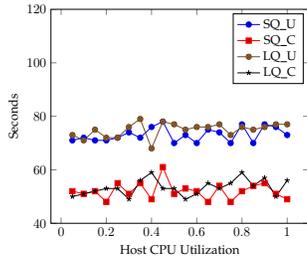


Fig. 10: VM Booting Time under Different CPU Utilization for QCOW2 Image

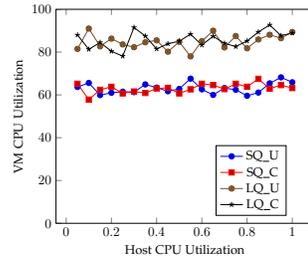


Fig. 11: Peak VM CPU Utilization under Different CPU Utilization for QCOW2 Image

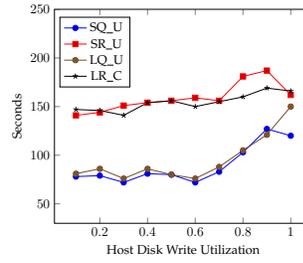


Fig. 14: VM Launching Time under Different IO Utilization using Uncached Images

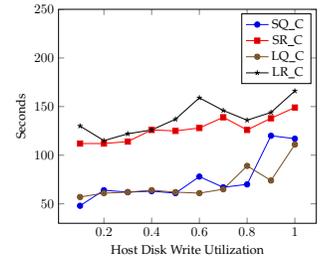


Fig. 15: VM Launching Time under Different IO Utilization using Cached Images

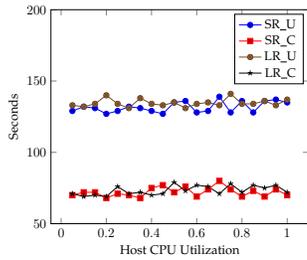


Fig. 12: VM Booting Time under Different CPU Utilization for Raw Image

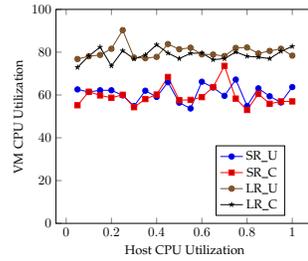


Fig. 13: Peak VM CPU Utilization under Different CPU Utilization for Raw Image

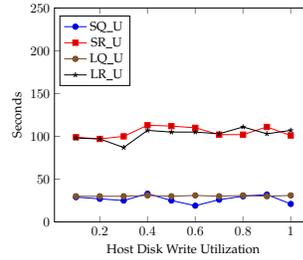


Fig. 16: VM Prologue Time under Different IO Utilization using Uncached Images

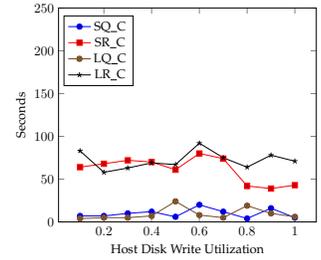


Fig. 17: VM Prologue Time under Different IO Utilization using Cached Images

stantly consume the host CPU utilization. We use *cgroup* to control the CPU usage of the script. Each time, we increase 5% of the CPU utilization consumed by the script. Hence, we can simulate the scenario that VMs are launched under different host CPU utilizations.

Fig. 10 depicts the launching times of VMs with QCOW2 images. As can be seen from the figure, the launching time of the VMs are relatively steady. For the uncached images, the maximum launching time is 79 seconds while the minimum launching time is 68 seconds for both small and large VM instances. For the cached images, the range of the VMs' launching times is from 48 seconds to 61 seconds. Comparing with the base VM launching times listed in Table 1, we consider the variations of the launch times are in normal condition. Hence, we can conclude that the host CPU utilization does not impact the VMs' launching times. As illustrated in Fig. 11, the peak VM CPU utilization during the booting process are also quite steady (60% for the small instances and 80% for the large instances). Hence, the host CPU utilization does not impact the peak VM CPU utilization during the booting process.

Fig. 12 and Fig. 13 illustrate the launching times and peak VM CPU utilization under different host CPU utilizations for the raw images. Similar to the QCOW2 images, the launching times and peak VM CPU utilizations are insensitive to the host CPU utilization changes. Hence, we can conclude that the host CPU utilization does not impact the VMs' launching process.

5.5 Disk Write Utilization Impact

In this experiment, we test the VM launching overhead under different host disk write utilizations. We use *dd* command to constantly write files to the disk., and use *cgroup* to control the write speed of the *dd* command. At each step, we increase 10% of the disk write bandwidth for *dd* command. Hence, we can simulated the scenario that VMs are launching under different host disk write utilizations.

Fig. 14 and Fig. 15 show the VMs' launching times under different host disk write utilizations. Overall, when the host disk write utilization increases, VMs need more time to be launched as compared with the base VM launching times. However, for the cached images, i.e., as illustrated in Fig. 15, the increasing trends are not so obvious when compared with the uncached images (Fig. 14). As for the VM prologue times, the prologue times for uncached images are steady and close to a constant. However, if the image is cached, the VMs' prologue times have larger variations. For the QCOW2 images, the variation is from 4 seconds to 24 seconds, and for a raw image, the variation is from 39 seconds to 92 seconds. This large variation is caused by cache override. Since the *dd* command also write files into the cache, it is possible that the cached image is overridden by the *dd* command. Hence, the missing part of the image need to be re-transferred from the data repository. In the worst case, the entire cached image is overridden by the *dd* command, and the entire image needs to be copied from the image repository.

Fig. 18 and Fig. 19 illustrate the booting times for un-

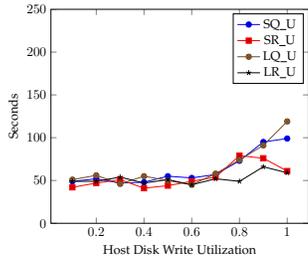


Fig. 18: VM Booting Time under Different IO Utilization using Uncached Images

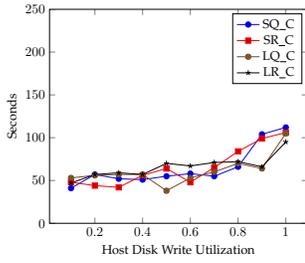


Fig. 19: VM Booting Time under Different IO Utilization using Cached Images

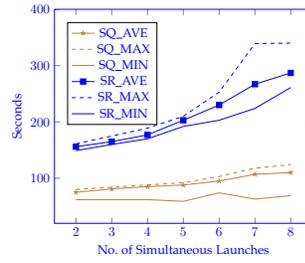


Fig. 22: VM Launching Time under Different Simultaneous Launches

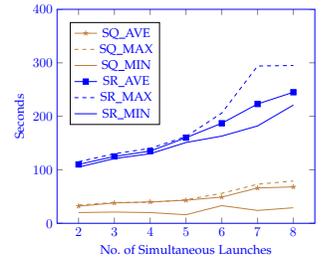


Fig. 23: VM Prologue Time under Different Simultaneous Launches

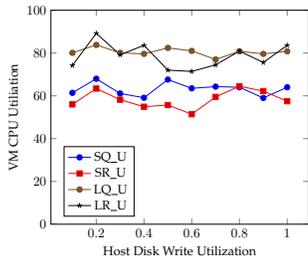


Fig. 20: VM Peak CPU Utilization under Different IO Utilization using Uncached Images

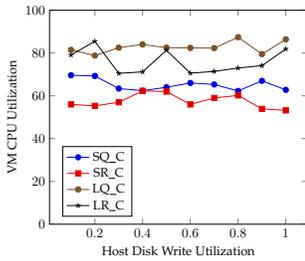


Fig. 21: VM Peak CPU Utilization under Different IO Utilization using Cached Images

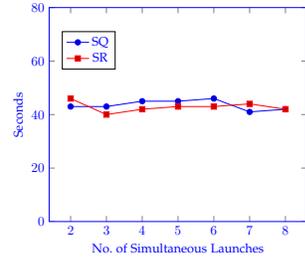


Fig. 24: Ave. VM Booting Time under Different Simultaneous Launches

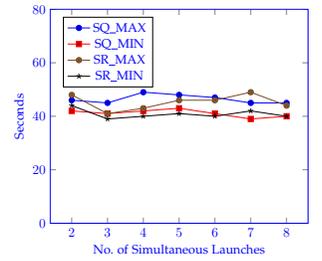


Fig. 25: Max and Min VM Booting Time under Different Simultaneous Launches

cached images and cached images, respectively. It is observed that when the host disk write utilization increases, the booting time of the VMs also increases. When the host disk write utilization reaches 100%, it almost takes two times long to boot a VM when compared to the base VMs' booting time. In addition, as shown in Fig. 20 and Fig. 21, the host disk write utilization does not have much impact on the peak VM CPU utilization during booting processes.

5.6 Image Repository Impact

Under the FermiCloud architecture, as shown in Fig. 3, all the machines are connected to the SAN. Hence, when large number of VMs are launched simultaneously, all the VM images are read and copied from the SAN at the same time which may lead to significant increase of the VM's launching overheads. We set up another set of experiments to evaluate the impact of sudden large number of simultaneous launches on the VM launching overhead. Since for cached images, the images are directly read from the host machine's memory, hence, the SAN activities does not affect the VM launching process for cached image. We therefore, only evaluate uncached images. In particular, we launch a VM on a host, and simultaneously launch more VMs on other different hosts.

Fig. 22 shows the average launching times and maximum and minimum launching times for the SQ_U and SR_U test cases. X-axis presents the number of simultaneous launches. It is clear that when the number of simultaneous launches increases, the VM launching times also increase.

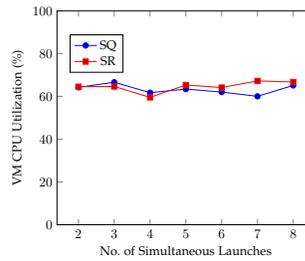


Fig. 26: VM Peak CPU Utilization under Different Simultaneous Launches

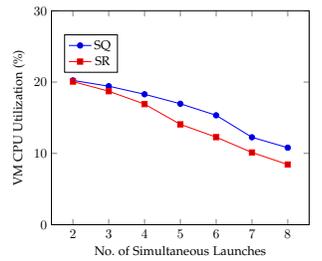


Fig. 27: VM Average Prologue CPU Utilization under Different Simultaneous Launches

As the number of simultaneous launches only affect the image read speed from the SAN, once the image is copied to the local host machine, the booting process becomes the same as the base VM booting process. Fig. 24 and 25 show the average VM booting time and maximum and minimum VM booting time, respectively. Fig. 26 depicts the average peak VM CPU utilization during the VM booting process. It is clear that the VMs' average booting times and average peak VM CPU utilization during the booting process are not influenced much by the number of simultaneous launches and aligned well with the base VMs' booting patterns as stated in Table 1.

As shown in Fig. 23, the number of simultaneous launches has significant impact on the image transmission process. When the number of simultaneous launches increases, the image transfer time variation also increases. This is because when the number of simultaneous launches increases, the number of requests for read bandwidth of

the SAN also increases. As a result, the read speed for each request is decreased. Fig. 27 shows when the read speed decreases, the average VM CPU utilization during the prologue process also decreases linearly. As mentioned before, the local disk read bandwidth is 500MB/s. When the read bandwidth is fully utilized, the host CPU utilization consumption is 70% on a single core. When eight VMs are being launched simultaneously, the measured average read speed on the SAN is 62MB/s for the tested VM. Hence, the theoretical VM CPU utilization during the prologue process is $62/(500/0.7) = 8.6\%$ which is consistent with the measured average value(8.4%).

5.7 Network Impact

Because the FermiCloud is using SAN as its storage, all the image transmission processes are treated as local disk activities. Hence, the network activities will not have impact on the VMs' launching process. However, the SAN architecture itself is a special network where the network bandwidth is much larger than the disk write/read speed and the network bandwidth is fully dedicated to its connected machines. In the environment where the machines are connected by the high speed Ethernet, the actual network bandwidth (at GB level) is still much larger than the disk write/read speed (at hundreds MB level). We also conducted the same set of experiments that we have done on the SAN architecture FermiCloud is built upon on the Ethernet environment [21]. The experimental results are the same as the observations we have on the SAN architecture. The disk write/read speed is also the dominant factor that significantly impact the VMs' launching overhead.

5.8 Discussion

From these experiments, we can conclude the following:

- *VM launching overhead mainly contains two parts: prologue (image copying/transferring) overhead and booting overhead;*
- *booting overhead is relatively steady, i.e., has less variations when it is compared to the prologue overhead;*
- *prologue overhead, on the other hand, has significant variations when the disk read speed on image repository varies; and*
- *disk write utilization has significant impact on both prologue overhead and booting overhead.*

As [5] states that different VM and cloud management tools have significant performance differences on VM process. In the experiments, we are trying to minimize the impact of OpenNebula by discounting its default scheduler overhead from VM launching process. We do believe that different VM management tools may impact VM performance on resource consumption. Using different VM management tool, for instance, Xen may lead different peak CPU utilization during the VM booting process, but we believe that the patterns of the image transferring process and VM booting process are the same regardless of the VM and cloud management tools.

Another major factor that impacts the VM launching process is the cloud infrastructure. In our earlier work, we have performed the same set of experiments on a different cloud

infrastructure where host machines and image repository are connected by Ethernet and regular NFS file system [21]. The patterns we have observed from VM launching process are the same on both cloud infrastructures. Hence, we believe that the patterns of VM launching process are generally the same in most of the private cloud systems. One of our future work is to verify the hypothesis.

In the next section, we present a reference model for the VM launching overhead based on the data obtained.

6 VM LAUNCHING OVERHEAD REFERENCE MODEL

Before we present the reference model for the VM launching overhead in private cloud, we first introduce notations to be used in defining the reference model.

A VM in a private cloud is defined as $v_i = (f_i, t_i, H_i)$, where f_i is the image size of the VM v_i , t_i is the VM start time and H_i is the host machine that the VM is to be deployed on. For each host H_i , we use $V_{H_i} = \{v_1, v_2, \dots, v_n\}$ to denote the set of VMs deployed on the host H_i . In the set V_{H_i} , virtual machines are sorted according to their start time in non-decreasing order. For each host H_i , p_i and m_i are the number of cores and the total memory owned by the host H_i , respectively. The $B_{H_i}^w$, $B_{H_i}^r$ and $B_{H_i}^c$ denote host machine disk write bandwidth, disk read bandwidth, and cache bandwidth, respectively; $S_w(H_i, t)$, $S_r(H_i, t)$ and $S_c(H_i, t)$ denote the file write speed, file read speed, and file cache speed on host H_i at time t , respectively.

The proposed reference model contains three different overheads: CPU utilization overhead, disk write utilization overhead, and timing overhead which is the time needed for launching a VM until it is ready to execute tasks. We first model the CPU utilization and disk write utilization that a single VM consumes on the host machine during the launching process. Then we model the host machine's entire system CPU utilization and disk write utilization. As discussed in section 5, the complete VM launching process mainly consists of two parts: prologue and boot process. We discuss the reference models for these two steps below.

6.1 Prologue Overhead Model

The prologue overhead we model here also contains three different parts: CPU utilization overhead, disk write utilization overhead, and timing overhead which is the time needed for transferring an image from image repository to the host machine. Since the prologue overhead for uncached image and cached image are different, we have separate models for the uncached and the cached images, respectively.

6.1.1 Uncached Image

The image transferring time for VM $v_i = \{f_i, t_i, H_i\}$ with uncached image is defined below:

$$Trans_i = \frac{f_i}{S_r(H_i, t_i)} \quad (1)$$

From the experiments we know that if local disk read bandwidth is fully utilized by a process, the process also utilizes 70% of one physical core of the host machine. For

different systems, the ratio may vary. We denote such ratio as β . For a given system, the β is a constant. We first define the base VM CPU utilization of prologue as follows:

$$U_{b_pro}(i, t) = \frac{1}{1 + e^{-0.5(Trans_i+t_i)(t-t_i)}} - \frac{1}{1 + e^{-0.5(Trans_i+t_i)(t-(Trans_i+t_i))}} \quad (2)$$

From the observation of image transferring process in previous section we know that when the image transferring process starts, it consumes all the I/O utilization in a very short period and occupied the I/O resources until it finishes transferring, then it releases the I/O resource. Equation (2) is given to match such utilization variations. The VM CPU utilization of transferring an image for VM v_i is modeled as:

$$U_{tr}(i, t) = \begin{cases} \frac{S_r(H_i, t_i)}{B_{H_i}^v/\beta} * U_{tr_base}(i, t) & t_i \leq t \leq t_i + Trans_i \\ 0 & otherwise \end{cases} \quad (3)$$

The disk write utilization consumed by a VM's prologue process is more complicated. As the write process is not synchronized with the prologue process, we first need to determine the start time of the disk write process. As discussed above, by Linux default setting, a file is written into disk at the time when the dirty page reaches 10% of the memory or 30 seconds whichever happens first. Hence, we define the start time of write process for VM v_i as:

$$st_w^{UC}(v_i) = t_i + \min\{30, \frac{\min\{f_i, m_i/10\}}{S_r(H_i, t_i)}\} \quad (4)$$

We then need to define the time points that the VM launching process actually writes disk. As discussed above, if the disk write speed is larger than the read speed, the write process sleeps for 5 seconds after writing all the content in the dirty page. We calculate the first time duration that the process writes file into the disk.

$$fst_w^{UC}(v_i) = \min\left\{\frac{f_i}{S_w(H_i, t_i)}, \min\{30, \frac{\min\{f_i, m_i/10\}}{S_r(H_i, t_i)}\} \cdot \frac{S_r(H_i, t_i)}{S_w(H_i, t_i) - S_r(H_i, t_i)}\right\} \quad (5)$$

After the first write duration, the remaining size of the file is $rf_i = f_i - fst_w^{UC}(v_i) * S_w(H_i, t_i)$. Hence, the total time of writing the remaining file into the disk is $rf_i/S_w(H_i, t_i)$. The write time after each 5 second sleep is $wt_{sleep} = 5 * S_r(H_i, t_i)/(S_w(H_i, t_i) - S_r(H_i, t_i))$. Then the total number of sleeps is $N_{sleep} = \lceil rf_i/S_w(H_i, t_i)/wt_{sleep} \rceil$. Hence, we can obtain the time points that the entire write process finishes as below:

$$lt_w^{UC}(v_i) = st_w^{UC}(v_i) + fst_w^{UC}(v_i) + rf_i/S_w(H_i, t_i) + N_{sleep} * 5 \quad (6)$$

The time point set that the write process actual writes file into disk is defined as:

$$\begin{aligned} \mathcal{T} = & \{[st_w^{UC}(v_i), st_w^{UC}(v_i) + fst_w^{UC}(v_i)] \cup \\ & [st_w^{UC}(v_i) + fst_w^{UC}(v_i) + 1 * 5, \\ & st_w^{UC}(v_i) + fst_w^{UC}(v_i) + 1 * 5 + wt_{sleep}] \cup \dots \cup \\ & [st_w^{UC}(v_i) + fst_w^{UC}(v_i) + (N_{sleep} - 1) * 5, \\ & st_w^{UC}(v_i) + fst_w^{UC}(v_i) + (N_{sleep} - 1) * 5 + wt_{sleep}] \cup \\ & [st_w^{UC}(v_i) + fst_w^{UC}(v_i) + N_{sleep} * 5, \\ & st_w^{UC}(v_i) + fst_w^{UC}(v_i) + N_{sleep} * 5 \\ & + (rf_i/S_w(H_i, t_i)) \bmod wt_{sleep}]\} \end{aligned} \quad (7)$$

At last, we define the disk write utilization consumed by the VM prologue process and the host CPU utilization it consumes. We first define the base host disk write utilization as:

$$IO_{w_base}^{UC}(i, t) = \frac{1}{1 + e^{-0.5(lt_w^{UC}(v_i)(t-st_w^{UC}(v_i)))}} - \frac{1}{1 + e^{-0.5(lt_w^{UC}(v_i)(t-lt_w^{UC}(v_i)))}} \quad (8)$$

Then the host disk write utilization for the prologue process is defined as:

$$IO_w^{UC}(i, t) = \begin{cases} \frac{S_w(H_i, t_i)}{B_{H_i}^v} * IO_{w_base}^{UC}(i, t) & t \in \mathcal{T} \\ 0 & otherwise \end{cases} \quad (9)$$

The host CPU utilization consumption of the write process is:

$$U_w^{UC}(i, t) = \frac{1}{p_i} IO_w^{UC}(i, t) \quad (10)$$

6.1.2 Cached Image

The prologue time and VM CPU utilization during the prologue process can be calculated using equation (1) and (3) by replacing the $S_r(h_i, t_i)$ with $S_c(h_i, t_i)$, respectively. For the host disk write utilization during the prologue process, the model is much simpler compared with the model for the uncached images.

The start time point $st_w(v_i)$ of the write process also can be calculated using equation (4) by replacing $S_r(h_i, t_i)$ with $S_c(h_i, t_i)$. The finish time point of the write process is:

$$lt_w(v_i) = st_w(v_i) + \frac{f_i}{S_c(H_i, t_i)} \quad (11)$$

Hence, the base host disk write utilization for cached images is defined as:

$$IO_{w_base}(i, t) = \frac{1}{1 + e^{-0.5(lt_w)(t-st_w(v_i))}} - \frac{1}{1 + e^{-0.5(lt_w)(t-lt_w(v_i))}} \quad (12)$$

Then the host disk write utilization for the prologue process is:

$$IO_w^C(i, t) = \begin{cases} \frac{S_c(H_i, t_i)}{B_{H_i}^v} * IO_{w_base}(i, t) & st_w(v_i) \leq t \leq lt_w(v_i) \\ 0 & otherwise \end{cases} \quad (13)$$

The host CPU utilization consumption of the write process is:

$$U_w^C(i, t) = \frac{1}{p_i} IO_w^C(i, t) \quad (14)$$

In general, the host disk write utilization during the VM prologue process is defined as:

$$IO_w(i, t) = \begin{cases} IO_w^{UC}(i, t) & \text{uncached image} \\ IO_w^C(i, t) & \text{cached image} \end{cases} \quad (15)$$

The host CPU utilization consumption of the write process is:

$$U_w(i, t) = \begin{cases} U_w^{UC}(i, t) & \text{uncached image} \\ U_w^C(i, t) & \text{cached image} \end{cases} \quad (16)$$

Note that the model for uncached images only validates for the scenario when $S_w(H_i, t_i) > S_r(H_i, t_i)$ and images are not cached. When $S_w(H_i, t_i) \leq S_r(H_i, t_i)$, we need to use the cached image model by replacing $S_c(H_i, t_i)$ with $S_r(H_i, t_i)$.

6.2 Booting Overhead Model

The VM booting overhead also refers to the timing overhead and CPU utilization overhead. As once the image is copied to a host, it will not consume any disk write utilization for the booting process. We consider that there is no disk write overhead for the virtual machine booting process. The experiments also indicate that the host disk write utilization impacts the booting overhead. Hence, we model the CPU utilization overhead for the VM v_i 's booting process as follows:

$$U_b(i, t) = \begin{cases} a * (t - Trans_i - t_i) & , t < Trans_i + t_i + \frac{f(v_i)}{a} \\ f(v_i) \frac{1}{m} e^{-t' \gamma (\alpha (1 + IO_s(H_i, t-1)) + \frac{\lambda}{\gamma + \lambda})} & , otherwise \end{cases} \quad (17)$$

where $f(v_i)$ is the function related to the virtual CPU cores that v_i has and it is used to control the peak VM CPU utilization during the booting process. In equation(17), a , γ , α and λ are constants, and γ and λ dominate the function decay rate while α determines the minimum decay rate, m is the number of cores on the host machine and $IO_s(H_i, t-1)$ represents the system's disk write utilization at time $t-1$. $t' = t - Trans_i - t_i - f(v_i)/a$. We will formally define the system disk IO utilization in section 6.5.

In OpenNebula, VMs are not immediately ready for use until all the necessary services, such as *SSH*, are started. As there is no accurate way to tell the actual time when a virtual machine is booted and ready for use unless entering into a running virtual machine and check the log, therefore, we base our estimation of the time points on the variation of the VM's CPU utilization consumption. If the VM's CPU utilization consumption remain stable, then we consider the VM is booted and ready for use. We define the time point $t_b(i)$ at which a VM v_i is ready to use as:

$$t_b(i) = \max\{t | |U_b'(i, t)| \leq \epsilon\} \quad (18)$$

where $U_b'(i, t)$ is the first derivative of $U_b(i, t)$ and ϵ is the threshold to determine whether the virtual machine's CPU utilization consumption become stable. Then, we can calculate the VM booting time is as $(t_b(i) - Trans_i)$.

6.3 Virtual Machine Launching Overhead Model

We have formally modeled image transfer overhead and virtual machine booting overhead. Combining the two components together, we derive VM launching overhead functions. In particular, combining equation (3) and equation (17), the VM v_i 's launching CPU utilization function is modeled as:

$$U(i, t) = \begin{cases} U_{tr}(i, t) & t \leq t_{tran} \\ U_b(i, t) & t > t_{tran} \end{cases} \quad (19)$$

Since IO utilization consumed by the VM booting process is negligible, the IO utilization function for VM v_i 's launching process is the same as equation (15).

The total time needed for launching a VM v_i then can be calculated as image copying time plus VM booting time. It is formally defined as follow:

$$t_{overhead}(i) = t_b(i) - t_i \quad (20)$$

6.4 Virtual Machine Utilization Consumption Model

The complete VM utilization functions consist of the VM launching overhead utilization functions and the utilization functions after workloads are deployed on the virtual machine. We assume at time $t' \geq t_b(i)$, the VM v_i starts executing tasks; and the CPU and disk IO utilization consumption function of v_i at t' are $U_e(t)$ and $IO_e(t)$, respectively. Then the VM CPU utilization consumption model is defined below:

$$U_c(i, t) = \begin{cases} U_{tr}(i, t) & t \leq t_{tran} \\ U_b(i, t) & t > t_{tran} \\ U_e(i, t) & t \geq t' \end{cases} \quad (21)$$

The VM IO utilization consumption model is defined as:

$$IO_c(i, t) = \begin{cases} IO_w(i, t) & t_i \leq t \leq t_b(i) \\ IO_e(i, t) & otherwise \end{cases} \quad (22)$$

6.5 System Utilization Model

We assume that host machines only run VMs and all other critical system services consume a small portion of the system CPU and IO utilization. Then we can calculate the system CPU and disk IO utilization as the summation of the VMs' CPU and IO utilization consumptions. The system CPU utilization of host h_i is modeled below:

$$U_s(H_i, t) = \max\{1, \sum_{j=1}^{|V_{H_i}|} \{U_c(j, t)\} + \sum_{j=1}^{|V_{H_i}|} \{U_w(j, t)\}\} \quad (23)$$

The system IO utilization of host h_i can be modeled as:

$$IO_s(H_i, t) = \max\{1, \sum_{j=1}^{|V_{H_i}|} \{IO_c(j, t)\}\} \quad (24)$$

Intuitively, the mathematical equations used to model the VM launching process in this section are obtained by finding the best match equations to the plotted data shown in Section 5. There may exist other equations that can also represent the variations of the real data. To balance the trade offs between accuracy and complexity, we choose the equations that we believe give fair accuracy within a short computational time period. In next section, we use data obtained from real operation cloud to verify the accuracy of the developed reference model.

7 EVALUATION

We build the reference model for virtual machine launching overhead based on a large amount of data obtained from a real production system. However, we cannot guarantee the accuracy of the model unless we compare the calculated data using the model we built with the real system data and prove the accuracy of the model. Since some of the parameters we use for modeling are system dependent, we focus the evaluation on the same given system, i.e., FermiCloud. We first use base VM launching overhead values shown in Section 5 to determine all the parameters. Once the parameters are determined, they are fixed for all the evaluation experiments. To evaluate the performance of the developed model, we first launch VMs on FermiCloud under different system loads. Then we use the developed reference model to simulate the launch process use the same VM release pattern.

We use mean square weighted deviation to evaluate the accuracy of our developed model from four aspects, i.e., VM CPU Utilization, host CPU utilization, host I/O utilization and VM launching time. We denote N as the total number of sampling points. The mean square weighted deviation is defined as follows:

$$MSWD = \frac{1}{N} \sum_{i=1}^N \frac{\sum_{j=1}^n (U_s(i) - U_r(i, j))^2}{\sigma^2} \quad (25)$$

where n is the number of the repetition of one experiment. $U_r(i, j)$ is the real data from the i^{th} sampling point and j^{th} repetition. U_s represents the calculated data at i^{th} sampling point. σ is the standard deviation. To check the real time point for the VM that is ready for use, we use the VM system log to check the starting time point of the *SSHD* service.

We first compare the base overhead obtained by calculating the value based on the model proposed in section 6, and the real data obtained on FermiCloud.

Fig. 28(a), Fig. 28(b), Fig. 28(c), Fig. 28(d), Fig. 29(a), Fig. 29(b), Fig. 29(c), and Fig. 29(d), draws the host CPU utilization, host disk write utilization and VM CPU utilization for SQ_U SQ_C LQ_U LQ_C SR_U SR_C LR_U LR_C test cases using the developed VM launching overhead model, respectively. Compare the graph with the utilization variations shown in Fig. 4, Fig. 5, Fig. 6, and Fig. 7 obtained from the real operation data from FermiCloud. The calculated data using our developed model is very close to the real data.

Table 2 gives a more detailed comparison between the real data and calculated data. From the table, we can observe that the maximum mean square weighted deviation for the VM CPU utilization of base test cases is 4.55 and the minimum mean square weighted deviation for the base test cases is 0.41. As the real data for the base cases shown in Fig. 4(a) to Fig. 7(b), the VM CPU utilization during the booting process drops for a small duration after it reaches the peak utilization, then immediately rises a little bit and finally decreases continuously. While in our model, the VM CPU utilization for the booting process keeps decreasing after reaches its peak utilization. Hence the range of the mean square weighted deviation for the VM CPU utilization for the base cases is from 0.41 to 4.55. The average mean square weighted deviation for VM CPU utilization of all base cases

| | VM CPU Util. | Host CPU U. | Host IO U. | Time |
|------------|--------------|-------------|------------|------|
| Base SQ_U | 2.79 | 2.32 | 1.61 | 2.35 |
| Base SQ_C | 1.31 | 2.02 | 1.61 | 3.16 |
| Base LQ_U | 4.28 | 0.71 | 1.60 | 4.21 |
| Base LQ_C | 1.88 | 2.53 | 1.86 | 2.87 |
| Base SR_U | 1.62 | 2.13 | 2.03 | 3.18 |
| Base SR_C | 4.53 | 3.55 | 2.12 | 2.58 |
| Base LR_U | 0.41 | 1.12 | 1.15 | 2.41 |
| Base LR_C | 4.55 | 3.21 | 0.34 | 2.85 |
| Sim. Lau. | 2.39 | 1.98 | 1.38 | 2.43 |
| Rand. Lau. | 1.78 | 1.83 | 1.40 | 2.27 |
| Overall | 2.55 | 2.14 | 1.51 | 2.82 |

TABLE 2: Mean Square Weighted Deviation for Calculated Overheads in VM CPU Util., Host CPU Util., Host IO Util., and Time

is 2.67. While the mean square weighted deviation for host disk write utilization for base cases only varies from 0.34 to 2.12. And the average mean square weighted deviation for host disk write utilization for all base cases is 1.54. As a result, the average mean square weighted deviation of host CPU utilization for all base cases is 2.19. The range of the mean square weighted deviation for the predicted VM launching time for base cases is from 2.35 to 4.21. The average mean square weighted deviation for VM launching time predicted by our model for all base cases is 2.93.

We further evaluate when more than one VMs are launched on the same host machine simultaneously. The number of simultaneous launch increases from 2 to 4. The VMs that to be launched are arbitrary selected from our four different test cases. The obtained data is given in Table 2. The mean square weighted deviation for calculated overheads in VM CPU utilization, host CPU utilization, host disk write utilization and VM launching time are 2.39, 1.98, 1.38 and 2.43, respectively. The performance of our developed model remain the same level compared to the base test cases (average mean square deviation for VM CPU utilization, host CPU utilization, host disk write utilization and VM launching time are 2.67, 2.19, 1.54 and 2.93, respectively).

For the last set of evaluations, we launch VMs under a random release pattern. Differ from previous evaluations, as soon as a VM is launched, it immediately executes an application deployed on it. Hence, the host machine has different CPU and IO utilization at different time instances when a new VM is released. We use the reference model to simulate the VM launching process in a real cloud environment using the same release pattern. From the Table 2, it is clear that the developed reference model accurately reflects the VM launching overhead, the mean square weighted deviation of calculated data is less than 2.5 from all aspects.

Overall, for all our test cases, the average mean square weighted deviation for VM CPU utilization is 2.55, the average mean square weighted deviation for host disk write utilization is 1.51, the average mean square weighted deviation for host CPU utilization is 2.14, and the average mean square weighted deviation for launch time is 2.82. With an average mean square weighted deviation less than 3 from all four aspects, we believe that our developed model can accurately reflect the VM launching process.

8 CONCLUSION

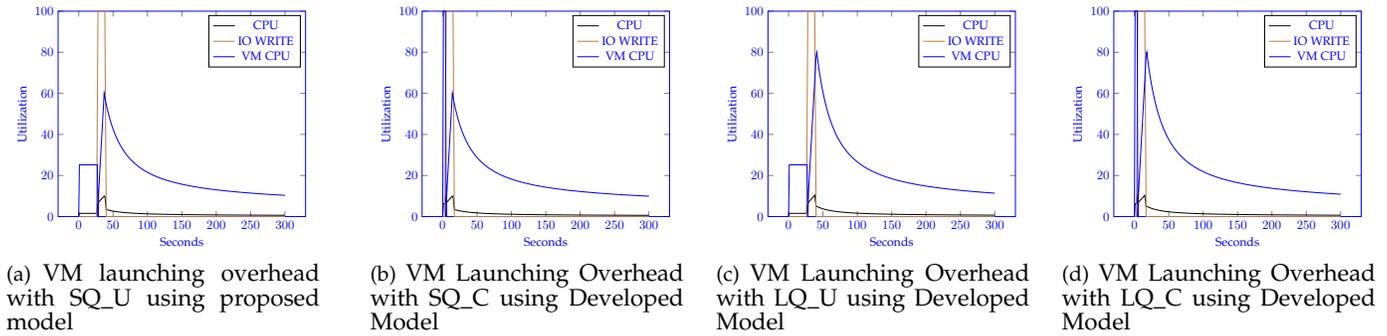


Fig. 28: VM Launching Overhead with QCOW2 Image using Developed Model

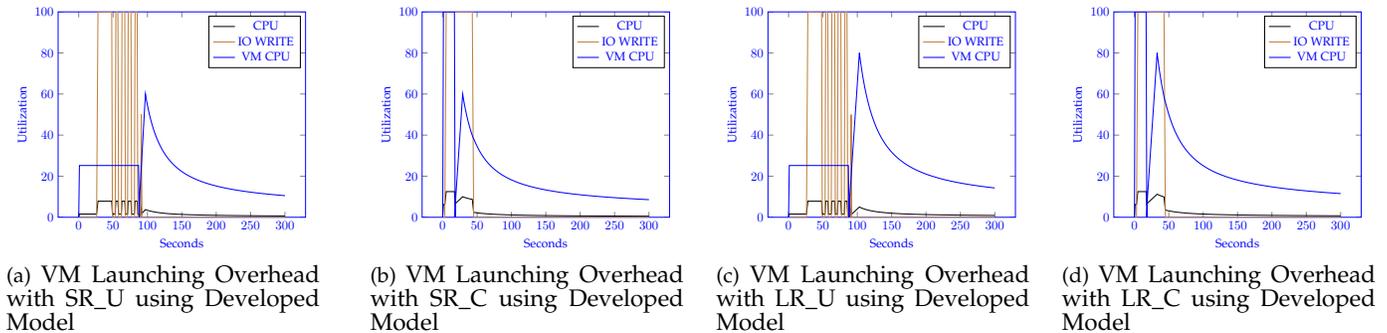


Fig. 29: VM Launching Overhead with Raw Image using Developed Model

One of the main challenges in developing the cloud bursting module is to decide *when* and *where* to launch a VM so that all resources are most effectively utilized and the system performance is optimized. We have found that the VM launching overhead has a large variation under different system states. The CPU and I/O utilizations caused by VM launching process can have significant impact on cloud bursting strategies. Hence, being able to model accurately the dependency between VM launching overhead and system resource utilization is critical in deciding *when* and *where* a VM should be launched. This paper has studied the VM launching overhead patterns based on data obtained on FermiCloud and presented a VM launching overhead reference model to represent such overhead. The evaluation shows that our proposed model can accurately predict the VM launching overhead within a mean square weighted deviation less than 3 from all four aspects, i.e. VM CPU utilization, system CPU utilization, system I/O utilization and VM launching time.

The model developed in this paper is based on SAN-based cloud infrastructures with OpenNebula and KVM as its cloud and VM management tool, respectively. We do believe that the patterns we modeled for VM launching process is applicable to other private cloud in general. It is our future work to verify our hypothesis that the model does fit different virtualization techniques, i.e. fully virtualization and hardware-assisted virtualization; different cloud management tools, i.e., OpenStack etc.; different VM management tools, i.e., Xen; and different cloud infrastructures. In addition, following the motivation of developing the reference model, our immediate work is to integrate the

reference model into the cloud bursting decision algorithms.

REFERENCES

- [1] Feature - clouds make way for STAR to shine. <http://www.isgtw.org/feature/isgtw-feature-clouds-make-way-star-shine>.
- [2] <https://www.scientificlinux.org/>.
- [3] Opennebula. <http://opennebula.org>.
- [4] Opennebula managing virtual machines. http://opennebula.org/documentation:archives:rel3.0:vm_guide_2.
- [5] P. Armstrong, A. Agarwal, A. Bishop, A. Charbonneau, R. Desmarais, K. Fransham, N. Hill, I. Gable, S. Gaudet, S. Goliath, et al. Cloud scheduler: a resource manager for distributed compute clouds. *arXiv preprint arXiv:1007.0050*, 2010.
- [6] A. Beloglazov and R. Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 577–578. IEEE, 2010.
- [7] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.
- [8] R. N. Calheiros, R. Ranjan, and R. Buyya. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 295–304. IEEE, 2011.
- [9] N. Huber, M. von Quast, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *CLOSER*, pages 563–573, 2011.
- [10] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on amazon ec2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. IEEE, 2009.

- [11] H. A. Lagar-Cavilla, J. A. Whitney, R. Bryant, P. Patchin, M. Brudno, E. de Lara, S. M. Rumble, M. Satyanarayanan, and A. Scannell. Snowflake: Virtual machine cloning as a first-class cloud primitive. *ACM Transactions on Computer Systems (TOCS)*, 29(1):2, 2011.
- [12] J. Lauret, M. Walker, S. Goasguen, and L. Hajdu. From grid to cloud, the star experience, 2010.
- [13] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012.
- [14] Y. Z. Mengxia Zhu, Qishi Wu. A cost-effective scheduling algorithm for scientific workflows in cloud. *Proceedings of 31st IEEE International Performance Computing and Communications Conference*, 2012.
- [15] R. Moreno-Vozmediano, R. Montero, and I. Llorente. IaaS cloud architecture: from virtualized data centers to federated cloud infrastructures. 2012.
- [16] S.-Y. Noh, S. C. Timm, and H. Jang. vcluster: a framework for auto scalable virtual cluster system in heterogeneous clouds. *Cluster Computing*, pages 1–9, 2013.
- [17] J. Qiu, J. Ekanayake, T. Gunarathne, J. Y. Choi, S.-H. Bae, H. Li, B. Zhang, T.-L. Wu, Y. Ruan, S. Ekanayake, et al. Hybrid cloud and cluster computing paradigms for life science applications. *BMC bioinformatics*, 11(Suppl 12):S3, 2010.
- [18] A. N. Toosi, R. N. Calheiros, R. K. Thulasiram, and R. Buyya. Resource provisioning policies to increase IaaS provider's profit in a federated cloud environment. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 279–287. IEEE, 2011.
- [19] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, and B. Berriman. Experiences using cloud computing for a scientific workflow application. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 15–24. ACM, 2011.
- [20] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, H. W. Kim, K. Chadwick, H.-J. Jang, and S.-Y. Noh. Automatic cloud bursting under fermicloud. *Workshop on Cloud Services and Systems*, 2013.
- [21] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, and S.-Y. Noh. A reference model for virtual machine launching overhead. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014)*, 2014.



Dr. Gabriele Garzoglio is head of the Grid and Cloud Services Department of the Scientific Computing Division at Fermilab and he is deeply involved in the project management of the Open Science Grid. He oversees the operations of the Grid services at Fermilab and sponsors the Cloud program in the division. Gabriele Garzoglio has a Laura degree in Physics from University of Genova, Italy, and a PhD in Computer Science from DePaul University, Chicago.

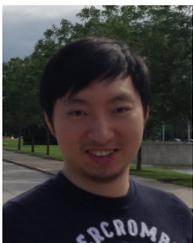


in 1995.

Dr. Steven Timm is the associate head for Cloud Computing of the Grid and Cloud Services Department at Fermilab. He has led the Fermi-Cloud project since its inception in early 2010, and been a member of the Fermilab staff since 2000. In this role he coordinates working with visiting students and guest researchers from other laboratories doing research and development on innovative techniques in distributed computing and cloud computing. He completed his PhD. Studies at Carnegie Mellon University



Gerard Bernabeu is an IT engineer and researcher of the Grid and Cloud Services Department of the Scientific Computing Division at Fermilab. He is Linux DevOps enthusiast with hand-on experience in IP networks, storage and Cloud Computing in dynamic, collaborative research communities. He received MSc in High Performance Computing and Information Theory by the Universitat Autnoma de Barcelona (UAB).



Hao Wu is now a Ph.D candidate in Computer Science Department at Illinois Institute of Technology. He received B.E in Information Security from Sichuan University, Chengdu, China, 2007. He received M.S. in Computer Science from University of Bridgeport, Bridgeport, CT, 2009. His current research interests mainly focus on cloud computing, real-time distributed open systems, Cyber-Physical System, parallel and distributed systems, and real-time applications.



Dr. Keith Chadwick is ITIL Availability and Service Continuity Manager at fermilab. During 2009 - 2013, he was the head of the Grid and Cloud Services Department of the Scientific Computing Division at Fermilab. He has been a member of the Fermilab staff since 1987. Dr. Keith received B.S. and M.A. in Physics from the University of Rochester in 1978 and 1980, respectively. He received Ph.D in Physics from the University of Rochester in 1984.



computing, and application-aware many-core virtualization for embedded and real-time applications.

Dr. Shangping Ren is an associate professor in Computer Science Department at the Illinois Institute of Technology. She earned her Ph.D from UIUC in 1997. Before she joined IIT in 2003, she worked in software and telecommunication companies as software engineer and then lead software engineer. Her current research interests include coordination models for real-time distributed open systems, real-time, fault-tolerant and adaptive systems, Cyber-Physical System, parallel and distributed systems, cloud



computer Engineering from Chungbuk National University in Korea and his M.S. and Ph.D. in Computer Science from Iowa State University, respectively. His research interests are including scientific data management, cloud & scientific computing, Linux platforms, databases, and natural language processing.

Dr. Seo-Young Noh is a principal researcher in National Institute of Supercomputing and Networking at Korea Institute of Science and Technology Information and an associate professor at Korea University of Science and Technology. He is leading the development of virtual cluster system called vcluster in conjunction with KISTI-FNAL joint project. Before joining the institutes, he worked for LG Electronics in the fields of embedded database systems and Linux mobile platforms. He received his B.E and M.E in Computer Engineering from Chungbuk National University in Korea and his

Overhead-Aware-Best-Fit (OABF) Resource Allocation Algorithm for Minimizing VM Launching Overhead

Hao Wu^{*}
Illinois Institute of Technology
10 w 31 St.
Chicago, IL, 60616
hwu28@hawk.iit.edu

Shangping Ren[†]
Illinois Institute of Technology
10 w 31 St.
Chicago, IL, 60616
ren@iit.edu

Steven Timm[‡]
Fermi National Accelerator
Laboratory
Batavia, IL, USA
timm@fnal.gov

Gabriele Garzoglio
Fermi National Accelerator
Laboratory
Batavia, IL, USA
garzogli@fnal.gov

Seo-Young Noh[§]
National Institute of
Supercomputing and
Networking,
Korea Institute of Science and
Technology Information
Daejeon, Korea
rsyoung@kisti.re.kr

ABSTRACT

FermiCloud is a private cloud developed in Fermi National Accelerator Laboratory to provide elastic and on-demand resources for different scientific research experiments. The design goal of the FermiCloud is to automatically allocate resources for different scientific applications so that the QoS required by these applications is met and the operational cost of the FermiCloud is minimized. Our earlier research shows that VM launching overhead has large variations. If such variations are not taken into consideration when making resource allocation decisions, it may lead to poor performance and resource waste. In this paper, we show how we may use an VM launching overhead reference model to minimize VM launching overhead. In particular, we first present a training algorithm that automatically tunes a given reference model to accurately reflect FermiCloud environment. Based on the tuned reference model for virtual machine launching overhead, we develop an overhead-aware-best-fit resource allocation algorithm that decides *where* and *when*

^{*}Hao Wu works as an intern in Fermi National Accelerator Laboratory, Batavia, IL, USA

[†]The research is supported in part by NSF under grant number CAREER 0746643 and CNS 1018731.

[‡]This work is supported by the US Department of Energy under contract number DE-AC02-07CH11359

[§]This work is supported by KISTI under a joint Cooperative Research and Development Agreement CRADA-FRA 2013-0001 / KISTI-C13013.

to allocate resources so that the average virtual machine launching overhead is minimized. The experimental results indicate that the developed overhead-aware-best-fit resource allocation algorithm can significantly improved the VM launching time when large number of VMs are simultaneously launched.

1. INTRODUCTION

Because of its elasticity and flexibility, cloud technology has not only benefited general purpose computing we encounter in our daily life, such as Gmail, Google docs, iCloud, to name a few. It also brings new opportunities to scientific applications. For instance, the Nimbus team at Argonne National Laboratory successfully migrates the STAR experiment at the Brookhaven National Laboratory to Amazon EC2 to avoid the shortage from local grid service [2]. Another successful example of deploying scientific applications on computer clouds is the ATLAS experiment at the Large Hadron Collider at CERN which uses Google Cloud to improve the efficiency of its research process [4].

One of the main advantages of deploying scientific applications on computer cloud is that computer cloud has "infinite" amount of resources. Scientific applications often require large amount computational resources and have long execution time. With a traditional grid system, if local resources are fully occupied by some applications, the newly arrived applications have to wait until one of the running applications finishes and releases its resources. However, with the cloud technology, even if local resources are fully occupied, public cloud resources are always available to execute newly arrived applications.

Many institutions and companies have already foreseen the advantages of deploying scientific applications on cloud and have developed their specific cloud services for scientific applications. For instance, Amazon provides a HPC cloud for scientific applications [1], Microsoft [3] and Google [4] also provide specific cloud services for scientific applications.

Fermi National Laboratory (Fermilab), as a leading physics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

research institution in United States, has developed its own private cloud – the FermiCloud to support scientific applications within Fermilab and its collaboration institutions. Since the establishment of the FermiCloud in 2010, the Fermilab Grid and Cloud Services department has smoothly integrated its grid computing infrastructure with the FermiCloud. The FermiCloud project has made significant progress through the collaboration between Fermilab, Korea Institute of Science Technology and Information (KISTI) global science experimental data hub center and Illinois Institute of Technology(IIT). In particular, we have successfully implemented an automation tool the “*vcluster*”, which allows grid worker virtual machines to run on the cloud in response to increased demand of grid jobs [11, 13]. The design goal of the *vcluster* is to automatically provisioning resources for different scientific applications so that the QoS of the scientific application is met and the operational cost of the FermiCloud is minimized.

Many solutions have been proposed by researchers to achieve the objectives that are similar to the FermiCloud design goal. The research includes minimizing application’s makespan in cloud [10], reducing energy consumption of datacenters [12], minimizing cost of the application execution [8], to name a few. However, these researches assume that virtual machine launching overhead is a constant or even negligible. As Mao *et al.* pointed out that the virtual machine launching overhead can have a very large variation in public cloud [9]. Without considering the variations of virtual machine launching overhead when designing resource allocation algorithm in cloud may lead to cost increase and resource waste.

Our earlier work has studied virtual machine launching overhead under different system conditions and developed a reference model to predict virtual machine launching overhead [14]. However, as each physical host machine has its own characteristics. Even two host machines that have the same configuration may differ in performances. In order to obtain accurate predictions, the parameters of the reference model need to be adjusted for each host machine. In this paper, we first present a training algorithm that automatically tunes the accuracy of the virtual machine launching overhead reference model for a given physical platform. Based on the tuned virtual machine launching overhead reference model, we present an overhead-aware-best-fit (OABF) resource allocation algorithm that decides *where* and *when* to allocate resources so that the average virtual machine launching overhead is minimized is proposed.

The rest of the paper is organized as follows: In Section 2, we present our prior work regarding virtual machine launching overhead and the software we have developed for virtual resource management. Section 3 discuss the details of the automatic model training algorithm. The OABF algorithm is presented in Section 4. Section 5 discusses the experimental results. We conclude our work in Section 6.

2. PRELIMINARY WORK

2.1 The *vcluster* System

The *vcluster* is a middleware jointly developed by KISTI and Fermilab for automatically managing virtual resources according based on batch job load in system [11, 13]. It consists of four major components: batch system plugin interface, cloud plugin interface, monitoring module and load balancer plugin interface. The batch system plugin interface

provides supports for communicating with different batch job systems such as HTCondor, Torque, and Sun Grid Engine (SGE). The cloud plugin interface is responsible for communicating with different cloud platforms such as FermiCloud in Fermilab, GCloud in KISTI, and public clouds such as Amazon EC2. The monitoring module collects all the information from connected batch job systems and cloud platforms. It translates the data from different batch job systems and cloud platforms into a uniform data format that can be used by load balancer interface. With the design of pluggable modules for different batch job system, cloud platforms and load balancers, the *vcluster* can be easily extended and modified if needed.

The *vcluster* has been successfully implemented on FermiCloud GCloud and Amazon EC2 [11, 13]. The source code of the *vcluster* can be found in [6, 5]. In this paper, we present the design of resource allocation mechanism used in the *vcluster*’s load balancer module which decide *when* and *where* to deploy virtual machines so that the average virtual machine launching overhead is minimized.

2.2 VM Launching Overhead Reference Model

Our early study reveals that virtual machine launching overhead has large variations under different system conditions. Based on large set of experiments conducted under operational FermiCloud [14], we have established a virtual machine launching overhead reference model. It models the CPU utilization overhead and IO utilization overhead during the process of launching a virtual machine. The CPU utilization overhead consists of two main parts. One is at the virtual machine image transferring time:

$$U_{T_t}(t) = \frac{1}{1 + e^{-0.5(T_t+t)(t-t_r)}} - \frac{1}{1 + e^{-0.5(T_t+t_r)(t-(T_t+t_r))}} \quad (1)$$

where T_t is the image transferring time and t_r is the virtual machine release time. The image transferring time is impacted by the system IO utilization and network bandwidth.

The second part of CPU utilization overhead is at the virtual machine booting time:

$$U_b(t) = c * \frac{1}{m} e^{-\gamma(1-IO_s(h,t-1))(t-T_t)} \quad (2)$$

where c and γ are two system configuration dependent constants, m is the number of cores on the host machine and $IO_s(h, t)$ represents the system’s disk IO utilization at time t . The CPU utilization overhead is also impacted by the IO utilization.

The main IO utilization overhead occurs at the time when virtual machine image is transferred. The overhead is proportional to the ratio of available IO bandwidth to the total IO bandwidth on the physical host machine. The virtual machine launching time t_b can be predicted based on the CPU utilization overhead, i.e.,

$$t_b = \max\{t | U'_b(t) \leq \epsilon\} \quad (3)$$

where $U'_b(t)$ is the first derivative of $U_b(t)$ and ϵ is the threshold to determine whether the virtual machine’s CPU utilization consumption become stable. The virtual machine launch time is calculated by adding image transfer time and boot time.

In [14], we have empirically shown that the developed reference model can accurately predict the virtual machine launching overhead. However, under different systems, the parameters of the model may change. In the next section, we present a training mechanism that can automatically adjust the parameters of the model to accurately reflect a given physical platform.

3. CALIBRATING VM LAUNCHING OVERHEAD REFERENCE MODEL ON FERMI-CLOUD PLATFORM

3.1 Prediction Accuracy Evaluation

There are many different benchmark methods to evaluate the accuracy of predictions. One of the most widely used approach is to compare the absolute error between the actual data and predicted data. Benchmarks such as mean square error (MSE), root mean square error (RMSE), mean absolute error (MAE), and median absolute error (MdAE) are the different ways to measure the absolute errors. However, as pointed by Hyndman *et al.*, one of the disadvantages the absolute error based measurements face is that they are scale-dependent [7].

Another category of measurement methods is based on the percentage error. In contrast to the absolute error based measurements, the advantage of percentage error based measurements is that they are scale-independent. The percentage error based measurement methods include mean absolute percentage error (MAPE), median absolute percentage error (MdAPE), root mean square percentage error (RM-SPE), and root median square percentage error (RMdSPE). However, the percentage error based measurements do not come flawless either. One of the main concerns with the percentage error based measurements is that the error can be infinite or undefined if the actual value is close to zero or being zero [7].

To overcome the disadvantages of the measurement methods mentioned above, Hyndman *et al.* proposed a new measurement method, i.e., the Mean Absolute Scaled Error [7]. The idea of their method is to scale the absolute error based on the in-sample MAE obtained from a benchmark prediction method. The scaled error is defined as:

$$q_t = \frac{e_t}{\frac{1}{n-1} \sum_{i=2}^n |Y_i - Y_{i-1}|} \quad (4)$$

where, e_t is the absolute error between the actual data Y_t and predicted data F_t at time point t ; and Y_i represents the actual data from i^{th} prediction. The Mean Absolute Scaled Error is:

$$\text{MASE} = \text{mean}(|q_t|) \quad (5)$$

In our model training mechanism, we use MASE to calibrate the VM launching overhead reference model for FermiCloud platforms. It is worth pointing out that though the paper focuses on FermiCloud platforms, the methodology developed in this paper can be applied to any physical cloud platforms.

3.2 Model Training Algorithm

The basic idea of the proposed training algorithm is to adjust the ϵ in equation(3) so that the mean absolute scaled error of the predicted VM launching time is maintained within a reasonable error range. The detailed algorithm is illustrated in Algorithm 1.

Each time when a new actual virtual machine launching time is read from the system, the training algorithm is executed to calibrate the accuracy of the virtual machine launching overhead model. As Line 1 to Line 4 in Algorithm 1 indicate, if the new MASE value is within the designed error range, no new calibration needs to be performed. Otherwise, a new ϵ needs to be calculated. We use an example to explain how to calculate the new ϵ .

Assume the calibrated value from last round training is $\epsilon = 0.032$. Intuitively, since all historical predictions are accurate with ϵ , when a new actual value is observed, if needed, just small calibration needs to be performed on ϵ to ensure MASE within the error range. In this case, the precision of ϵ is at thousandth level (calculated by Line 5 and obtain 0.001), then the calibration is performed at thousandth precision level. To start calibration, we first treat $\epsilon = 0.032$ as $\epsilon' = 0.030$ (Line 6) and use ϵ' as a middle start point. Then, we search the new ϵ above and below the ϵ' in a small range ($\pm 0.001 \times 10$). In this case, the search range is $[0.021, 0.039]$.

Next, the algorithm calculates the MASE of the each corresponding ϵ' from 0.021 to 0.039 and selects the ϵ' that has the smallest MASE (Line 10 to Line 30). If the smallest MASE is within the error range, the search is terminated. Otherwise, the algorithm increases the precision (Line 31) and repeat the search procedure. For instance, suppose the $\epsilon' = 0.033$, and the precision is increased to 0.0001. Since $\epsilon' = 0.033$ gives smallest MASE in the range $[0.021, 0.039]$, the desired ϵ must exist in the range of $(0.0320, 0.0340)$. The algorithm continues the search in the range of $(0.0320, 0.0340)$ starting from 0.0330 (Line 9 to Line 31). The search terminates when a ϵ' is found such that the MASE value calculated by it satisfies the error range. It is possible that the search never find such an ϵ' . Hence the other terminate condition for the algorithm is the precision reaches a predefined precision threshold.

4. OVERHEAD-AWARE-BEST-FIT RESOURCE ALLOCATION DESIGN AND IMPLEMENTATION

The previous section has discussed how we automatically calibrate the accuracy of the virtual machine launching overhead model. In this section, we present an overhead-aware-best-fit resource allocation algorithm, i.e., the OABF algorithm, that aims to reduce average virtual machine launching time and show an implementation of the OABF on the FermiCloud.

4.1 The OABF Algorithm

Our preliminary study [14] reveals that when multiple virtual machines are launched simultaneously, the VM launching time increases as the number of simultaneous launches increases. We also have noticed that when a virtual machine that being deployed uses the same image as the last virtual machine that has been deployed on the same host machine, the virtual machine takes less time to launched. In other words, the VM launching time is significantly reduced if memory cache can be used.

Algorithm 1: Model Training Algorithm

Input : Threshold ϵ , Predicted Time Set T_P , Actual Time Set T_A
Output: Threshold ϵ

```
1  $e \leftarrow \text{calculateMASE}(T_P, T_A)$ 
2 if  $e \leq \text{ErrorThreshold}$  then
3   | return  $\epsilon$ 
4 end
5  $g \leftarrow \text{calculateCurrentPrecision}(\epsilon)$ 
6  $e' \leftarrow \max\{\lfloor \frac{\epsilon}{g \times 10} \rfloor, 1\} \times g \times 10$ 
7 Recalculate predicted time set  $T_P$  using  $e'$ 
8  $e \leftarrow \text{calculateMASE}(T_P, T_A)$ 
9 do
10 for  $i \leftarrow 1$  to 9 do
11   |  $e'' \leftarrow e' - i \times g$ 
12   | if  $e'' \leq 0$  then
13     | | break
14   | end
15   | Recalculate predicted time set  $T_P$  using  $e''$ 
16   |  $e' \leftarrow \text{calculateMASE}(T_P, T_A)$ 
17   | if  $e' \leq e$  then
18     | |  $e \leftarrow e'$ 
19     | |  $\epsilon \leftarrow e''$ 
20   | end
21 end
22 for  $i \leftarrow 1$  to 9 do
23   |  $e'' \leftarrow e' + i \times g$ 
24   | Recalculate predicted time set  $T_P$  using  $e''$ 
25   |  $e' \leftarrow \text{calculateMASE}(T_P, T_A)$ 
26   | if  $e' \leq e$  then
27     | |  $e \leftarrow e'$ 
28     | |  $\epsilon \leftarrow e''$ 
29   | end
30 end
31  $g \leftarrow g/10$ 
32 while  $e \leq \text{ErrorThreshold} \vee g \leq \text{PrecisionThreshold}$ ;
33 return  $\epsilon$ 
```

Based on our previous experimental work and experimental observation, we develop an overhead-aware-best-fit resource allocation algorithm. The fundamental drive behind the algorithm is to avoid simultaneous launches and to deploy virtual machines with the same image on the same host machine sequentially. Algorithm 2 gives the pseudo code of the OABF algorithm.

Each virtual machine v in the system is characterized by its release time t_r , host h that v is deployed on and waiting time t_w that denotes the offset from its release time. Once a virtual machine is released, its release time is recorded but without any host and waiting information. By running Algorithm 2, the virtual machine is assigned to the host that is predicted to have the shortest launching time. Its waiting time that indicates actual deploy time point offsets from its release time is given.

In particular, the algorithm compares the predicted launch time for v when it is deployed on each of the hosts (Line 3 to Line 22). For each host, the algorithm calculates the virtual machine launching overhead using equation 1 and 2 and its predicted launch time using equation 3. Then it compares the predicted launch time for v when v starts at different time points (Line 9 to Line 21). As mentioned before, the intuition of the OABF algorithm is trying to avoid simultaneous launch, hence we only check the time points that the virtual machines that have been deploy on the same host before v have predicted to finish image trans-

Algorithm 2: overhead-aware-best-fit Algorithm

Input : Empty Virtual Machine $v = \{t_r, h, t_w\}$, Host set $H = \{h_1, \dots, h_n\}$, VM waiting queue $Q = \{v_1^q, \dots, v_m^q\}$
Output: Virtual Machine $v = \{t_r, h, t_w\}$ with host and waiting time information

```
1  $v.h \leftarrow \text{null}; v.t_w \leftarrow 0; h' \leftarrow \text{null}$ 
2  $t'_w \leftarrow 0; t_b \leftarrow \infty; t'_r \leftarrow v.t_r$ 
3 for  $i \leftarrow 1$  to  $n$  do
4   |  $v.t_r \leftarrow t'_r$ 
5   |  $t_p \leftarrow \text{calculatePredictLaunchTime}(h_i, v)$ 
6   | if  $t_p \leq t_b$  then
7     | |  $t_b \leftarrow t_p; h' \leftarrow h_i$ 
8   | end
9   | for  $j \leftarrow 1$  to  $m$  do
10    | | if  $v_j^q$  is deployed on  $h_i$  then
11      | | |  $t_t \leftarrow v_j^q$ 's predicted image transfer time
12      | | | if  $t_t + v_j^q.t_r + v_j^q.t_w \geq v.t_r$  then
13        | | | |  $v.t_r \leftarrow t_t + v_j^q.t_r + v_j^q.t_w$ 
14        | | | |  $t_p \leftarrow \text{calculatePredictLaunchTime}(h_i, v)$ 
15        | | | | if  $t_p + t_t + v_j^q.t_r + v_j^q.t_w - v.t_r \leq t_b$  then
16          | | | | |  $t_b \leftarrow t_p; h' \leftarrow h_i$ 
17          | | | | |  $t'_w \leftarrow t_t + v_j^q.t_r + v_j^q.t_w - v.t_r$ 
18        | | | | end
19      | | | end
20    | | end
21  | end
22 end
23  $v.t_r \leftarrow t'_r; v.h \leftarrow h'; v.t_w \leftarrow t'_w$ 
24 return  $v$ 
```

ferring process(Line 10 to Line 20). Finally, a best fit host h with shortest predicted virtual machine launching time and waiting time t_w are assigned to v (Line 23).

4.2 Implementation

The implementation of the resource allocation automation process is embedded into the load balancer module that in the *vcluster*. Figure 1 depicts the architecture of *load balancer* in the *vcluster*.

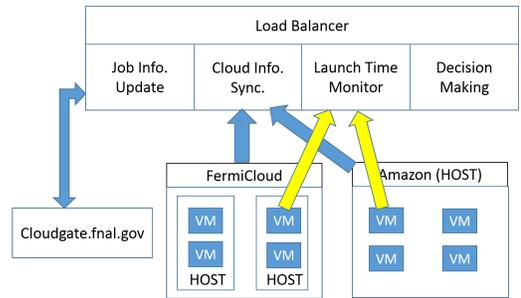


Figure 1: Architecture of Load Balancer

As shown in Figure 1, the load balancer has four sub-modules: job information update module, cloud information synchronization module, launch time monitoring module, and decision making module. The job information update module is responsible for fetching information from the batch job system. Since this paper focuses on reducing the virtual machine launching overhead, we skip the details of the job information update module.

The cloud information synchronization module is used to synchronize real time virtual machine information, host in-

formation and cloud platform information with the information predicted by load balancer. Due to the system error or manual operations, the information predicted and kept in load balancer may not be consistent with the real system information. Hence, the cloud information synchronization module is to check the consistence of stored predicted data and real system information, and ensure the correct information is provided to decision making process.

The launch time monitoring module is used to collect virtual machine’s actual launching time. Once a virtual machine is actually launched and running, it reports the time stamp to the launch time monitoring module. After the launch time monitoring module receives the time stamp, it calculates the launch time for that virtual machine and record the time for the virtual machine launching overhead training process. In the system, there is a virtual machine waiting queue contains all the virtual machine that yet to be launched or the virtual machine under launching process. Once a virtual machine that in the waiting queue reports its actual launching time, the virtual machine is removed from the waiting queue.

The decision making module executes the resource allocation algorithm. It takes the information from other three modules to decide *where*, *when* and *what* to launch virtual machines. In this paper, we only focus on *when* and *where* to launch the virtual machine so that the average virtual machine launching overhead is minimized.

Figure 2 depicts the workflow of resource allocation automation process. It illustrates how different modules cooperate to automatically allocate resources and calibrate the virtual machine launching overhead reference model.

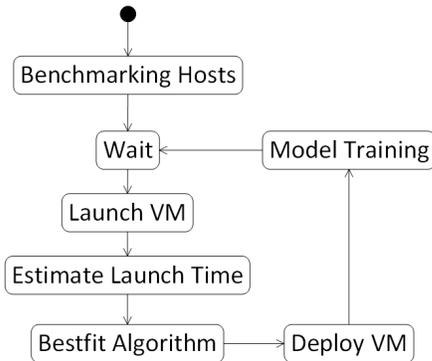


Figure 2: Resource Allocation Automation Workflow

When the load balancer starts, it first benchmarks all host machine’s performance, i.e. disk I/O bandwidth, network bandwidth, etc. It then goes into waiting state. Once a virtual machine request is released, the load balancer needs to decide *when* and *where* to launch the virtual machine. Based on the virtual machine launching overhead reference model, a predicted launch time is calculated. The predicted launch time is adapted by OABF algorithm to determine the host machine where the virtual machine should be deployed on. Finally, the virtual machine is initialized and deployed on the assigned host machine. After the virtual machine is launched, the load balancer uses the actual launch time to calibrate the accuracy of the model using Algorithm 1.

5. EVALUATION

The overhead-aware-best-fit algorithm is evaluated under real cloud environment–FermiCloud. Since FermiCloud is designed for scientific applications and it is very likely that when an application that needs large amount resources is submitted to the system, large number of virtual machines are needed to be launched simultaneously. Hence, our evaluation focuses on the performance of the propose OABF algorithm under larger number of simultaneous launches.

5.1 Experiment Setting

The experiments are performed under FermiCloud. We use total ten host machines for the experiment. All ten hosts are configured with 8-core Intel(R) Xeon(R) CPU X5355 @ 2.66GHz and 16GB memory. All these machines are connected through high speed Ethernet. We use OpenNebula as the cloud platform. The OpenNebula front end server has 16-core Intel(R) Xeon(R) CPU E5640 @ 2.67GHz, 48GB memory.

5.2 Launching Time Comparison

In order to evaluate the performance of the developed OABF algorithm, we compare the virtual machine launching time when the OpenNebula default scheduler is used with the virtual machine launching time when the OABF algorithm is used. Each time, we launch seventy (70) virtual machines simultaneously. All virtual machines are using the same image. Same set of contextualization scripts are used for launching VMs. The first experiment is to launch virtual machines using QEMU Copy On Write images (QCOW2) with size 2.6GB.

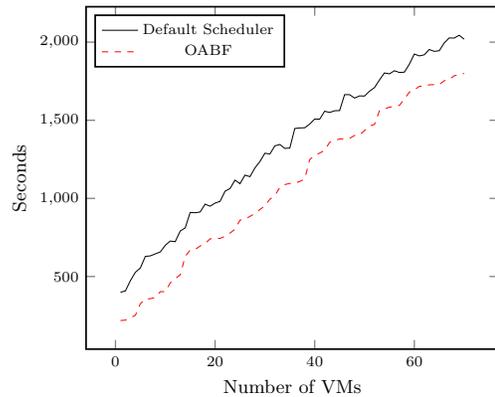


Figure 3: VM Launching Time Comparison using QCOW2 Image

Figure 3 depicts the comparison of virtual machine launching time with the OpenNebula default scheduler and virtual machine launching time with the OABF algorithm. All the virtual machines are ordered by their launching time in increasing order. From the Figure 3, it is clear that with the OABF algorithm, virtual machines take less time to launch than using OpenNebula default scheduler. The launching time reduction from the OABF is rather stable from the first virtual machine to the last virtual machine. The maximum reduction among the seventy virtual machine is 349 seconds. On average, the virtual machine launching time reduction is 245.44 seconds, which is more than four minutes reduction compared to the OpenNebula default scheduler.

The second experiment is to test the performance on large images. We also launch seventy (70) virtual machines at a

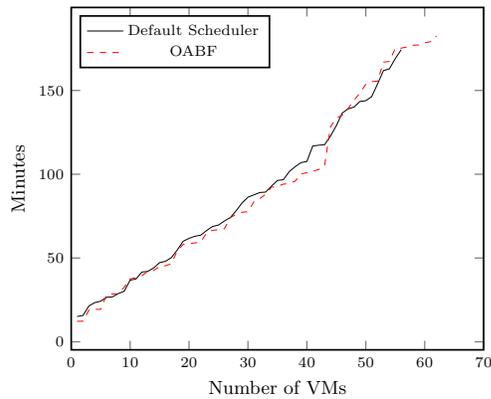


Figure 4: VM Launching Time Comparison using RAW image

time. Each virtual machine now use RAW image with size 16 GB. Figure 4 dispatches the comparison results between virtual machines' launching time with the OpenNebula default scheduler and virtual machines' launching time with the OABF algorithm. As shown in Figure 4, when the image size is large, the virtual machine launching time reduction from OABF can be clearly observed. The maximum reduction from OABF algorithm is 907 seconds which is about 15 minutes saving.

In our experiments, we have also observed that after 50 virtual machines being launched, the remaining virtual machines takes longer time to launch when using the OABF algorithm than with the OpenNebula default scheduler. This is because most of the remaining virtual machines (15 out of 20) are failed to launch under OpenNebula default scheduler due to large amount I/O operations, hence there are just few virtual machines are actually undergoing launching process. On the other hand, when using the OABF algorithm, only five virtual machines are failed to launch. The OABF algorithm not only reduces the virtual machines' launching times when large amount simultaneous launches occur but also can improve the success rate of such extreme scenario. With the OABF algorithm, on average, the virtual machine saves 103 seconds on launching process compared with the OpenNebula default scheduler.

6. CONCLUSION

The FermiCloud is a private cloud built in Fermi National Accelerator Laboratory to provide on-demand resources for different scientific applications. The design goal of FermiCloud is to automatically provision resources for different scientific applications so that the QoS of the scientific application is met and the operational cost of FermiCloud is minimized. The main challenge of designing the FermiCloud system is to decide *when* and *where* to allocate resources so that the goals are met. In this paper, we present a mechanism to automatically train the VM launching overhead reference model that we previously developed. Based on the virtual machine launching overhead reference model, we have developed in this paper an overhead-aware-best-fit resource allocation algorithm to help the cloud reduce the average VM launching time. In the paper, we have also presented an implementation of the developed OABF algorithm on FermiCloud. The experimental results indicate that the

OABF can significantly reduce the VM launching time (reduced VM launch time by 4 minutes on average) when large number of VMs are launched simultaneously.

7. REFERENCES

- [1] AWS HPC cloud computing. <http://aws.amazon.com/hpc/>.
- [2] Feature - clouds make way for STAR to shine. <http://www.isgtw.org/feature/isgtw-feature-clouds-make-way-star-shine>.
- [3] High performance computing on microsoft azure for scientific and technical applications. <http://research.microsoft.com/en-us/projects/azure/high-perf-computing-on-windows-azure.pdf>.
- [4] Mapping the secrets of the universe with google compute engine. <https://cloud.google.com/developers/articles/mapping-the-secrets-of-the-universe-with-google-compute-engine?hl=ja>.
- [5] Source code for FermiCloud version vcluster. <https://github.com/philip-wu5/project/tree/fermi>.
- [6] Source code for KISTI version vcluster. <https://github.com/vcluster/project>.
- [7] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, 22(4):679–688, 2006.
- [8] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.
- [9] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012.
- [10] Y. Z. Mengxia Zhu, Qishi Wu. A cost-effective scheduling algorithm for scientific workflows in cloud. *Proceedings of 31st IEEE International Performance Computing and Communications Conference*, 2012.
- [11] S.-Y. Noh, S. C. Timm, and H. Jang. vcluster: A framework for auto scalable virtual cluster system in heterogeneous clouds. *Cluster Computing*. To appear.
- [12] A. M. Sampaio and J. G. Barbosa. Optimizing energy-efficiency in high-available scientific cloud environments. In *Cloud and Green Computing (CGC), 2013 Third International Conference on*, pages 76–83. IEEE, 2013.
- [13] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, H. W. Kimy, K. Chadwick, H. Jang, and S.-Y. Noh. Automatic cloud bursting under fermicloud. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 681–686. IEEE, 2013.
- [14] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, and S.-Y. Noh. Modeling the virtual machine launching overhead under fermicloud. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 374–383. IEEE, 2014.

Automatic Installation and Deployment of Network File System and On-Demand Caching Service on Dynamically Instantiated Large Scale Batch of Virtual Machines On Private and Public Clouds

Sandeep Palur* Steven Timm† Dr. Ioan Raicu*

psandeep@hawk.iit.edu timmm@fnal.gov iraicu@cs.iit.edu

*Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA

†Scientific Computing Division, Fermi National Accelerator Laboratory, Batavia IL, USA

Abstract – *Big scientific experiments usually need a lot of virtual machines for running scientific workflows on private and public clouds. Virtual machine images can be large and the software they need inside them can be constantly changing. The best solution is to use CERN Virtual Machine File System - read only, runtime download, and caching http file system. It is a read-only network file system that provides access to files from a CVMFS Server over HTTP. When CVMFS client runs on a groups of worker nodes that share the same cloud, a HTTP web proxy, inside the same cloud, can be used to cache the file system contents, so that all subsequent requests for that file will be delivered from the local HTTP web proxy) and doesn't have to hit the Internet. Typically, a High Energy Physics (HEP) computing site has a local or regional Squid HTTP web proxy, with the central CVMFS servers located at the main laboratory, such as CERN for the LHC experiments. Each VM has a list of the available Squid servers and, in most cases, the Squids are remote. The optimal Squid may be different depending on the location of the cloud. Further, one can imagine dynamically instantiating Squid servers in an opportunistic cloud environment to meet application demand. As a result, we use Shoal as a service that can dynamically publish and advertise the available Squid servers. In this work, we automate installation and deployment of CVMFS(network file system), Squid(on-demand caching service) and Shoal(squid cache publishing and advertising tool designed to work in fast changing environments) on dynamically instantiated large scale batch of virtual machines on Fermi Cloud (private cloud) and Amazon Web Services (Public cloud).*

I. INTRODUCTION

The CERN Virtual Machine File System (CVMFS) [1] is widely adopted by the High Energy Physics

(HEP) community for the distribution of project software. CVMFS is a read-only network file system that provides access to files from a CVMFS Server over HTTP. When CVMFS is used on a cluster of worker nodes, a HTTP web proxy can be used to cache the file system contents, so that all subsequent requests for that file will be delivered from the local HTTP proxy server. Typically, a HEP computing site has a local or regional Squid HTTP web proxy [2], with the central CVMFS servers located at the main laboratory, such as CERN for the LHC experiments.

The use of IaaS cloud resources is becoming a realistic solution for HEP workloads [3, 4], and CVMFS is an effective means of providing the software to the virtual machines (VMs). Each VM has a list of the available Squid servers and, in most cases, the Squids are remote. The optimal Squid may be different depending on the location of the cloud. Further, one can imagine dynamically instantiating Squid servers in an opportunistic cloud environment to meet application demand. However, there is currently no mechanism other than Shoal for locating the optimal Squid server. As a result, we use Shoal as a service that can dynamically publish and advertise the available Squid servers. Shoal is ideal for an environment using both static and dynamic Squid servers.

A. CERN Virtual Machine File System

The CernVM File System (CernVM-FS) provides a scalable, reliable and low maintenance software distribution service. It was developed to assist High Energy Physics (HEP) collaborations to deploy software on the worldwide-distributed computing infrastructure used to run data processing applications. CernVM-FS is implemented as a POSIX read-only file system in user space (a FUSE module). Files and directories are hosted on standard web servers and mounted in the universal namespace /cvmfs. Internally, it uses content-addressable storage and Merkle trees in order to maintain file data and meta-data. CernVM-FS uses outgoing HTTP connections only, thereby it avoids most of the firewall issues of other network file systems. It is actively used by small and large HEP collaborations.

In many cases, it replaces package managers and shared software areas on cluster file systems as means to distribute the software used to process experiment data.

B. Squid

Squid is a caching proxy for the Web supporting HTTP, HTTPS, FTP, and more. It reduces bandwidth and improves response times by caching and reusing frequently-requested web pages. Squid has extensive access controls and makes a great server accelerator. It runs on most available operating systems, including Windows and is licensed under the GNU GPL.

C. Shoal

Shoal is divided into three logical modules, a server, an agent, and a client. Each package is uploaded to the Python Package Index [5] (the standard method of distributing new components in the Python language).

Each component is designed to provide the functionality of different parts of the system as follows:

Shoal Server - is responsible for the following key tasks:

1. Maintaining a list of active Squid servers in volatile memory and handling AMQP messages sent from active Squid servers.
2. Providing a RESTful interface for Shoal Clients to retrieve a list of geographically closest Squid servers.
3. Providing basic support for Web Proxy Auto-Discovery Protocol (WPAD).
4. Providing a web user interface to easily view Squid servers being tracked.

Shoal Agent - is a daemon process run on Squid servers to send an Advanced Message Query Protocol (AMQP) [6] message to Shoal Server on a set interval. Every Squid server wishing to publish its existence runs Shoal Agent on boot. Shoal Agent sends periodic heartbeat messages to the Shoal Server (typically every 30 seconds).

Shoal Client - is used by worker nodes to query Shoal Server to retrieve a list of geographically nearest Squid servers via the REST interface. Shoal Client is designed to be simple (less than 100 lines of Python) with no dependencies beyond a standard Python installation.

Shoal Server runs at a centralized location with a public IP address. For agents (i.e. Squid servers), Shoal Server will consume the heartbeat messages sent and maintain an up-to-date list of active Squids. For clients, Shoal Server will return a list of Squids organized by geographical distance and load. For regular users of Shoal Server, a web server is provided. The web server generates dynamic web pages that display an overview of Shoal. All of the tracked Squid servers are displayed and updated periodically on Shoal Server's web user interface, and all client requests are available in the access logs.

AMQP forms the communications backbone of Shoal Server. All information exchanges between Shoal Agent (Squid Servers) and Shoal Server are done using this protocol, and all messages are routed through a RabbitMQ [7] Server.

II. DESIGN AND IMPLEMENTATION

This project aims to automate installation and deployment of CVMFS(network file system), Squid(on-demand caching service) and Shoal(squid cache publishing and advertising tool designed to work in fast changing environments) on dynamically instantiated large scale batch of virtual machines on Fermi Cloud (private cloud) using Puppet Master/Agent and Amazon Web Services (Public cloud) using Serverless Puppet.

As a part of this work, we developed puppet modules and scripts for installing and deploying shoal client, agent and server, script to dynamically update the proxy address. We also fixed potential bugs in Shoal Server and made it suitable to publish Squid Servers running on EC2 instances that does not have a static public IP.

A. Architecture

The architecture of large scale batch of dynamically instantiated FermiCloud and EC2 worker nodes provided with network file system, on-demand caching service and a cache publishing and advertising tool is shown in Figure1. It consists of the following components:

a) **Worker Node Installed with Shoal Client** – When a worker node is instantiated dynamically, CVMFS client and Shoal Client are installed on start up of the machine. Shoal Client is a cron job that queries the Shoal Server using the REST interface to get the closest Squid Server and is configured to run every 2 hours. Shoal Client updates the proxy address in the CVMFS configuration file. So that when

CVMFS client tries to download any software from CVMFS server, the request passes through the Squid Server, whose IP address is configured in CVMFS configuration file.

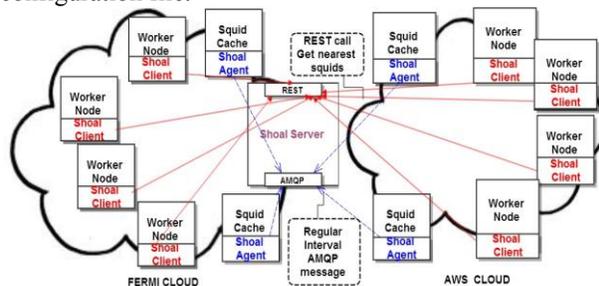


Figure1: **Architecture of Dynamically Instantiated Fermi Cloud and EC2 Instances with On-Demand Caching Service and a Cache Publishing Service**

b) Squid Server Installed with Shoal Agent –

When a server node is instantiated dynamically, Squid Server and Shoal Agent are installed on the start up of the machine. Shoal Agent running alongside with Squid Server, sends periodic heartbeat messages (IP Address, Load, etc) to the Shoal Server typically every 30 seconds.

c) Shoal Server Installed with RabbitMQ Server and Apache Server-

When a server node is instantiated dynamically, Shoal Server is installed on start up of the machine. Shoal Server provides two major functions: provides a RESTful interface (hosted on Apache server) for Shoal Clients to retrieve a list of geographically closest Squid servers and maintains a list of active Squid servers (active squid servers send AMQP messages to RabbitMQ server) in volatile memory.

Since we have our Worker Nodes split over in two different clouds (AWS and FermiCloud), the idea is to install sufficient Squid Servers on both the clouds and restrict the Worker Nodes to use the Squid Servers in their local cloud for the following reasons:

- 1) We don't want to open Fermilab cache servers to outside internet
- 2) Reduce data transfer from Internet.
- 3) Faster data transfers.
- 4) Reduce Latency

When a Shoal Client on any Worker Node queries for nearest Squid Servers, it is responded back with the IP addresses of Squid Servers running inside the local cloud because the Shoal Server finds the closest Squid server to the Worker Node. Thus only the first

Worker Node in each cloud downloads the software from Internet (CVMFS server) and rest of the Worker Nodes that needs the same software, takes it from the local Squid Server.

III. CONCLUSION AND FUTURE WORK

We automate installation and deployment of CVMFS(network file system), Squid(on-demand caching service) and Shoal(squid cache publishing and advertising tool designed to work in fast changing environments) on dynamically instantiated large scale batch of virtual machines on Fermi Cloud (private cloud) and AWS(public cloud) by installing and deploying all the required software on start up of the instances and also restrict the Worker Nodes to use the Squid Servers running on the local cloud thereby reducing the number of hits to the Internet.

Our future work includes:

- a) Installation and deployment on a sum of 1000 virtual machines on both AWS and Fermi Cloud
- b) Benchmarking this work at high scales.

IV. REFERENCES

[1] J. Blomer et al, Status and future perspectives of CernVM-FS J. Phys.: Conf. Ser. 396052013, doi:10.1088/1742-6596/396/5/052013
 [2] Squid - HTTP proxy server <http://www.squid-cache.org>
 [3] F. H. B. Megino et al. Exploiting Virtualization and Cloud Computing in ATLAS J. Phys.: Conf. Ser. 396032011, doi:10.1088/1742-6596/396/3/032011
 [4] I. Gable et al, A batch system for HEP applications on a distributed IaaS cloud J. Phys.: Conf. Ser. 331062010, doi:10.1088/1742-6596/331/6/062010
 [5] Python Package Index <https://pypi.python.org/>
 [6] S.Vinoski, Advanced Message Queuing Protocol, IEEE Internet Computing 10 87, doi:10.1109/MIC.2006.116
 [7] RabbitMQ - AMQP Messaging software, <http://www.rabbitmq.com>

```

File: puppetrepo-shoal/files/set-session-proxy.sh
proxy="/usr/bin/shoal-client -d`
#echo $proxy
if [[ $proxy =~ 'export http_proxy=http://' ]];
then
$proxy > /usr/run-export-command.sh
chmod 755 /usr/run-export-command.sh
source /usr/run-export-command.sh
echo "Proxy set successfully!"
else
echo "Proxy not set!"
echo "No squid servers are active currently!"
fi
-----
-----
-----

```

```

File: puppetrepo-shoal/files/shoal-client
#!/usr/bin/python
"""
    Very simple client script used to get nearest squid server using the RESTful API.
"""
import urllib2
import sys
# import json
import re
import os
import logging
import time

from shoal_client import config as config

```

```

from optparse import OptionParser
from urllib2 import urlopen

server = config.shoal_server_url
cvdfs_config = config.cvdfs_config
default_http_proxy = config.default_squid_proxy

data = None
dump = False
closest_http_proxy = ''
http_proxy_formatted = ''
cvdfs_http_proxy = "CVMFS_HTTP_PROXY="

logging.basicConfig(filename="shoal_client.log", format='%(asctime)s %(message)s')

def get_args():
    """
        gets server and dump variables from command line arguments
    """
    global server
    global dump

    p = OptionParser()

```

```

p.add_option("-s", "--server", action="store", type="string", dest="server",
             help="Also needs string for specifying the shoal server to use. " +
             "Takes presedence over the option in config file")
p.add_option("-d", "--dump", action="store_true", dest="dump",
             help="Print closest proxies to terminal for debugging "+
             "instead of over writing the CVMFS config file")
(options, args) = p.parse_args()

if options.server:
    server = options.server
if options.dump:
    dump = True

def convertServerData(val):
    """
    converts val to digits if it's not already or else return None
    """
    if val.isdigit():
        return int(val)
    else:
        try:
            return float(val)
        except:
            if "null" in val:
                return None
            else:
                return unicode(val.strip("\\"))

# TODO is this parser sufficient or should a full JSON parser be implemented?
# Seperating out the list of properties should be done but does support
# for arbitrary json strings add anything?
def parseServerData(jsonStr):
    """
    creates a multidimensional server data dictionary indexed by
    unicode integers with dataTypes, geo_data and geoDataTypes. Each
    respective entry holds the appropriate dataTypes and geoDataTypes
    found in jsonStr
    """

    # TODO should load this from a config file as it has to match the server
    # Nested properties (i.e geo_data) needs to be handled separately
    dataTypes = ["load", "distance", "squid_port", "last_active", "created",
                 "external_ip", "hostname", "public_ip", "private_ip"]

    geoDataTypes = ["city", "region_name", "area_code", "time_zone", "dma_code",
                    "metro_code", "country_code3", "latitude", "postal_code",
                    "longititude", "country_code", "country_name", "continent"]

    # don't really care about data here

```

```

# it is just a simple way to get number of nearest squids
p = re.compile("\"" + dataTypes[0] + "\": ([^,]+)")
numNearestSquids = len(p.findall(jsonStr))
## compiles regex "load": ([^,]+), although it doesn't really matter that fact that it's a load
## this will find the number of above matches in jsonStr and return into numNearestSquids
## therefore each match in json is a 'nearest' squid

# initialize the dictionaries
outDict = {}
for i in range(0, numNearestSquids):
    outDict[unicode(str(i))] = {}
    outDict[unicode(str(i))][unicode("geo_data")] = {}
## creates a multidimensional dict with each key 1 being u'i' (i in unicode)
## and key 2 being "geo_data" for all entries

# TODO probably don't need seperate regexes
# test using geodata one for both
## for each item in dataTypes, compile a regex for that item and find all the matches with jsonStr
## and put those matches in dataList.
for dataType in dataTypes:
    p = re.compile("\"" + dataType + "\": ([^,]+)[,|]\""
    dataList = p.findall(jsonStr)
    for i, val in enumerate(dataList):
        outDict[unicode(str(i))][unicode(dataType)] = convertServerData(val)

```

```

## outDict is a multidimensional dict that now holds a val in each dataType per i
## same as above just for geoDataTypes
for geoDataType in geoDataTypes:
    p = re.compile("\"" + geoDataType + "\": ([\"^\"]*|[,]*)")
    dataList = p.findall(jsonStr)
    for i, val in enumerate(dataList):
        outDict[unicode(str(i))][unicode("geo_data")][unicode(geoDataType)] = convertServerData(val)
## outDict in geo_data for each geoDataType holds a val
return outDict
get_args()

"""
opens up a url to a server, parses server data from there
from that data, creates addresses to each squid, stores in cvmfs_http_proxy
overwrites cvmfs config file with cvmfs_http_proxy if dump is not specified
"""
## CVMFS = CERN Virtual Machine File System

## reads in server data (if it can be read in) into a dictionary called data
try:
    f = urlopen(server)
    # data = json.loads(f.read())
    data = parseServerData(f.read())
except (urllib2.URLError, ValueError), e:
    logging.error("Unable to open url. %s" % e)

```

```

data = None

if data:
    ## iterates through the data dict and uses all hostname and squid_port keys
    ## to create addresses for each squid in closest_http_proxy
    for i in range(0, len(data)):
        try:
            closest_http_proxy += 'http://%s:%s;' % (data['%s%i']['hostname'], data['%s%i']['squid_port'])
            http_proxy_formatted+='http://%s:%s' % (data['%s%i']['hostname'], data['%s%i']['squid_port'])+ "/"
        except KeyError, e:
            logging.error("The data returned from '%s' was missing the key: %s. Please ensure the url is running
the latest Shoal-Server." % (server, e))
            sys.exit(1)

cvmfs_http_proxy += "\"" + closest_http_proxy + "\"\n"

## if dump -> don't overwrite CVMFS config file
if dump:
    #print "%s would have been written to the CVMFS config file", cvmfs_http_proxy
    print "export http_proxy="+http_proxy_formatted

else:
    # attempt to read the cvmfs_config file, no point in continuing if it can't be read
    try:

```

```

        f = open(cvmfs_config)
        lines = f.readlines()
    except:
        logging.error("Could not open and read the CVMFS config file")
        sys.exit(1)

f.close()
# create a list of tuples of lines/line numbers that have "CVMFS_HTTP_PROXY"
CVMFS_proxy_lines = [t for t in enumerate(lines) if "CVMFS_HTTP_PROXY" in t[1]]
# if there is only one can just replace it with the new proxy
if len(CVMFS_proxy_lines) == 1:
    lines[CVMFS_proxy_lines[0][0]] = cvmfs_http_proxy
# add a line if it doesn't exist
elif len(CVMFS_proxy_lines) == 0:
    lines += cvmfs_http_proxy
# something is wrong with the CVMFS config; there are multiple entries
# fixing it by changing the first and removing the extras
else:
    logging.error("CVMFS file had duplicate CVMFS_HTTP_PROXY entries;" +
        " writing the first deleting the rest")
    lines[CVMFS_proxy_lines[0][0]] = cvmfs_http_proxy
    for line in CVMFS_proxy_lines[1:]:
        lines[line[0]] = ""
# open the file again this time for writing and replace its contents with the modified lines

```

```

try:
    try:
        f = open(cvmfs_config, "w")
        f.writelines(lines)
    except Exception:
        logging.error("Could not write CVMFS config file")
finally:
    f.close()

```

```

-----
-----
-----

```

```

File: puppetrepo-shoal/files/shoal-server.py
#!/usr/bin/python
import os
import sys
import logging

from threading import Thread
from time import sleep
from shoal_server import config
from shoal_server.shoal import ThreadMonitor, WebpyServer

def main():

    shoal_list = {}
    threads = []

    # establishes ThreadMonitor and its thread
    monitor_thread = ThreadMonitor(shoal_list)
    monitor_thread.daemon = True
    threads.append(monitor_thread)

    # establishes WebpyServer and its thread
    webpy_thread = WebpyServer(shoal_list)
    webpy_thread.daemon = True
    threads.append(webpy_thread)

    # starts running threads

```

```

for thread in threads:
    thread.start()

# keep running threads until KeyboardInterrupt
#try:
#while True:
#    for thread in threads:
#        if not thread.is_alive():
#            logging.error('{0} died.'.format(thread))
#            sys.exit()
#    #sleep(1)
#except KeyboardInterrupt:
#    sys.exit()
if __name__ == '__main__':
    # sets up logging file
    shoal_dir = config.shoal_dir
    log_file = config.log_file
    log_format = '%(asctime)s - %(levelname)s - [%(filename)s:%(lineno)s] - %(message)s'

    try:
        logging.basicConfig(level=logging.ERROR, format=log_format, filename=log_file)
    except IOError as e:
        sys.exit(1)

    # change working directory so webpy static files load correctly.
    try:
        os.chdir(shoal_dir)
    except OSError as e:
        sys.exit(1)
    main()

```

```

-----
-----
-----

```

```

File: puppetrepo-shoal/files/shoal-server.sh
#!/bin/bash
#This script requires puppet installed.
#One needs to be root to apply this.
#This script install shoal server

if [ "$(whoami)" != "root" ]; then
    echo "This should be run by root"
    exit 1
fi

```

```

mkdir -p /etc/puppet/modules

if [ ! -f "/usr/bin/git" ]; then
    yum install git -y
fi

if [ ! -d "/etc/puppet/modules/shoal" ]; then
    git clone ssh://p-puppetrepo@cdcvcs.fnal.gov/cvs/projects/puppetrepo-shoal
    mkdir -p /etc/puppet/modules/shoal && cp -r puppetrepo-shoal/* /etc/puppet/modules/shoal
else
    cd puppetrepo-shoal
    git pull
    cd ..
    cp -rf puppetrepo-shoal/* /etc/puppet/modules/shoal
fi

if [ ! -d "/etc/puppet/modules/rabbitmq" ]; then
    git clone https://github.com/jcochard/puppet-rabbitmq.git
    mkdir -p /etc/puppet/modules/rabbitmq && mv puppet-rabbitmq/* /etc/puppet/modules/rabbitmq
fi

if [ ! -d "/etc/puppet/modules/erlang" ]; then
    puppet module install dcarley/erlang
fi

if [ ! -d "/etc/puppet/modules/epel" ]; then
    puppet module install stahma/epel
fi

if [ ! -d "/etc/puppet/modules/stdlib" ]; then
    puppet module install puppetlabs/stdlib
fi

cat << EOF | puppet apply
include epel
include shoal::server_dependencies
include erlang
include rabbitmq
include shoal::server
include shoal::host_server
Class['epel'] -> Class['shoal::server_dependencies'] -> Class['erlang'] -> Class[rabbitmq] ->
Class['shoal::server'] -> Class['shoal::host_server']
EOF

```

```

-----
-----

```

```

-----

File: puppetrepo-shoal/files/shoal.conf
WSGIDaemonProcess shoal user=apache group=apache threads=10 processes=1
WSGIScriptAlias / /var/www/shoal/scripts/shoal_wsgi.py
WSGIProcessGroup shoal

```

```
Alias /static /var/www/shoal/static/
```

```
AddType text/html .py
```

```
<Directory /var/www/shoal/>
    Order deny,allow
    Allow from all
</Directory>
```

```

-----
-----

```

```

File: puppetrepo-shoal/files/squid.conf
acl NET_LOCAL src 10.0.0.0/8 172.16.0.0/12 192.168.0.0/16
acl HOST_MONITOR src 131.225.152.0/23
acl snmppublic snmp_community HOST_MONITOR
acl all src all
acl manager proto cache_object
acl localhost src 127.0.0.1/32
acl to_localhost dst 127.0.0.0/8 0.0.0.0/32
acl localnet src 10.0.0.0/8 # RFC1918 possible internal network
acl localnet src 172.16.0.0/12 # RFC1918 possible internal network
acl localnet src 192.168.0.0/16 # RFC1918 possible internal network
acl SSL_ports port 443
acl Safe_ports port 80 # http
acl Safe_ports port 21 # ftp
acl Safe_ports port 443 # https
acl Safe_ports port 70 # gopher
acl Safe_ports port 210 # wais
acl Safe_ports port 1025-65535 # unregistered ports
acl Safe_ports port 280 # http-mgmt
acl Safe_ports port 488 # gss-http
acl Safe_ports port 591 # filemaker
acl Safe_ports port 777 # multiling http
acl CONNECT method CONNECT
http_access allow manager localhost
http_access deny manager
http_access deny !Safe_ports

```

```

http_access deny CONNECT !SSL_ports
http_access allow NET_LOCAL
acl our_networks src 131.225.0.0/16 127.0.0.1
http_access allow our_networks
http_access allow localhost
http_access deny all
acl PURGE method PURGE
http_access allow PURGE localhost
http_access deny PURGE
reply_body_max_size 1000000000 allow all
icp_access allow localnet
icp_access deny all
http_port 3128
hierarchy_stoplist cgi-bin
cache_mem 1024 MB
maximum_object_size_in_memory 256 KB
cache_dir ufs /var/spool/squid 290000 16 256
maximum_object_size 10 GB
logformat awstats %>a %ui %un [%d/%b/%Y:%H:%M:%S +0000]t1l "%rm %ru HTTP/%rv" %Hs %<st %Ss:%Sh %tr "%{X-
Frontier-Id}>h" "%{Referer}>h" "%{User-Agent}>h"
access_log /var/log/squid/access.log awstats
logfile_daemon /usr/libexec/squid/logfile-daemon
cache_log /var/log/squid/cache.log
cache_store_log none
mime_table /etc/squid/mime.conf
pid_filename /var/run/squid/squid.pid
strip_query_terms off
unlinkd_program /usr/libexec/squid/unlinkd
refresh_pattern ^ftp: 1440 20% 10080
refresh_pattern ^gopher: 1440 0% 1440
refresh_pattern -i /cgi-bin/ 0 0% 0
refresh_pattern \.crl$ 60 25% 1440
refresh_pattern \.der$ 60 25% 1440
refresh_pattern \.pem$ 60 25% 1440
refresh_pattern \.r0$ 60 25% 1440
refresh_pattern \.pacman$ 60 10% 1440
refresh_pattern . 60 20% 4320
negative_ttl 1 minute
acl shoutcast rep_header X-HTTP09-First-Line ^ICY.[0-9]
upgrade_http0.9 deny shoutcast
acl apache rep_header Server ^Apache
broken_vary_encoding allow apache
collapsed_forwarding on
connect_timeout 30 seconds
read_timeout 1 minute
request_timeout 1 minute
client_lifetime 1 hour
cache_mgr fermigrid-help@fnal.gov

```

```

cache_effective_user squid
cache_effective_group squid
umask 022
snmp_access allow snmppublic HOST_MONITOR
snmp_access deny all
icp_port 0
icon_directory /usr/share/squid/icons
error_directory /usr/share/squid/errors/English
ignore_ims_on_miss on
coredump_dir /var/spool/squid

```

```

-----
-----
-----

```

```

File: puppetrepo-shoal/manifests/agent.pp
# The shoal agent runs in the squid server and announces it to the shoal server

```

```

class shoal::agent(
  $shoal_server_ip = undef,
)
{
  if($shoal_server_ip)
  {
    package { "shoal-agent":
      ensure => installed,
    }

    file_line { '/etc/shoal/shoal_agent.conf':
      path => '/etc/shoal/shoal_agent.conf',
      line => "amqp_server_url = ${shoal_server_ip}",
      match => "amqp_server_url =.*$",
      require => Package["shoal-agent"],
      notify => Service["shoal-agent"],
    }

    service { 'shoal-agent':
      ensure => running,
      enable => true,
      hasstatus => false,
      #hasrestart => true,
      path => "/usr/bin",
      #require => File_line["shoal-agent"],
    }
  }
}

```

```

}
else{
  warning('Shoal agent is NOT INSTALLED. please pass a valid ip to the parameter "shoal_server_ip" and run it
again' )
  warning('eg: shoal_server_url => shoalserver.domain')
}
}
}

```

```

-----
-----
-----

```

File: puppetrepo-shoal/manifests/client.pp

The shoal client runs in the WorkeNodes and sets up the squid proxy according to what the server tells

```

class shoal::client(
  $shoal_server_url = undef,
)
{
  if($shoal_server_url)
  {
    package { 'git':
      ensure => installed,
      before => Exec['git-shoal-client'],
    }

    exec { 'git-shoal-client':
      command => "git clone git://github.com/hep-gc/shoal.git",
      path => "/usr/bin",
      cwd => "/usr",
      require => Package['git'],
      before => Exec['install-shoal-client'],
      creates => "/usr/shoal"
    }

    exec { 'install-shoal-client':
      command => "python setup.py install",
      cwd => "/usr/shoal/shoal-client/",
      path => "/usr/bin",
      require => Exec['git-shoal-client'],
      creates => "/etc/shoal/shoal_client.conf",
    }
  }
}

```

```

file_line { '/etc/shoal/shoal_client.conf':
  path => '/etc/shoal/shoal_client.conf',
  line => "shoal_server_url = ${shoal_server_url}",
  require => Exec['install-shoal-client'],
  match => "shoal_server_url =.*$",
}

file { '/usr/bin/shoal-client':
  ensure => present,
  owner => 'root',
  group => 'root',
  mode => '0755',
  source => "puppet:///modules/shoal/shoal-client",
  require => Exec['install-shoal-client'],
}

exec { 'run-shoal-client':
  command => "shoal-client",
  path => "/usr/bin",
  require => File['/usr/bin/shoal-client'],
  refreshonly => true,
  subscribe => File["/usr/bin/shoal-client"],
}

cron::job{ 'shoal-client':
  minute => '0,30',
  hour => '*',
  date => '*',
  month => '*',
  weekday => '*',
  user => 'root',
  command => '/usr/bin/shoal-client >> /var/log/cron_shoal_client.log',
  environment => [ 'MAILTO=psandeep@hawk.iit.edu' ];
}

file { '/usr/bin/set-session-proxy.sh':
  ensure => present,
  owner => 'root',
  group => 'root',
  mode => '0755',
  source => "puppet:///modules/shoal/set-session-proxy.sh",
}
}

```

```
    else {
      warning('Shoal client is NOT INSTALLED. please pass a valid address to the parameter "shoal_server_url" and
run it again' )
      warning('eg: shoal_server_url => \'http://shoalserver.domain/nearest\' ')
    }
  }
}
#exec { 'run-set-session-proxy-script':
# command => "source set-session-proxy.sh",
# path => "/usr/bin",
# require => File['Add script to set session proxy'],
#}

-----
-----
-----
```

```
File: puppetrepo-shoal/manifests/frontier.pp
# == Class: frontier::squid

#

# Installation and configuration of a frontier squid

#

# === Parameters

#

# [*customize_file*]
# The customization config file to be used.
#

# [*customize_template*]
# The customization config template to be used.
#
```

```
# [*cache_dir*]
# The cache directory.
#

# [*install_resource*]
# The cache directory.
#

# [*resource_path*]
# The cache directory.
#

# === Examples

#

# class { frontier::squid:
#   customize_file => 'puppet:///modules/mymodule/customize.sh',
#   cache_dir      => '/var/squid/cache'
# }

#

# === Authors

#

# Alessandro De Salvo <Alessandro.DeSalvo@romal.infn.it>

#

# === Copyright

#

# Copyright 2014 Alessandro De Salvo

#

# Added comment
```

```

class shoal::frontier (
  $customize_file = undef,
  $customize_template = undef,

  $cache_dir = $shoal::frontier_params::frontier_cache_dir,
  $install_resource = false,
  $resource_path = $shoal::frontier_params::resource_agents_path
) inherits shoal::frontier_params {
  yumrepo {'cern-frontier':
    baseurl => 'http://frontier.cern.ch/dist/rpms/',
    enabled => 1,
    gpgcheck => 1,
    gpgkey  => 'http://frontier.cern.ch/dist/rpms/cernFrontierGpgPublicKey'
  }

  package {$shoal::frontier_params::frontier_packages:
    ensure => latest,
    require => Yumrepo['cern-frontier'],
    notify => Service[$shoal::frontier_params::frontier_service]
  }

  if ($cache_dir) {
    file { $cache_dir:
      ensure => directory,

```

```

      owner  => squid,
      group => squid,
      mode  => 0755,
      require => Package[$shoal::frontier_params::frontier_packages],
      notify => Service[$shoal::frontier_params::frontier_service]
    }
  }

  if ($customize_file) {
    file {$shoal::frontier_params::frontier_customize:
      ensure => file,
      owner  => squid,
      group => squid,
      mode  => 0555,
      source => $customize_file,
      require => Package[$shoal::frontier_params::frontier_packages],
      notify => Service[$shoal::frontier_params::frontier_service]
    }
  }

  if ($customize_template) {
    file {$shoal::frontier_params::frontier_customize:
      ensure => file,
      owner  => squid,

```

```

        group => squid,
        mode  => 0755,
        content => template($customize_template),
        require => Package[$shoal::frontier_params::frontier_packages],
        notify => Service[$shoal::frontier_params::frontier_service]
    }
}

if ($install_resource) {
    file { $resource_path:
        ensure => directory,
        owner  => "root",
        group  => "root",
        mode   => 0755,
    }

    file { "${resource_path}/FrontierSquid":
        ensure => file,
        owner  => "root",
        group  => "root",
        mode   => 0755,
        source => "puppet:///modules/frontier/FrontierSquid",
        require => File[$resource_path]
    }
}

```

```

}

service {$shoal::frontier_params::frontier_service:
    ensure => running,
    enable => true,
    hasrestart => true,
    require => Package[$shoal::frontier_params::frontier_packages]
}

file { "/var/spool/squid":
    ensure => "directory",
    #owner  => "root",

    #group => "wheel",
    mode   => 766,
}

file { 'Replacing default squid config with new config content':
    ensure => present,
    owner  => 'root',
    group  => 'root',
    mode   => '0755',
    path   => '/etc/squid/squid.conf',
    content => inline_template("acl NET_LOCAL src 10.0.0.0/8 172.16.0.0/12 192.168.0.0/16

```

```
acl HOST_MONITOR src 131.225.152.0/23
acl snmppublic snmp_community HOST_MONITOR
acl all src all
acl manager proto cache_object
acl localhost src 127.0.0.1/32
acl to_localhost dst 127.0.0.0/8 0.0.0.0/32
acl localnet src 10.0.0.0/8 # RFC1918 possible internal network
acl localnet src 172.16.0.0/12 # RFC1918 possible internal network
acl localnet src 192.168.0.0/16 # RFC1918 possible internal network
acl SSL_ports port 443
acl Safe_ports port 80 # http
acl Safe_ports port 21 # ftp
acl Safe_ports port 443 # https
acl Safe_ports port 70 # gopher
acl Safe_ports port 210 # wais
acl Safe_ports port 1025-65535 # unregistered ports
acl Safe_ports port 280 # http-mgmt
acl Safe_ports port 488 # gss-http
acl Safe_ports port 591 # filemaker
acl Safe_ports port 777 # multiling http
acl CONNECT method CONNECT
http_access allow manager localhost
http_access deny manager
http_access deny !Safe_ports
```

```
http_access deny CONNECT !SSL_ports
http_access allow NET_LOCAL
acl our_networks src 131.225.0.0/16 127.0.0.1
http_access allow our_networks
http_access allow localhost

http_access deny all
acl PURGE method PURGE
http_access allow PURGE localhost
http_access deny PURGE

reply_body_max_size 1000000000 allow all
icp_access allow localnet
icp_access deny all
http_port 3128
hierarchy_stoplist cgi-bin
cache_mem 1024 MB
maximum_object_size_in_memory 256 KB
cache_dir ufs /var/spool/squid 290000 16 256
maximum_object_size 10 GB

logformat awstats %>a %ui %un [%{%d/%b/%Y:%H:%M:%S +0000}t1] \"%rm %ru HTTP/%rv\" %Hs %<st %Ss:%Sh %tr \"%{X-
Frontier-Id}>h\" \"%{Referer}>h\" \"%{User-Agent}>h\"

access_log /var/log/squid/access.log awstats
logfile_daemon /usr/libexec/squid/logfile-daemon
cache_log /var/log/squid/cache.log
cache_store_log none
```

```

mime_table /etc/squid/mime.conf
pid_filename /var/run/squid/squid.pid
strip_query_terms off
unlinkd_program /usr/libexec/squid/unlinkd
refresh_pattern ^ftp:          1440  20%  10080
refresh_pattern ^gopher:      1440  0%   1440
refresh_pattern -i /cgi-bin/   0      0%   0
refresh_pattern \\.crl$        60    25%  1440
refresh_pattern \\.der$        60    25%  1440
refresh_pattern \\.pem$        60    25%  1440
refresh_pattern \\.r0$         60    25%  1440
refresh_pattern \\.pacman$     60    10%  1440
refresh_pattern .              60    20%  4320
negative_ttl 1 minute
acl shoutcast rep_header X-HTTP09-First-Line ^ICY.[0-9]
upgrade_http0.9 deny shoutcast
acl apache rep_header Server ^Apache
broken_vary_encoding allow apache
collapsed_forwarding on
connect_timeout 30 seconds
read_timeout 1 minute

request_timeout 1 minute
client_lifetime 1 hour
cache_mgr fermigrid-help@fnal.gov

```

```

cache_effective_user squid
cache_effective_group squid
umask 022

snmp_access allow snmppublic HOST_MONITOR
snmp_access deny all
icp_port 0
icon_directory /usr/share/squid/icons
error_directory /usr/share/squid/errors/English
ignore_ims_on_miss on
coredump_dir /var/spool/squid
"),
notify => Service[$shoal::frontier_params::frontier_service],
}

}

```

```

-----
-----
-----

```

```

File: puppetrepo-shoal/manifests/frontier_params.pp
#contains parameters for shoal::frontier class
class shoal::frontier_params {
  case $::osfamily {
    'RedHat': {
      $frontier_release_provider = 'rpm'
      $frontier_release_package = 'frontier-release-1.0-1.noarch.rpm'
      $frontier_release_package_url = "http://frontier.cern.ch/dist/rpms/RPMS/noarch/${frontier_release_package}"
      $frontier_packages = ['frontier-squid']
      $frontier_service = 'frontier-squid'
    }
  }
}

```

```

    $frontier_customize = '/etc/squid/customize.sh'
    $frontier_cache_dir = '/var/cache/squid'
    $resource_agents_path = '/usr/lib/ocf/resource.d/lcg'
  }
  default: {
  }
}
}
-----
-----
-----

```

```

File: puppetrepo-shoal/manifests/host_server.pp
# Installs wsgi module for apache and hosts shoal server
class shoal::host_server{

```

```

  $rabbitmq_server_url = "localhost"

  file { ["/var/run/wsgi":
    ensure => "directory",
  ]

  package { [httpd:
    ensure => installed,
  ]

  exec { [install-wsgi:
    command => "yum install mod_wsgi -y",
    path => "/usr/bin",
    creates => "/usr/lib64/httpd/modules/mod_wsgi.so",
  ]

  file_line { [Add wsgi module to /etc/httpd/conf/httpd.conf':
    path => '/etc/httpd/conf/httpd.conf',
    line => 'LoadModule wsgi_module modules/mod_wsgi.so',
    require => Exec['install-wsgi'],
    notify => Service["httpd"],
  ]

  file_line { ["/etc/httpd/conf/httpd.conf':
    path => '/etc/httpd/conf/httpd.conf',
    line => 'WSGISocketPrefix /var/run/wsgi',
    require => Exec['install-wsgi'],
    notify => Service["httpd"],
  ]
}

```

```

exec { [Host shoal on apache':
  command => "mv /var/shoal/ /var/www/",
  path => "/bin",
  creates => "/var/www/shoal",
]

file_line { [changing shoal dir in /etc/shoal/shoal_server.conf':
  path => '/etc/shoal/shoal_server.conf',
  line => 'shoal_dir = /var/www/shoal/',
  require => Exec['install-wsgi'],
  match => "shoal_dir =.*$",
]

file_line { [changing amqp server url in /etc/shoal/shoal_server.conf':
  path => '/etc/shoal/shoal_server.conf',
  line => "amqp_server_url = ${rabbitmq_server_url}",
  require => Exec['install-wsgi'],
  match => "amqp_server_url =.*$",
  notify => Service["httpd"],
]

file { ["/etc/httpd/conf.d/shoal.conf':
  ensure => present,
  source => "puppet:///modules/shoal/shoal.conf",
]

file { ["/var/log/shoal_server.log":
  mode => '777',
  #recurse => true,
]

file { ["/var/www/shoal/scripts/shoal_wsgi.py":
  mode => '755',
  #recurse => true,
]

service { [httpd':
  ensure => running,
  path => "/usr/sbin/",
  require => Exec['install-wsgi'],
]
}
}

```

```
-----  
-----  
-----
```

```
File: puppetrepo-shoal/manifests/replace_squid_conf.pp  
class shoal::replace_squid_conf{
```

```
  file { ["/var/spool/squid":  
    ensure => "directory",  
    mode   => 766,  
  ] }  
  
  file { ["/etc/squid/squid.conf":  
    ensure => present,  
    owner  => 'root',  
    group  => 'root',  
    mode   => '0755',  
    source => "puppet:///modules/shoal/squid.conf",  
    #notify => Service['frontier-squid'],  
    require => File['/var/spool/squid'],  
  ] }  
  
  exec { ['restart frontier-squid':  
    command => "service frontier-squid restart",  
    require => File['/etc/squid/squid.conf'],  
    path    => '/sbin'  
  ] }  
}
```

```
-----  
-----  
-----
```

```
File: puppetrepo-shoal/manifests/repository.pp  
# shoal repository form where we install the shoal agent package
```

```
class shoal::repository {  
  package { ["yum-conf-epel":  
    ensure => installed,  
  ] }  
}
```

```
# exec { ['yum-update':  
  # command => "yum update -y",  
  # path => "/usr/bin",  
  # before => Exec["curl-shoal-repo"],  
  # require => Package["yum-conf-epel"],  
  # creates => "/usr/bin/shoal-agent",  
#}]  
  
exec { ['curl-shoal-repo':  
  command => "curl http://shoal.heprc.uvic.ca/repo/shoal.repo -o /etc/yum.repos.d/shoal.repo",  
  path => "/usr/bin",  
  before => Exec["rpm-import"],  
  # require => Exec["yum-update"],  
  require => Package["yum-conf-epel"],  
  creates => "/etc/yum.repos.d/shoal.repo",  
  ] }  
  
exec { ['rpm-import':  
  command => "rpm --import http://hepnetcanada.ca/pubkeys/igable.asc",  
  path => "/bin",  
  require => Exec["curl-shoal-repo"],  
  creates => "/usr/bin/shoal-agent",  
  ] }  
}
```

```
-----  
-----  
-----
```

```
File: puppetrepo-shoal/manifests/server.pp
```

```
# Installs the shoal server  
class shoal::server{  
  
  exec { ['git-shoal-server':  
    command => "git clone https://github.com/SandeepPalur/shoal.git",  
    cwd => "/usr",  
    path => "/usr/bin",  
    before => Exec["install-shoal-server"],  
    creates => "/usr/shoal",  
  ] }  
  
  exec { ['install-shoal-server':  
    command => "python setup.py install",  
    cwd => "/usr/shoal/shoal-server",  
    path => "/usr/bin",  
  ] }  
}
```

```

require => Exec["git-shoal-server"],
creates => "/etc/shoal/shoal_server.conf",
}

exec { "wget-pip":
  command => "wget https://bootstrap.pypa.io/get-pip.py --no-check-certificate",
  before => Exec["install-pip"],
  require => Exec["install-shoal-server"],
  path => "/usr/bin",
  cwd => "/usr",
  creates => "/usr/bin/pip",
}

exec { "install-pip":
  path => "/usr/bin",
  cwd => "/usr",
  command => "python get-pip.py",
  require => Exec["wget-pip"],
  before => Exec["install-webpy"],
  creates => "/usr/bin/pip",
}

exec { "install-webpy":
  path => "/usr/bin",
  command => "pip install web.py",
  require => Exec["install-pip"],
  before => Exec["install-pygeoip"],
  creates => "/usr/lib/python2.6/site-packages/web.py-0.37-py2.6.egg-info",
}

exec { "install-pygeoip":
  path => "/usr/bin",
  command => "pip install pygeoip",
  require => Exec["install-webpy"],
  before => Exec["install-pika"],
  creates => "/usr/lib/python2.6/site-packages/pygeoip",
}

exec { "install-pika":
  path => "/usr/bin",
  command => "pip install pika",
  creates => "/usr/lib/python2.6/site-packages/pika",
}

file { '/usr/bin/shoal-server.py':
  require => Exec["install-pika"],
  ensure => present,
}

```

```

owner => 'root',
source => "puppet:///modules/shoal/shoal-server.py",
}

exec { "Run shoal server script":
  path => "/usr/bin",
  command => "python /usr/bin/shoal-server.py",
  require => File['/usr/bin/shoal-server.py'],
  refreshonly => true,
  subscribe => File["/usr/bin/shoal-server.py"],
}
}

```

```

-----
-----
-----

```

File: puppetrepo-shoal/manifests/server_dependencies.pp

Installs shoal server dependency packages

```

class shoal::server_dependencies{
  Package { ensure => "installed" }

```

```

  package { "wget": }
  package { "gcc": }
  package { "make": }
  package { "ncurses":}
  package { "ncurses-devel":}
  package { "openssl-devel":}
  package { "libxslt":}
  package { "zip":}
  package { "unzip":}
  package { "nc":}
  package { "git":}

```

```

}

```

```

-----
-----
-----

```

X.509 Authentication/Authorization in FermiCloud

Hyunwoo Kim, Steven C. Timm

Scientific Computing Division
Fermi National Accelerator Laboratory
Batavia, U.S.A
hyunwoo@fnal.gov, timm@fnal.gov

Abstract—We present a summary of how X.509 authentication and authorization are used with OpenNebula in FermiCloud. We also describe a history of why the X.509 authentication was needed in FermiCloud, and review X.509 authorization options, both internal and external to OpenNebula. We show how these options can be and have been used to successfully run scientific workflows on federated clouds, which include OpenNebula on FermiCloud and Amazon Web Services as well as other community clouds. We also outline federation options being used by other commercial and open-source clouds and cloud research projects.

Keywords—Cloud; X.509; Authentication; Authorization; FermiCloud

I. INTRODUCTION

A. X.509 Certificates For Identity Authentication

FermiCloud relies on X.509 certificates [1] to achieve identity authentication. X.509 provides us with a way to verify the user’s identity is in fact who he or she claims to be. The identity of a user can be verified by a chain of signing authorities.

Three basic concepts, identification, authentication and authorization, must be considered with equal importance in order to make sure the right users are doing the right things in our system. Identification is how users assert who they are to our system. It can be your user name if the system is relying on username and password authentication. Authentication is how users prove their identity assertion. In other words, it is a presentation of secret that only the owner must know and the system can then verify the identity with. If it is password, the server should keep the secret and use it to verify the user when the user types in the password when the user wants to sign in. Furthermore all communications can be encrypted by using SSL. Since it can be presumed that only a user knows his or her secret, the user’s claim can be validated by authentication.

Identification and authentication in X.509 scheme are closely related. Identity in X.509 scheme can be the subject name on a X.509 digital certificate. In analogy with username and password scheme, the secret could be the private key associated with a digital certificate. The difference between username and password scheme and X.509 scheme is how

users present the secret. With the simple username and password scheme, as described above, the server must keep the secret and the user has to type in the password in a browser. In the X.509 scheme, users present their Distinguished Name when their account is created. Each time they authenticate, they present the full certificate and private key credential, using the passphrase to decrypt the private key. The login process will sign a simple text (username in case of OpenNebula) with the private key and transmit this private-key-signed text to the server. The server verifies this text using X.509 certificate that came along with the document and then extract the DN from the X.509 certificate. Then the server compares this DN against the list of DNs stored in the server database. The process described above is one of basic cryptographic assurance that is provided by public key cryptography. In principle, with RSA[2], we can use private key to encrypt the entire plaintext. This provides both identity authentication and message authentication because only the person that holds the private key can encrypt a document and because that person alone could guarantee the authenticity of a document by encrypting the entire document. What are transmitted are the plaintext and the encryption result. While PKC uses public and private keys to achieve cryptographic assurances, we still need to prove the ownership of the public key, which should be distributed in a manageable way. In other words, public key must be distributed as digital certificates. Public key infrastructure (PKI) is one way to achieve this and X.509 is an ITU-T standard for PKI [16].

B. Authorization with X.509 Certificates

Authentication on its own is not sufficient to allow users to use resources. We also need to know what actions, if any, a user is authorized to undertake. The existing OpenNebula[13] authorization scheme is based on Access Control Lists for resources. All resources in OpenNebula, including virtual networks, machine images, and templates, have user, group, and world permissions similar to Unix permissions. There are also per-user and per-group quotas of how many machines can be launched.

FermiCloud is developing a new X.509 based authorization module for OpenNebula that also can determine the correct user and group given a grid identity. It uses information from Virtual Organization Membership Service (VOMS) [3] and Grid User Management System (GUMS) [4]. With a user’s subject line in X.509 certificate, we first contact a local

VOMS in FermiCloud to acquire a list of Fully Qualified Attributes Name (FQAN) assigned to that user. We present this list to the user prompting for the user's selection. Then we construct an XACML (eXtensible Access Control Markup Language) [5] request using an XACML Java client library called privilege package, developed at Fermilab, and send this request to a local GUMS server. We expect two answers from GUMS: a pair of FermiCloud-specific user ID and group ID mapped by GUMS and secondly whether the user is authorized to undertake the Role and Capability in the FQAN. We use this information to finally authorize the user and also record which VO a virtual machine is running under the name of. Technical details are found in a later section.

C. History of why X.509 Authentication was needed

X.509 based authentication and authorization became widespread in the scientific community with the widespread adoption of grid computing. The Grid Security Infrastructure (GSI) [6] includes a set of certificate authorities that are recognized by the International Grid Trust Federation and accepted worldwide by grid computing sites. All Fermilab-hosted experiments participate in grid computing via the FermiGrid [7] campus grid and the Open Science Grid [8] of which FermiGrid is a major part. When the FermiCloud [17] project was initiated in 2009 we identified a requirement to have stronger security than the default username/password or access key / secret key mechanism could provide. We had several years of successful operation of X.509 authentication and authorization on the grid and an interoperability protocol for authorization based on XACML authorization[18] which we hoped to reuse. By using X.509 authentication and authorization we could know exactly who is running virtual machines on our cloud. An X.509 authorization scheme also allows us to transparently let a user with a single X.509 Distinguished Name be part of more than one group or organization, and transparently change between them. Because Fermilab operates its own Short Lived Credential Service (SLCS) certificate authority we can further auto-generate short-lived certificates on behalf of our own users and revoke them at any point. In practice much of the X.509 certificate manipulation is transparent to the user and invoked in the login script as they log into the interface node.

II. OPENNEBULA IMPLEMENTATION OF X.509

A. Token-based Authentication in OpenNebula before using X.509

Besides X.509 authentication, OpenNebula also provides another token-based authentication method that uses ssh keys. When a new user is signed up, the new user has to use ssh-keygen command to generate a pair of ssh keys and register the public ssh key in OpenNebula system. For sign-in, the regular login command with the option of using ssh keys will create a Single Sign On (SSO) token and this token will be used for subsequent uses of user commands.

B. X.509 Authentication in Command Line Interface

FermiCloud developed X.509 module for OpenNebula. The basic idea is token-based SSO authentication using user's X.509 certificate and associated private key. A user initially executes a login command with X.509 certificate and private key. Internally this command uses the user's private key to sign a text document, base64-encodes it and produce eventually a token as a secure file in the user's private area. Subsequent commands issued by the user will present this token to the OpenNebula server. The server first retrieves the user's X.509 certificate and uses it to verify (authenticate) the identity of the user. This implementation was incorporated into the main OpenNebula 3.0 code base in 2012 and has continued to be available since that time in all subsequent versions.

FermiCloud implementation of X.509 authentication follows a common approach to achieve X.509-based authentication. We note that this is also the case for OpenStack PKI-signed token-based authentication that we will review in a later section.

1. An X.509 authentication scheme must provide a tool a user can use to sign in. This command will require both X.509 certificate and associated private key from a user. The command then will generate a token and sign it with the private key.
2. The server side, first of all, should be able to acquire the user's X.509 certificate. In the above description, the client tool requires the user's X.509 certificate too besides the private key. In case of OpenNebula, the client tool appends the X.509 certificate to the signed token. In case of OpenStack, each service endpoint (such as Nova) downloads a X.509 certificate from a pre-defined location.
3. The next question is, how to transmit this signed token to the server side. OpenNebula CLI generates a token in a form of a file. The user must set an environment variable to the location of this file so that the OpenNebula command line tools can transmit the appropriate authorization data to the "oned" daemon. As OpenStack only supports RESTful services (via direct use of URL in a tool such as cURL, OpenStack SDK or OpenStack CLI), OpenStack adopts a different method for a transmission of the signed token and use X-Auth-Token HTTP header for this purpose.
4. The server side can conduct a simple verify operation against a private-key-signed token with a X.509 certificate of OpenNebula user or the Keystone signing certificate in the case of OpenStack.

After verification, OpenNebula extracts the DN of the user from X.509 certificate and uses it to identify the user against the list of Distinguished Names stored in the user pool table of the database. OpenStack simply uses the username for identification. As mentioned above, this scheme is used by X.509 authentication of OpenNebula CLI and OpenStack

Keystone PKI-based token authentication. Note that the scheme described above cannot be done with the normal use of web browser because the browser doesn't allow the same flexibility as CLI utilities of using user certificate and private key for cryptographical purposes.

C. X.509 Authentication in Sunstone OpenNebula Web Interface

OpenNebula Sunstone is basically a Sinatra-based web application with Thin [9] in front of it as a Ruby web browser. Sunstone is usually placed behind Apache HTTPD with SSL module. There are two types of clients that will access Sunstone. Users can upload their certificate and private key pair to web browsers to access REST services provided by Sunstone. Web browsers allow only PKCS12 [10] format and require the encryption password when the certificate and key are imported into the browser. The certificate is then protected by the password of the certificate store of the browser, which must be given when the certificate is used at a given site for the first time. The Sunstone service and other such services can also be accessed with CLI tools such as the cURL command. The cURL command for example also requires the users to provide both private key and certificate at the same time and prompts for the password (if any) associated with the private key. The cURL command also goes through SSL handshake with the SSL module attached to Apache server that is in front of Sunstone. In both cases, the SSL module for Apache on the server side is configured to use the option to verify client for a client-authentication. After the SSL handshake finishes successfully, the SSL module sets an environment variable called `HTTP_SSL_CLIENT_CERT` equal to the base-64 encoded PEM string of the full client certificate that was transmitted from the browser or the CLI tool. Sunstone code uses the X.509 Certificate utility in Ruby OpenSSL library in order to extract the user's Distinguished Name from the PEM string and uses the extracted DN to compare against a list of DN's in the user pool in order to acquire the username that matches the DN. This look-up process is common to both username-password and X.509 certificate schemes, but using X.509 certificate, we can identify a user who is verified by Certificate Authority (CA). We note that this authentication scheme in OpenNebula Sunstone is functionally similar to the external authentication option that OpenStack supports besides the regular methods using username-password or token.

D. X.509 authentication in EC2 emulation

OpenNebula EC2 emulation is also a Sinatra application with Thin web server behind Apache HTTPD with SSL module. Current implementation of OpenNebula EC2 emulation client code is using AWS Ruby SDK to generate REST/Query requests to OpenNebula EC2 emulation server using AWS signature algorithm to sign the request with Access Key ID and Secret Access Key. FermiCloud modifies OpenNebula EC2 emulation codes in order to enable the use of X.509 certificates. We replace AWS EC2 library with Ruby cURL library to generate REST requests. This way we can

exploit the options for X.509 certificate and private key available in the Ruby cURL library and achieve the same client authentication as what happens between the cURL command and the OpenNebula Sunstone. In current implementation of OpenNebula, the EC2 emulation server shares the same X.509 authentication code with the Sunstone. Apache with SSL module processes and forwards secure information to EC2 emulation Sinatra server. EC2 emulation server uses this information to reconstruct the user's X.509 certificate with X.509 Certificate utility in Ruby OpenSSL library and extracts the user's DN to compare against the list of DN's in user pool. It was necessary to modify the code slightly to allow the server to accept X.509 proxy certificates as well as full certificates.

E. X.509 Authorization developed by FermiCloud

The existing OpenNebula authorization is based on Access Control List. After a user is authenticated, relevant access control information or permissions related to the user are examined to determine the authorization results. We started by modifying CLI to use Local Credential Mapping Service (LCMAPS) [11] developed by NIKHEF. When a user signs in with OpenNebula CLI, both proxy certificate PEM string and personal certificate PEM string are available in OpenNebula server side where we call LCMAPS C function via Ruby C binding in order contact FermiCloud GUMS server. Then, we extended this solution to Sunstone. The fact that we need both proxy and personal certificates to call LCMAPS function means that we need to plant both proxy certificate and personal certificate into the browser, which is technically possible. The problem was with the transmission of certificates from web browsers to the server. The proxy certificate is available in the server side as a PEM string, but the personal certificate that was used to generate the proxy certificate is not transmitted. Technically this is believed to originate from the fact that web browsers do not recognize the personal certificate as a proper CA that signed the proxy certificate. Using the cURL command, the server sets these variables properly. For this reason, we do not consider using LCMAPS to contact GUMS because it does not work with the web browsers. We decided to use an XACML client library to build an XACML request. This client library is called privilege package and was developed in Java programming language by Fermilab. In order to invoke this Java privilege package from a Ruby code, we use Ruby Java Bridge (RJB). A request to GUMS requires user's DN, VO and FQAN. Our solution to acquire user's VO and FQAN is contacting VOMS-Admin server in FermiCloud. We then present this list to the user asking for the user's selection. Then with the selected VO and FQAN, Sunstone constructs a request to GUMS using privilege package. The query to VOMS-Admin for all the possible groups and roles has the benefit that we can use a grid proxy or certificate rather than one with extended VOMS attributes and these are easier to manage in the browser. This successful implementation in Sunstone could also be applied to command line and EC2 emulation interfaces of OpenNebula.

We also tried to use GridSite package as a module for Apache HTTPD. GridSite can be used to retrieve the VO and FQAN from a VOMS-signed certificate that is transmitted from a web browser. In our testing, GridSite package worked properly only when cURL command was used and it failed when web browser was used.

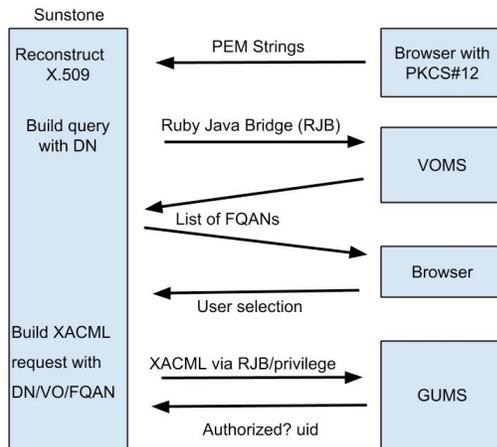


Figure 1. FermiCloud new Authorization Scheme

F. Scientific Workflows on Federated Clouds

Fermilab uses the GlideinWMS [19] workflow management system and the Jobsub client/server submission system to federate heterogeneous resources including grids and clouds. Users use their automatically generated SLCS-based X.509 certificates to authenticate to the server and submit jobs. The GlideinWMS system uses its own certificates and AWS access keys to obtain grid job slots and virtual machines on behalf of the federation of users, and then these resources are matched to user jobs based on the requirements of these jobs. We use this infrastructure to support Fermilab physics experiments. We have successfully run the Cosmic Ray simulation of the NOvA neutrino detectors on Amazon AWS services, FermiCloud, and a collection of sites on the Open Science Grid. We continue to increase the amount of virtual machines we can run simultaneously.

III. X.509 IN AMAZON WEB SERVICES EC2

AWS basically supports two APIs: REST and SOAP. If we want to use SOAP API, there are two possible ways. We can use AWS EC2 SDK to generate SOAP requests or we can use AWS EC2 CLI to generate SOAP requests. In both cases, we can use X.509 certificate and private key for authentication. AWS will discontinue the support of SOAP APIs at the end of 2014. If we want to use REST API, there are three ways. We can access REST API directly by constructing a query with a command line tool. Or we can use AWS EC2 SDK or CLI to generate Query requests. The Amazon EC2 REST API provides HTTP or HTTPS requests that use HTTP GET or POST methods and a Query parameter named Action. These AWS REST requests are signed using Access Keys that

consist of Access Key ID and Secret Access Key. Note that the AWS Command Line Interface or the AWS SDKs automatically sign requests for us. But if we construct a Query request directly, we must sign the requests manually, using the procedure described in AWS signing algorithm.

Also we note that there are several AWS credentials types for different purposes.

1. Email address and password: when we sign up for AWS, we provide an email address and password that is associated with our AWS account. We use these credentials to sign in to secure AWS web pages.
2. Access Keys: we use access keys to sign requests to AWS whether we're accessing the REST API via the AWS SDK, CLI or direct access.
3. X.509 Certificates: we are recommended to use X.509 certificates only to sign SOAP-based requests. In all other cases, we are recommended to use access keys.
4. Key Pairs: for Amazon EC2, we use key pairs to access Amazon EC2 instances, such as when we use SSH to log in to a Linux instance.

IV. EGI FEDERATED CLOUD

European Grid Infrastructure (EGI) [12] recently launched a project called Federated Cloud (FC). General structure of EGI FC is rOCCI server in front of cloud resources using OpenNebula or OpenStack. The rOCCI server consists of a Rails web application and Apache HTTPD with SSL module and Passenger [12] module. We can issue an occi client command with options for X.509 authentication and the use of VOMS. We are interested in how the rOCCI OpenNebula backend conducts X.509 authentication. The backend invokes OpenNebula UserPool method that is available in a local OpenNebula distribution that resides in rOCCI server deployment. This UserPool contacts the main OpenNebula instance via the regular xml-rpc channel to receive a list of users that are registered. Then local methods in the rOCCI OpenNebula backend such as do_auth will see if a valid username is returned from a query using the regular X.509 DN when auth x509 option is used with occi command or using the extended DN with FQAN when auth x509 and VOMS options are used together in occi command. When the OpenNebula instance that is running behind rOCCI server is needed to support rOCCI's VOMS authentication type, each user should be created with a DN extended with FQAN. This can be done manually or by using Perun script. In details, the X.509 based authentication that is conducted by rOCCI OpenNebula backend relies on the list of users returned from the actual OpenNebula instance running behind rOCCI server and is equivalent to what the ordinary OpenNebula does for X.509 authentication. We are also interested in understanding how Perun is used in association with rOCCI's VOMS authentication. As mentioned in the previous paragraph, a user can be created with a DN extended with FQAN by using Perun script. This script first contacts a Perun server to retrieve an up-to-date list of users and associated virtual

organizations and accordingly update OpenNebula's user pool with the list and this update process will create a user, if necessary, with an extended DN with FQAN.

We note that the authentication and authorization model in EGI Federated Cloud is using information from VOMS only for authentication purpose and we will still need our new development for X.509 authorization even if FermiCloud OpenNebula is placed behind rOCCI server.

V. OPENSTACK AND FEDERATION

First of all, we note that both AWS and OpenStack support only RESTful Web Services. There are two ways to access RESTful Web Services. In a direct way we can use cURL or other external RESTful clients. Here, we need to construct the request for ourselves and interpret the raw response of XML or JSON. In an indirect way we can use SDK or CLI. They both need endpoint URL. They will both access the URL just like the direct way, but the difference is that the indirect way via SDK or CLI will process the raw response of XML or JSON and returned a formatted response. Any statements with regard to OpenStack refer to OpenStack version Icehouse, the current version at the writing of this paper.

A. Regular Authentication in Keystone

OpenStack consists of several Services. Keystone is the one that handles the Identity Service. Another example is Nova that handles the Compute Service. Keystone supports four authentication plugins, which are specified in the [auth] section of the configuration file: password, token, external and federation. Suppose a user has obtained a credential, i.e. username and password. This user might use username and password to issue a Identity API request of a token to the Keystone. Or if this user already has a token that is still valid, the user could use the token to issue a new Identity API request to Keystone. In either case, after a token is acquired, the user can use this token to issue subsequent API requests such as Compute API requests to the NOVA service. This is Single Sign On. The user can also still use username and password, to issue subsequent API requests.. This use of username and password as authentication method in Identity service and other service such as Compute is similar to the use of Access Keys in Amazon Web Services. When a token is used, Keystone uses PKI to sign and verify the tokens. Further, Keystone uses SSL to encrypt the communications. Use of token is similar to X.509 authentication in OpenNebula.

B. External Authentication

Web servers like Apache HTTPD support many methods of authentication. When Keystone is executed in a web server like Apache HTTPD, Keystone can profit from this feature and let the authentication be done in the web server. When a web server is in charge of authentication, it is normally possible to set the REMOTE_USER environment variable so that it can be used in the underlying application (Keystone). Keystone can be configured to use that environment variable

if set. This user must exist in advance in the identity backend to get a token from the controller. This way, we can use X.509 authentication or Kerberos, for example, instead of using the username and password combination. To use this method, Keystone should be running on HTTPD and Apache should be configured to enable SSL. Note that while it is possible to use an external authentication method besides password and tokens, it is also possible to use an external method for identity provider besides the SQL database backend. Popular choice is LDAP directory service.

C. OpenStack and Federation

Keystone can be placed behind Apache HTTPD for two reasons: to use external authentication method, besides the ordinary two methods, which are password and token. Second reason is to use federation. New feature in the latest version of OpenStack is Identity Federation using SAML [14]. External users authenticate with Identity Provider (IdP). The IdP communicates the authentication result to Keystone using SAML assertions. Keystone maps the SAML assertions to Keystone user groups and assignments created in Keystone. In order to make this possible, Keystone should be configured accordingly. First, Keystone should be driven by Apache httpd and Shibboleth should be installed. Secondly, Shibboleth itself should be configured. Third, there is an extension called OS-FEDERATION. This should be enabled. Lastly OS-FEDERATION extension should be configured.

D. Authorization in OpenStack Keystone

Role is a personality that a user assumes when performing a specific set of operations. A role includes a set of rights and privileges. A user assuming that role inherits those rights and privileges. In OpenStack Identity, a token that is issued to a user includes the list of roles that user can assume. Services that are being called by that user determine how they interpret the set of roles a user has and to which operations or resources each role grants access. It is up to individual services such as the Compute service and Image service to assign meaning to these roles. As far as the Identity service is concerned, a role is an arbitrary name assigned by the user.

VI. NIMBUS AUTHENTICATION

The Nimbus project [15] implemented X.509 authentication and authorization using a full Grid Security Infrastructure system. This included a WSRF (Web Services Resource Framework) application container based on the Globus toolkit, which was capable of authentication and authorizations using certificate/key pairs or proxies. There was also a gridftp-based image transfer service used by the cloud client. The project also later added emulations of the SOAP and REST API's of Amazon EC2 as well as the Cumulus storage element, which emulates the Amazon S3 API.

VII. SUMMARY

We learned from this study that X.509 authentication in SOAP based API is decreasing in popularity and RESTful API is the current trend. We reviewed how FermiCloud developed X.509 authentication for OpenNebula command line interface. The idea is similar to how OpenStack is using PKI-based token for single-sign-on type authentication in REST requests. This approach is one way to support X.509 based authentication in RESTful services whereas using username and password in RESTful services is dominantly popular as in AWS. We have also shown a proof of principle of a unified procedure based on callouts to VOMS-Admin and GUMS for X.509-based authorization in OpenNebula. We also reviewed how OpenNebula Sunstone is using X.509 authentication via Apache HTTPD with SSL module. This is also similar to how OpenStack is using Apache HTTPD with SSL module as an option for external authentication. We also reviewed how EGI Federated Cloud is using rOCCI to federate cloud facilities using OpenNebula and OpenStack and how OpenStack is using SAML based external identity provider to federate users with external identities. Our plan is to keep looking for the best way to achieve authentication and authorization on top of restful cloud services as many new concepts and technologies are being developed and made available publicly.

ACKNOWLEDGMENT

We thank the developers of OpenNebula for their continued cooperation in adding authentication and authorization features that we have requested. We also acknowledge the significant contribution of former Fermilab employee Ted Hesselroth who was a member of the project through the fall of 2011 and was largely responsible for the X.509 authentication code that was contributed to OpenNebula. This work is supported by the US Department of Energy under contract number DE-AC02-07CH11359 and by KISTI under a joint Cooperative Research and Development Agreement. CRADA-FRA 2014-0002/KISTI-C14014.

REFERENCES

- [1] R.Hously et al, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile" <https://www.ietf.org/rfc/rfc2459>
- [2] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signature and Public-key Cryptosystems", *Communications of the ACM* 21 120-126 1978.
- [3] R. Alfieri et al. 2004. VOMS, an authorization system for virtual organizations *Proceedings of European across Grids conference No1, Santiago De Compostela, Spain 2970* 33-40
- [4] M. Lorch, D. Kafura, I. Fisk, K. Keahey, G. Carcassi, T. Freeman, T. Peremutov, A. S. Rana. 2005. Authorization and account management in the Open Science Grid *The 6th IEEE/ACM International Workshop on Grid Computing, 2005*
- [5] <http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cos01-en.html>
- [6] <http://toolkit.globus.org/toolkit/security/>
- [7] <http://fermigrid.fnal.gov>
- [8] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Wurthwein, I. Foster, R. Gardner, M. Wilde, A. Blatecky, J. McGee, and R. Quick 2007. The Open Science Grid *Journal of Physics: Conference Series*, 78 15
- [9] <http://code.macourmoyer.com/thin/>
- [10] <https://tools.ietf.org/html/rfc7292>
- [11] <https://wiki.nikhef.nl/grid/LCMAPS>
- [12] <http://www.egi.eu>
- [13] R. Moreno-Vozmediano, R. S. Monero, I. M. Llorente, IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures, *IEEE Computer*, vol. 45, pp. 65-72, Dec. 2012
- [14] <https://www.oasis-open.org/committees/download.php/13525/sstc-saml-exec-overview-2.0-cd-01-2col.pdf>
- [15] K. Keahey, I. Foster, T. Freeman, X. Zhang, D. Galron, Virtual Workspaces In The Grid, *Europar 2005*, Lisbon, Portugal, Sep. 2005.
- [16] <http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=X.509>
- [17] S. Timm, K. Chadwick, G. Garzoglio, S. Y. Noh, Grids, virtualization, and Clouds at Fermilab, in *Proceedings of the 20th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2013)*, *Journal of Physics: Conference Series* 513 (2014). D. L. Groep and D. Bonacorsi, eds. IOP Publishing.
- [18] G. Garzoglio, J. Bester, K. Chadwick, D. Dykstra, D. Groep, J. Gu, T. Hesselroth et al. "Adoption of a SAML-XACML Profile for Authorization Interoperability across Grid Middleware in OSG and EGEE." In *Journal of Physics: Conference Series*, vol. 331, no. 6, p. 062011. IOP Publishing, 2011.
- [19] P. Mhashilkar, A. Tiaradani, B. Holzman, K. Larson, I. Sfiligoi, and M. Rynge, Cloud Bursting With GlideinWMS: Means to satisfy ever increasing needs for Scientific Workflows. In *Journal of Physics: Conference Series* 513 (2014). D. L. Groep and D. Bonacorsi, eds., IOP Publishing.

Puppet Procedure to run Fermi2AWS Export

1. For a Paravirtual image on AWS: (Copy these for other images and rename them)

```
- vi /etc/puppet/modules/awsexport/manifests/gcso_sl6_pv.pp
- change parameters to your credentials.
- *Note: Obtain HVM worker instance id from aws console (caws_worker_instance_id => 'i-7788c97c')

- #contains parameters for the gcso_sl6 PV image conversion
- awsexport::awsexport_params {'gcso_sl6_pv':
-   cvm_file_location => '/opt/gcso/awsexport',
-   cvm_image_location => 'oneadmin@fcl008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5',
-   ckernel_ver => '2.6.32-431.23.3.el6.x86_64',
-   cvm_number => '103',
-   cvm_name => 'gcso_sl6',
-   cvm_owner => 'your Fermi username',
-   caws_image_name => 'GCSO_SL6_PV',
-   caws_image_owner => 'gcso',
-   caws_instance => 'm3.medium',
-   caws_key => 'add aws owner key here',
-   caws_secret_key => 'add aws secret owner key here',
-   caws_pem_name => 'gcso.pem',
-   caws_worker_instance_id => 'i-7788c97c',
-   caws_owner_keypair_name => 'gcso',
-   caws_eph_mount => '/ ephemeral_mount_dir or none',
- }
```

2. For a HVM image on AWS: (Copy these for other images and rename them)

```
- vi /etc/puppet/modules/awsexport/manifests/gcso_sl6_hvm.pp
- change parameters to your credentials.
- *Note: DO NOT CHANGE (caws_worker_instance_id => 'hvm') leave as 'hvm' to create a worker vm on aws

- #contains parameters for the gcso_sl6 HVM image conversion
- awsexport::awsexport_params {'gcso_sl6_hvm':
-   cvm_file_location => '/opt/gcso/awsexport',
-   cvm_image_location => 'oneadmin@fcl008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5',
-   ckernel_ver => '2.6.32-431.23.3.el6.x86_64',
-   cvm_number => '103',
-   cvm_name => 'gcso_sl6',
-   cvm_owner => 'your Fermi username',
-   caws_image_name => 'GCSO_SL6_HVM',
-   caws_image_owner => 'gcso',
-   caws_instance => 'm3.medium',
-   caws_key => 'add aws owner key here',
-   caws_secret_key => 'add aws secret owner key here',
-   caws_pem_name => 'gcso.pem',
-   caws_worker_instance_id => 'hvm',
-   caws_owner_keypair_name => 'gcso',
-   caws_eph_mount => '/ ephemeral_mount_dir or none',
- }
```

3. To setup Crontab jobs:

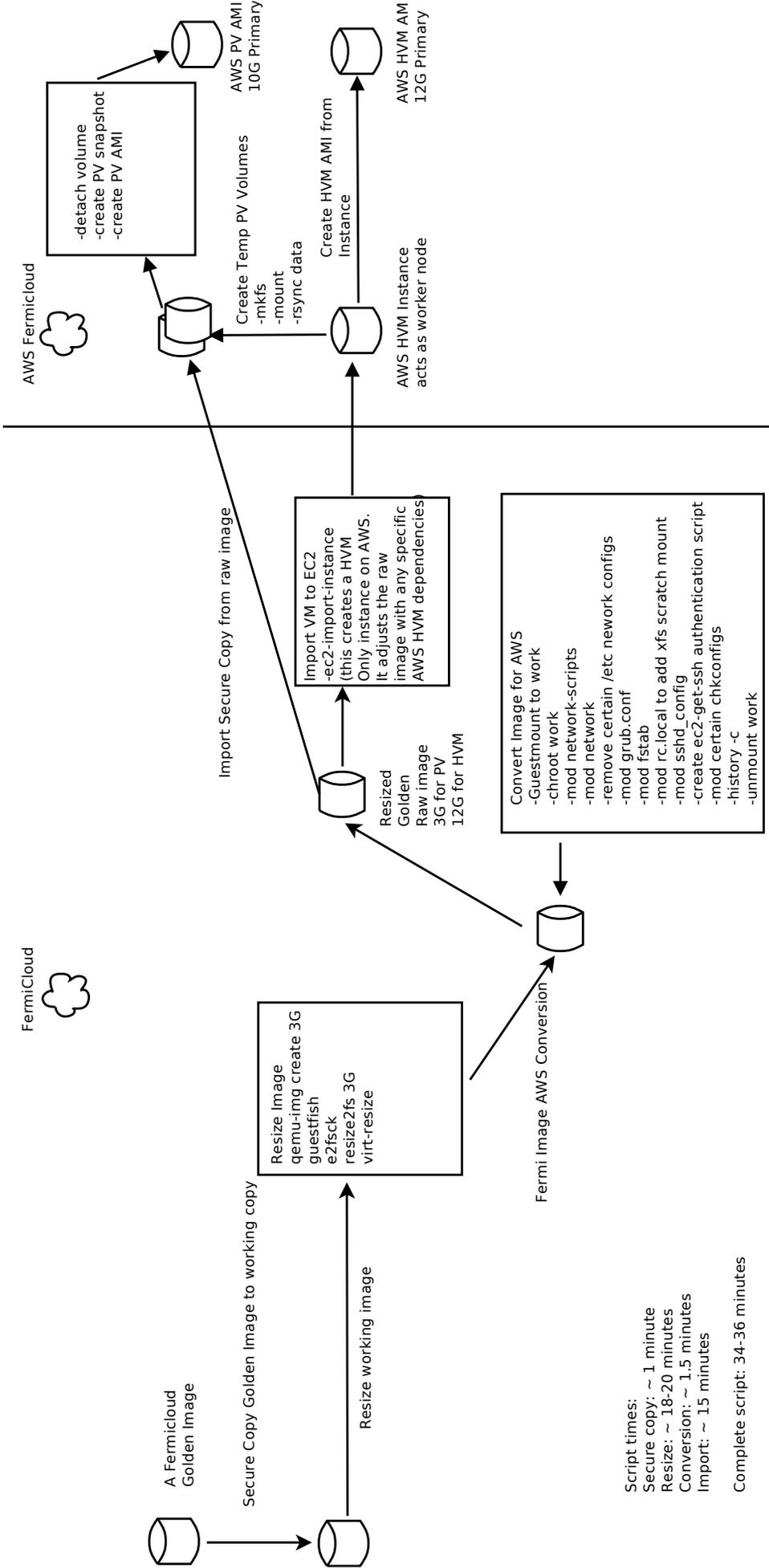
```
- vi /etc/puppet/modules/awsexport/manifests/awsexport_params.pp
- Change MAILTO=username@fnal.gov to your email address to receive cron job completion emails.
- Change the runtime schedule to what you want:
- cron {'awsexport':
-   minute => '55',
-   hour => '10',
-   monthday => '14',
-   month => '8',
-   weekday => '*',
```

4. Run Puppet apply to set crontab job for AWS HVM worker:

- run 'puppet apply /etc/puppet/modules/awsexport/manifests/gcso_sl6_hvm.pp'
- wait for cron completion email (in about 1.5 hours for a HVM Conversion)
- detail job log is located at: /opt/gcso/awsexport/aws_image_convert.log
- obtain aws console HVM worker node 'instance id' for subsequent PV conversion runs.
- *Note: this HVM worker node is needed only once. It can be run again for other images, if you want to provide HVM AMI's and instances on AWS.

5. Run Puppet apply to set crontab job for AWS PV Images:

- run 'puppet apply /etc/puppet/modules/awsexport/manifests/gcso_sl6_pv.pp'
- wait for cron completion email (in about 55 minutes for a PV Conversion)
- detail job log is located at: /opt/gcso/awsexport/aws_image_convert.log
- check aws console to see PV AMI's and instances.
- *Note: this job can run, as needed, to obtain a latest AWS image. The AWS AMI's and instances are time-stamped to identify the latest version.



Script times:
 Secure copy: ~ 1 minute
 Resize: ~ 18-20 minutes
 Conversion: ~ 1.5 minutes
 Import: ~ 15 minutes
 Complete script: 34-36 minutes

```

File: puppetrepo-awsexport/files/opt/gcso/awsexport/Convert.py
#!/usr/bin/env python
# encoding: utf-8
'''
FermiCloud.AWS.Images -- Convert FermiCloud KVM Images to AWS Xen Images

@author:      khs

@copyright:   2014 Fermilab. All rights reserved.

@license:     license

@contact:    kirkshal@fnal.gov
'''

import sys
import os
#import random
#import string
import subprocess
import logging
import textwrap
#import socket
import datetime
import time
#import httplib

from argparse import ArgumentParser
from argparse import RawDescriptionHelpFormatter

__all__ = []
__version__ = 0.1
__date__ = '2014-07-15'
__updated__ = '2014-07-15'

DEBUG = 1
TESTRUN = 0
PROFILE = 0

class CLIErr(Exception):
    '''Generic exception to raise and log different fatal errors.'''
    def __init__(self, msg):
        super(CLIErr).__init__(type(self))
        self.msg = "E: %s" % msg
    def __str__(self):
        return self.msg
    def __unicode__(self):
        return self.msg

    return self.msg

class Timer:
    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, *args):
        self.end = time.time()
        self.interval = self.end - self.start

def get_help():
    """This function outputs the general usage of the script and uses
    the text wrapper class to specify 80 columns for display.
    """
    wrapper = textwrap.TextWrapper(width=80,
        initial_indent=" " * 2,
        subsequent_indent=" " * 2,
        break_long_words=False,
        replace_whitespace=False,
        expand_tabs=True,
        break_on_hyphens=False)
    help_text= ("(Run Fermi's Kerberos Initialize before running this script: kinit userid). This script takes
    fifteen (15) arguments: "
        " <script dir location>, <vm image location>, <vm kernel version>, <fermicloud worker vm number>,
    <fermicloud vm name>, <fermicloud vm owner>, <aws image name>,"
        " <aws image owner>, <aws instance type>, <aws key>, <aws secret key>, <aws pem name>,"
        " <aws worker instance id>, <aws owner keypair name> and </ephemeral_mount_dir or none>."
        " A pre-requisite is to have a Fermicloud worker vm setup ahead of time
    (root@fermicloudnfn.fnl.gov) with a /data directory created to store files."
        " The script will accept a VM image
    (oneadmin@fc1008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5),"
        " with kernel (2.6.32-431.23.3.el6.x86_64) and will make a worker copy of the image ready for AWS
    conversion,"
        " then convert the image (fermi cleanse and resize) and import to AWS by the specified <aws image
    name>."
        " The script requires 5 parameters to be obtained ahead of time from the AWS Console. They are:"
        " <aws owner security pem name> <aws instance id of the HVM worker image, use 'hvm' here to
    create HVM worker before creating PV's.> <aws owner keypair name>"
        " <aws key> and <aws secret key> which are obtained from the AWS console under EC2 instances and
    the IAM security tab prior to running this script."
        " For example, to convert a Fermicloud VM named gcso_slf6 with owner oneadmin,"
        " and to create a AWS image named SLF6Vanilla, with owner oneadmin, and a instance type of
    m3.medium and with a worker"
        " vm of fermicloud103.fnl.gov you would run the following script:\n"
        " **\n")

```

```

    "/opt/gcso/awsexport/Convert.py /opt/gcso/awsexport
oneadmin@fc1008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5 2.6.32-431.23.3.el6.x86_64 103
gcso_slf6 oneadmin SLF6Vanilla gcso m3.medium awskey awssecretkey gcso.pem i-hvminstanceid gcso /scratch/n
    **\n"
    "...Available.AWS.Instance.Types.listed.below...\n"
    "Type.....vCPU..ECU..Memory.(GiB)..Instance Storage (GB)..Cost per hour\n"
    "General.Purpose.--Current.Generation\n"
    "t2.micro...1....Variable....1....EBSOnly....$0.013 per Hour\n"
    "t2.small...1....Variable....2....EBSOnly....$0.026 per Hour\n"
    "t2.medium...2....Variable....4....EBSOnly....$0.052 per Hour\n"
    "m3.medium...1.....3.....3.75...1.x.4.SSD...$0.070 per Hour\n"
    "m3.large...2.....6.5.....7.5...1.x.32.SSD...$0.140 per Hour\n"
    "m3.xlarge...4.....13.....15...2.x.40.SSD...$0.280 per Hour\n"
    "m3.2xlarge..8.....26.....30...2.x.80.SSD...$0.560 per Hour\n"
    "Compute.Optimized.--Current.Generation\n"
    "c3.large...2.....7.....3.75...2.x.16.SSD...$0.105 per Hour\n"
    "c3.xlarge...4.....14.....7.5...2.x.40.SSD...$0.210 per Hour\n"
    "c3.2xlarge..8.....28.....15...2.x.80.SSD...$0.420 per Hour\n"
    "c3.4xlarge..16.....55.....30...2.x.160.SSD...$0.840 per Hour\n"
    "c3.8xlarge..32...108.....60...2.x.320.SSD...$1.680 per Hour\n"
    "GPU.Instances.--Current.Generation\n"
    "g2.2xlarge..8.....26.....15.....60.SSD...$0.650 per Hour\n"
    "Memory.Optimized.--Current.Generation\n"
    "r3.large...2.....6.5.....15...1.x.32.SSD...$0.175 per Hour\n"
    "r3.xlarge...4.....13.....30.5...1.x.80.SSD...$0.350 per Hour\n"
    "r3.2xlarge..8.....26.....61...1.x.160.SSD...$0.700 per Hour\n"
    "r3.4xlarge..16.....52.....122...1.x.320.SSD...$1.400 per Hour\n"
    "r3.8xlarge..32...104.....244...2.x.320.SSD...$2.800 per Hour\n"
    "Storage.Optimized.--Current.Generation\n"
    "i2.xlarge...4.....14.....30.5...1.x.800.SSD...$0.853 per Hour\n"
    "i2.2xlarge..8.....27.....61...2.x.800.SSD...$1.705 per Hour\n"
    "i2.4xlarge..16.....53.....122...4.x.800.SSD...$3.410 per Hour\n"
    "i2.8xlarge..32...104.....244...8.x.800.SSD...$6.820 per Hour\n"
    "hs1.8xlarge.16.....35.....117...24.x.2048....$4.600 per Hour\n")
print '\n', wrapper.fill(help_text)
print
# sys.exit()

def copy_to_image_location(vm_script_location, vm_number, vm_name, vm_owner, aws_pem_name, vm_image_location):
    """ This function copies the selected Fermicloud VM image to a worker VM image.
    """
    get_date = datetime.date.today()
    time_differential = datetime.timedelta(days=7)
    delete_date = str(get_date - time_differential)
    logging.info("Start: Copying the selected Fermicloud VM image. Function copy_to_image_location.")
    """ Change to location of your scripts below.
    """
    cp_command = (

```

```

        "cp -f {v_script_location}/Fermi_AWS_Modifications.sh /data"
        "&& cp -f {v_script_location}/{v_aws_pem_name} /data"
        "&& cp -f {v_script_location}/Fermi_AWS_Resize.sh /data"
        "&& cp -f {v_script_location}/ec2-get-ssh /data"
        "&& cp -f {v_script_location}/cloud.cfg /data"
        "&& cp -f {v_script_location}/Fermi_AWS_CLI_Setup.sh /root"
        "&& cp -f {v_script_location}/Fermi_AWS_Import.sh /root"
        " "&& kinit -k -t /var/adm/krb5/cloudadminpp.keytab
cloudadmin/cron/fermicloudpp.fnal.gov@FNAL.GOV"
        " "&& scp {v_image_location} /data/{v_name}.qcow2tmp"
# The 2 lines above need to be modified for production to accept a parameter driven image
the one above is pre-production
# " "&& scp oneadmin@fc1008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5
/data/{v_name}.qcow2tmp"
# " "&& ssh -o StrictHostKeyChecking=no -l {v_owner} fermicloud.fnal.gov 'scp -Cq
{v_owner}@fermicloud.fnal.gov:{v_name}.qcow2 root@fermicloud{v_number}.fnal.gov:/data/{v_name}.qcow2tmp"
# " "&& rm -f {v_owner}@fermicloud.fnal.gov:{v_name}.qcow2.{remove} && exit'"
).format (v_owner=vm_owner, today=str(get_date), v_name=vm_name, v_number=vm_number,
v_script_location=vm_script_location, v_aws_pem_name=aws_pem_name, v_image_location=vm_image_location,
remove=delete_date)
with open('/opt/gcso/awsexport/aws_image_convert.log', 'a') as my_log:

    try:
        with Timer() as t:
            subprocess.check_call(cp_command, shell=True, stdout=my_log, stderr=my_log)
            logging.info("Stop: Completed Copying the selected Fermicloud VM image. Function
copy_to_image_location.")
        except:
            logging.error("Couldn't copy image to temp area")
            raise Exception("Couldn't copy image to temp area! Aborting.")
            sys.exit(1)
        finally:
            min_interval = t.interval / 60
            print('Copying took %.03f minutes.' % min_interval)

def convert_image(vm_number, vm_name, vm_owner, aws_worker_instance_id, aws_instance, kernel_ver, eph_mount):
    """ This function converts the worker Fermicloud VM image for AWS specifics.
    """
    get_date = datetime.date.today()
    logging.info("Start: Converting the worker Fermicloud VM image. Function convert_image.")
    convert_command = (
        # "ssh root@fermicloud{v_number}.fnal.gov 'mkdir -p /data/work"
        "mkdir -p /data/work"
        "&& guestmount -a /data/{v_name}.raw -m /dev/sdal /data/work"
        "&& mv /data/Fermi_AWS_Modifications.sh /data/work"
        "&& mv /data/ec2-get-ssh /data/work"
        "&& mv /data/cloud.cfg /data/work"

```

```

        " && /usr/sbin/chroot /data/work ./Fermi_AWS_Modifications.sh {v_aws_worker_instance_id}
{v_aws_instance} {v_kernel_ver} {v_eph_mount}"
        " && sleep 10"
        " && rm -f /data/work/Fermi_AWS_Modifications.sh && rm -f /data/work/ec2-get-ssh && rm -f
/data/work/cloud.cfg"
        " && rm -f /data/Fermi_AWS_Modifications.sh && rm -f /data/ec2-get-ssh && rm -f
/data/cloud.cfg"
        " && fusermount -uz /data/work && rmdir /data/work"
        #
        " && exit"
    ).format (v_owner=vm_owner, today=str(get_date), v_name=vm_name, v_number=vm_number,
v_aws_worker_instance_id=aws_worker_instance_id, v_aws_instance=aws_instance, v_kernel_ver=kernel_ver,
v_eph_mount=eph_mount)
    with open('/opt/gcso/awsexport/aws_image_convert.log', 'a') as my_log:

        try:
            with Timer() as t:
                subprocess.check_call(convert_command, shell=True, stdout=my_log, stderr=my_log)
            logging.info("Stop: Completed Converting the worker Fermicloud VM image. Function
convert_image.")
        except:
            logging.error("Couldn't convert image in temp area")
            raise Exception("Couldn't convert image in temp area! Aborting.")
            sys.exit(1)
        finally:
            min_interval = t.interval / 60
            print('Converting took %.03f minutes.' % min_interval)

def resize_image(vm_number, vm_name, vm_owner, aws_worker_instance_id):
    """ This function resizes the worker Fermicloud VM image for AWS image import.
    """
    get_date = datetime.date.today()
    logging.info("Start: Resizing the worker Fermicloud VM image. Function resize_image.")
    resize_command = (
        #
        "ssh root@fermicloud{v_number}.fnal.gov 'cd /data"
        "cd /data"
        " && ./Fermi_AWS_Resize.sh {v_name} {v_aws_worker_instance_id}"
        " && rm -f /data/Fermi_AWS_Resize.sh"
        #
        " && exit'"
    ).format (v_owner=vm_owner, today=str(get_date), v_name=vm_name, v_number=vm_number,
v_aws_worker_instance_id=aws_worker_instance_id)
    with open('/opt/gcso/awsexport/aws_image_convert.log', 'a') as my_log:

        try:
            with Timer() as t:
                subprocess.check_call(resize_command, shell=True, stdout=my_log, stderr=my_log)
            logging.info("Stop: Completed Resizing the worker Fermicloud VM image. Function resize_image.")
        except:
            logging.error("Couldn't resize image in temp area")

```

```

        raise Exception("Couldn't resize image in temp area! Aborting.")
        sys.exit(1)
    finally:
        min_interval = t.interval / 60
        print('Resizing took %.03f minutes.' % min_interval)

def import_image(vm_number, vm_name, vm_owner, aws_instance, aws_key, aws_secret_key, aws_pem_name,
aws_worker_instance_id, aws_image_owner, aws_image_name, aws_owner_keypair_name, my_env):
    """ This function sets up the AWS CLI tools and imports the resized raw worker Fermicloud VM image to a AWS
HVM Instance.
    """
    get_date = datetime.date.today()
    logging.info("Start: Importing the worker Fermicloud VM image. Function import_image.")
    import_command = (
        #
        "ssh root@fermicloud{v_number}.fnal.gov 'cd /root && chmod +x
Fermi_AWS_CLI_Setup.sh"
        "cd /root && chmod +x Fermi_AWS_CLI_Setup.sh"
        " && ./Fermi_AWS_CLI_Setup.sh {v_aws_key} {v_aws_secret_key}"
        " && rm -f /root/Fermi_AWS_CLI_Setup.sh && cd /root && chmod +x Fermi_AWS_Import.sh"
        " && ./Fermi_AWS_Import.sh {v_aws_key} {v_aws_secret_key} {v_name} {v_aws_instance}
{v_aws_pem_name} {v_aws_worker_instance_id} {v_aws_image_owner} {v_aws_image_name} {v_aws_owner_keypair_name}"
        " && rm -f /root/Fermi_AWS_Import.sh"
        #
        " && exit'"
    ).format (v_owner=vm_owner, today=str(get_date), v_name=vm_name,
v_number=vm_number, v_aws_instance=aws_instance, v_aws_key=aws_key,
v_aws_secret_key=aws_secret_key, v_aws_pem_name=aws_pem_name,
v_aws_worker_instance_id=aws_worker_instance_id,
v_aws_image_owner=aws_image_owner, v_aws_image_name=aws_image_name,
v_aws_owner_keypair_name=aws_owner_keypair_name)
    with open('/opt/gcso/awsexport/aws_image_convert.log', 'a') as my_log:

        try:
            with Timer() as t:
                subprocess.check_call(import_command, env=my_env, shell=True, stdout=my_log, stderr=my_log)
            logging.info("Stop: Completed Importing the worker Fermicloud VM image. Function import_image.")
        except:
            logging.error("Couldn't import image to AWS")
            raise Exception("Couldn't import image to AWS! Aborting.")
            sys.exit(1)
        finally:
            min_interval = t.interval / 60
            print('Importing took %.03f minutes.' % min_interval)

def main(argv=None): # IGNORE:C0111
    """ Change location of Log file.
    """
    logging.basicConfig(level=logging.DEBUG,
format='%(asctime)s %(levelname)-8s %(message)s',

```

```

        datefmt='%b %d %H:%M:%S',
        filename='/opt/gcso/awsexport/aws_image_convert.log')

my_env = os.environ.copy()

'''Command line options.'''

if argv is None:
    argv = sys.argv
else:
    sys.argv.extend(argv)

get_help()
program_name = os.path.basename(sys.argv[0])
program_version = "v%s" % __version__
program_build_date = str(__updated__)
program_version_message = '%%(prog)s %s (%s)' % (program_version, program_build_date)
program_shortdesc = __import__('__main__').__doc__.split("\n")[1]
program_license = ' '

Created by khs on %s.
Copyright 2014 Fermilab. All rights reserved.

Licensed under the Apache License 2.0
http://www.apache.org/licenses/LICENSE-2.0

Distributed on an "AS IS" basis without warranties
or conditions of any kind, either express or implied.

USAGE

''' % (program_shortdesc, str(__date__))
try:
    with Timer() as t:
        # Setup argument parser
        parser = ArgumentParser(description=program_license, formatter_class=RawDescriptionHelpFormatter)
        parser.add_argument('-V', '--version', action='version', version=program_version_message)
        parser.add_argument(dest="cvm_script_location", help="Location of all scripts",
metavar="cvm_script_location")
        parser.add_argument(dest="cvm_image_location", help="Location of Fermi Image",
metavar="cvm_image_location")
        parser.add_argument(dest="ckernel_ver", help="Kernel version of Fermi Image", metavar="ckernel_ver")
        parser.add_argument(dest="cvm_number", help="Fermicloud Work VM number", metavar="cvm_number")
        parser.add_argument(dest="cvm_name", help="Fermicloud VM name", metavar="cvm_name")
        parser.add_argument(dest="cvm_owner", help="Fermicloud Owner name", metavar="cvm_owner")
        parser.add_argument(dest="caws_image_name", help="AWS AMI VM name", metavar="caws_image_name")
        parser.add_argument(dest="caws_image_owner", help="AWS AMI Owner name", metavar="caws_image_owner")
        parser.add_argument(dest="caws_instance", help="AWS AMI Instance Type", metavar="caws_instance")

        parser.add_argument(dest="caws_key", help="AWS key", metavar="caws_key")
        parser.add_argument(dest="caws_secret_key", help="AWS Secret Key", metavar="caws_secret_key")
        parser.add_argument(dest="caws_pem_name", help="AWS PEM Name", metavar="caws_pem_name")
        parser.add_argument(dest="caws_worker_instance_id", help="AWS Worker Instance ID from AWS Console",
metavar="caws_worker_instance_id")
        parser.add_argument(dest="caws_owner_keypair_name", help="AWS Owner Keypair Name",
metavar="caws_owner_keypair_name")
        parser.add_argument(dest="caws_eph_mount", help="AWS ephemeral mount dir (/ephemeral_mount_dir or
none)", metavar="caws_eph_mount")

        # Process arguments
        args = parser.parse_args()
        cvm_script_location = args.cvm_script_location
        cvm_image_location = args.cvm_image_location
        ckernel_ver = args.ckernel_ver
        cvm_number = args.cvm_number
        cvm_name = args.cvm_name
        cvm_owner = args.cvm_owner
        caws_image_name = args.caws_image_name
        caws_image_owner = args.caws_image_owner
        caws_instance = args.caws_instance
        caws_key = args.caws_key
        caws_secret_key = args.caws_secret_key
        caws_pem_name = args.caws_pem_name
        caws_worker_instance_id = args.caws_worker_instance_id
        caws_owner_keypair_name = args.caws_owner_keypair_name
        caws_eph_mount = args.caws_eph_mount

        logging.info('Begin script run')
        print("Arguments supplied:")
        print("location of all files and scripts->", cvm_script_location)
        print("location of fermicloud vm image->", cvm_image_location)
        print("kernel version of fermicloud vm image->", ckernel_ver)
        print("fermicloud worker vm number->", cvm_number)
        print("fermicloud vm image name->", cvm_name)
        print("fermicloud vm owner->", cvm_owner)
        print("aws vm image name->", caws_image_name)
        print("aws vm image owner->", caws_image_owner)
        print("aws instance type->", caws_instance)
        print("aws owner key->", caws_key)
        print("aws owner secret key->", caws_secret_key)
        print("aws owner pem name->", caws_pem_name)
        print("aws hvm worker instance id from aws console for PV or 'hvm' for HVM->",
caws_worker_instance_id)
        print("aws owner keypair name->", caws_owner_keypair_name)
        print("aws ephemeral mount->", caws_eph_mount)
        print("Starting AWS VM conversion. This job can take up to 60 minutes for PV and 84 for HVM...")
        print("Copying the Golden Fermicloud VM image takes under 1 minute...")

```

```

        copy_to_image_location(cvm_script_location, cvm_number, cvm_name, cvm_owner, caws_pem_name,
cvm_image_location)
        print("Resizing the worker Fermicloud VM image takes up to 20 minutes for PV and 10 for HVM...")
        resize_image(cvm_number, cvm_name, cvm_owner, caws_worker_instance_id)
        print("Converting the worker Fermicloud VM image takes over 24 minutes...")
        convert_image(cvm_number, cvm_name, cvm_owner, caws_worker_instance_id, caws_instance, ckernel_ver,
caws_eph_mount)
        print("Importing the raw image to AWS takes up to 15 minutes for PV and 49 for HVM...")
        import_image(cvm_number, cvm_name, cvm_owner, caws_instance, caws_key, caws_secret_key,
caws_pem_name, caws_worker_instance_id, caws_image_owner, caws_image_name, caws_owner_keypair_name, my_env)
        print("Completed AWS VM conversion.")
        logging.info('End script run')
        return 0
    except KeyboardInterrupt:
        ### handle keyboard interrupt ###
        return 0
    except Exception, e:
        if DEBUG or TESTRUN:
            raise(e)
            indent = len(program_name) * " "
            sys.stderr.write(program_name + ": " + repr(e) + "\n")
            sys.stderr.write(indent + " for help use --help")
        return 2
    finally:
        min_interval = t.interval / 60
        print('Job took %.03f minutes. Thank you.' % min_interval)

if __name__ == "__main__":
    sys.exit(main())
-----
-----
-----

```

```

File: puppetrepo-awsexport/files/opt/gcso/awsexport/Fermi_AWS_CLI_Setup.sh
#!/bin/bash

```

```

# Use the following steps to install the Amazon API Tools and the Amazon AMI Tools on the Linux platform. 2
positional parameters are needed; You will need your AWS_ACCESS_KEY as the 1st parameter and AWS_SECRET_KEY as
the second parameter.
# Remember to 'chmod +x Fermi_AWS_CLI_Setup.sh' after copying this script to your root directory
# Syntax Format is ./Fermi_AWS_CLI_Setup.sh <AWS_ACCESS_KEY> <AWS_SECRET_KEY>

# Shell Login Script

# Add the following environment variables to your shell login script (i.e. /root/.bashrc). Make any necessary
changes for your specific environment by replacing AWS_ACCOUNT_NUMBER, AWS_ACCESS_KEY_ID, and

```

```

AWS_SECRET_ACCESS_KEY with your AWS account number and security credentials. Make certain to remove the < and >
characters when providing your values.

```

```

cp /root/.bashrc /root/.bashrc.backup
cd /root
rm -f .bashrc

echo '# .bashrc' >> .bashrc
echo '# ' >> .bashrc
echo '# User specific aliases and functions' >> .bashrc
echo '# ' >> .bashrc
echo "alias rm='rm -i'" >> .bashrc
echo "alias cp='cp -i'" >> .bashrc
echo "alias mv='mv -i'" >> .bashrc
echo '# ' >> .bashrc
echo '# Source global definitions' >> .bashrc
echo 'if [ -f /etc/bashrc ]; then' >> .bashrc
echo '    . /etc/bashrc' >> .bashrc
echo 'fi' >> .bashrc
echo '# ' >> .bashrc
echo 'PATH=$PATH:$HOME/bin' >> .bashrc
echo '# ' >> .bashrc
echo 'export PATH' >> .bashrc
echo '# ' >> .bashrc
echo 'export EC2_BASE=/usr/local/ec2' >> .bashrc
echo 'export EC2_HOME=$EC2_BASE/ec2-api-tools-1.7.1.0' >> .bashrc
echo 'export EC2_PRIVATE_KEY=$EC2_BASE/certificates/ec2-pk.pem' >> .bashrc
echo 'export EC2_CERT=$EC2_BASE/certificates/ec2-cert.pem' >> .bashrc
echo 'export EC2_URL=https://ec2.us-west-2.amazonaws.com' >> .bashrc
echo 'export AWS_ACCOUNT_NUMBER=159067897602' >> .bashrc
echo 'export AWS_ACCESS_KEY_ID=$1' >> .bashrc
echo 'export AWS_ACCESS_KEY=$1' >> .bashrc
echo 'export AWS_SECRET_ACCESS_KEY=$2' >> .bashrc
echo 'export AWS_SECRET_KEY=$2' >> .bashrc
echo 'export PATH=$PATH:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin:$EC2_HOME/bin' >> .bashrc
echo 'export JAVA_HOME=/usr/lib/jvm/jre-1.6.0-openjdk.x86_64' >> .bashrc

source ~/.bashrc

cp /root/.bash_profile /root/.bash_profile.backup
cd /root
rm -f .bash_profile

echo '# Get the aliases and functions' >> .bash_profile
echo 'if [ -f ~/.bashrc ]; then' >> .bash_profile
echo '    . ~/.bashrc' >> .bash_profile
echo 'fi' >> .bash_profile
echo '# ' >> .bash_profile

```

```

echo '# User specific environment and startup programs' >> .bash_profile
echo ' ' >> .bash_profile
echo 'PATH=$PATH:$HOME/bin' >> .bash_profile
echo ' ' >> .bash_profile
echo 'export PATH' >> .bash_profile
echo ' ' >> .bash_profile
echo 'export EC2_BASE=/usr/local/ec2' >> .bash_profile
echo 'export EC2_HOME=$EC2_BASE/ec2-api-tools-1.7.1.0' >> .bash_profile
echo 'export EC2_PRIVATE_KEY=$EC2_BASE/certificates/ec2-pk.pem' >> .bash_profile
echo 'export EC2_CERT=$EC2_BASE/certificates/ec2-cert.pem' >> .bash_profile
echo 'export EC2_URL=https://ec2.us-west-2.amazonaws.com' >> .bash_profile
echo 'export AWS_ACCOUNT_NUMBER=159067897602' >> .bash_profile
echo 'export AWS_ACCESS_KEY_ID=$1' >> .bash_profile
echo 'export AWS_ACCESS_KEY=$1' >> .bash_profile
echo 'export AWS_SECRET_ACCESS_KEY=$2' >> .bash_profile
echo 'export AWS_SECRET_KEY=$2' >> .bash_profile
echo 'export PATH=$PATH:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin:$EC2_HOME/bin' >>
.bash_profile
echo 'export JAVA_HOME=/usr/lib/jvm/jre-1.6.0-openjdk.x86_64' >> .bash_profile

source ~/.bash_profile

# Install Java

# The EC2 API Tools and Amazon EC2 AMI Tools are Java based. If you don't already have a version of Java
installed, do so now.

yum -y install java-1.6.0-openjdk

# The JAVA_HOME environment variable should be set to the appropriate home directory in your shell login script
(i.e. /root/.bashrc) which was handled in the previous step. Verify the JAVA_HOME environment variable is set for
the current shell and confirm that Java is installed correctly.

echo $JAVA_HOME

# java -version

# Install the Amazon EC2 Tools

# Download the Amazon EC2 API Tools.

mkdir -p $EC2_HOME

curl -o /tmp/ec2-api-tools.zip http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip

unzip -qq -o /tmp/ec2-api-tools.zip -d /tmp

cp -r /tmp/ec2-api-tools-*/ $EC2_HOME

```

```

# Download the Amazon EC2 AMI Tools to the EC2 image.

curl -o /tmp/ec2-ami-tools.zip http://s3.amazonaws.com/ec2-downloads/ec2-ami-tools.zip

unzip -qq -o /tmp/ec2-ami-tools.zip -d /tmp

cp -rf /tmp/ec2-ami-tools-*/ $EC2_HOME

# EC2 Private Certificate Key File and EC2 Certificate File

# If used (Certificates are being deprecated), Copy your X.509 Certificate (private key file and certificate
file) to appropriate directory. For the purpose of this example, I will be renaming my private key file from pk-
2L7LZYRTNEAC4KGZMPPZWA0Z4KYCTCA4.pem to ec2-pk.pem and my certificate file from cert-
2L7LZYRTNEAC4KGZMPPZWA0Z4KYCTCA4.pem to ec2-cert.pem.

mkdir -p $EC2_BASE/certificates

# cp pk-2L7LZYRTNEAC4KGZMPPZWA0Z4KYCTCA4.pem $EC2_BASE/certificates/ec2-pk.pem

# cp cert-2L7LZYRTNEAC4KGZMPPZWA0Z4KYCTCA4.pem $EC2_BASE/certificates/ec2-cert.pem

# Verify Amazon EC2 Tools

# Verify that the Amazon EC2 Tools have been installed correctly.

# Test the ec2-describe-regions script which is found in the EC2 API Tools to list the regions you have access
to.

# ec2-describe-regions

# Install Expect used for interaction 'yes' prompts throughout script

yum -y install tcl-devel tk-devel expect-devel expectk

# Package updates completed
sleep 1
echo 'AWS CLI Tools Setup Completed.'
-----
-----

File: puppetrepo-awsexport/files/opt/gcso/awsexport/Fermi_AWS_Import.sh
#!/bin/bash

# Use the following steps to perform a VM import to create a HVM Instance on AWS.
# Remember to 'chmod +x Fermi_AWS_Import.sh' after copying this script to your root directory

```

```

# Syntax Format is ./Fermi_AWS_Import.sh <AWS_ACCESS_KEY> <AWS_SECRET_KEY> <Fermicloud VM Name> <AWS Instance
Type> <PEM Name> <AWS Worker Instance ID> <AWS image owner> <AWS image name> <AWS owner keypair name>

cd /data
dte=`date +%Y%m%d%H%M%S`
chmod 400 $5

if [ "$6" == "hvm" ]; then
# VM Import used to create AWS HVM Worker Image that is used for building Paravirtual Images
# Note: That the size of the primary boot partition must stay at 12G to prevent freeze at 53% conversion
import=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2iin -f RAW -t $4 -a x86_64 -b fcloudimport$dte -o $1 -O $1 -w
$2 -W $2 -p Linux -g sg-a6a010c3 -s 13 -d $8_$dte -K $9 --region us-west-2 -z us-west-2a $3.raw | grep
IMPORTINSTANCE | awk {'print($4)'}
sleep 5
# poll every 60 seconds to see if import is completed
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-conversion-tasks -O $1 -W $2 $import | awk
'/IMPORTINSTANCE/{print $8}')
while [ "$status" != "completed" ] ; do
sleep 60
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-conversion-tasks -O $1 -W $2 $import | awk
'/IMPORTINSTANCE/{print $8}')
done
sleep 5

volimp=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-conversion-tasks -O $1 -W $2 $import | grep
DISKIMAGE | awk {'print($7)'}
sleep 2
vmimpinstance=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-conversion-tasks -O $1 -W $2 $import | grep
IMPORTINSTANCE | awk {'print($10)'}
sleep 2
# Create Snapshot to be used to create HVM AMI
snap=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-create-snapshot -O $1 -W $2 --region us-west-2 -d "$8_$dte HVM
Snapshot" $volimp | grep SNAPSHOT | awk {'print($2)'}
sleep 5
# poll every 10 seconds to see if snapshot is completed
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-snapshots -O $1 -W $2 $snap | awk
'/SNAPSHOT/{print $4}')
while [ "$status" != "completed" ] ; do
sleep 10
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-snapshots -O $1 -W $2 $snap | awk
'/SNAPSHOT/{print $4}')
done
sleep 5

# Create HVM AMI
ami=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-register -O $1 -W $2 -n "$8_$dte" -d "$8_$dte" -b
"/dev/sd1=$snap:13:true:standard" -b "/dev/sdb=ephemeral0" --architecture x86_64 --region us-west-2 --
virtualization-type hvm | grep IMAGE | awk {'print($2)'}

```

```

sleep 30

# Create and run Instance from HVM AMI (Comment out for golive)
instances=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-run-instances -O $1 -W $2 --region us-west-2 -z us-west-2a
$ami -g sg-a6a010c3 -n 1 -t $4 -k $9 | grep INSTANCE | awk {'print($2)'}
sleep 5
# poll every 10 seconds to see if instance status is running
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $instance | awk
'/INSTANCE/{print $6}')
while [ "$status" != "running" ] ; do
sleep 10
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $instance | awk
'/INSTANCE/{print $6}')
done
sleep 5

# Get Volume for create tags
vol2=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-volumes -O $1 -W $2 | grep $instance | awk
{'print($2)'}
sleep 10

# Create and attach a work volume for ongoing PV conversions
vol3=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-create-volume -O $1 -W $2 --size 20 --region us-west-2 -z us-
west-2a --type standard | grep VOLUME | awk {'print($2)'}
sleep 10

/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-attach-volume $vol3 -O $1 -W $2 --instance $instance --device
/dev/sdf

# Create tags for all resources
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-create-tags -O $1 -W $2 $vmimpinstance $volimp $snap $ami $instance
$vol2 $vol3 --tag "Name=$8_$dte" --tag "User=$7"
sleep 10

# Stop the AWS HVM Worker and delete temp work volume
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-stop-instances $instance -O $1 -W $2
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-terminate-instances $vmimpinstance -O $1 -W $2
sleep 90
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-delete-volume $volimp -O $1 -W $2
pip install --upgrade awscli
aws s3 rb s3://fcloudimport$dte --force

echo 'AWS HVM VM Import Completed.'
exit
fi

# Launch new HVM AMI Instance (AWS Instance ID is needed as a parameter to the script)
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-start-instances $6 -O $1 -W $2

```

```

# poll every 10 seconds to see if instance status is running
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $6 | awk '/INSTANCE/{print $6}')
while [ "$status" != "running" ] ; do
sleep 10
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $6 | awk '/INSTANCE/{print $6}')
done
sleep 5

# Wait for instance to launch and get public ip address and setup staging area
ip=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $6 | grep NICASSOCIATION | awk {'print($2)'}^
sleep 10

# Create new volume on AWS Worker to be used for PV snapshot
vol=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-create-volume -O $1 -W $2 --size 10 --region us-west-2 -z us-west-2a | grep VOLUME | awk {'print($2)'}^
sleep 10

# Attach volume on AWS Worker and Wait for attachment to complete and setup mount for data
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-attach-volume -O $1 -W $2 --instance $6 --device /dev/sdg $vol --region us-west-2
sleep 10

# build the expect script in bash

expect1_sh=$(expect -c "
spawn ssh -i $5 root@$ip
expect \"yes/no\"
send \"yes\r\"
expect \"#\"
send \"mkfs -t ext4 /dev/xvdf\r\"
expect \"#\"
send \"mount -t ext4 /dev/xvdf /mnt\r\"
expect \"#\"
send \"cd /mnt\r\"
expect \"#\"
send \"mkdir -p images\r\"
expect \"#\"
send \"mkfs -t ext4 /dev/xvdg\r\"
expect \"#\"
send \"mkdir -p /opt/ec2/mnt\r\"
expect \"#\"
send \"mount -t ext4 /dev/xvdg /opt/ec2/mnt\r\"
expect \"#\"
send \"exit\r\"
)

```

```

expect eof
puts \"spawned process completed...\"
exit
)

# run the expect script and then exit it
echo \"$expect1_sh\"

echo \"Back to script. Please wait...\"
sleep 5

# Upload Fericloud prepaed raw image to AWS Worker stage area (about 3 minutes)
# rsync or scp can be used, rsync requies root password to be passed (so default to scp)
echo \"Copying (SCP) Raw image to AWS. Please wait...\"
# rsync -S -z -q -e ssh -i $5 /data/$3.raw root@$ip:/mnt/images
scp -Cq -i $5 /data/$3.raw root@$ip:/mnt/images

sleep 5

# Prepare raw image to become extracted boot volume on AWS Worker

# build the expect script in bash

expect3_sh=$(expect -c "
spawn ssh -i $5 root@$ip
expect \"#\"
send \"cd /mnt/images\r\"
expect \"#\"
send \"mkdir -p raw\r\"
expect \"#\"
send \"kpartx -a $3.raw\r\"
expect \"#\"
send \"mount /dev/mapper/loop0p1 /mnt/images/raw\r\"
expect \"#\"
send \"cd /mnt/images/raw\r\"
expect \"#\"
send \"rsync -aqHx /mnt/images/raw/ /opt/ec2/mnt\r\"
sleep 120
expect \"#\"
send \"rsync -aqHx /mnt/images/raw/dev /opt/ec2/mnt\r\"
expect \"#\"
send \"cd /opt/ec2/mnt\r\"
expect \"#\"
send \"tune2fs -L '\/' /dev/xvdg\r\"
expect \"#\"
send \"sync;sync;sync;sync\r\"
expect \"#\"
send \"cd /mnt/images\r\"
)

```

```

        expect \"#\
send \"umount /mnt/images/raw\r\"
        expect \"#\
send \"kpartx -d $3.raw\r\"
        expect \"#\
send \"exit\r\"
        expect eof
        puts \"spawned process completed...\"
        exit
    )

    # run the expect script and then exit it
    echo \"$expect3_sh\"

echo \"Back to script. Please wait...\"
sleep 5

# Detach temp worker volume
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-detach-volume $vol -O $1 -W $2
sleep 60

# Create Snapshot to be used to create PV AMI
snap=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-create-snapshot -O $1 -W $2 --region us-west-2 -d \"$8_sdte PV
Snapshot\" $vol | grep SNAPSHOT | awk {'print($2)'} `
sleep 5
# poll every 10 seconds to see if snapshot is completed
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-snapshots -O $1 -W $2 $snap | awk
'/SNAPSHOT/{print $4}')
while [ \"$status\" != \"completed\" ] ; do
sleep 10
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-snapshots -O $1 -W $2 $snap | awk
'/SNAPSHOT/{print $4}')
done
sleep 5

# Create PV AMI
ami=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-register -O $1 -W $2 -n \"$8_sdte\" -d \"$8_sdte\" -b
/dev/sd1=$snap:10:true:standard\" -b \"/dev/sdb=ephemeral0\" --architecture x86_64 --kernel aki-fc8f11cc --region
us-west-2 | grep IMAGE | awk {'print($2)'} `
sleep 30

# Create and run Instance from PV AMI (Comment out for golive)
instances=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-run-instances -O $1 -W $2 --region us-west-2 -z us-west-2a
$ami -g sg-a6a010c3 -n 1 --kernel aki-fc8f11cc -t $4 -k $9 | grep INSTANCE | awk {'print($2)'} `
sleep 5
# poll every 10 seconds to see if instance status is running
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $instance | awk
'/INSTANCE/{print $6}')

```

```

while [ \"$status\" != \"running\" ] ; do
sleep 10
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $instance | awk
'/INSTANCE/{print $6}')
done
sleep 5

# Get Volume for create tags (Comment out for golive)
vol2=`/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-volumes -O $1 -W $2 | grep $instance | awk
{'print($2)'} `
sleep 10

# Create tags for all resources
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-create-tags -O $1 -W $2 $vol $snap $ami $instance $vol2 --tag
\"Name=$8_sdte\" --tag \"User=$7\"
sleep 10

# Stop the AWS HVM Worker and new PV instances and delete temp HVM work volume
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-stop-instances $6 -O $1 -W $2
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-stop-instances $instance -O $1 -W $2
sleep 5
# poll every 10 seconds to see if instance status is stopped
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $instance | awk
'/INSTANCE/{print $5}')
while [ \"$status\" != \"stopped\" ] ; do
sleep 10
status=$(/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-describe-instances -O $1 -W $2 $instance | awk
'/INSTANCE/{print $5}')
done
sleep 5
/usr/local/ec2/ec2-api-tools-1.7.1.0/bin/ec2-delete-volume $vol -O $1 -W $2

echo 'AWS VM Import Completed.'

-----
-----
-----

```

```

File: puppetrepo-awsexport/files/opt/gcso/awsexport/Fermi_AWS_Modifications.sh
#!/bin/bash
# This is a script to add or modify several OS networking files required for the AWS image conversion

# Remove glideinwms-vm-one (OpenNebula), if exists, and Install glideinwms-vm-ec2 (AWS)

yum -y remove glideinwms-vm-one
yum -y install glideinwms-vm-core
yum -y install glideinwms-vm-ec2

```

```

# Install cloud-init for aws metadata user data and scripts

rpm -Uvh http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
yum -y install cloud-init
yum -y install cloud-utils
yum -y install http://repos.fedorapeople.org/repos/openstack/openstack-havana/epel-6/python-backports-1.0-4.el6.x86_64.rpm
cd /
mv -f cloud.cfg /etc/cloud

# Modify ifcfg-eth0
cd /etc/sysconfig/network-scripts
rm -f ifcfg-eth0
echo DEVICE=eth0 >> ifcfg-eth0
echo BOOTPROTO=dhcp >> ifcfg-eth0
echo ONBOOT=yes >> ifcfg-eth0
echo TYPE=Ethernet >> ifcfg-eth0

# Modify network
cd /etc/sysconfig
rm -f network
echo NETWORKING=yes >> network

# Remove fermi network resolv and hosts that AWS will not use
cd /etc
rm -f /etc/resolv.conf
rm -f /etc/hosts
rm -f /etc/hosts.allow
rm -f /etc/hosts.deny

# Remove fermi specifics that AWS will not use
rm -f /etc/yp.conf
rm -f /etc/auto.master
rm -f /etc/auto.misc
rm -f /etc/rc.d/rc3.d/K99.credentials
rm -f /etc/rc.d/rc3.d/S09one-context
rm -f /etc/rc.d/rc3.d/K84one-context
rm -f /etc/init.d/one-context
rm -f /etc/init.d/credentials*

# Create grub.conf for AWS paravirtual
cd /boot/grub
rm -f grub.conf
echo default=0 >> grub.conf
echo timeout=0 >> grub.conf
echo 'title Scientific Linux Fermi ('$3')' >> grub.conf
if [ "$1" == "hvm" ]; then

```

```

echo 'root (hd0,0)' >> grub.conf
echo 'kernel /boot/vmlinuz-'$3' ro root=/dev/xvda1 rd_NO_PLYMOUTH' >> grub.conf
else
echo 'root (hd0)' >> grub.conf
echo 'kernel /boot/vmlinuz-'$3' ro root=/dev/xvde1 rd_NO_PLYMOUTH' >> grub.conf
fi
echo 'initrd /boot/initramfs-'$3'.img' >> grub.conf

# Create symbolic link for boot
cd /boot; ln -s . boot

# Create fstab for AWS paravirtual
cd /etc
rm -f fstab

if [ "$1" == "hvm" ]; then
echo '/dev/xvda1 / ext3 defaults 1 1' >> fstab
if [ "$4" != "none" ] && [ "$2" != "t2.micro" ] && [ "$2" != "t2.small" ] && [ "$2" != "t2.medium" ]; then
echo '/dev/xvdb $4 xfs defaults 0 0' >> fstab
fi
else
echo '/dev/xvde1 / ext4 defaults 1 1' >> fstab
if [ "$4" != "none" ] && [ "$2" != "t2.micro" ] && [ "$2" != "t2.small" ] && [ "$2" != "t2.medium" ]; then
echo '/dev/xvdf $4 xfs defaults 0 0' >> fstab
fi
fi

echo 'tmpfs /dev/shm tmpfs defaults 0 0' >> fstab
echo 'devpts /dev/pts devpts gid=5,mode=620 0 0' >> fstab
echo 'sysfs /sys sysfs defaults 0 0' >> fstab
echo 'proc /proc proc defaults 0 0' >> fstab

# Modify rc.local final init script to add additional ephemeral drive and mount scratch there
if [ "$4" != "none" ]; then
cd /etc/rc.d
echo ' ' >>rc.local
echo '# Create additional Scratch directory' >>rc.local
echo 'mkdir -p $4' >>rc.local
echo ' ' >>rc.local
echo '# Mount scratch as XFS' >>rc.local
if [ "$1" == "hvm" ]; then
echo 'mkfs.xfs -f /dev/xvdb' >>rc.local
if [ "$2" != "t2.micro" ] && [ "$2" != "t2.small" ] && [ "$2" != "t2.medium" ]; then
echo 'mount /dev/xvdb $4' >>rc.local
fi
else
echo 'mkfs.xfs -f /dev/xvdf' >>rc.local

```

```

if [ "$2" != "t2.micro" ] && [ "$2" != "t2.small" ] && [ "$2" != "t2.medium" ]; then
echo 'mount /dev/xvdf '$4'' >>rc.local
fi
fi
fi

# Modify sshd_config
cd /etc/ssh
rm -f sshd_config
echo 'SyslogFacility AUTHPRIV' >> sshd_config
echo 'RSAAuthentication no' >> sshd_config
echo 'PubkeyAuthentication yes' >> sshd_config
echo 'AuthorizedKeysFile .ssh/authorized_keys' >> sshd_config
echo 'PasswordAuthentication yes' >> sshd_config
echo 'KerberosAuthentication no' >> sshd_config
echo 'KerberosOrLocalPasswd no' >> sshd_config
echo 'KerberosTicketCleanup no' >> sshd_config
echo 'GSSAPIAuthentication no' >> sshd_config
echo 'GSSAPICleanupCredentials no' >> sshd_config
echo 'UsePAM no' >> sshd_config
echo 'AllowTcpForwarding yes' >> sshd_config
echo 'X11Forwarding yes' >> sshd_config
echo 'UseLogin no' >> sshd_config
echo 'UseDNS no' >> sshd_config

# Create ec2-get-ssh authentication script for public keypair
cd /
mv ec2-get-ssh /etc/init.d
/bin/chmod +x /etc/init.d/ec2-get-ssh

# Modify /sbin/chkconfig services
/sbin/chkconfig ec2-get-ssh on
/sbin/chkconfig rpcbind off
/sbin/chkconfig postfix off
# /sbin/chkconfig autofs off
/sbin/chkconfig vmcontext off
# Prevent 10 minute automatic vm shutdown for testing (**turn back on after testing**)
# /sbin/chkconfig glideinwms-pilot off

# Clear history
history -c

# exit chroot
exit
-----
-----
-----

```

```

File: puppetrepo-awsexport/files/opt/gcso/awsexport/Fermi_AWS_Resize.sh
#!/bin/bash
# This is a script to resize the converted image (accepts param $1=vmimage name $2="hvm") from a QCOW2 256G image
down to a 12G primary partiton required for the AWS HVM image upload or a 3G primary partiton for a AWS PV image
upload

# Remove previous raw image, if exists, and formats new raw image
cd /data
rm -f $1.raw

# Use guestfish to delete 2nd and 3rd partitions, if present, clean up count errors and resize primary partiton
content (about 8.5 minutes)
guestfish -a $1.qcow2tmp <<_EOF1_
run
part-del /dev/sda 2
part-del /dev/sda 3
_EOF1_
sleep 2
guestfish -a $1.qcow2tmp <<_EOF2_
run
e2fsck-f /dev/sda1
_EOF2_
sleep 2

if [ "$2" == "hvm" ]; then

    guestfish -a $1.qcow2tmp <<_EOF3_
    run
    resize2fs-size /dev/sda1 12G
_EOF3_
sleep 2
# Resize boot partiton takes about 3.5 minutes
qemu-img create -f raw $1.raw 12291M
virt-resize --resize /dev/sda1=12G $1.qcow2tmp $1.raw

else

    guestfish -a $1.qcow2tmp <<_EOF4_
    run
    resize2fs-size /dev/sda1 3G
_EOF4_
sleep 2
# Resize boot partiton takes about 3.5 minutes
qemu-img create -f raw $1.raw 3075M
virt-resize --resize /dev/sda1=3G $1.qcow2tmp $1.raw

fi

```

```

# Status of converted raw file
gemu-img info $1.raw
-----
-----

File: puppetrepo-awsexport/files/opt/gcso/awsexport/cloud.cfg
#Bare-bone cloud.cfg, add parameters as needed for FermiCloud

user: root

#If this is not explicitly false, cloud-init will change things so that root
#login via ssh is disabled. Set it false to allow root login via ssh keypair.

disable_root: false

#add additional cloud-init output logging

output: {all: '| tee -a /var/log/cloud-init-output.log'}

#Since cloud-init runs at multiple stages of boot, this needs to be set so
#it can log in all of them to /var/log/cloud-init.

syslog_fix_perms: null

#This is the piece that makes userdata work. You need this to have userdata
#scripts be run by cloud-init.

datasource_list: [Ec2]
datasource:
  Ec2:
    metadata_urls: ['http://169.254.169.254']

#modules that run early in boot

cloud_init_modules:
- bootcmd #for running commands during boot. Commands can be defined in cloud-config userdata.

#modules that run after boot

cloud_config_modules:
- runcmd #like bootcmd, but runs after boot. Use this instead of bootcmd for after boot processing.

#modules that run at some point after config is finished

cloud_final_modules:
- scripts-per-once #all of these run scripts at specific events. Like bootcmd, can be defined in cloud-config.

```

```

- scripts-per-boot
- scripts-per-instance
- scripts-user
- phone-home #if defined, can make a post request to a specified url when done booting
- final-message #if defined, can write a specified message to the log
- power-state-change #if defined, can trigger stuff based on power state changes

system_info:
  distro: rhel

# vim:syntax=yaml
-----
-----

File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/AWS Convert files Inventory.txt
FermiCloud AWS Convert documentation and script files:

1. AWS Convert files Inventory.txt -- This document inventory file
2. Fermi2AWS Readme 1st.txt -- Read this file 1st to get overview of conversion process
3. fermi2aws.pdf -- Diagram of script process flow
4. fermi_VM_Convert.txt -- Manual working notes to identify script process flow
5. keypair.pem -- AWS Private key pair user file (Retrieved from AWS EC2 Console under key pairs - when created
it will ask you to save, save it to this directory)
6. Convert.py -- Python script to run conversion (you can modify log output location otherwise will default to
this directory)
7. ec2-get-ssh -- bash script to setup ec2 ssh authentication (loads public key pair for user on server)
8. cloud.cfg -- Cloud Init package bare-bone config file
9. Fermi_AWS_Modifications.sh -- bash script to perform aws conversion modifications to fermi golden image
10. Fermi_AWS_Resize.sh -- bash script to perform resizing of 256G qcw2 image to 12G for HVM and 3G for PV
images
11. Fermi_AWS_CLI_Setup.sh -- bash script to load ec2 api tools, setup ec2 environment variables and expect
package
12. Fermi_AWS_Import.sh -- bash script to perform import of raw image and ec2 steps to create instances, ami,
snapshot and volumes used in conversion process

```

13. aws_image_convert_hvm.log -- example output log file of actual run for HVM
14. aws_image_convert_pv.log -- example output log file of actual run for PV
15. HVM Run.txt -- capture of terminal console during HVM script run
16. PV Run.txt -- capture of terminal console during PV script run
17. create_samplemime.txt -- example userdata (metadata) that creates a multipart mime txt file for cloud init
18. samplemime.txt -- example userdata (metadata) that executes on AWS vm boot up
19. Puppet Procedure to run Fermi2AWS Export.pdf -- Puppet procedures to run cron jobs

```
-----
-----
-----
```

File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/AWS2Fermi PreProduction/AWS2Fermi Manual Procedures.txt
Importing an Amazon Web Services (AWS) PV VM into FermiCloud

Suppose to have two running VMs on Amazon EC2, both of them in the same region (e.g. us-west-2c): one of these (e.g. i-a966649d) is an Amazon Linux image, with has got all the EC2 tools (CLI and AMI) by default; the other (e.g. i-f61f83c2) is the Red Hat instance that you want to import on FermiCloud. The EC2 tools are assumed well configured. This means that you have set up the environment with the variables AWS_ACCESS_KEY, AWS_SECRET_KEY and EC2_URL.

To begin you have to connect to the first one, using for example SSH. So, from your computer's terminal, run:

```
$ ssh -i foo gcso.pem root@nn.nn.nn.nn
```

Then obtain root access using:

```
$ sudo su
```

Now you can get a raw image of the second VM with the following instructions:

```
# cd /tmp/
# ec2-stop-instances i-f61f83c2
# ec2-detach-volume vol-5f6f3e36
# ec2-attach-volume vol-5f6f3e36 -i i-a966649d -d /dev/sda2

# dd if=/dev/xvdd | cp --sparse=always /dev/stdin gcso_pv.img

# chmod 644 gcso_pv.img
# exit
$ exit
```

Then, you will find your PC's terminal. So get the image with the command:

```
$ rsync -S -z -v --progress -e "ssh -i gcso.pem" root@54.191.161.121:/tmp/gcso_pv.img .
```

After that you can begin the conversion procedure. So:

```
$ su
# dd if=/dev/zero of=newimage.img bs=1M seek=10240 count=0
# losetup -fv newimage.img
```

Take now the path of the loop device you have created (e.g. /dev/loop1) and use it in the next instruction:

```
cdisk /dev/loop1
```

Go ahead and create a New Primary Bootable partition. Then Write the partition table and Quit. Then find the partition beginning, ending, number of blocks, number of cylinders, and block size using:

```
fdisk -l -u /dev/loop1
```

In our example we have:

```
Disk /dev/loop1: 10.7 GB, 10737418240 bytes
255 heads, 63 sectors/track, 1305 cylinders, total 20971520 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000
```

| Device | Boot | Start | End | Blocks | Id | System |
|--------------|------|-------|----------|-----------|----|--------|
| /dev/loop1p1 | * | 63 | 20971519 | 10485728+ | 83 | Linux |

Then create a new loop device (e.g. /dev/loop2), using the beginning of the image and the block size:

```
# losetup -fv -o $((512*63)) newimage.img
```

After that create the filesystem on the image. Use the end, the beginning and the block size again:

```
# mkfs.ext3 -b 4096 /dev/loop2 $(((20971519 - 63)*512/4096))
```

Now fill the image:

If you have boot problem with the VM, maybe you have to change your kernel with a KVM compatible one. To do that substitute all your /boot/ folder with a SLF /boot/ folder.

```
# mkdir /mnt/gcso
# mount gcso_pv.img /mnt/gcso/ -o loop
# mkdir /mnt/loop
# mkdir /mnt/loop/2
```

>>> Below procedures are for OpenNebula command line launch <<<<<<
Now the image is ready to be deployed in FermiCloud with OpenNebula 3.2. Let's use the EC2 interface:

```
$ rsync -S -z -v --progress -e ssh gcso_sl6_pv.qcow2 fcl316.fnal.gov:-  
$ ssh fcl316.fnal.gov  
$ export ONE_LOCATION=/opt/one32  
$ export PATH=$PATH:/opt/one32/bin  
$ export ONE_AUTH=/cloud/login/user/.one/one_x509  
$ econe-upload --url https://fermicloudpp.fnal.gov:8444/ --access-key /tmp/x509up_u502 --secret-key  
/tmp/x509up_u502 ~/gcso_sl6_pv.qcow2
```

The last command returns the AMI ID (e.g. ami-00000106). With this you can finally launch your instance:

```
$ econe-run-instances --url https://fermicloudpp.fnal.gov:8444/ --access-key /tmp/x509up_u502 --secret-key  
/tmp/x509up_u502 ami-00000106 --user-data abc -t ml.small.francesco
```

The template used is ml.small.francesco. To define it you need to edit /cloud/app/one/3.2/etc/econe.conf. It has to look like:

```
# Configuration for the image repository  
# IMAGE_DIR will store the Cloud images, check space!  
IMAGE_DIR=/var/lib/one/local/pub_scratch  
  
# OpenNebula sever contact information  
:one_xmlrpc: http://fermicloudpp.fnal.gov:2633/RPC2  
  
# Host and port where econe server will run  
:server: localhost  
:port: 4567  
  
# SSL proxy that serves the API (set if is being used)  
:ssl_server: https://fermicloudpp.fnal.gov:8444/  
  
# Authentication driver for incoming requests  
# ec2, default Access key and Secret key scheme  
# x509, for x509 certificates based authentication  
#:auth: ec2  
:auth: x509  
  
# Authentication driver to communicate with OpenNebula core  
# cipher, for symmetric cipher encryption of tokens  
# x509, for x509 certificate encryption of tokens  
#:core_auth: cipher  
:core_auth: x509  
  
# VM types allowed and its template file (inside templates directory)
```

```
:instance_types:  
  :ml.small:  
    :template: ml.small.erb  
  :ml.small.francesco:  
    :template: ml.small.francesco.erb
```

Then you have to edit the file /cloud/app/one/3.2/etc/ec2query_templates/ml.small.francesco.erb. For example you can use:

```
NAME = eco-vm  
CPU = 1  
VCPU = 1  
MEMORY = 1024  
OS = [ ARCH = x86_64 ]  
  
DISK = [ IMAGE_ID = <%= erb_vm_info[:img_id] %> ]  
  
DISK = [  
  type = swap,  
  size = 5120 ]  
  
#NIC=[NETWORK_ID=<EC2-VNET-ID>]  
NIC=[NETWORK_ID=0,  
  MODEL = virtio]  
  
#IMAGE_ID = <%= erb_vm_info[:ec2_img_id] %>  
#INSTANCE_TYPE = <%= erb_vm_info[:instance_type] %>  
  
FEATURES=[ acpi="yes" ]  
  
GRAPHICS = [  
  type = "vnc",  
  listen = "127.0.0.1",  
  port = "-1",  
  autoport = "yes",  
  keymap="en-us" ]  
  
#<% if erb_vm_info[:user_data] %>  
#CONTEXT = [  
  # EC2_USER_DATA="<%= erb_vm_info[:user_data] %>",  
  # TARGET="hdc"  
  # ]  
#<% end %>  
  
<% if erb_vm_info[:user_data] %>  
CONTEXT=[
```



```

# Add back fermi network resolv and hosts
cd /etc
cp -f /data/readd/resolv.conf .
cp -f /data/readd/hosts .
cp -f /data/readd/hosts.allow .
cp -f /data/readd/hosts.deny .

# Add back fermi specifics
cp -f /data/readd/yp.conf .
cp -f /data/readd/auto.master .
cp -f /data/readd//auto.misc .

cd /etc/init.d
cp -f /data/readd/.credentials .
# for production only
cp -f /data/readd/one-context.prod one-context

# Recreate symbolic links
ln -s /etc/init.d/.credentials /etc/rc.d/rc3.d/K99.credentials
# for production only
ln -s /etc/init.d/one-context /etc/rc.d/rc3.d/S09one-context
ln -s /etc/init.d/one-context /etc/rc.d/rc3.d/K84one-context

# copy vmcontext from pp for pp testing and prod for prod testing
cd /etc/init.d

# for preproduction only
cp -f /data/readd/vmcontext.pp vmcontext

# for production only
cp -f /data/readd/vmcontext.prod vmcontext

# Delete ifcfg-eth0
cd /etc/sysconfig/network-scripts
rm -f ifcfg-eth0

# Modify sshd_config
cd /etc/ssh
cp -f /data/readd/sshd_config .

# modify grub.conf using uuid

cd /boot/grub
cp -f /data/readd/grub.conf .

# default=0
# timeout=0

```

```

# title Scientific Linux Fermi (2.6.32-431.23.3.el6.x86_64)
# root (hd0,0)
# kernel /boot/vmlinuz-2.6.32-431.23.3.el6.x86_64 ro root=UUID=17898259-6979-4bb3-9d73-26ae917e8ed9 rd_NO_LUKS
rd_NO_LVM LANG=en_US.UTF-8 rd_NO_MD SYSFONT=latarcyrheb-sun16 crashkernel=auto KEYBOARDTYPE=pc KEYTABLE=us
rd_NO_DM rhgb quiet edd=off
# initrd /boot/initramfs-2.6.32-431.23.3.el6.x86_64.img

# modify fstab using uuid
cd /etc
cp -f /data/readd/fstab .

# UUID=17898259-6979-4bb3-9d73-26ae917e8ed9 /          ext3      defaults    1 1
# tmpfs      /dev/shm     tmpfs defaults    0 0
# devpts     /dev/pts     devpts gid=5,mode=620  0 0
# sysfs      /sys         sysfs defaults    0 0
# proc/proc  proc         defaults    0 0

# Modify /sbin/chkconfig services
/sbin/chkconfig ec2-get-ssh off
/sbin/chkconfig rpcbind on
/sbin/chkconfig postfix on
# /sbin/chkconfig autofs off
/sbin/chkconfig vmcontext on
# Prevent 10 minute automatic vm shutdown for testing (**turn back on after testing**)
# /sbin/chkconfig glideinwms-pilot off

# Clear history
history -c

# exit chroot
exit

# Unmount image
cd /data
fusermount -u /mnt
-----
-----

File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/AWS2Fermi PreProduction/opennebtemplate.txt
Update image properties

BUS=virtio
DEV_PREFIX=hd
DRIVER=gcow2
-----

```

Update template properties (change IMAGE_ID= and TEMPLATE_ID= when creating new images under OpenNebula)

```
CONTEXT=[
  CTX_USER="$USER[TEMPLATE]",
  ETH0_DNS="$NETWORK[DNS, NETWORK=\"DynamicIP\"]",
  ETH0_GATEWAY="$NETWORK[GATEWAY, NETWORK=\"DynamicIP\"]",
  ETH0_IP="$NIC[IP, NETWORK=\"DynamicIP\"]",
  ETH0_MASK="$NETWORK[NETWORK_MASK, NETWORK=\"DynamicIP\"]",
  ETH0_NETWORK="$NETWORK[NETWORK_ADDRESS, NETWORK=\"DynamicIP\"]",
  FILES="/cloud/images/OpenNebula/scripts/one3.2/contextualization/init.sh
/cloud/images/OpenNebula/scripts/one3.2/contextualization/credentials.sh
/cloud/images/OpenNebula/scripts/one3.2/contextualization/kerberos.sh",
  GATEWAY="$NETWORK[GATEWAY, NETWORK=\"DynamicIP\"]",
  INIT_SCRIPTS="init.sh credentials.sh kerberos.sh",
  IP_PUBLIC="$NIC[IP, NETWORK=\"DynamicIP\"]",
  NETMASK="$NETWORK[NETWORK_MASK, NETWORK=\"DynamicIP\"]",
  NETWORK=YES,
  ROOT_PUBKEY=id_dsa.pub,
  TARGET=hdc,
  USERNAME=opennebula,
  USER_PUBKEY=id_dsa.pub ]
CPU=0.5
DISK=[
  IMAGE_ID=223,
  IMAGE_UNAME=oneadmin,
  TARGET=vda ]
DISK=[
  SIZE=2048,
  TARGET=vdb,
  TYPE=swap ]
FEATURES=[
  ACPI=yes ]
GRAPHICS=[
  AUTOPOST=yes,
  KEYMAP=en-us,
  LISTEN=127.0.0.1,
  PORT=-1,
  TYPE=vnc ]
MEMORY=2048
NAME="aws hvm export"
NIC=[
  MODEL=virtio,
  NETWORK=DynamicIP,
  NETWORK_UNAME=oneadmin ]
NPTYPE=NPERNLM
OS=[
  ARCH=x86_64,
```

```
BOOT=hd ]
RANK=FREEMEMORY
RAW=[
  DATA="
      <devices>
      <serial type='pty'>
        <target port='0' />
      </serial>
      <console type='pty'>
        <target type='serial' port='0' />
      </console>
    </devices>",
  TYPE=kvm ]
REQUIREMENTS="HYPERVISOR=\"kvm\" & HOSTNAME=\"fcl010.fnal.gov\""
TEMPLATE_ID=139
VCPU=1
-----
-----
-----
```

File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/Fermi2AWS Readme 1st.txt
Date: August 14, 2014

Steve/Gerard/Nick,

I have successfully implemented a conversion of a Fermicloud Golden image to a launched AWS Para virtual image using a python script with several subprocess bash scripts and puppet modules. The entire process takes about 60 to 84 minutes for a PV and HVM conversion. The actual processing time is less than that, but asynchronous conditional sleeps on the amazon side were implemented to make sure those ec2 steps completed before going on to the next step.

If you would like to test this script, you can run the puppet apply procedures. (See 'Puppet Procedure to run Fermi2AWS Export.pdf'). Change the parameters as described in that document, prior to running the script.

****Note:** the following 4 steps are a pre-requisite prior to running this puppet module:

1. kinit username
2. Obtain a fermicloud worker vm instance from the OpenNebula Fermicloud. This is used as a interim work area for the conversions. Identify the 3 digit assigned number to the worker vm as in fermicloud(nnn).fnal.gov.
3. This script also requires 5 parameters and files to be obtained ahead of time from the AWS Console. They are:
 - <aws owner security pem name> (keypair_name.pem) (Download this file from the AWS console to your terminal root directory. *Important* You must run 'chmod 400 keypair_name.pem' to allow a ssh session to connect to a aws instance.)
 - <aws instance id of the HVM worker image> (use 'hvm' here to create AWS HVM worker before creating AWS PV's, Otherwise view ec2 hvm under aws ec2 console and get i-xxxxxxx for hvm worker)

```
- <aws owner keypair name> (create in aws console ec2-key pairs - create key pair)
- <aws key> (create in aws console IAM Users - Security Credentials tab and download)
- <aws secret key> (create in aws console IAM Users - Security Credentials tab and download)
(all of these are obtained from the AWS console under EC2 instances and the IAM Users-Security Credentials tab
prior to running this script).
```

4. Run the puppet procedure with 15 positional parameters and adjust your crontab start time (as documented in 'Puppet Procedure to run Fermi2AWS Export.pdf').

A log file (aws_image_convert.log) is created for each run in the /opt/gcso/awsexport directory for review. The Python module can be modified to point that log file to another writeable directory that GCSO's monitoring system runs from, if you wish to have it monitored there.

To support the use of userdata (metadata), to install 3rd party packages, you will need to create a multipart mime txt file to be used in advanced userdata when launching a AWS instance from a AMI. (see create_samplemime.txt and samplemime.txt files)

```
Kirk
-----
-----
-----
```

```
File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/HVM Run.txt
(Run Fermi's Kerberos Initialize before running this script: kinit userid).
This script takes twelve (12) arguments: <script dir location>, <fermicloud
worker vm number>, <fermicloud vm name>, <fermicloud vm owner>, <aws image
name>, <aws image owner>, <aws instance type>, <aws key>, <aws secret key>,
<aws pem name>, <aws worker instance id>, and <aws owner keypair name>. A
pre-requisite is to have a Fermicloud worker vm setup ahead of time
(root@fermicloudnfn.fnl.gov) with a /data directory created to store files.
The script will accept a VM image that is pre-loaded on your
username@fermicloud.fnl.gov root directory, (it must have a qcw2 extension),
and will make a worker copy of the image ready for AWS conversion, then
convert the image (fermi cleanse and resize) and import to AWS by the
specified <aws image name>. The script requires 5 parameters to be obtained
ahead of time from the AWS Console. They are: <aws owner security pem name>
<aws instance id of the HVM worker image, use 'hvm' here to create HVM worker
before creating PV's.> <aws owner keypair name> <aws key> and <aws secret key>
which are obtained from the AWS console under EC2 instances and the IAM
security tab prior to running this script. For example, to convert a
Fermicloud VM named gcso_slf6 with owner oneadmin, and to create a AWS image
named SLF6Vanilla, with owner oneadmin, and a instance type of m3.medium and
with a worker vm of fermicloud103.fnl.gov you would run the following script:
**
./Convert.py /Users/terminalroot 103 gcso_slf6 oneadmin SLF6Vanilla
oneadmin m3.medium awskey awssecretkey awsoowner.pem i-hvminstanceid
aws_owner_keypair_name
```

```
**
...Available.AWS.Instance.Types.listed.below...
Type.....vCPU..ECU..Memory.(GiB)..Instance Storage (GB)..Cost per hour
General.Purpose.--Current.Generation
t2.micro...1...Variable...1...EBSONly.....$0.013 per Hour
t2.small...1...Variable...2...EBSONly.....$0.026 per Hour
t2.medium...2...Variable...4...EBSONly.....$0.052 per Hour
m3.medium...1.....3.....3.75...1.x.4.SSD...$0.070 per Hour
m3.large...2.....6.5.....7.5...1.x.32.SSD...$0.140 per Hour
m3.xlarge...4.....13.....15...2.x.40.SSD...$0.280 per Hour
m3.2xlarge...8.....26.....30...2.x.80.SSD...$0.560 per Hour
Compute.Optimized.--Current.Generation
c3.large...2.....7.....3.75...2.x.16.SSD...$0.105 per Hour
c3.xlarge...4.....14.....7.5...2.x.40.SSD...$0.210 per Hour
c3.2xlarge...8.....28.....15...2.x.80.SSD...$0.420 per Hour
c3.4xlarge...16...55.....30...2.x.160.SSD...$0.840 per Hour
c3.8xlarge...32...108.....60...2.x.320.SSD...$1.680 per Hour
GPU.Instances.--Current.Generation
g2.2xlarge...8.....26.....15.....60.SSD...$0.650 per Hour
Memory.Optimized.--Current.Generation
r3.large...2.....6.5.....15...1.x.32.SSD...$0.175 per Hour
r3.xlarge...4.....13.....30.5...1.x.80.SSD...$0.350 per Hour
r3.2xlarge...8.....26.....61...1.x.160.SSD...$0.700 per Hour
r3.4xlarge...16...52.....122...1.x.320.SSD...$1.400 per Hour
r3.8xlarge...32...104.....244...2.x.320.SSD...$2.800 per Hour
Storage.Optimized.--Current.Generation
i2.xlarge...4.....14.....30.5...1.x.800.SSD...$0.853 per Hour
i2.2xlarge...8.....27.....61...2.x.800.SSD...$1.705 per Hour
i2.4xlarge...16...53.....122...4.x.800.SSD...$3.410 per Hour
i2.8xlarge...32...104.....244...8.x.800.SSD...$6.820 per Hour
hs1.8xlarge...16...35.....117...24.x.2048.....$4.600 per Hour
```

```
Arguments supplied:
('location of all scripts->', '/Users/kshallcross')
('fermicloud work vm number->', '103')
('fermicloud vm name->', 'gcso_sl6')
('fermicloud vm owner->', 'kirkshal')
('aws image name->', 'GCSO_SL6_HVM')
('aws image owner->', 'kirkshal')
('aws instance type->', 'm3.medium')
('aws key->', 'xxxxx')
('aws secret key->', 'xxxxxx')
('aws owner pem name->', 'khs-fermi.pem')
('aws owner keypair name->', 'khs-fermi')
('aws worker instance id from aws console->', 'hvm')
Starting AWS VM conversion. This job can take up to 52 minutes for PV and 77 for HVM...
Copying the Golden Fermicloud VM image takes under 1 minute...
Copying took 0.908 minutes.
```

```

Resizing the worker Fermicloud VM image takes up to 20 minutes for PV and 10 for HVM...
Resizing took 9.787 minutes.
Converting the worker Fermicloud VM image takes over 18 minutes...
Converting took 19.033 minutes.
Importing the raw image to AWS takes up to 13 minutes for PV and 48 for HVM...
Importing took 44.233 minutes.
Completed AWS VM conversion.
Job took 73.961 minutes. Thank you.
-----
-----

```

```

File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/PV Run.txt
(Run Fermi's Kerberos Initialize before running this script: kinit userid).
This script takes twelve (12) arguments: <script dir location>, <fermicloud
worker vm number>, <fermicloud vm name>, <fermicloud vm owner>, <aws image
name>, <aws image owner>, <aws instance type>, <aws key>, <aws secret key>,
<aws pem name>, <aws worker instance id>, and <aws owner keypair name>. A
pre-requisite is to have a Fermicloud worker vm setup ahead of time
(root@fermicloudnfn.gov) with a /data directory created to store files.
The script will accept a VM image that is pre-loaded on your
username@fermicloud.fnl.gov root directory, (it must have a qcw2 extension),
and will make a worker copy of the image ready for AWS conversion, then
convert the image (fermi cleanse and resize) and import to AWS by the
specified <aws image name>. The script requires 5 parameters to be obtained
ahead of time from the AWS Console. They are: <aws owner security pem name>
<aws instance id of the HVM worker image, use 'hvm' here to create HVM worker
before creating PV's.> <aws owner keypair name> <aws key> and <aws secret key>
which are obtained from the AWS console under EC2 instances and the IAM
security tab prior to running this script. For example, to convert a
Fermicloud VM named gcso_slf6 with owner oneadmin, and to create a AWS image
named SLF6Vanilla, with owner oneadmin, and a instance type of m3.medium and
with a worker vm of fermicloud103.fnl.gov you would run the following script:
**

```

```

./Convert.py /Users/terminalroot 103 gcso_slf6 oneadmin SLF6Vanilla
oneadmin m3.medium awskey awssecretkey awsowner.pem i-hvminstanceid
aws_owner_keypair_name
**

```

```

...Available.AWS.Instance.Types.listed.below...
Type.....vCPU..ECU..Memory.(GiB)..Instance Storage (GB)..Cost per hour
General.Purpose.--Current.Generation
t2.micro....1....Variable....1....EBSOnly.....$0.013 per Hour
t2.small....1....Variable....2....EBSOnly.....$0.026 per Hour
t2.medium...2....Variable....4....EBSOnly.....$0.052 per Hour
m3.medium...1.....3.....3.75...1.x.4.SSD...$0.070 per Hour
m3.large...2.....6.5.....7.5...1.x.32.SSD...$0.140 per Hour
m3.xlarge...4.....13.....15....2.x.40.SSD...$0.280 per Hour

```

```

m3.2xlarge..8.....26.....30....2.x.80.SSD...$0.560 per Hour
Compute.Optimized.--Current.Generation
c3.large...2.....7.....3.75...2.x.16.SSD...$0.105 per Hour
c3.xlarge...4.....14.....7.5...2.x.40.SSD...$0.210 per Hour
c3.2xlarge..8.....28.....15....2.x.80.SSD...$0.420 per Hour
c3.4xlarge..16.....55.....30....2.x.160.SSD...$0.840 per Hour
c3.8xlarge..32....108.....60....2.x.320.SSD...$1.680 per Hour
GPU.Instances.--Current.Generation
g2.2xlarge..8.....26.....15.....60.SSD...$0.650 per Hour
Memory.Optimized.--Current.Generation
r3.large...2.....6.5.....15....1.x.32.SSD...$0.175 per Hour
r3.xlarge...4.....13.....30.5...1.x.80.SSD...$0.350 per Hour
r3.2xlarge..8.....26.....61....1.x.160.SSD...$0.700 per Hour
r3.4xlarge..16....52.....122...1.x.320.SSD...$1.400 per Hour
r3.8xlarge..32....104.....244...2.x.320.SSD...$2.800 per Hour
Storage.Optimized.--Current.Generation
i2.xlarge...4.....14.....30.5...1.x.800.SSD...$0.853 per Hour
i2.2xlarge..8.....27.....61....2.x.800.SSD...$1.705 per Hour
i2.4xlarge..16....53.....122...4.x.800.SSD...$3.410 per Hour
i2.8xlarge..32....104.....244...8.x.800.SSD...$6.820 per Hour
hs1.8xlarge.16....35.....117...24.x.2048....$4.600 per Hour

```

```

Arguments supplied:
('location of all scripts->', '/Users/kshallcross')
('fermicloud work vm number->', '103')
('fermicloud vm name->', 'gcso_sl6')
('fermicloud vm owner->', 'kirkshal')
('aws image name->', 'GCSO_SL6_PV')
('aws image owner->', 'kirkshal')
('aws instance type->', 'm3.medium')
('aws key->', 'xxx')
('aws secret key->', 'xxxxx')
('aws owner pem name->', 'khs-fermi.pem')
('aws owner keypair name->', 'khs-fermi')
('aws worker instance id from aws console->', 'i-94fe889f')
Starting AWS VM conversion. This job can take up to 53 minutes for PV and 75 for HVM...
Copying the Golden Fermicloud VM image takes under 1 minute...
Copying took 0.906 minutes.
Resizing the worker Fermicloud VM image takes up to 20 minutes for PV and 10 for HVM...
Resizing took 19.422 minutes.
Converting the worker Fermicloud VM image takes over 19 minutes...
Converting took 20.174 minutes.
Importing the raw image to AWS takes up to 13 minutes for PV and 45 for HVM...
Importing took 12.136 minutes.
Completed AWS VM conversion.
Job took 52.638 minutes. Thank you.
-----
-----

```

```
-----
File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/aws_image_convert_hvm.log
Aug 02 14:52:36 INFO      Begin script run
Aug 02 14:52:36 INFO      Start: Copying the selected Fermicloud VM image. Function copy_to_image_location.
Aug 02 14:53:31 INFO      Stop: Completed Copying the selected Fermicloud VM image. Function
copy_to_image_location.
Aug 02 14:53:31 INFO      Start: Resizing the worker Fermicloud VM image. Function resize_image.
*stdin*:2: libguestfs: error: e2fsck_f: /dev/sdal: 94023/786432 files (0.3% non-contiguous), 550839/3145728
blocks
Formatting 'gcso_sl6.raw', fmt=raw size=12888047616
Examining gcso_sl6.qcow2tmp ...
*****
```

Summary of changes:

/dev/sdal: This partition will be left alone.

There is a surplus of 864.0K. The surplus space is not large enough for an extra partition to be created and so it will just be ignored.

Setting up initial partition table on gcso_sl6.raw ...
Copying /dev/sdal ...

Resize operation completed with no errors. Before deleting the old disk, carefully check that the resized disk boots and works correctly.

image: gcso_sl6.raw
file format: raw
virtual size: 12G (12888047616 bytes)
disk size: 12G

Aug 02 15:03:18 INFO Stop: Completed Resizing the worker Fermicloud VM image. Function resize_image.

Aug 02 15:03:18 INFO Start: Converting the worker Fermicloud VM image. Function convert_image.

warning: /var/tmp/rpm-tmp.Udfdjd: Header V3 RSA/SHA256 Signature, key ID 0608b895: NOKEY

Retrieving http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm

Preparing...

epel-release

Loaded plugins: security

Setting up Install Process

Resolving Dependencies

--> Running transaction check

----> Package cloud-init.noarch 0:0.7.4-2.el6 will be installed

--> Processing Dependency: python-boto >= 2.6.0 for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: python-requests for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: python-prettytable for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: python-jsonpatch for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: python-configobj for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: python-cheetah for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: python-argparse for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: policycoreutils-python for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: libselinux-python for package: cloud-init-0.7.4-2.el6.noarch

--> Processing Dependency: PyYAML for package: cloud-init-0.7.4-2.el6.noarch

--> Running transaction check

----> Package PyYAML.x86_64 0:3.10-3.el6 will be installed

--> Processing Dependency: libyaml-0.so.2()(64bit) for package: PyYAML-3.10-3.el6.x86_64

----> Package libselinux-python.x86_64 0:2.0.94-5.3.el6 will be installed

----> Package policycoreutils-python.x86_64 0:2.0.83-19.30.el6 will be installed

--> Processing Dependency: libsemanage-python >= 2.0.43-4 for package: policycoreutils-python-2.0.83-

19.30.el6.x86_64

--> Processing Dependency: audit-libs-python >= 1.4.2-1 for package: policycoreutils-python-2.0.83-

19.30.el6.x86_64

--> Processing Dependency: setools-libs-python for package: policycoreutils-python-2.0.83-19.30.el6.x86_64

--> Processing Dependency: libgroup for package: policycoreutils-python-2.0.83-19.30.el6.x86_64

----> Package python-argparse.noarch 0:1.2.1-2.el6 will be installed

----> Package python-boto.noarch 0:2.27.0-1.el6 will be installed

----> Package python-cheetah.x86_64 0:2.4.1-1.el6 will be installed

--> Processing Dependency: python-pygments for package: python-cheetah-2.4.1-1.el6.x86_64

--> Processing Dependency: python-markdown for package: python-cheetah-2.4.1-1.el6.x86_64

----> Package python-configobj.noarch 0:4.6.0-3.el6 will be installed

----> Package python-jsonpatch.noarch 0:1.2-2.el6 will be installed

--> Processing Dependency: python-jsonpointer for package: python-jsonpatch-1.2-2.el6.noarch

----> Package python-prettytable.noarch 0:0.7.2-1.el6 will be installed

----> Package python-requests.noarch 0:1.1.0-4.el6 will be installed

--> Processing Dependency: python-urllib3 for package: python-requests-1.1.0-4.el6.noarch

--> Processing Dependency: python-ordereddict for package: python-requests-1.1.0-4.el6.noarch

--> Processing Dependency: python-charadet for package: python-requests-1.1.0-4.el6.noarch

--> Running transaction check

----> Package audit-libs-python.x86_64 0:2.2-2.el6 will be installed

----> Package libgroup.x86_64 0:0.37-7.el6 will be installed

----> Package libsemanage-python.x86_64 0:2.0.43-4.2.el6 will be installed

----> Package libyaml.x86_64 0:0.1.6-1.el6 will be installed

----> Package python-charadet.noarch 0:2.0.1-1.el6 will be installed

----> Package python-jsonpointer.noarch 0:1.0-3.el6 will be installed

----> Package python-markdown.noarch 0:2.0.1-3.1.el6 will be installed

----> Package python-ordereddict.noarch 0:1.1-2.el6 will be installed

----> Package python-pygments.noarch 0:1.1.1-1.el6 will be installed

--> Processing Dependency: python-setuptools for package: python-pygments-1.1.1-1.el6.noarch

----> Package python-urllib3.noarch 0:1.5-7.el6 will be installed

--> Processing Dependency: python-six for package: python-urllib3-1.5-7.el6.noarch

--> Processing Dependency: python-backports-ssl_match_hostname for package: python-urllib3-1.5-7.el6.noarch

----> Package setools-libs-python.x86_64 0:3.3.7-4.el6 will be installed

--> Processing Dependency: setools-libs = 3.3.7-4.el6 for package: setools-libs-python-3.3.7-4.el6.x86_64

--> Processing Dependency: libpol.so.1(VERS 1.3)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64

--> Processing Dependency: libpoldiff.so.1(VERS 1.3)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64

--> Processing Dependency: libapol.so.4(VERS 4.1)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64

--> Processing Dependency: libpol.so.1(VERS 1.2)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64

```

--> Processing Dependency: libqpol.so.1(VERS 1.4)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libseaudit.so.4(VERS 4.2)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libpoldiff.so.1(VERS 1.2)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libsefs.so.4(VERS 4.0)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libapol.so.4(VERS 4.0)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libseaudit.so.4(VERS 4.1)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libseaudit.so.4()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libqpol.so.1()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libapol.so.4()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libpoldiff.so.1()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Processing Dependency: libsefs.so.4()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
--> Running transaction check
----> Package python-backports-ssl_match_hostname.noarch 0:3.4.0.2-1.el6 will be installed
--> Processing Dependency: python-backports for package: python-backports-ssl_match_hostname-3.4.0.2-1.el6.noarch
----> Package python-setuptools.noarch 0:0.6.10-3.el6 will be installed
----> Package python-six.noarch 0:1.6.1-1.el6 will be installed
----> Package setools-libs.x86_64 0:3.3.7-4.el6 will be installed
--> Running transaction check
--> Package python-backports.x86_64 0:1.0-3.el6 will be installed
--> Finished Dependency Resolution

```

Dependencies Resolved

| Package | Arch | Version | Repository | Size |
|-------------------------------------|--------|------------------|------------|-------|
| Installing: | | | | |
| cloud-init | noarch | 0.7.4-2.el6 | epel | 487 k |
| Installing for dependencies: | | | | |
| PyYAML | x86_64 | 3.10-3.el6 | epel | 157 k |
| audit-libs-python | x86_64 | 2.2-2.el6 | slf | 58 k |
| libcgroup | x86_64 | 0.37-7.el6 | slf | 110 k |
| libselinux-python | x86_64 | 2.0.94-5.3.el6 | slf | 201 k |
| libsemanage-python | x86_64 | 2.0.43-4.2.el6 | slf | 80 k |
| libyaml | x86_64 | 0.1.6-1.el6 | epel | 52 k |
| policycoreutils-python | x86_64 | 2.0.83-19.30.el6 | slf | 341 k |
| python-argparse | noarch | 1.2.1-2.el6 | epel | 48 k |
| python-backports | x86_64 | 1.0-3.el6 | epel | 5.3 k |
| python-backports-ssl_match_hostname | noarch | 3.4.0.2-1.el6 | epel | 12 k |
| python-boto | noarch | 2.27.0-1.el6 | epel | 1.6 M |
| python-charDET | noarch | 2.0.1-1.el6 | epel | 225 k |
| python-cheetah | x86_64 | 2.4.1-1.el6 | slf | 364 k |
| python-configobj | noarch | 4.6.0-3.el6 | slf | 181 k |
| python-jsonpatch | noarch | 1.2-2.el6 | epel | 14 k |
| python-jsonpointer | noarch | 1.0-3.el6 | epel | 9.2 k |
| python-markdown | noarch | 2.0.1-3.1.el6 | slf | 117 k |
| python-ordereddict | noarch | 1.1-2.el6 | epel | 7.6 k |

| | | | | |
|---------------------|--------|--------------|------|-------|
| python-prettytable | noarch | 0.7.2-1.el6 | epel | 37 k |
| python-pygments | noarch | 1.1.1-1.el6 | slf | 561 k |
| python-requests | noarch | 1.1.0-4.el6 | epel | 71 k |
| python-setuptools | noarch | 0.6.10-3.el6 | slf | 335 k |
| python-six | noarch | 1.6.1-1.el6 | epel | 25 k |
| python-urllib3 | noarch | 1.5-7.el6 | epel | 41 k |
| setools-libs | x86_64 | 3.3.7-4.el6 | slf | 399 k |
| setools-libs-python | x86_64 | 3.3.7-4.el6 | slf | 221 k |

Transaction Summary

Install 27 Package(s)

Total download size: 5.6 M

Installed size: 25 M

Downloading Packages:

```

-----
Total                               562 kB/s | 5.6 MB    00:10
Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
warning: rpmts_HdrFromFdno: Header V3 RSA/SHA256 Signature, key ID 0608b895: NOKEY
Importing GPG key 0x0608B895:
  Userid : EPEL (6) <epel@fedoraproject.org>
  Package: slf-release-6.4-1.x86_64 (@anaconda-ScientificLinuxFermi-201304091346.x86_64/6)
  From   : /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
Warning: RPMDB altered outside of yum.

```

| | |
|---|------|
| Installing : python-ordereddict-1.1-2.el6.noarch | 1/27 |
| Installing : libselinux-python-2.0.94-5.3.el6.x86_64 | 2/27 |
| Installing : python-argparse-1.2.1-2.el6.noarch | 3/27 |
| Installing : libsemanage-python-2.0.43-4.2.el6.x86_64 | 4/27 |
| Installing : python-boto-2.27.0-1.el6.noarch | 5/27 |
| Installing : python-six-1.6.1-1.el6.noarch | 6/27 |
| Installing : python-configobj-4.6.0-3.el6.noarch | 7/27 |
| Installing : python-setuptools-0.6.10-3.el6.noarch | 8/27 |
| Installing : python-pygments-1.1.1-1.el6.noarch | 9/27 |

| | |
|---|-------|
| Installing : python-markdown-2.0.1-3.1.el6.noarch | 10/27 |
| Installing : python-cheetah-2.4.1-1.el6.x86_64 | 11/27 |
| Installing : python-chardet-2.0.1-1.el6.noarch | 12/27 |
| Installing : python-prettytable-0.7.2-1.el6.noarch | 13/27 |
| Installing : libcgrou-0.37-7.el6.x86_64 | 14/27 |
| Installing : setools-libs-3.3.7-4.el6.x86_64 | 15/27 |
| Installing : setools-libs-python-3.3.7-4.el6.x86_64 | 16/27 |
| Installing : libyaml-0.1.6-1.el6.x86_64 | 17/27 |
| Installing : PyYAML-3.10-3.el6.x86_64 | 18/27 |
| Installing : audit-libs-python-2.2-2.el6.x86_64 | 19/27 |
| Installing : policycoreutils-python-2.0.83-19.30.el6.x86_64 | 20/27 |
| Installing : python-backports-1.0-3.el6.x86_64 | 21/27 |
| Installing : python-backports-ssl_match_hostname-3.4.0.2-1.el6.noarch | 22/27 |
| Installing : python-urllib3-1.5-7.el6.noarch | 23/27 |
| Installing : python-requests-1.1.0-4.el6.noarch | 24/27 |
| Installing : python-jsonpointer-1.0-3.el6.noarch | 25/27 |
| Installing : python-jsonpatch-1.2-2.el6.noarch | 26/27 |
| Installing : cloud-init-0.7.4-2.el6.noarch | 27/27 |
| Verifying : python-jsonpointer-1.0-3.el6.noarch | 1/27 |
| Verifying : python-backports-1.0-3.el6.x86_64 | 2/27 |
| Verifying : libselinux-python-2.0.94-5.3.el6.x86_64 | 3/27 |
| Verifying : audit-libs-python-2.2-2.el6.x86_64 | 4/27 |
| Verifying : setools-libs-python-3.3.7-4.el6.x86_64 | 5/27 |
| Verifying : libyaml-0.1.6-1.el6.x86_64 | 6/27 |

| | |
|--|-------|
| Verifying : setools-libs-3.3.7-4.el6.x86_64 | 7/27 |
| Verifying : libcgrou-0.37-7.el6.x86_64 | 8/27 |
| Verifying : python-jsonpatch-1.2-2.el6.noarch | 9/27 |
| Verifying : python-pygments-1.1.1-1.el6.noarch | 10/27 |
| Verifying : python-prettytable-0.7.2-1.el6.noarch | 11/27 |
| Verifying : python-backports-ssl_match_hostname-3.4.0.2-1.el6.noarch | 12/27 |
| Verifying : python-urllib3-1.5-7.el6.noarch | 13/27 |
| Verifying : python-chardet-2.0.1-1.el6.noarch | 14/27 |
| Verifying : python-requests-1.1.0-4.el6.noarch | 15/27 |
| Verifying : python-cheetah-2.4.1-1.el6.x86_64 | 16/27 |
| Verifying : policycoreutils-python-2.0.83-19.30.el6.x86_64 | 17/27 |
| Verifying : cloud-init-0.7.4-2.el6.noarch | 18/27 |
| Verifying : python-markdown-2.0.1-3.1.el6.noarch | 19/27 |
| Verifying : python-setuptools-0.6.10-3.el6.noarch | 20/27 |
| Verifying : python-configobj-4.6.0-3.el6.noarch | 21/27 |
| Verifying : python-six-1.6.1-1.el6.noarch | 22/27 |
| Verifying : PyYAML-3.10-3.el6.x86_64 | 23/27 |
| Verifying : python-boto-2.27.0-1.el6.noarch | 24/27 |
| Verifying : python-ordereddict-1.1-2.el6.noarch | 25/27 |
| Verifying : libsemanage-python-2.0.43-4.2.el6.x86_64 | 26/27 |
| Verifying : python-argparse-1.2.1-2.el6.noarch | 27/27 |

Installed:
cloud-init.noarch 0:0.7.4-2.el6

Dependency Installed:
PyYAML.x86_64 0:3.10-3.el6
audit-libs-python.x86_64 0:2.2-2.el6

```

libcgroup.x86_64 0:0.37-7.el6
libselinux-python.x86_64 0:2.0.94-5.3.el6
libsemanage-python.x86_64 0:2.0.43-4.2.el6
libyaml.x86_64 0:0.1.6-1.el6
policycoreutils-python.x86_64 0:2.0.83-19.30.el6
python-argparse.noarch 0:1.2.1-2.el6
python-backports.x86_64 0:1.0-3.el6
python-backports-ssl_match_hostname.noarch 0:3.4.0.2-1.el6
python-boto.noarch 0:2.27.0-1.el6
python-charDET.noarch 0:2.0.1-1.el6
python-cheetah.x86_64 0:2.4.1-1.el6
python-configobj.noarch 0:4.6.0-3.el6
python-jsonpatch.noarch 0:1.2-2.el6
python-jsonpointer.noarch 0:1.0-3.el6
python-markdown.noarch 0:2.0.1-3.1.el6
python-orderdict.noarch 0:1.1-2.el6
python-prettytable.noarch 0:0.7.2-1.el6
python-pygments.noarch 0:1.1.1-1.el6
python-requests.noarch 0:1.1.0-4.el6
python-setuptools.noarch 0:0.6.10-3.el6
python-six.noarch 0:1.6.1-1.el6
python-urllib3.noarch 0:1.5-7.el6
setools-libs.x86_64 0:3.3.7-4.el6
setools-libs-python.x86_64 0:3.3.7-4.el6

Complete!
Loaded plugins: security
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package cloud-utils.x86_64 0:0.27-10.el6 will be installed
--> Processing Dependency: qemu-img for package: cloud-utils-0.27-10.el6.x86_64
--> Processing Dependency: python-paramiko for package: cloud-utils-0.27-10.el6.x86_64
--> Processing Dependency: euca2ools for package: cloud-utils-0.27-10.el6.x86_64
--> Processing Dependency: cloud-utils-growpart for package: cloud-utils-0.27-10.el6.x86_64
--> Running transaction check
--> Package cloud-utils-growpart.x86_64 0:0.27-10.el6 will be installed
--> Package euca2ools.noarch 0:2.1.4-1.el6 will be installed
--> Processing Dependency: m2crypto for package: euca2ools-2.1.4-1.el6.noarch
--> Package python-paramiko.noarch 0:1.7.5-2.1.el6 will be installed
--> Processing Dependency: python-crypto >= 1.9 for package: python-paramiko-1.7.5-2.1.el6.noarch
--> Package qemu-img.x86_64 2:0.12.1.2-2.415.el6_5.10 will be installed
--> Processing Dependency: libusbredirparser.so.1()(64bit) for package: 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64
--> Processing Dependency: libgfxdr.so.0()(64bit) for package: 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64
--> Processing Dependency: libgfrpc.so.0()(64bit) for package: 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64
--> Processing Dependency: libgfapi.so.0()(64bit) for package: 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64
--> Running transaction check
--> Package glusterfs-api.x86_64 0:3.4.0.36rhs-1.el6 will be installed

```

```

--> Package glusterfs-libs.x86_64 0:3.4.0.36rhs-1.el6 will be installed
--> Package m2crypto.x86_64 0:0.20.2-9.el6 will be installed
--> Package python-crypto.x86_64 0:2.0.1-22.el6 will be installed
--> Package usbredir.x86_64 0:0.5.1-1.el6 will be installed
--> Finished Dependency Resolution

```

Dependencies Resolved

| Package | Arch | Version | Repository | Size |
|------------------------------|--------|---------------------------|--------------|-------|
| Installing: | | | | |
| cloud-utils | x86_64 | 0.27-10.el6 | epel | 43 k |
| Installing for dependencies: | | | | |
| cloud-utils-growpart | x86_64 | 0.27-10.el6 | epel | 25 k |
| euca2ools | noarch | 2.1.4-1.el6 | epel | 326 k |
| glusterfs-api | x86_64 | 3.4.0.36rhs-1.el6 | slf-security | 44 k |
| glusterfs-libs | x86_64 | 3.4.0.36rhs-1.el6 | slf-security | 225 k |
| m2crypto | x86_64 | 0.20.2-9.el6 | slf | 470 k |
| python-crypto | x86_64 | 2.0.1-22.el6 | slf | 157 k |
| python-paramiko | noarch | 1.7.5-2.1.el6 | slf | 727 k |
| qemu-img | x86_64 | 2:0.12.1.2-2.415.el6_5.10 | slf-security | 595 k |
| usbredir | x86_64 | 0.5.1-1.el6 | slf | 39 k |

Transaction Summary

```
Install 10 Package(s)
```

```
Total download size: 2.6 M
Installed size: 13 M
Downloading Packages:
```

```
Total 589 kB/s | 2.6 MB 00:04
```

```
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
```

```

Installing : glusterfs-libs-3.4.0.36rhs-1.el6.x86_64 1/10
Installing : glusterfs-api-3.4.0.36rhs-1.el6.x86_64 2/10
Installing : usbredir-0.5.1-1.el6.x86_64 3/10
Installing : 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64 4/10
Installing : python-crypto-2.0.1-22.el6.x86_64 5/10

```

```

Installing : python-paramiko-1.7.5-2.1.el6.noarch           6/10
Installing : cloud-utils-growpart-0.27-10.el6.x86_64      7/10
Installing : m2crypto-0.20.2-9.el6.x86_64                8/10
Installing : euca2ools-2.1.4-1.el6.noarch                 9/10
Installing : cloud-utils-0.27-10.el6.x86_64              10/10
Verifying  : 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64   1/10
Verifying  : euca2ools-2.1.4-1.el6.noarch                 2/10
Verifying  : m2crypto-0.20.2-9.el6.x86_64                3/10
Verifying  : cloud-utils-growpart-0.27-10.el6.x86_64     4/10
Verifying  : glusterfs-libs-3.4.0.36rhs-1.el6.x86_64    5/10
Verifying  : python-crypto-2.0.1-22.el6.x86_64           6/10
Verifying  : python-paramiko-1.7.5-2.1.el6.noarch        7/10
Verifying  : cloud-utils-0.27-10.el6.x86_64              8/10
Verifying  : usbredir-0.5.1-1.el6.x86_64                 9/10
Verifying  : glusterfs-api-3.4.0.36rhs-1.el6.x86_64     10/10

```

```

Installed:
cloud-utils.x86_64 0:0.27-10.el6

```

```

Dependency Installed:
cloud-utils-growpart.x86_64 0:0.27-10.el6
euca2ools.noarch 0:2.1.4-1.el6
glusterfs-api.x86_64 0:3.4.0.36rhs-1.el6
glusterfs-libs.x86_64 0:3.4.0.36rhs-1.el6
m2crypto.x86_64 0:0.20.2-9.el6
python-crypto.x86_64 0:2.0.1-22.el6
python-paramiko.noarch 0:1.7.5-2.1.el6
qemu-img.x86_64 2:0.12.1.2-2.415.el6_5.10
usbredir.x86_64 0:0.5.1-1.el6

```

```

Complete!
Loaded plugins: security
Setting up Install Process
Examining /var/tmp/yum-root-bFavOG/python-backports-1.0-4.el6.x86_64.rpm: python-backports-1.0-4.el6.x86_64

```

```

Marking /var/tmp/yum-root-bFavOG/python-backports-1.0-4.el6.x86_64.rpm as an update to python-backports-1.0-3.el6.x86_64
Resolving Dependencies
--> Running transaction check
---> Package python-backports.x86_64 0:1.0-3.el6 will be updated
---> Package python-backports.x86_64 0:1.0-4.el6 will be an update
--> Finished Dependency Resolution

```

```
Dependencies Resolved
```

```

=====
Package      Arch    Version      Repository                               Size
=====
Updating:
python-backports x86_64 1.0-4.el6 /python-backports-1.0-4.el6.x86_64 318

```

```
Transaction Summary
```

```
=====
Upgrade      1 Package(s)

```

```

Total size: 318
Downloading Packages:
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction

```

```

Updating   : python-backports-1.0-4.el6.x86_64           1/2
Cleanup    : python-backports-1.0-3.el6.x86_64         2/2
Verifying  : python-backports-1.0-4.el6.x86_64         1/2
Verifying  : python-backports-1.0-3.el6.x86_64         2/2

```

```

Updated:
python-backports.x86_64 0:1.0-4.el6

```

```

Complete!
Aug 02 15:22:20 INFO      Stop: Completed Converting the worker Fermicloud VM image. Function convert_image.
Aug 02 15:22:20 INFO      Start: Importing the worker Fermicloud VM image. Function import_image.
Loaded plugins: refresh-packagekit, security
Setting up Install Process
Package 1:java-1.6.0-openjdk-1.6.0.0-6.1.13.4.el6_5.x86_64 already installed and latest version
Nothing to do
/usr/lib/jvm/jre-1.6.0-openjdk.x86_64
% Total    % Received % Xferd   Average Speed   Time    Time       Time  Current
           % Dload  % Upload   Total   Spent    Left     Speed

```



```
Copying /dev/sdal ...
Expanding /dev/sdal using the 'resize2fs' method ...
```

```
Resize operation completed with no errors. Before deleting the old
disk, carefully check that the resized disk boots and works correctly.
```

```
image: gcs0_sl6.raw
```

```
file format: raw
```

```
virtual size: 3.0G (3224371200 bytes)
```

```
disk size: 3.0G
```

```
Aug 02 16:35:19 INFO Stop: Completed Resizing the worker Fermicloud VM image. Function resize_image.
```

```
Aug 02 16:35:19 INFO Start: Converting the worker Fermicloud VM image. Function convert_image.
```

```
warning: /var/tmp/rpm-tmp.61H0MT: Header V3 RSA/SHA256 Signature, key ID 0608b895: NOKEY
```

```
Retrieving http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
```

```
Preparing...
```

```
epel-release
```

```
Loaded plugins: security
```

```
Setting up Install Process
```

```
Resolving Dependencies
```

```
--> Running transaction check
```

```
--> Package cloud-init.noarch 0:0.7.4-2.el6 will be installed
```

```
--> Processing Dependency: python-boto >= 2.6.0 for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: python-requests for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: python-prettytable for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: python-jsonpatch for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: python-configobj for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: python-cheetah for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: python-argparse for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: policycoreutils-python for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: libselinux-python for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Processing Dependency: PyYAML for package: cloud-init-0.7.4-2.el6.noarch
```

```
--> Running transaction check
```

```
--> Package PyYAML.x86_64 0:3.10-3.el6 will be installed
```

```
--> Processing Dependency: libyaml-0.so.2()(64bit) for package: PyYAML-3.10-3.el6.x86_64
```

```
--> Package libselinux-python.x86_64 0:2.0.94-5.3.el6 will be installed
```

```
--> Package policycoreutils-python.x86_64 0:2.0.83-19.30.el6 will be installed
```

```
--> Processing Dependency: libsemanage-python >= 2.0.43-4 for package: policycoreutils-python-2.0.83-
```

```
19.30.el6.x86_64
```

```
--> Processing Dependency: audit-libs-python >= 1.4.2-1 for package: policycoreutils-python-2.0.83-
```

```
19.30.el6.x86_64
```

```
--> Processing Dependency: setools-libs-python for package: policycoreutils-python-2.0.83-19.30.el6.x86_64
```

```
--> Processing Dependency: libcgroup for package: policycoreutils-python-2.0.83-19.30.el6.x86_64
```

```
--> Package python-argparse.noarch 0:1.2.1-2.el6 will be installed
```

```
--> Package python-boto.noarch 0:2.27.0-1.el6 will be installed
```

```
--> Package python-cheetah.x86_64 0:2.4.1-1.el6 will be installed
```

```
--> Processing Dependency: python-pygments for package: python-cheetah-2.4.1-1.el6.x86_64
```

```
--> Processing Dependency: python-markdown for package: python-cheetah-2.4.1-1.el6.x86_64
```

```
--> Package python-configobj.noarch 0:4.6.0-3.el6 will be installed
```

```
--> Package python-jsonpatch.noarch 0:1.2-2.el6 will be installed
```

```
--> Processing Dependency: python-jsonpointer for package: python-jsonpatch-1.2-2.el6.noarch
```

```
--> Package python-prettytable.noarch 0:0.7.2-1.el6 will be installed
```

```
--> Package python-requests.noarch 0:1.1.0-4.el6 will be installed
```

```
--> Processing Dependency: python-urllib3 for package: python-requests-1.1.0-4.el6.noarch
```

```
--> Processing Dependency: python-ordereddict for package: python-requests-1.1.0-4.el6.noarch
```

```
--> Processing Dependency: python-charset for package: python-requests-1.1.0-4.el6.noarch
```

```
--> Running transaction check
```

```
--> Package audit-libs-python.x86_64 0:2.2-2.el6 will be installed
```

```
--> Package libcgroup.x86_64 0:0.37-7.el6 will be installed
```

```
--> Package libsemanage-python.x86_64 0:2.0.43-4.2.el6 will be installed
```

```
--> Package libyaml.x86_64 0:0.1.6-1.el6 will be installed
```

```
--> Package python-charset.noarch 0:2.0.1-1.el6 will be installed
```

```
--> Package python-jsonpointer.noarch 0:1.0-3.el6 will be installed
```

```
--> Package python-markdown.noarch 0:2.0.1-3.1.el6 will be installed
```

```
--> Package python-ordereddict.noarch 0:1.1-2.el6 will be installed
```

```
--> Package python-pygments.noarch 0:1.1.1-1.el6 will be installed
```

```
--> Processing Dependency: python-setuptools for package: python-pygments-1.1.1-1.el6.noarch
```

```
--> Package python-urllib3.noarch 0:1.5-7.el6 will be installed
```

```
--> Processing Dependency: python-six for package: python-urllib3-1.5-7.el6.noarch
```

```
--> Processing Dependency: python-backports-ssl_match_hostname for package: python-urllib3-1.5-7.el6.noarch
```

```
--> Package setools-libs-python.x86_64 0:3.3.7-4.el6 will be installed
```

```
--> Processing Dependency: setools-libs = 3.3.7-4.el6 for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libgpol.so.1(VERS_1.3)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libpoldiff.so.1(VERS_1.3)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libapol.so.4(VERS_4.1)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libgpol.so.1(VERS_1.2)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libgpol.so.1(VERS_1.4)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libseaudit.so.4(VERS_4.2)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libpoldiff.so.1(VERS_1.2)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libsefs.so.4(VERS_4.0)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libapol.so.4(VERS_4.0)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libseaudit.so.4(VERS_4.1)(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libseaudit.so.4()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libgpol.so.1()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libapol.so.4()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libpoldiff.so.1()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Processing Dependency: libsefs.so.4()(64bit) for package: setools-libs-python-3.3.7-4.el6.x86_64
```

```
--> Running transaction check
```

```
--> Package python-backports-ssl_match_hostname.noarch 0:3.4.0-2-1.el6 will be installed
```

```
--> Processing Dependency: python-backports for package: python-backports-ssl_match_hostname-3.4.0-2-1.el6.noarch
```

```
--> Package python-setuptools.noarch 0:0.6.10-3.el6 will be installed
```

```
--> Package python-six.noarch 0:1.6.1-1.el6 will be installed
```

```
--> Package setools-libs.x86_64 0:3.3.7-4.el6 will be installed
```

```
--> Running transaction check
```

```
--> Package python-backports.x86_64 0:1.0-3.el6 will be installed
```

```
--> Finished Dependency Resolution
```

```
Dependencies Resolved
```

```

=====
Package                               Arch    Version      Repository
-----
Installing:
cloud-init                             noarch  0.7.4-2.el6  epel  487 k
Installing for dependencies:
PyYAML                                  x86_64  3.10-3.el6   epel  157 k
audit-libs-python                       x86_64  2.2-2.el6    slf   58 k
libcgroup                                x86_64  0.37-7.el6   slf  110 k
libselinux-python                       x86_64  2.0.94-5.3.el6 slf  201 k
libsemanage-python                     x86_64  2.0.43-4.2.el6 slf   80 k
libyaml                                  x86_64  0.1.6-1.el6  epel   52 k
policycoreutils-python                  x86_64  2.0.83-19.30.el6 slf  341 k
python-argparse                          noarch  1.2.1-2.el6  epel   48 k
python-backports                         x86_64  1.0-3.el6    epel   5.3 k
python-backports-ssl_match_hostname     noarch  3.4.0.2-1.el6 epel   12 k
python-boto                              noarch  2.27.0-1.el6 epel  1.6 M
python-chardet                           noarch  2.0.1-1.el6  epel  225 k
python-cheetah                           x86_64  2.4.1-1.el6  slf  364 k
python-configobj                         noarch  4.6.0-3.el6  slf  181 k
python-jsonpatch                         noarch  1.2-2.el6    epel   14 k
python-jsonpointer                       noarch  1.0-3.el6    epel   9.2 k
python-markdown                          noarch  2.0.1-3.1.el6 slf  117 k
python-orderdict                         noarch  1.1-2.el6    epel   7.6 k
python-prettytable                      noarch  0.7.2-1.el6  epel   37 k
python-pygments                          noarch  1.1.1-1.el6  slf  561 k
python-requests                          noarch  1.1.0-4.el6  epel   71 k
python-setuptools                        noarch  0.6.10-3.el6 slf  335 k
python-six                               noarch  1.6.1-1.el6  epel   25 k
python-urllib3                           noarch  1.5-7.el6    epel   41 k
setools-libs                             x86_64  3.3.7-4.el6  slf  399 k
setools-libs-python                     x86_64  3.3.7-4.el6  slf  221 k

Transaction Summary
=====
Install      27 Package(s)

Total download size: 5.6 M
Installed size: 25 M
Downloading Packages:
-----
Total                    532 kB/s | 5.6 MB    00:10
Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
warning: rpmts_HdrFromFdno: Header V3 RSA/SHA256 Signature, key ID 0608b895: NOKEY
Importing GPG key 0x0608B895:
  Userid : EPEL (6) <epel@fedoraproject.org>

```

```

Package: slf-release-6.4-1.x86_64 (@anaconda-ScientificLinuxFermi-201304091346.x86_64/6)
From : /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
Warning: RPMDB altered outside of yum.

Installing : python-orderdict-1.1-2.el6.noarch                1/27
Installing : libselinux-python-2.0.94-5.3.el6.x86_64        2/27
Installing : python-argparse-1.2.1-2.el6.noarch              3/27
Installing : libsemanage-python-2.0.43-4.2.el6.x86_64       4/27
Installing : python-boto-2.27.0-1.el6.noarch                 5/27
Installing : python-six-1.6.1-1.el6.noarch                   6/27
Installing : python-configobj-4.6.0-3.el6.noarch             7/27
Installing : python-setuptools-0.6.10-3.el6.noarch           8/27
Installing : python-pygments-1.1.1-1.el6.noarch              9/27
Installing : python-markdown-2.0.1-3.1.el6.noarch            10/27
Installing : python-cheetah-2.4.1-1.el6.x86_64              11/27
Installing : python-chardet-2.0.1-1.el6.noarch               12/27
Installing : python-prettytable-0.7.2-1.el6.noarch           13/27
Installing : libcgroup-0.37-7.el6.x86_64                     14/27
Installing : setools-libs-3.3.7-4.el6.x86_64                 15/27
Installing : setools-libs-python-3.3.7-4.el6.x86_64         16/27
Installing : libyaml-0.1.6-1.el6.x86_64                      17/27
Installing : PyYAML-3.10-3.el6.x86_64                         18/27
Installing : audit-libs-python-2.2-2.el6.x86_64             19/27
Installing : policycoreutils-python-2.0.83-19.30.el6.x86_64 20/27

```

```
Installing : python-backports-1.0-3.el6.x86_64 21/27
Installing : python-backports-ssl_match_hostname-3.4.0.2-1.el6.noarch 22/27
Installing : python-urllib3-1.5-7.el6.noarch 23/27
Installing : python-requests-1.1.0-4.el6.noarch 24/27
Installing : python-jsonpointer-1.0-3.el6.noarch 25/27
Installing : python-jsonpatch-1.2-2.el6.noarch 26/27
Installing : cloud-init-0.7.4-2.el6.noarch 27/27
Verifying : python-jsonpointer-1.0-3.el6.noarch 1/27
Verifying : python-backports-1.0-3.el6.x86_64 2/27
Verifying : libselinux-python-2.0.94-5.3.el6.x86_64 3/27
Verifying : audit-libs-python-2.2-2.el6.x86_64 4/27
Verifying : setools-libs-python-3.3.7-4.el6.x86_64 5/27
Verifying : libyaml-0.1.6-1.el6.x86_64 6/27
Verifying : setools-libs-3.3.7-4.el6.x86_64 7/27
Verifying : libcgroupp-0.37-7.el6.x86_64 8/27
Verifying : python-jsonpatch-1.2-2.el6.noarch 9/27
Verifying : python-pygments-1.1.1-1.el6.noarch 10/27
Verifying : python-prettytable-0.7.2-1.el6.noarch 11/27
Verifying : python-backports-ssl_match_hostname-3.4.0.2-1.el6.noarch 12/27
Verifying : python-urllib3-1.5-7.el6.noarch 13/27
Verifying : python-charDET-2.0.1-1.el6.noarch 14/27
Verifying : python-requests-1.1.0-4.el6.noarch 15/27
Verifying : python-cheetah-2.4.1-1.el6.x86_64 16/27
Verifying : policycoreutils-python-2.0.83-19.30.el6.x86_64 17/27
```

```
Verifying : cloud-init-0.7.4-2.el6.noarch 18/27
Verifying : python-markdown-2.0.1-3.1.el6.noarch 19/27
Verifying : python-setuptools-0.6.10-3.el6.noarch 20/27
Verifying : python-configobj-4.6.0-3.el6.noarch 21/27
Verifying : python-six-1.6.1-1.el6.noarch 22/27
Verifying : PyYAML-3.10-3.el6.x86_64 23/27
Verifying : python-boto-2.27.0-1.el6.noarch 24/27
Verifying : python-ordereddict-1.1-2.el6.noarch 25/27
Verifying : libsemanage-python-2.0.43-4.2.el6.x86_64 26/27
Verifying : python-argparse-1.2.1-2.el6.noarch 27/27
```

Installed:

```
cloud-init.noarch 0:0.7.4-2.el6
```

Dependency Installed:

```
PyYAML.x86_64 0:3.10-3.el6
audit-libs-python.x86_64 0:2.2-2.el6
libcgroupp.x86_64 0:0.37-7.el6
libselinux-python.x86_64 0:2.0.94-5.3.el6
libsemanage-python.x86_64 0:2.0.43-4.2.el6
libyaml.x86_64 0:0.1.6-1.el6
policycoreutils-python.x86_64 0:2.0.83-19.30.el6
python-argparse.noarch 0:1.2.1-2.el6
python-backports.x86_64 0:1.0-3.el6
python-backports-ssl_match_hostname.noarch 0:3.4.0.2-1.el6
python-boto.noarch 0:2.27.0-1.el6
python-charDET.noarch 0:2.0.1-1.el6
python-cheetah.x86_64 0:2.4.1-1.el6
python-configobj.noarch 0:4.6.0-3.el6
python-jsonpatch.noarch 0:1.2-2.el6
python-jsonpointer.noarch 0:1.0-3.el6
python-markdown.noarch 0:2.0.1-3.1.el6
python-ordereddict.noarch 0:1.1-2.el6
python-prettytable.noarch 0:0.7.2-1.el6
python-pygments.noarch 0:1.1.1-1.el6
python-requests.noarch 0:1.1.0-4.el6
python-setuptools.noarch 0:0.6.10-3.el6
python-six.noarch 0:1.6.1-1.el6
python-urllib3.noarch 0:1.5-7.el6
```

```
setools-libs.x86_64 0:3.3.7-4.el6
setools-libs-python.x86_64 0:3.3.7-4.el6
```

Complete!

Loaded plugins: security
Setting up Install Process
Resolving Dependencies

```
--> Running transaction check
--> Package cloud-utils.x86_64 0:0.27-10.el6 will be installed
--> Processing Dependency: qemu-img for package: cloud-utils-0.27-10.el6.x86_64
--> Processing Dependency: python-paramiko for package: cloud-utils-0.27-10.el6.x86_64
--> Processing Dependency: euca2ools for package: cloud-utils-0.27-10.el6.x86_64
--> Processing Dependency: cloud-utils-growpart for package: cloud-utils-0.27-10.el6.x86_64
--> Running transaction check
--> Package cloud-utils-growpart.x86_64 0:0.27-10.el6 will be installed
--> Package euca2ools.noarch 0:2.1.4-1.el6 will be installed
--> Processing Dependency: m2crypto for package: euca2ools-2.1.4-1.el6.noarch
--> Package python-paramiko.noarch 0:1.7.5-2.1.el6 will be installed
--> Processing Dependency: python-crypto >= 1.9 for package: python-paramiko-1.7.5-2.1.el6.noarch
--> Package qemu-img.x86_64 2:0.12.1.2-2.415.el6_5.10 will be installed
--> Processing Dependency: libusbredirparser.so.l()(64bit) for package: 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64
--> Processing Dependency: libgfxdr.so.0()(64bit) for package: 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64
--> Processing Dependency: libgfxrpc.so.0()(64bit) for package: 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64
--> Processing Dependency: libgfxapi.so.0()(64bit) for package: 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64
--> Running transaction check
--> Package glusterfs-api.x86_64 0:3.4.0.36rhs-1.el6 will be installed
--> Package glusterfs-libs.x86_64 0:3.4.0.36rhs-1.el6 will be installed
--> Package m2crypto.x86_64 0:0.20.2-9.el6 will be installed
--> Package python-crypto.x86_64 0:2.0.1-22.el6 will be installed
--> Package usbredir.x86_64 0:0.5.1-1.el6 will be installed
--> Finished Dependency Resolution
```

Dependencies Resolved

```
=====
Package                Arch      Version                               Repository      Size
=====
Installing:
cloud-utils             x86_64   0.27-10.el6                           epel             43 k
Installing for dependencies:
cloud-utils-growpart   x86_64   0.27-10.el6                           epel             25 k
euca2ools               noArch   2.1.4-1.el6                           epel            326 k
glusterfs-api          x86_64   3.4.0.36rhs-1.el6                     slf-security    44 k
glusterfs-libs         x86_64   3.4.0.36rhs-1.el6                     slf-security    225 k
m2crypto               x86_64   0.20.2-9.el6                           slf             470 k
python-crypto          x86_64   2.0.1-22.el6                           slf             157 k
python-paramiko        noarch   1.7.5-2.1.el6                           slf             727 k
qemu-img               x86_64   2:0.12.1.2-2.415.el6_5.10             slf-security    595 k
=====
```

```
usbredir                x86_64   0.5.1-1.el6                           slf              39 k
```

Transaction Summary

```
-----
Install      10 Package(s)
```

Total download size: 2.6 M

Installed size: 13 M

Downloading Packages:

```
-----
Total                               570 kB/s | 2.6 MB    00:04
```

Running rpm_check_debug

Running Transaction Test

Transaction Test Succeeded

Running Transaction

```
Installing : glusterfs-libs-3.4.0.36rhs-1.el6.x86_64           1/10
Installing : glusterfs-api-3.4.0.36rhs-1.el6.x86_64           2/10
Installing : usbredir-0.5.1-1.el6.x86_64                       3/10
Installing : 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64        4/10
Installing : python-crypto-2.0.1-22.el6.x86_64                 5/10
Installing : python-paramiko-1.7.5-2.1.el6.noarch              6/10
Installing : cloud-utils-growpart-0.27-10.el6.x86_64          7/10
Installing : m2crypto-0.20.2-9.el6.x86_64                     8/10
Installing : euca2ools-2.1.4-1.el6.noarch                       9/10
Installing : cloud-utils-0.27-10.el6.x86_64                   10/10
Verifying  : 2:qemu-img-0.12.1.2-2.415.el6_5.10.x86_64        1/10
Verifying  : euca2ools-2.1.4-1.el6.noarch                      2/10
Verifying  : m2crypto-0.20.2-9.el6.x86_64                     3/10
Verifying  : cloud-utils-growpart-0.27-10.el6.x86_64          4/10
Verifying  : glusterfs-libs-3.4.0.36rhs-1.el6.x86_64          5/10
Verifying  : python-crypto-2.0.1-22.el6.x86_64                6/10
```

```
Verifying : python-paramiko-1.7.5-2.1.el6.noarch 7/10
Verifying : cloud-utils-0.27-10.el6.x86_64 8/10
Verifying : usbredir-0.5.1-1.el6.x86_64 9/10
Verifying : glusterfs-api-3.4.0.36rhs-1.el6.x86_64 10/10
```

```
Installed:
cloud-utils.x86_64 0:0.27-10.el6
```

```
Dependency Installed:
cloud-utils-growpart.x86_64 0:0.27-10.el6
euca2ools.noarch 0:2.1.4-1.el6
glusterfs-api.x86_64 0:3.4.0.36rhs-1.el6
glusterfs-libs.x86_64 0:3.4.0.36rhs-1.el6
m2crypto.x86_64 0:0.20.2-9.el6
python-crypto.x86_64 0:2.0.1-22.el6
python-paramiko.noarch 0:1.7.5-2.1.el6
qemu-img.x86_64 2:0.12.1.2-2.415.el6_5.10
usbredir.x86_64 0:0.5.1-1.el6
```

```
Complete!
Loaded plugins: security
Setting up Install Process
Examining /var/tmp/yum-root-DElfP9/python-backports-1.0-4.el6.x86_64.rpm: python-backports-1.0-4.el6.x86_64
Marking /var/tmp/yum-root-DElfP9/python-backports-1.0-4.el6.x86_64.rpm as an update to python-backports-1.0-3.el6.x86_64
Resolving Dependencies
--> Running transaction check
--> Package python-backports.x86_64 0:1.0-3.el6 will be updated
--> Package python-backports.x86_64 0:1.0-4.el6 will be an update
--> Finished Dependency Resolution
```

Dependencies Resolved

```
=====
Package Arch Version Repository Size
=====
Updating:
python-backports x86_64 1.0-4.el6 /python-backports-1.0-4.el6.x86_64 318
```

Transaction Summary

```
=====
Upgrade 1 Package(s)
```

```
Total size: 318
Downloading Packages:
```

```
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
```

```
Updating : python-backports-1.0-4.el6.x86_64 1/2
Cleanup : python-backports-1.0-3.el6.x86_64 2/2
Verifying : python-backports-1.0-4.el6.x86_64 1/2
Verifying : python-backports-1.0-3.el6.x86_64 2/2
```

```
Updated:
python-backports.x86_64 0:1.0-4.el6
```

```
Complete!
Aug 02 16:55:29 INFO Stop: Completed Converting the worker Fermicloud VM image. Function convert_image.
Aug 02 16:55:29 INFO Start: Importing the worker Fermicloud VM image. Function import_image.
```

```
Loaded plugins: refresh-packagekit, security
Setting up Install Process
Package 1:java-1.6.0-openjdk-1.6.0.0-6.1.13.4.el6_5.x86_64 already installed and latest version
Nothing to do
```

```
/usr/lib/jvm/jre-1.6.0-openjdk.x86_64
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
0 0 0 0 0 0 0 0 ---:--:-- ---:--:-- ---:--:-- 0
4 14.2M 4 708k 0 0 890k 0 0:00:16 ---:--:-- 0:00:16 915k
100 14.2M 100 14.2M 0 0 8873k 0 0:00:01 0:00:01 ---:--:-- 8997k
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
```

```
0 0 0 0 0 0 0 0 ---:--:-- ---:--:-- ---:--:-- 0
100 152k 100 152k 0 0 575k 0 ---:--:-- ---:--:-- ---:--:--Loaded plugins: refresh-packagekit, security
```

```
Setting up Install Process
630k
Package 1:tccl-devel-8.5.7-6.el6.x86_64 already installed and latest version
Package 1:tk-devel-8.5.7-5.el6.x86_64 already installed and latest version
Package expect-devel-5.44.1.15-2.el6.x86_64 already installed and latest version
Package expectk-5.44.1.15-2.el6.x86_64 already installed and latest version
Nothing to do
AWS CLI Tools Setup Completed.
INSTANCE i-94fe889f stopped pending
ATTACHMENT vol-af1680ae i-94fe889f /dev/sdg attaching 2014-08-02T21:56:45+0000
spawn ssh -i khs-fermi.pem root@54.187.12.76
The authenticity of host '54.187.12.76 (54.187.12.76)' can't be established.
```

RSA key fingerprint is ab:90:bl:c4:ef:65:f4:5f:c9:8f:db:df:9l:f6:6a:9c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.187.12.76' (RSA) to the list of known hosts.

Last login: Thu Jul 17 16:43:33 2014 from 131.225.170.210

NOTICE TO USERS

This is a Federal computer (and/or it is directly connected to a Fermilab local network system) that is the property of the United States Government. It is for authorized use only. Users (authorized or unauthorized) have no explicit or implicit expectation of privacy.

Any or all uses of this system and all files on this system may be intercepted, monitored, recorded, copied, audited, inspected, and disclosed to authorized site, Department of Energy and law enforcement personnel, as well as authorized officials of other agencies, both domestic and foreign. By using this system, the user consents to such interception, monitoring, recording, copying, auditing, inspection, and disclosure at the discretion of authorized site or Department of Energy personnel.

Unauthorized or improper use of this system may result in administrative disciplinary action and civil and criminal penalties. By continuing to use this system you indicate your awareness of and consent to these terms and conditions of use. LOG OFF IMMEDIATELY if you do not agree to the conditions stated in this warning.

Fermilab policy and rules for computing, including appropriate use, may be found at <http://www.fnal.gov/cd/main/cpolicy.html>
/usr/bin/xauth: creating new authority file /root/.Xauthority
[root@ip-172-31-38-72 ~]# mkfs -t ext4 /dev/xvdf
mke2fs 1.41.12 (17-May-2010)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
1310720 inodes, 5242880 blocks
262144 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=4294967296
160 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:

32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
4096000

Writing inode tables:
0/1601/1602/1603/1604/1605/1606/1607/1608/1609/16010/16011/16012/16013/16014/16015/16016/16017/16018/16019/16020/
16021/16022/16023/16024/16025/16026/16027/16028/16029/16030/16031/16032/16033/16034/16035/16036/16037/16038/16039/
16040/16041/16042/16043/16044/16045/16046/16047/16048/16049/16050/16051/16052/16053/16054/16055/16056/16057/1605/
8/16059/16060/16061/16062/16063/16064/16065/16066/16067/16068/16069/16070/16071/16072/16073/16074/16075/16076/160/
77/16078/16079/16080/16081/16082/16083/16084/16085/16086/16087/16088/16089/16090/16091/16092/16093/16094/16095/16/
096/16097/16098/16099/160100/160101/160102/160103/160104/160105/160106/160107/160108/160109/160110/160111/160112/
160113/160114/160115/160116/160117/160118/160119/160120/160121/160122/160123/160124/160125/160126/160127/160128/1/
60129/160130/160131/160132/160133/160134/160135/160136/160137/160138/160139/160140/160141/160142/160143/160144/16/
0145/160146/160147/160148/160149/160150/160151/160152/160153/160154/160155/160156/160157/160158/160159/160done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: mount -t ext4 /dev/xvdf /mnt
done

This filesystem will be automatically checked every 27 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.

[root@ip-172-31-38-72 ~]# mount -t ext4 /dev/xvdf /mnt
cd /mnt

[root@ip-172-31-38-72 ~]# cd /mnt
[root@ip-172-31-38-72 mnt]# mkdir -p images
[root@ip-172-31-38-72 mnt]# mkfs -t ext4 /dev/xvdf
mke2fs 1.41.12 (17-May-2010)

Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
655360 inodes, 2621440 blocks
131072 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=2684354560
80 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Writing inode tables:
0/801/802/803/804/805/806/807/808/809/8010/8011/8012/8013/8014/8015/8016/8017/8018/8019/8020/8021/8022/8023/8024/
8025/8026/8027/8028/8029/8030/8031/8032/8033/8034/8035/8036/8037/8038/80mknod -p /opt/ec2/mnt
39/8040/8041/8042/8043/8044/8045/8046/8047/8048/8049/8050/8051/8052/8053/8054/8055/8056/8057/8058/8059/8060/8061/
8062/8063/8064/8065/8066/8067/8068/8069/8070/8071/8072/8073/8074/8075/8076/8077/8078/8079/80done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: mount -t ext4 /dev/xvdf /opt/ec2/mnt
done

```
This filesystem will be automatically checked every 24 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@ip-172-31-38-72 mnt]# mkdir -p /opt/ec2/mnt
[root@ip-172-31-38-72 mnt]# mount -t ext4 /dev/xvdg /opt/ec2/mnt
[root@ip-172-31-38-72 mnt]# exit
logout
Connection to 54.187.12.76 closed.
```

```
spawned process completed...
Back to script. Please wait...
Copying (SCP) Raw image to AWS. Please wait...
spawn ssh -i khs-fermi.pem root@54.187.12.76
Last login: Sat Aug  2 16:56:57 2014 from 131.225.155.50
```

NOTICE TO USERS

This is a Federal computer (and/or it is directly connected to a Fermilab local network system) that is the property of the United States Government. It is for authorized use only. Users (authorized or unauthorized) have no explicit or implicit expectation of privacy.

Any or all uses of this system and all files on this system may be intercepted, monitored, recorded, copied, audited, inspected, and disclosed to authorized site, Department of Energy and law enforcement personnel, as well as authorized officials of other agencies, both domestic and foreign. By using this system, the user consents to such interception, monitoring, recording, copying, auditing, inspection, and disclosure at the discretion of authorized site or Department of Energy personnel.

Unauthorized or improper use of this system may result in administrative disciplinary action and civil and criminal penalties. By continuing to use this system you indicate your awareness of and consent to these terms and conditions of use. LOG OFF IMMEDIATELY if you do not agree to the conditions stated in this warning.

```
Fermilab policy and rules for computing, including appropriate use, may be found at http://www.fnal.gov/cd/main/cpolicy.html
[root@ip-172-31-38-72 ~]# cd /mnt/images
[root@ip-172-31-38-72 images]# mkdir -p raw
[root@ip-172-31-38-72 images]# kpartx -a gcso_sl6.raw
[root@ip-172-31-38-72 images]# mount /dev/mapper/loop0p1 /mnt/images/raw
[root@ip-172-31-38-72 images]# cd /mnt/images/raw
[root@ip-172-31-38-72 raw]# rsync -aqHx /mnt/images/raw/ /opt/ec2/mnt
[root@ip-172-31-38-72 raw]# rsync -aqHx /mnt/images/raw/dev /opt/ec2/mnt
```

```
[root@ip-172-31-38-72 raw]# cd /opt/ec2/mnt
[root@ip-172-31-38-72 mnt]# tune2fs -L '/' /dev/xvdg
tune2fs 1.41.12 (17-May-2010)
[root@ip-172-31-38-72 mnt]# sync;sync;sync;sync
[root@ip-172-31-38-72 mnt]# cd /mnt/images
[root@ip-172-31-38-72 images]# umount /mnt/images/raw
[root@ip-172-31-38-72 images]# kpartx -d gcso_sl6.raw
loop deleted : /dev/loop0
[root@ip-172-31-38-72 images]# exit
logout
Connection to 54.187.12.76 closed.
```

```
spawned process completed...
Back to script. Please wait...
ATTACHMENT vol-af1680ae i-94fe889f /dev/sdg detaching 2014-08-02T21:56:45+0000
TAG volume vol-af1680ae Name GCSO_SL6_PV_20140802165545
TAG volume vol-af1680ae User kirkshal
TAG snapshot snap-117fdee6 Name GCSO_SL6_PV_20140802165545
TAG snapshot snap-117fdee6 User kirkshal
TAG image ami-9fd2a9af Name GCSO_SL6_PV_20140802165545
TAG image ami-9fd2a9af User kirkshal
TAG instance i-abe593a0 Name GCSO_SL6_PV_20140802165545
TAG instance i-abe593a0 User kirkshal
TAG volume vol-79178178 Name GCSO_SL6_PV_20140802165545
TAG volume vol-79178178 User kirkshal
INSTANCE i-94fe889f running stopping
INSTANCE i-abe593a0 running stopping
VOLUME vol-af1680ae
AWS VM Import Completed.
Aug 02 17:07:38 INFO Stop: Completed Importing the worker Fermicloud VM image. Function import_image.
Aug 02 17:07:38 INFO End script run
```

```
-----
-----
-----
```

File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/create_samplemime.txt

```
$ vi my-user-script.txt
#!/bin/bash
echo "=====Hello Fermi======" > /root/fermihello1.txt
:wq

$ vi my-cloudconfig.txt
#cloud-config
cloud_final_modules:
- rightscale_userdata
```

```

- scripts-per-once
- scripts-per-boot
- scripts-per-instance
- [scripts-user, always]
- ssh-authkey-fingerprints
- keys-to-console
- phone-home
- final-message

runcmd:

- [ sh, -c, echo "====Hello Fermi====" > /root/fermihello2.txt ]
:wq

$ write-mime-multipart --output=samplemime.txt \
  my-user-script.txt:text/x-shellscript \
  my-cloudconfig.txt
-----
-----
-----

File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/fermi_VM_Convert.txt
# This file contains working steps to manually create a AWS working instance. From this I was able to create the
automated
# scripts. Note: Not every command was used in the automated scripts these are working notes only.

# Copy golden image from fermicloud.fnal.gov to data area on fermicloud vm (~2 minutes)
scp kirkshal@fermicloud.fnal.gov:/var/lib/one/local/pub_scratch/gcso_sl6.img
root@fermicloud103.fnal.gov:/data/gcso_sl6.qcow2
scp kirkshal@fermicloud.fnal.gov:/var/lib/one/local/images/fa5219771174d0a16dd4ba0b0cf9e012
root@fermicloud103.fnal.gov:/data/gcso_sl6.qcow2
scp kirkshal@fermicloud.fnal.gov:/var/lib/one/local/images/21022925deb7edd64800ebb3bf97f102
root@fermicloud103.fnal.gov:/data/gcso_sl6.qcow2

# Clean Up Image for conversion to aws
ssh root@fermicloud103.fnal.gov
virt-rescue /data/gcso_sl6.qcow2
fdisk -l
fdisk /dev/sda
p
d
2
w

#Exit virt-rescue
exit

```

```

cd /data
mkdir work
guestmount -a gcso_sl6.qcow2 -m /dev/sda1 work

# cp /grid/data/parag/gwms-cloudvm-rpms/glideinwms-vm-core-0.4-0.2.rc2.el6.noarch.rpm work
# cp /grid/data/parag/gwms-cloudvm-rpms/glideinwms-vm-ec2-0.4-0.2.rc2.el6.noarch.rpm work

chroot work

cd /etc/sysconfig/network-scripts
vi ifcfg-eth0
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
TYPE=Ethernet

:wq

cd /etc/sysconfig
vi /etc/sysconfig/network

from:

NETWORKING=yes
HOSTNAME=localhost.localdomain
GATEWAY=131.225.154.1

to:

NETWORKING=yes

:wq

cd /etc
rm /etc/resolv.conf
y
rm /etc/hosts
y
rm /etc/hosts.allow
y
rm /etc/hosts.deny
y

vi /boot/grub/grub.conf

from: (for Xen HVM Only)

```

```

# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE: You do not have a /boot partition. This means that
#           all kernel and initrd paths are relative to /, eg.
#           root (hd0,0)
#           kernel /boot/vmlinuz-version ro root=/dev/vda1
#           initrd /boot/initrd-[generic-]version.img
#boot=/dev/vda
default=0
timeout=5
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
hiddenmenu
title Scientific Linux Fermi (2.6.32-431.11.2.el6.x86_64)
    root (hd0,0)
    kernel /boot/vmlinuz-2.6.32-431.11.2.el6.x86_64 ro root=UUID=17898259-6979-4bb3-9d73-26ae917e8ed9
rd_NO_LUKS rd_NO_LVM LANG=en_US.UTF-8 rd_NO_MD SYSFONT=latarcyrheb-sun16 crashkernel=auto KEYBOARDTYPE=pc
KEYTABLE=us rd_NO_DM rhgb quiet
    initrd /boot/initramfs-2.6.32-431.11.2.el6.x86_64.img

to: (for Xen PV Only)

default=0
timeout=0
title Scientific Linux Fermi (2.6.32-431.11.2.el6.x86_64)
    root (hd0)
    kernel /boot/vmlinuz-2.6.32-431.11.2.el6.x86_64 ro root=/dev/xvdel rd_NO_PLYMOUTH
    initrd /boot/initramfs-2.6.32-431.11.2.el6.x86_64.img

:wq

cd /boot; ln -s . boot

vi /etc/fstab

from: (for Xen HVM Only)

#
# /etc/fstab
# Created by anaconda on Fri May  2 13:00:03 2014
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
UUID=17898259-6979-4bb3-9d73-26ae917e8ed9 /                ext3    defaults    1 1
UUID=2c7666fb-e233-4793-899e-c783e3e4f328 swap             swap    defaults    0 0
UUID=b0dcfd19-a2b1-41a1-8ff6-168732f909dd swap             swap    defaults    0 0

```

```

tmpfs          /dev/shm                tmpfs    defaults    0 0
devpts         /dev/pts                devpts   gid=5,mode=620 0 0
sysfs         /sys                    sysfs    defaults    0 0
proc          /proc                   proc     defaults    0 0

```

to: (for Xen PV Only)

```

/dev/xvdel    /                ext4    defaults    1 1
/dev/vdb     swap             swap    defaults    0 0
tmpfs        /dev/shm        tmpfs    defaults    0 0
devpts       /dev/pts        devpts   gid=5,mode=620 0 0
sysfs        /sys            sysfs    defaults    0 0
proc         /proc           proc     defaults    0 0
/dev/xvdf    /scratch        xfs     defaults    0 0

```

:wq

vi /etc/ssh/sshd_config

from:

```

Protocol 2
RSAAuthentication no
PubkeyAuthentication no
PasswordAuthentication no
ChallengeResponseAuthentication no
UsePAM yes
KerberosAuthentication yes
KerberosOrLocalPasswd no
KerberosTicketCleanup yes
GSSAPIAuthentication yes
GSSAPIKeyExchange yes
GSSAPICleanupCredentials yes
X11Forwarding yes
AllowTcpForwarding yes
UsePrivilegeSeparation yes
UseDNS no
PermitRootLogin yes

```

to:

```

SyslogFacility AUTHPRIV
RSAAuthentication no
PubkeyAuthentication yes
AuthorizedKeysFile .ssh/authorized_keys
PasswordAuthentication yes
KerberosAuthentication no

```

```

KerberosOrLocalPasswd no
KerberosTicketCleanup no
GSSAPIAuthentication no
GSSAPICleanupCredentials no
UsePAM no
AllowTcpForwarding yes
X11Forwarding yes
UseLogin no
UseDNS no

:wq

# Create a script that captures the public key credentials for your root login
vi /etc/init.d/ec2-get-ssh

#!/bin/bash
# chkconfig: 2345 95 20
# processname: ec2-get-ssh
# description: Capture AWS public key credentials for EC2 user

# Source function library
. /etc/rc.d/init.d/functions

# Source networking configuration
[ -r /etc/sysconfig/network ] && . /etc/sysconfig/network

# Replace the following environment variables for your system
export PATH=/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin

# Check that networking is configured
if [ "${NETWORKING}" = "no" ]; then
    echo "Networking is not configured."
    exit 1
fi

start() {
    if [ ! -d /root/.ssh ]; then
        mkdir -p /root/.ssh
        chmod 700 /root/.ssh
    fi
    # Retrieve public key from metadata server using HTTP
    curl -f http://169.254.169.254/latest/meta-data/public-keys/0/openssh-key > /tmp/my-public-key
    if [ $? -eq 0 ]; then
        echo "EC2: Retrieve public key from metadata server using HTTP."
        cat /tmp/my-public-key >> /root/.ssh/authorized_keys
        chmod 600 /root/.ssh/authorized_keys
        rm /tmp/my-public-key
    fi
}

stop() {
    echo "Nothing to do here"
}

restart() {
    stop
    start
}

# See how we were called.
case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    restart)
        restart
        ;;
    *)
        echo $"Usage: $0 {start|stop|restart}"
        exit 1
esac

exit $?
:wq

# Update the runlevel information for the new system service on the image.

/bin/chmod +x /etc/init.d/ec2-get-ssh

chkconfig --level 34 ec2-get-ssh on

chkconfig postfix off
chkconfig autofs off
chkconfig vmcontext off

# rpm -ivh /glideinwms-vm-core-0.4-0.2.rc2.el6.noarch.rpm
# rpm -ivh /glideinwms-vm-ec2-0.4-0.2.rc2.el6.noarch.rpm

# Prevent 10 minute vm shutdown for testing (turn back on after testing)
# chkconfig glideinwms-pilot off

```

```

}

stop() {
    echo "Nothing to do here"
}

restart() {
    stop
    start
}

# See how we were called.
case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    restart)
        restart
        ;;
    *)
        echo $"Usage: $0 {start|stop|restart}"
        exit 1
esac

exit $?
:wq

# Update the runlevel information for the new system service on the image.

/bin/chmod +x /etc/init.d/ec2-get-ssh

chkconfig --level 34 ec2-get-ssh on

chkconfig postfix off
chkconfig autofs off
chkconfig vmcontext off

# rpm -ivh /glideinwms-vm-core-0.4-0.2.rc2.el6.noarch.rpm
# rpm -ivh /glideinwms-vm-ec2-0.4-0.2.rc2.el6.noarch.rpm

# Prevent 10 minute vm shutdown for testing (turn back on after testing)
# chkconfig glideinwms-pilot off

```

```

# make sure /etc/puppet/modules directory is present if not create it
# download Puppet Module for CVMFS
# puppet module install desalvo-cvmfs

# install cvmfs
# vi /etc/puppet/modules/cvmfs/manifests/client.pp
# add "include cvmfs::client" to the end
# :wq
# puppet apply /etc/puppet/modules/cvmfs/manifests/client.pp

history -c
exit
fusermount -u work
rmdir /data/work

# Resize Image from 256G to 12.1G
# qemu-img create [-f format] [-o options] filename [size]

rm gcso_sl6.raw
y
qemu-img create -f raw gcso_sl6.raw 3075M

# guestfish
cd /data

guestfish -a gcso_sl6.qcow2tmp
e2fsck -f /dev/sdal
resize2fs-size /dev/sdal 3G
# virt-rescue -a gcso_sl6.qcow2tmp
# e2fsck -f /dev/sdal
# resize2fs /dev/sdal 3G
# fdisk -l
# blkid -c /dev/null
exit

# Resize takes about 7 minutes ?
virt-resize --delete /dev/sda2 --resize /dev/sdal=3G gcso_sl6.qcow2tmp gcso_sl6.raw

# Convert Image from QCOW2 to RAW for AWS VM Import (Not used as Virt-Resize replaces this)
# qemu-img convert -p -f qcow2 -O raw gcso_sl6.qcow2 ./gcso_sl6.raw

# Status of converted 3G raw file
qemu-img info gcso_sl6.raw

# Install AWS EC2 API Tools (ec2-api-tools.zip)
# http://docs.aws.amazon.com/AWSEC2/latest/CommandLineReference/set-up-ec2-cli-linux.html

```

```

# Import VM for HVM Snapshot
ec2iin -f RAW -t m3.medium -a x86_64 -b fcloudimport -o AKIAIDJ22XYOSDF3BRPQ -w
SvLUaCSB106Bj7KqVKEDW/AOY1qLcLgJ82THqc3C -p Linux --region us-west-2 -z us-west-2a gcso_sl6.raw

# Average speed was 22.738 MBps to upload import (~ 10 minutes)
# Approx. 20 minutes to convert to AWS Instance after upload
# Total time approx 30 minutes.

# Status of import
ec2-describe-conversion-tasks

# Create AMI from Imported Instance
# Snapshot auto created

# Launch new HVM AMI Instance
# Manual
ssh -i khs-fermi.pem root@nn.nn.nn.nn
# Auto
# ec2-start-instances instance_id [instance_id...]
ec2-start-instances i-650b6b6e
ip=`ec2-describe-instances i-l81e8913 | grep NICASSOCIATION | awk {'print($2)}`
echo $ip
ssh -i khs-fermi.pem root@$ip

# Below only used if pem auth not set up
# Accept key
# yes
# Use root pw (get from admin if needed)
# mkdir .ssh
# cd .ssh
# vi authorized_keys
# insert your public key
# :wq
# chmod 700 ~/.ssh
# chmod 600 ~/.ssh/authorized_keys
# exit
# ssh -i khs-fermi.pem root@nn.nn.nn.nn

# Create staged VM Image on AWS HVM 30G Worker Instance
fdisk -l
mkfs -t ext4 /dev/xvdf
mount -t ext4 /dev/xvdf /mnt
cd /mnt
mkdir -p /mnt/images

# Create new volume on AWS Worker from fermicloud103
vol=`ec2-create-volume --size 10 --region us-west-2 -z us-west-2a | grep VOLUME | awk {'print($2)}`

```

```

# Attach volume on AWS Worker from fermicloud103
ec2-attach-volume --instance i-181e8913 --device /dev/sdg vol-010a8600 --region us-west-2

# on AWS Worker
mkfs -t ext4 /dev/xvdg

mkdir -p /opt/ec2/mnt
mount -t ext4 /dev/xvdg /opt/ec2/mnt

# Import VM for PV Snapshot

# Copy .pem and metadata script from Desktop terminal to FermiCloud103 worker VM
scp -Cq /Users/kshallcross/khs-fermi.pem root@fermicloud103.fnal.gov:/data
scp -Cq -i khs-fermi.pem /Users/kshallcross/Desktop/ec2-metadata root@54.191.225.130:/root
# ON AWS Worker
chmod +x ec2-metadata
#List IP addresses
./ec2-metadata -o -v

# From FermiCloud103
cd /data
#rsync -S -z -v --progress -e "ssh -i khs-fermi.pem" /data/gcso_sl6.raw root@nn.nn.nn:/mnt/images
rsync -S -z -q -e "ssh -i khs-fermi.pem" /data/gcso_sl6.raw root@54.191.225.130:/mnt/images
yes to auth

# Rename grub.conf.pv and fstab.pv to grub.conf and fstab on fermicloud103

cd /mnt/images
mkdir raw
kpartx -a gcso_sl6.raw
mount /dev/mapper/loop0p1 /mnt/images/raw
cd /mnt/images/raw
cd etc
mv fstab fstab.hvm
mv fstab.pv fstab
cd ..
cd boot/grub
mv grub.conf grub.conf.hvm
mv grub.conf.pv grub.conf
cd /mnt/images

# Move image to new volume
rsync -aqHx /mnt/images/raw/ /opt/ec2/mnt

rsync -aqHx /mnt/images/raw/dev /opt/ec2/mnt

# Label the disk.
cd /opt/ec2/mnt

```

```

tune2fs -L '/' /dev/xvdf

# Flush all writes and unmount the volume.
sync;sync;sync;sync

cd /mnt/images
umount /mnt/images/raw
kpartx -d gcso_sl6.raw
# dmsetup remove /dev/mapper/loop0p1
# losetup -d /dev/loop0p1

# Move image to new volume (Not used, kpartx rsync above does this)
# sudo dd if=/opt/ec2/mnt/gcso_sl6.raw of=/dev/xvdg bs=10M (not needed)
# scp root@fermicloud103.fnal.gov:/data/gcso_sl6.raw root@54.187.254.70:/dev/xvdh (not needed)

# Detach Volume from fermicloud103
ec2-detach-volume vol-010a8600

# Create Snapshot to be used to create PV AMI
ec2-create-snapshot --region us-west-2 -d "gcso_sl6 pv snap" vol-010a8600

# Create PV AMI
# ec2-register -n "AMI-HK-SLF582" -d "SLF58-Test2" --root-device-name /dev/sda2 -b /dev/sda=snap-fdc5fa09:12:true
--architecture x86_64 --region us-west-2
ec2-register -n "AMI-GCSO-SL6-PV test" -d "AMI-GCSO-SL6-PV test" -b "/dev/sda=snap-fdc5fa09:10:true:gp2" -b
"/dev/sdb=ephemeral0" --architecture x86_64 --kernel aki-fc8f11cc --region us-west-2

# Create and run Instance from PV AMI
ec2-run-instances --region us-west-2 -z us-west-2a ami-d3bec6e3 -g sg-a6a010c3 -n 1 --kernel aki-fc8f11cc -t
m3.medium -k khs-fermi

# Create tags for all resources
ec2-create-tags vol-010a8600 snap-fdc5fa09 ami-d3bec6e3 i-6fe08164 vol-c525a9c4 --tag "Name=Fermi GCSO_SL6 PV" --
tag "User=kirkshal"

# On new instance mount Scratch on ephemeral0 SSD storage
# Modify (append) /etc/rc.d/rc.local
# mkfs.xfs /dev/xvdf (already ext3)

mkdir -p /scratch
mount /dev/xvdf /scratch

-----

#
#

```

```

# Supportive Documentation below...
# Prepare the EBS PV Volume

#
#
# Below are details that support above statements
# Log in to the EC2 instance HVM AMI and create an ext4 filesystem type on the partitionless EBS volume.

/bin/egrep '[xvsh]d[a-z].*$' /proc/partitions
# 202      65      10485760 xvda1
# 202      66      156352512 xvda2
# 202      67      917504 xvda3
# 202      144      10485760 xvdf

mkfs.ext4 /dev/xvdf

# Note that while we specified to attach the EBS volume as device /dev/sdg in the previous step, Amazon Linux
uses the Xen virtual disk notation /dev/xvda. The volume in my case was attached as /dev/xvdg.

# Create a mount point directory and mount the EBS volume.

mkdir -p /opt/ec2/mnt

mount -t ext4 /dev/xvdf /opt/ec2/mnt

mount

# /dev/xvda1 on / type ext4 (rw)
# none on /proc type proc (rw)
# none on /sys type sysfs (rw)
# none on /dev/pts type devpts (rw,gid=5,mode=620)
# none on /dev/shm type tmpfs (rw)
# /dev/xvda2 on /mnt/voll type ext4 (rw)
# none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
# /dev/xvdf on /opt/ec2/mnt type ext4 (rw)

# Remove any local instance storage entries from /etc/fstab if they exist. Booting from an EBS volume does not
use local instance storage by default. If you followed this guide to create the instance store-backed AMI, remove
the local instance storage entry in /etc/fstab.

cat /etc/fstab | grep -v mnt > /tmp/fstab

mv /etc/fstab /etc/fstab.bak

mv /tmp/fstab /etc/fstab

```

```

# Sync the root and dev file systems to the EBS volume.

rsync -S -z -v --progress -e "ssh -i khs-fermi.pem" /data/gcso_sl6.raw root@54.187.254.70:/opt/ec2/mnt

rsync -aqHx / /opt/ec2/mnt

rsync -aqHx /dev /opt/ec2/mnt

# Label the disk.

tune2fs -L '/' /dev/xvdg

# Flush all writes and unmount the volume.

sync;sync;sync;sync

umount /opt/ec2/mnt

# Detach Volume

# Create Snapshot from Volume to be used to create PV AMI

# Create PV AMI

# Create Instance from PV AMI

# Modify (append) /etc/rc.d/rc.local
# Make a Scratch file system
mkdir -p /scratch

# Mount and Unmount Paravirtual
mkfs.xfs -f /dev/xvdf
mount /dev/xvdf /scratch
umount /scratch

# Mount and Unmount HVM
mkfs.xfs -f /dev/xvdb
mount /dev/xvdb /scratch
umount /scratch

# display fs info
fdisk -l
df -h
blkid -c /dev/null

# Linux Release info
cat /etc/*release
uname -a

```

```

# Find files
find / -name xxx* 2> /dev/null

#Setup Expect
yum install tcl-devel tk-devel expect-devel expectk

-----
-----

File: puppetrepo-awsexport/files/opt/gcso/awsexport/docs/samplemime.txt
Content-Type: multipart/mixed; boundary="====3302562582348490243=="
MIME-Version: 1.0

-----3302562582348490243==
Content-Type: text/x-shellscript; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename="my-user-script.txt"

#!/bin/bash
echo "====Hello Fermi====" > /root/fermihello1.txt

-----3302562582348490243==
Content-Type: text/cloud-config; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename="my-cloudconfig.txt"

#cloud-config
cloud_final_modules:
- rightscale_userdata
- scripts-per-once
- scripts-per-boot
- scripts-per-instance
- [scripts-user, always]
- ssh-authkey-fingerprints
- keys-to-console
- phone-home
- final-message

runcmd:

- [ sh, -c, echo "====Hello Fermi====" > /root/fermihello2.txt ]

-----3302562582348490243==--

```

```

-----
-----

File: puppetrepo-awsexport/files/opt/gcso/awsexport/ec2-get-ssh
#!/bin/bash
# chkconfig: 2345 95 20
# processname: ec2-get-ssh
# description: Capture AWS public key credentials for EC2 user

# Source function library
. /etc/rc.d/init.d/functions

# Source networking configuration
[ -r /etc/sysconfig/network ] && . /etc/sysconfig/network

# Replace the following environment variables for your system
export PATH=/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin

# Check that networking is configured
if [ "${NETWORKING}" = "no" ]; then
    echo "Networking is not configured."
    exit 1
fi

start() {
    if [ ! -d /root/.ssh ]; then
        mkdir -p /root/.ssh
        chmod 700 /root/.ssh
    fi
    # Retrieve public key from metadata server using HTTP
    curl -f http://169.254.169.254/latest/meta-data/public-keys/0/openssh-key > /tmp/my-public-key
    if [ $? -eq 0 ]; then
        echo "EC2: Retrieve public key from metadata server using HTTP."
        cat /tmp/my-public-key >> /root/.ssh/authorized_keys
        chmod 600 /root/.ssh/authorized_keys
        rm /tmp/my-public-key
    fi
}

stop() {
    echo "Nothing to do here"
}

restart() {
    stop
    start
}

```

```

}

# See how we were called.
case "$1" in
  start)
    start
    ;;
  stop)
    stop
    ;;
  restart)
    restart
    ;;
  *)
    echo $"Usage: $0 {start|stop|restart}"
    exit 1
esac

exit $?
-----
-----
-----

File: puppetrepo-awsexport/files/opt/gcso/awsexport/gcso.pem
-----BEGIN RSA PRIVATE KEY-----

[SNIP]

-----END RSA PRIVATE KEY-----
-----
-----

File: puppetrepo-awsexport/manifests/awsexport.pp
# include awsexport
class { 'awsexport': }
-----
-----

File: puppetrepo-awsexport/manifests/awsexport_params.pp
#contains parameters for the awsexport::awsexport class

define awsexport::awsexport_params (
    $svm_file_location,

```

```

    $svm_image_location,
    $ckernel_ver,
    $svm_number,
    $svm_name,
    $svm_owner,
    $caws_image_name,
    $caws_image_owner,
    $caws_instance,
    $caws_key,
    $caws_secret_key,
    $caws_pem_name,
    $caws_worker_instance_id,
    $caws_owner_keypair_name,
    $caws_eph_mount
)
{
include awsexport

# create a cron job

$command = "/opt/gcso/awsexport/Convert.py $svm_file_location $svm_image_location $ckernel_ver $svm_number
$svm_name $svm_owner $caws_image_name $caws_image_owner $caws_instance $caws_key $caws_secret_key $caws_pem_name
$caws_worker_instance_id $caws_owner_keypair_name $caws_eph_mount"

$environmentv = [ 'MAILTO=kirkshal@fnal.gov', 'JAVA_HOME=/usr/lib/jvm/jre-1.6.0-openjdk.x86_64',
'EC2_BASE=/usr/local/ec2', 'EC2_HOME=/usr/local/ec2/ec2-api-tools-1.7.1.0', 'EC2_URL=https://ec2.us-west-
2.amazonaws.com', 'PATH=/usr/lib64/qt-
3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/usr/local/ec2/ec2-api-tools-
1.7.1.0/bin', 'AWS_ACCOUNT_NUMBER=159067897602', "AWS_ACCESS_KEY_ID=${caws_key}",
"AWS_SECRET_ACCESS_KEY=${caws_secret_key}" ]

cron {'awsexport':

    minute    => '15',

    hour      => '00',

    monthday  => '26',

    month     => '8',

    weekday   => '*',

    user      => 'root',

    command   => $command,

    environment => $environmentv;

```

```
}
}
-----
-----
-----
```

```
File: puppetrepo-awsexport/manifests/gcso_sl6_giwms_pv.pp
#contains parameters for the gcso_sl6 GIWMS PV image conversion
awsexport::awsexport_params {'gcso_sl6_giwms_pv':
  cvm_file_location => '/opt/gcso/awsexport',
  cvm_image_location => 'oneadmin@fcl008:/var/lib/one/local/images/6a50ad27120ad62979d75ba2dbfc8e98',
  ckernel_ver => '2.6.32-431.23.3.el6.x86_64',
  cvm_number => '103',
  cvm_name => 'gcso_sl6_giwms',
  cvm_owner => 'kirkshal',
  caws_image_name => 'GCSO_SL6_GIWMS_PV',
  caws_image_owner => 'gcso',
  caws_instance => 'm3.medium',
  caws_key => 'AKIAJX75FCYK6SAOW4DQ',
  caws_secret_key => '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',
  caws_pem_name => 'gcso.pem',
  caws_worker_instance_id => 'i-3eb6e435',
  caws_owner_keypair_name => 'gcso',
  caws_eph_mount => 'none',
}
-----
-----
-----
```

```
File: puppetrepo-awsexport/manifests/gcso_sl6_hvm.pp
#contains parameters for the gcso_sl6 HVM image conversion
awsexport::awsexport_params {'gcso_sl6_hvm':
  cvm_file_location => '/opt/gcso/awsexport',
  cvm_image_location => 'oneadmin@fcl008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5',
  ckernel_ver => '2.6.32-431.23.3.el6.x86_64',
  cvm_number => '103',
  cvm_name => 'gcso_sl6',
  cvm_owner => 'kirkshal',
  caws_image_name => 'GCSO_SL6_HVM',
  caws_image_owner => 'gcso',
  caws_instance => 'm3.medium',
  caws_key => 'AKIAJX75FCYK6SAOW4DQ',
  caws_secret_key => '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',
  caws_pem_name => 'gcso.pem',
```

```
  caws_worker_instance_id => 'hvm',
  caws_owner_keypair_name => 'gcso',
  caws_eph_mount => '/scratch',
}
-----
-----
-----
```

```
File: puppetrepo-awsexport/manifests/gcso_sl6_pv.pp
#contains parameters for the gcso_sl6 PV image conversion
awsexport::awsexport_params {'gcso_sl6_pv':
  cvm_file_location => '/opt/gcso/awsexport',
  cvm_image_location => 'oneadmin@fcl008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5',
  ckernel_ver => '2.6.32-431.23.3.el6.x86_64',
  cvm_number => '103',
  cvm_name => 'gcso_sl6',
  cvm_owner => 'kirkshal',
  caws_image_name => 'GCSO_SL6_PV',
  caws_image_owner => 'gcso',
  caws_instance => 'm3.medium',
  caws_key => 'AKIAJX75FCYK6SAOW4DQ',
  caws_secret_key => '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',
  caws_pem_name => 'gcso.pem',
  caws_worker_instance_id => 'i-3eb6e435',
  caws_owner_keypair_name => 'gcso',
  caws_eph_mount => '/scratch',
}
-----
-----
-----
```

```
File: puppetrepo-awsexport/manifests/init.pp
# This class deploys all the files for the awsexport scripts that Kirk has developed (Aug 2014).
include cron
class awsexport {

  # create awsexport source directory
  file {'/opt/gcso/awsexport':
    ensure => 'directory',
    owner  => 'root',
    group  => 'root',
    recurse => true,
    source => 'puppet:///files/opt/gcso/awsexport',
  }

  # create data directory
```

```
file { ["/data":
  ensure => "directory",
  owner  => 'root',
  group  => 'root',
  recurse => true,
]
}

# Setup by path properties for the different puppet resources

Exec {
  path => [ '/usr/lib64/qt-
3.3/bin', '/usr/krb5/bin', '/bin', '/usr/local/bin', '/usr/bin', '/usr/local/sbin', '/usr/sbin', '/sbin', '/root/bin', '/u
sr/local/ec2/ec2-api-tools-1.7.1.0/bin',
]
}

}
-----
-----
-----
```

Puppet Procedure to run Fermi2AWS Export

1. For a Paravirtual image on AWS: (Copy these for other images and rename them)

```
- vi /etc/puppet/modules/awsexport/manifests/gcso_sl6_pv.pp
- change parameters to your credentials.
- *Note: Obtain HVM worker instance id from aws console (caws_worker_instance_id => 'i-7788c97c')

- #contains parameters for the gcso_sl6 PV image conversion
- awsexport::awsexport_params {'gcso_sl6_pv':
-   cvm_file_location => '/opt/gcso/awsexport',
-   cvm_image_location => 'oneadmin@fcl008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5',
-   ckernel_ver => '2.6.32-431.23.3.el6.x86_64',
-   cvm_number => '103',
-   cvm_name => 'gcso_sl6',
-   cvm_owner => 'your Fermi username',
-   caws_image_name => 'GCSO_SL6_PV',
-   caws_image_owner => 'gcso',
-   caws_instance => 'm3.medium',
-   caws_key => 'add aws owner key here',
-   caws_secret_key => 'add aws secret owner key here',
-   caws_pem_name => 'gcso.pem',
-   caws_worker_instance_id => 'i-7788c97c',
-   caws_owner_keypair_name => 'gcso',
-   caws_eph_mount => '/ ephemeral_mount_dir or none',
- }
```

2. For a HVM image on AWS: (Copy these for other images and rename them)

```
- vi /etc/puppet/modules/awsexport/manifests/gcso_sl6_hvm.pp
- change parameters to your credentials.
- *Note: DO NOT CHANGE (caws_worker_instance_id => 'hvm') leave as 'hvm' to create a worker vm on aws

- #contains parameters for the gcso_sl6 HVM image conversion
- awsexport::awsexport_params {'gcso_sl6_hvm':
-   cvm_file_location => '/opt/gcso/awsexport',
-   cvm_image_location => 'oneadmin@fcl008:/var/lib/one/local/images/55c42a4cc7f87ea3390bc2bef14212c5',
-   ckernel_ver => '2.6.32-431.23.3.el6.x86_64',
-   cvm_number => '103',
-   cvm_name => 'gcso_sl6',
-   cvm_owner => 'your Fermi username',
-   caws_image_name => 'GCSO_SL6_HVM',
-   caws_image_owner => 'gcso',
-   caws_instance => 'm3.medium',
-   caws_key => 'add aws owner key here',
-   caws_secret_key => 'add aws secret owner key here',
-   caws_pem_name => 'gcso.pem',
-   caws_worker_instance_id => 'hvm',
-   caws_owner_keypair_name => 'gcso',
-   caws_eph_mount => '/ ephemeral_mount_dir or none',
- }
```

3. To setup Crontab jobs:

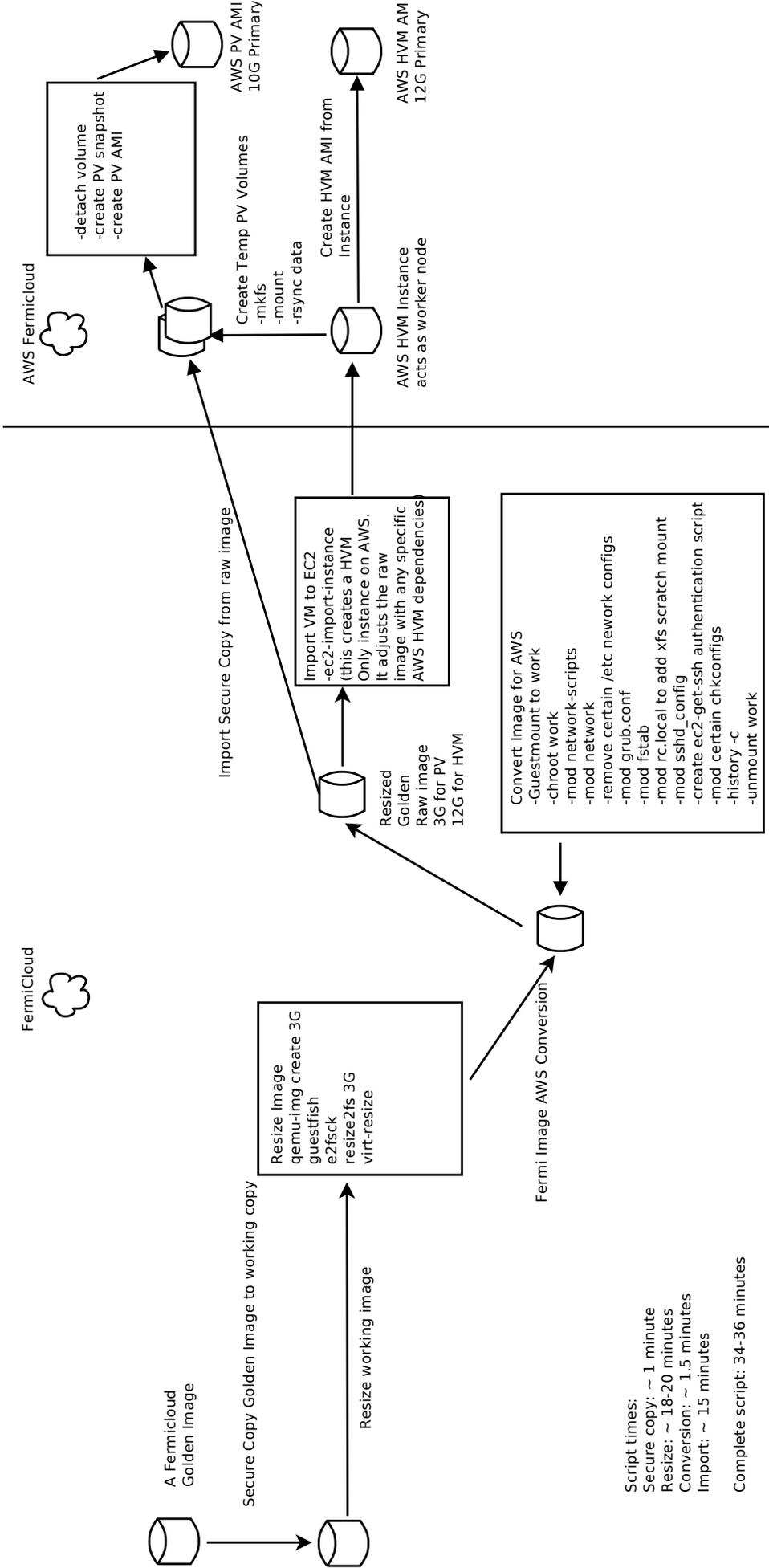
```
- vi /etc/puppet/modules/awsexport/manifests/awsexport_params.pp
- Change MAILTO=username@fnal.gov to your email address to receive cron job completion emails.
- Change the runtime schedule to what you want:
- cron {'awsexport':
-   minute => '55',
-   hour => '10',
-   monthday => '14',
-   month => '8',
-   weekday => '*',
```

4. Run Puppet apply to set crontab job for AWS HVM worker:

- run 'puppet apply /etc/puppet/modules/awsexport/manifests/gcso_sl6_hvm.pp'
- wait for cron completion email (in about 1.5 hours for a HVM Conversion)
- detail job log is located at: /opt/gcso/awsexport/aws_image_convert.log
- obtain aws console HVM worker node 'instance id' for subsequent PV conversion runs.
- *Note: this HVM worker node is needed only once. It can be run again for other images, if you want to provide HVM AMI's and instances on AWS.

5. Run Puppet apply to set crontab job for AWS PV Images:

- run 'puppet apply /etc/puppet/modules/awsexport/manifests/gcso_sl6_pv.pp'
- wait for cron completion email (in about 55 minutes for a PV Conversion)
- detail job log is located at: /opt/gcso/awsexport/aws_image_convert.log
- check aws console to see PV AMI's and instances.
- *Note: this job can run, as needed, to obtain a latest AWS image. The AWS AMI's and instances are time-stamped to identify the latest version.



Script times:
 Secure copy: ~ 1 minute
 Resize: ~ 18-20 minutes
 Conversion: ~ 1.5 minutes
 Import: ~ 15 minutes
 Complete script: 34-36 minutes

Microsoft Windows Azure Google Compute Engine

Quick reference on virtual machines management

Alessio Balsini

10/01/2014

Contents

| | |
|---|----|
| Azure and Google Cloud VM Instances Creation Through Web Interface..... | 3 |
| Azure..... | 3 |
| Instance Creation..... | 3 |
| Notes | 6 |
| Google Cloud | 7 |
| Instance Creation..... | 7 |
| Instance Image Upload and Instance Creation by Command Line | 10 |
| Microsoft Windows Azure Custom Data | 12 |
| Google Cloud: Google Compute Engine in Detail | 13 |
| Accessing the REST API | 13 |
| Prerequisites..... | 13 |
| Authentication..... | 14 |
| Rest API..... | 16 |
| Replica Pools and Autoscaler..... | 17 |
| Replica Pools management | 17 |
| Autoscaler management | 18 |
| Metadata Management..... | 19 |
| Setting Metadata..... | 19 |
| Getting Metadata | 20 |

Azure and Google Cloud VM Instances Creation Through Web Interface

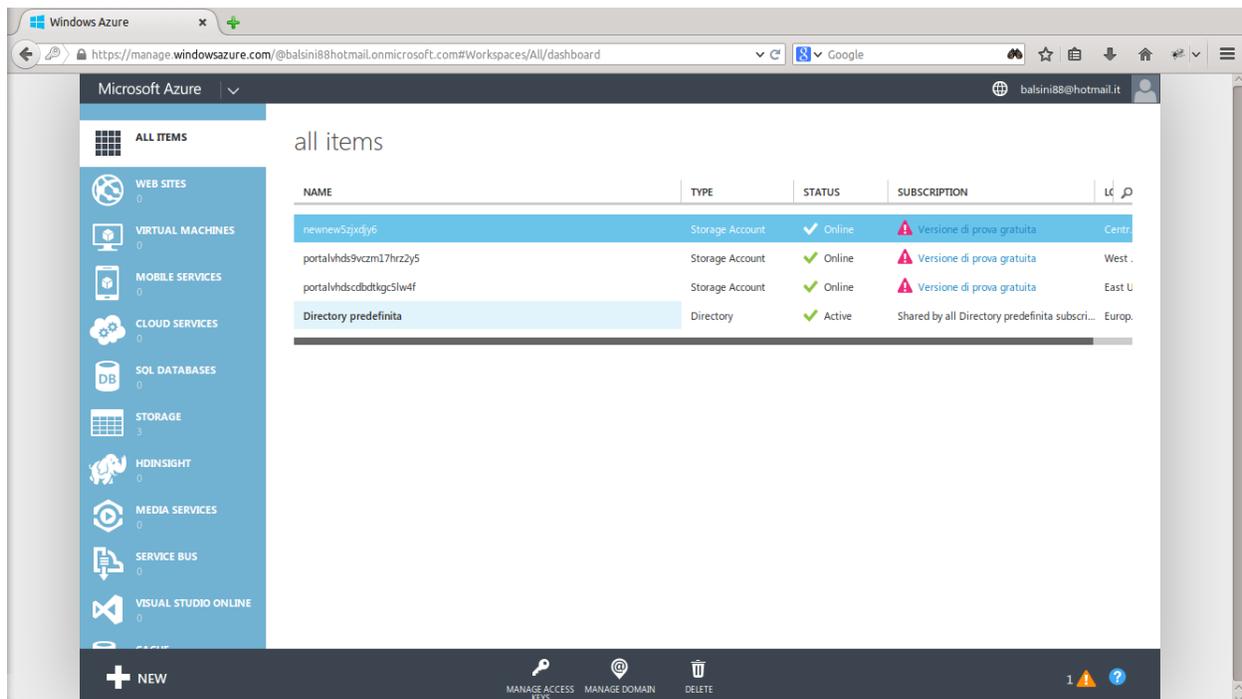
The Web interfaces are the easiest way of managing projects and a welcoming tool for the newcomers approaching the new environment.

Azure

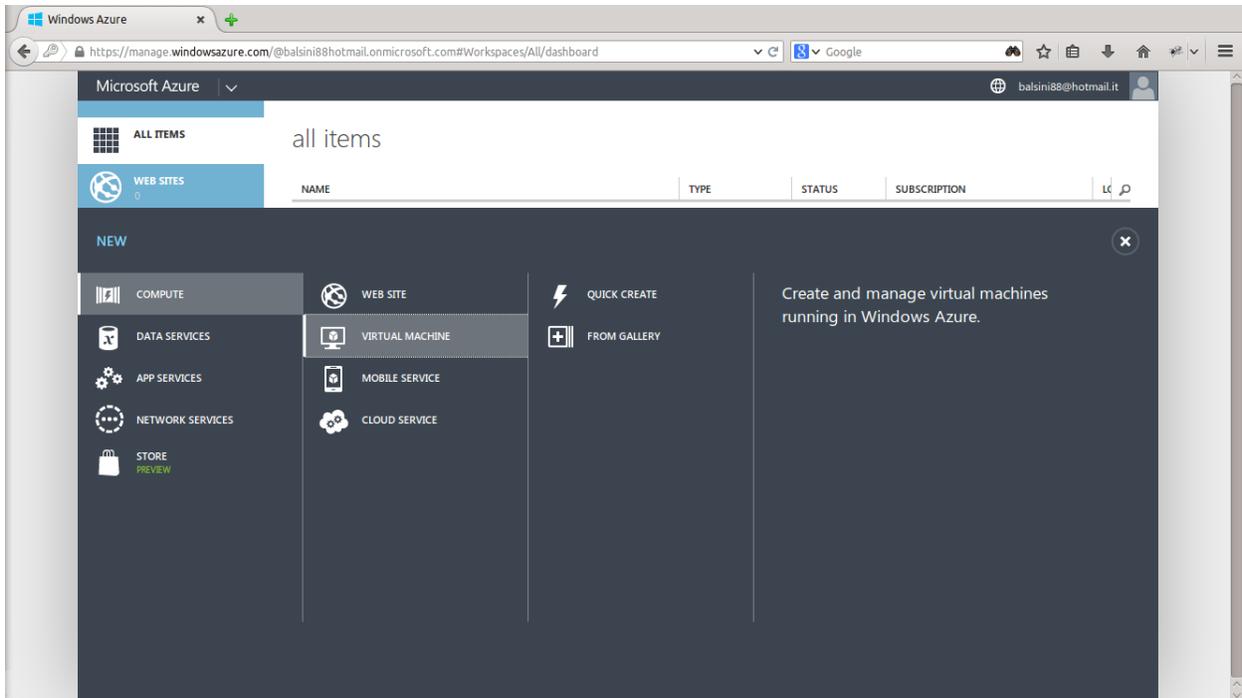
Instance Creation

Log into the Azure console at <https://manage.windowsazure.com/>.

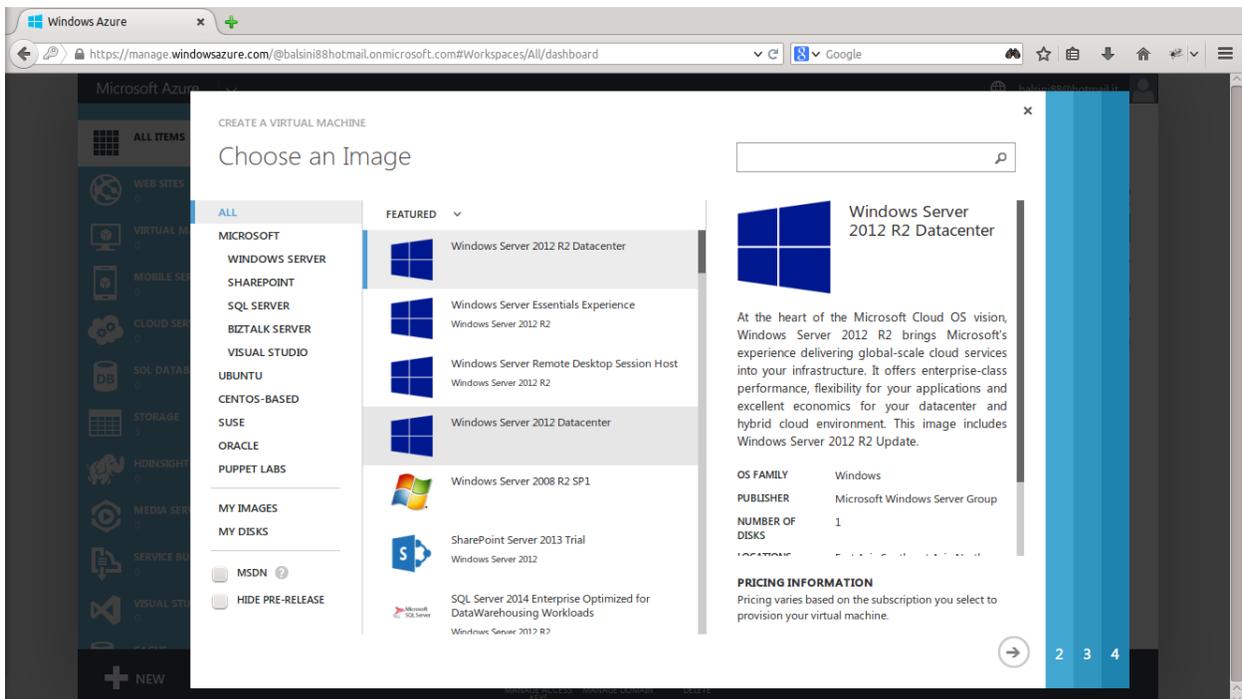
Click the "NEW" button in the bottom-left of the main window.



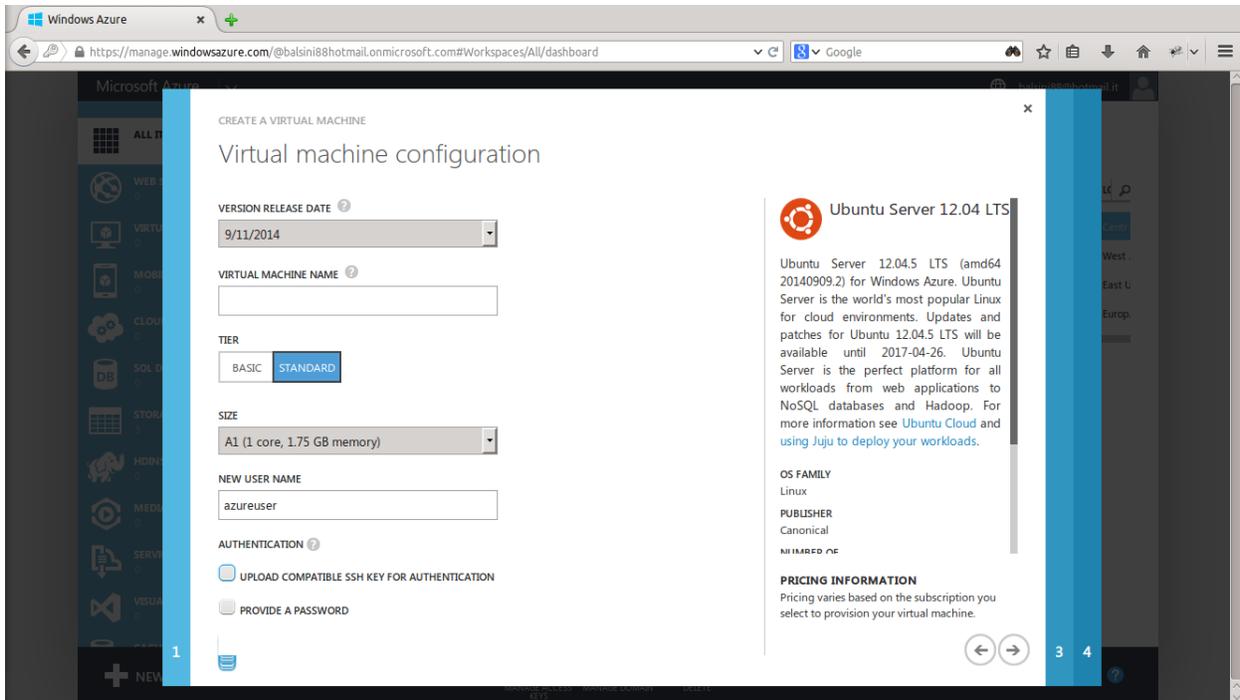
Now, select
COMPUTE -> VIRTUAL MACHINE -> FROM GALLERY



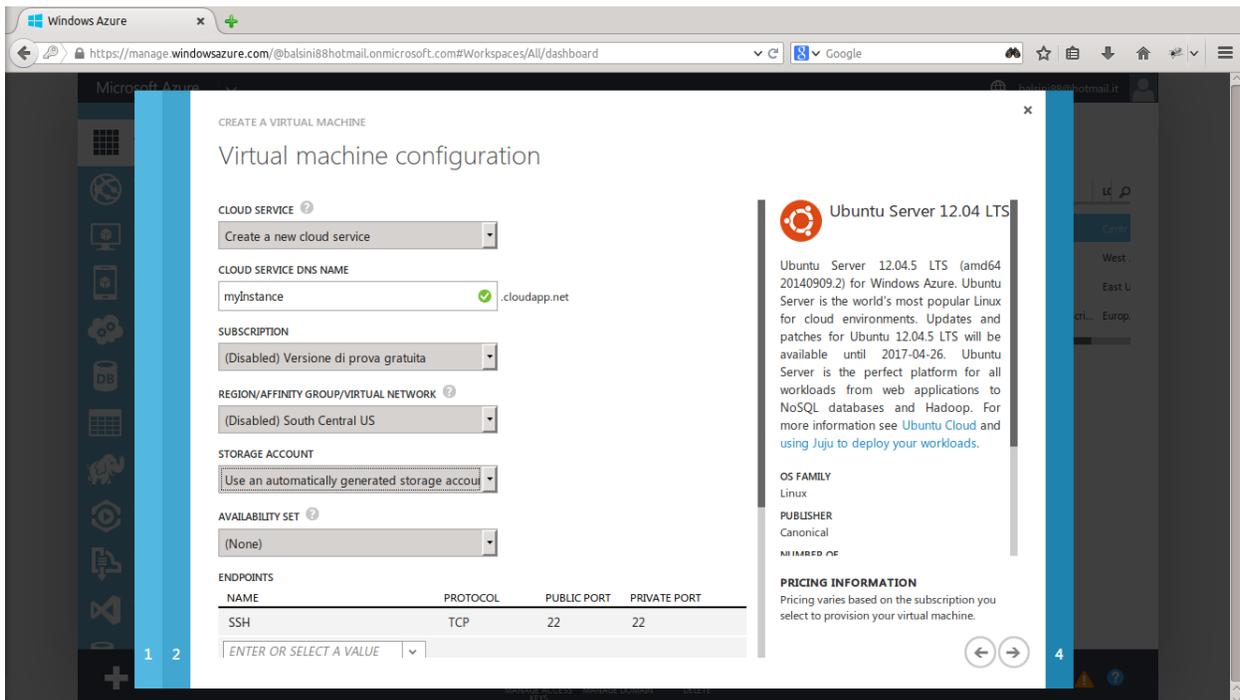
Choose your favorite disk image. In the example, an Ubuntu Server 12.04 is chosen, but it is also possible to choose custom images previously uploaded.

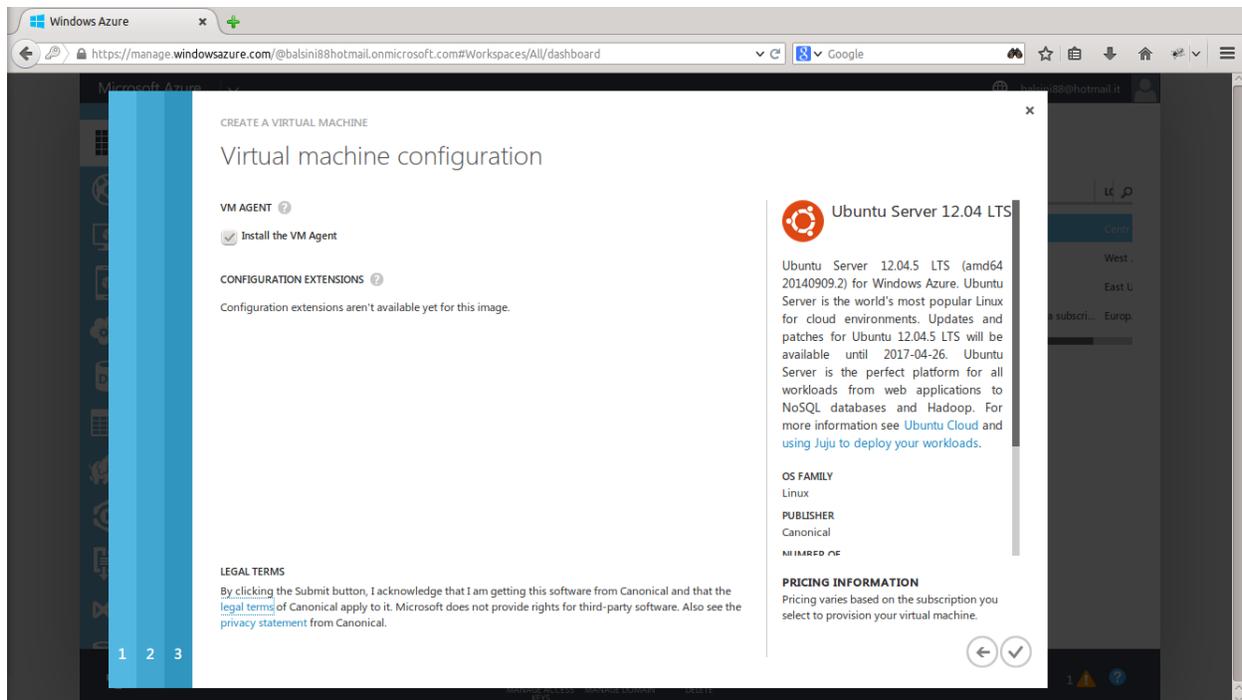


Choose the virtual machine name and authentication method.



Check any other preferences.





Notes

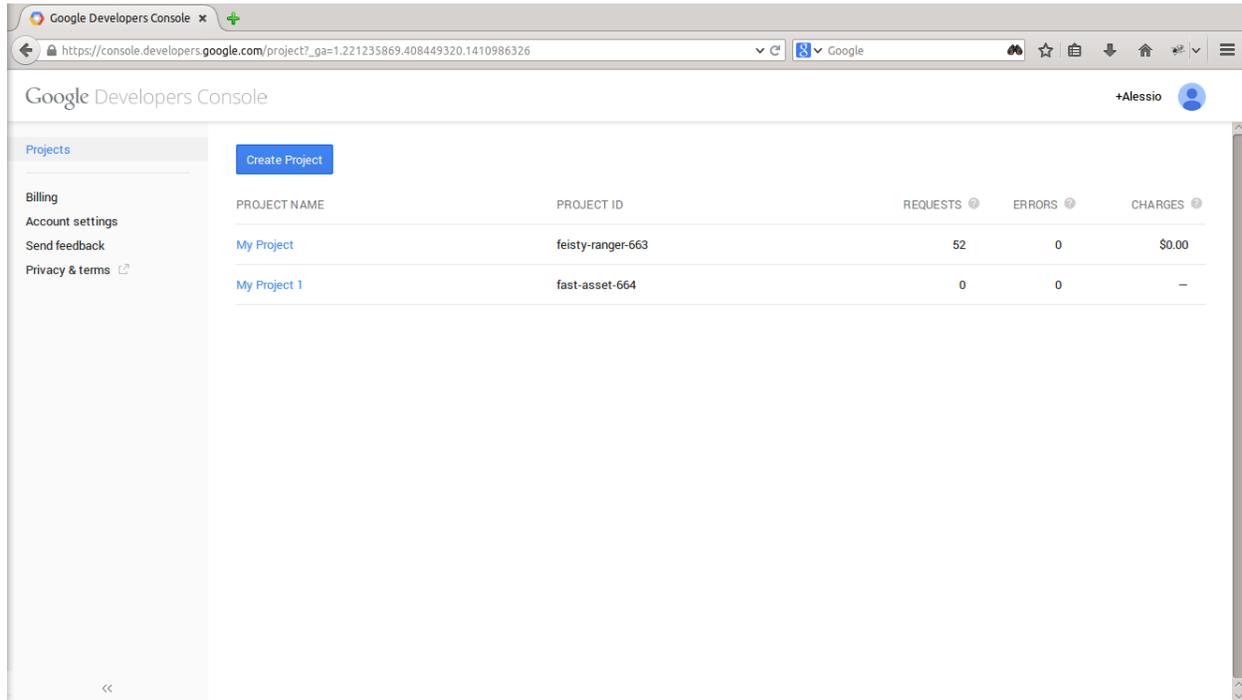
- There's no way to submit User Metadata (which Microsoft calls Custom Data) through the web interface. The VM must be initialized by command line at initialization phase.
- To manually upload custom disk images, follow the steps described here: <http://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-create-upload-vhd>

Google Cloud

Instance Creation

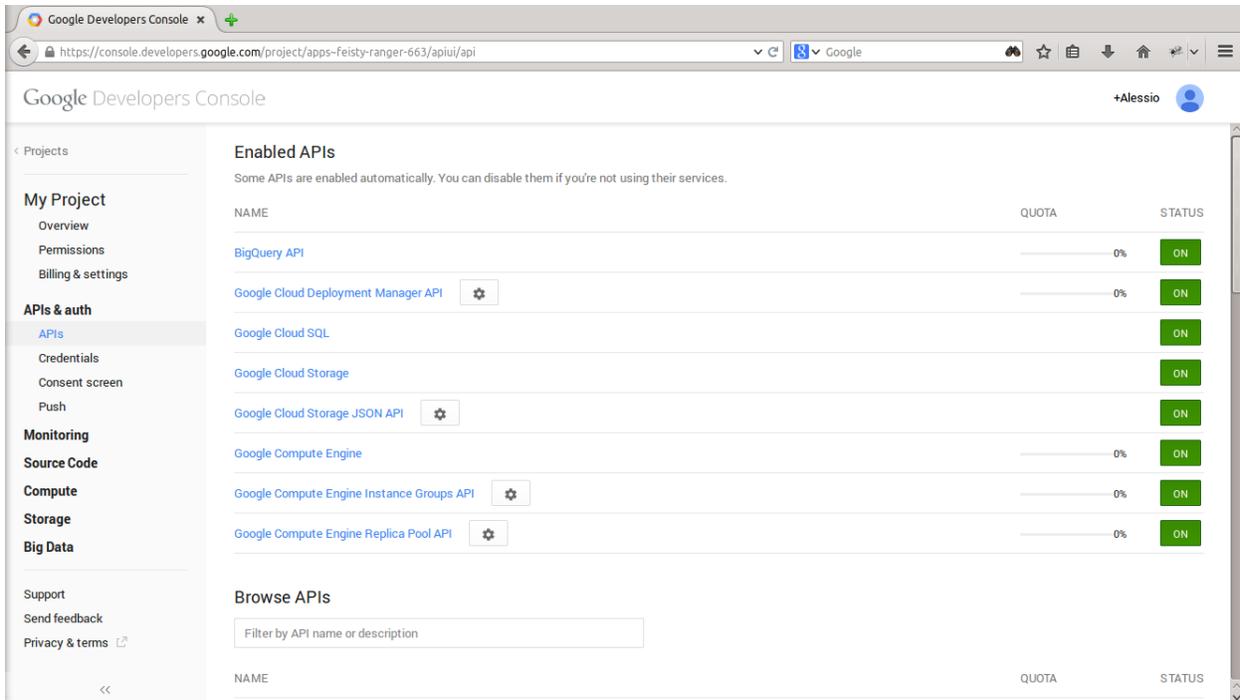
Log into the Google Cloud console at <https://console.developers.google.com/>.

Choose your project.

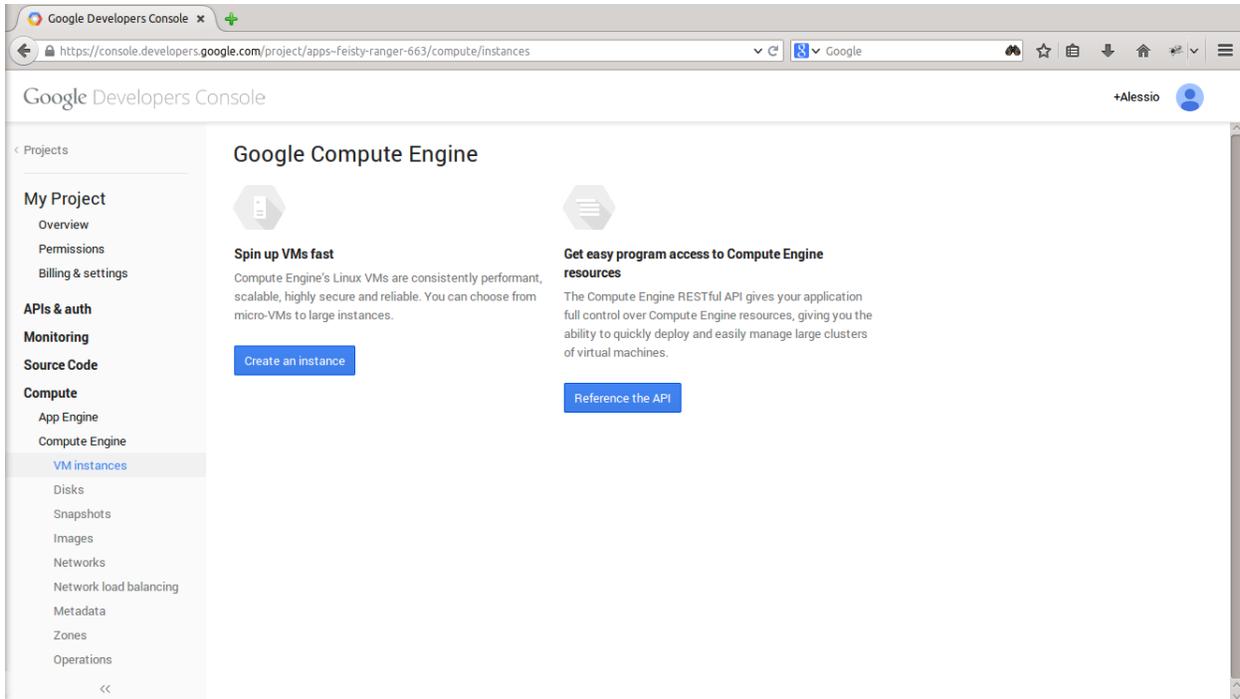


Make sure you have enabled the "Google Compute Engine" API in

APIs & auth -> APIs



Create a new Virtual Machine instance in
 Compute -> Compute Engine -> VM instances



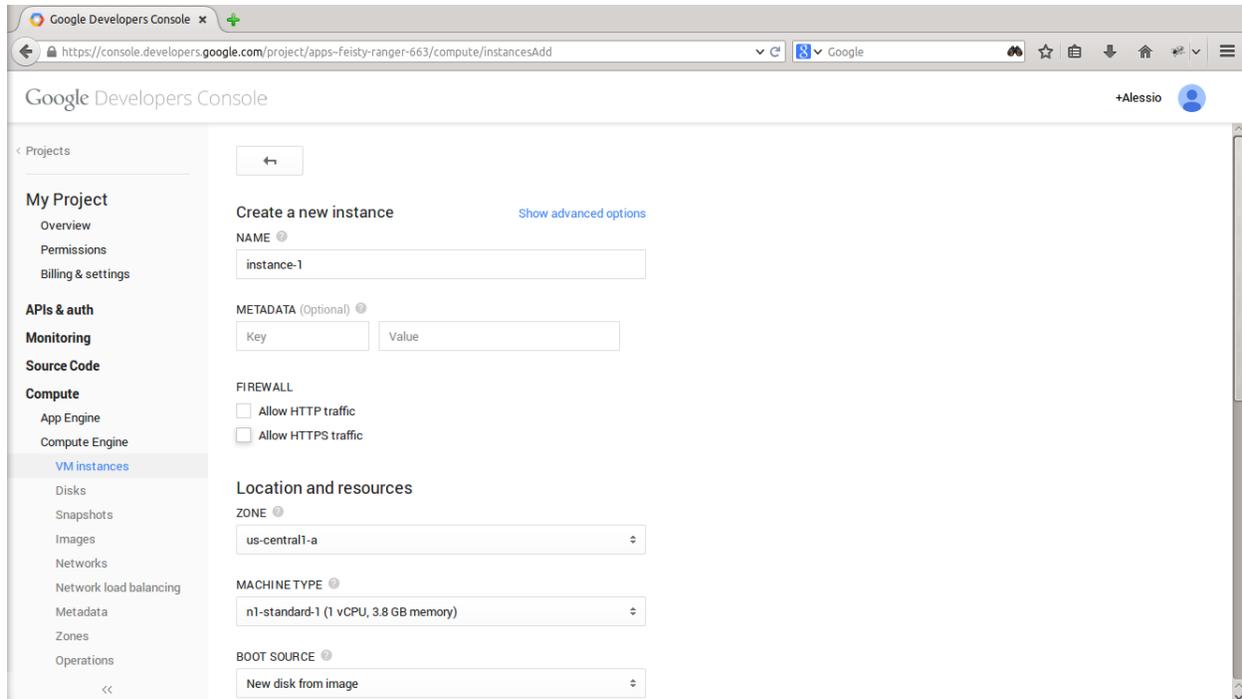
Set the VM's:

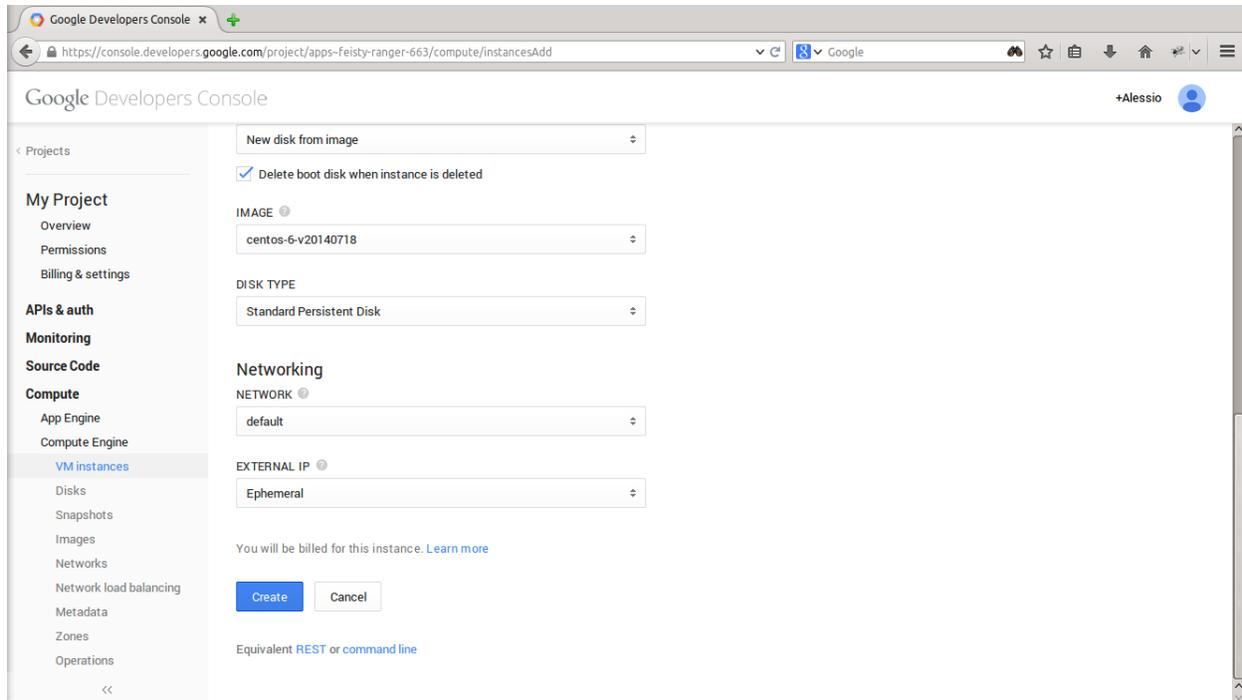
- name,

- meta keys (optional),
- zone,
- machine type,
- image.

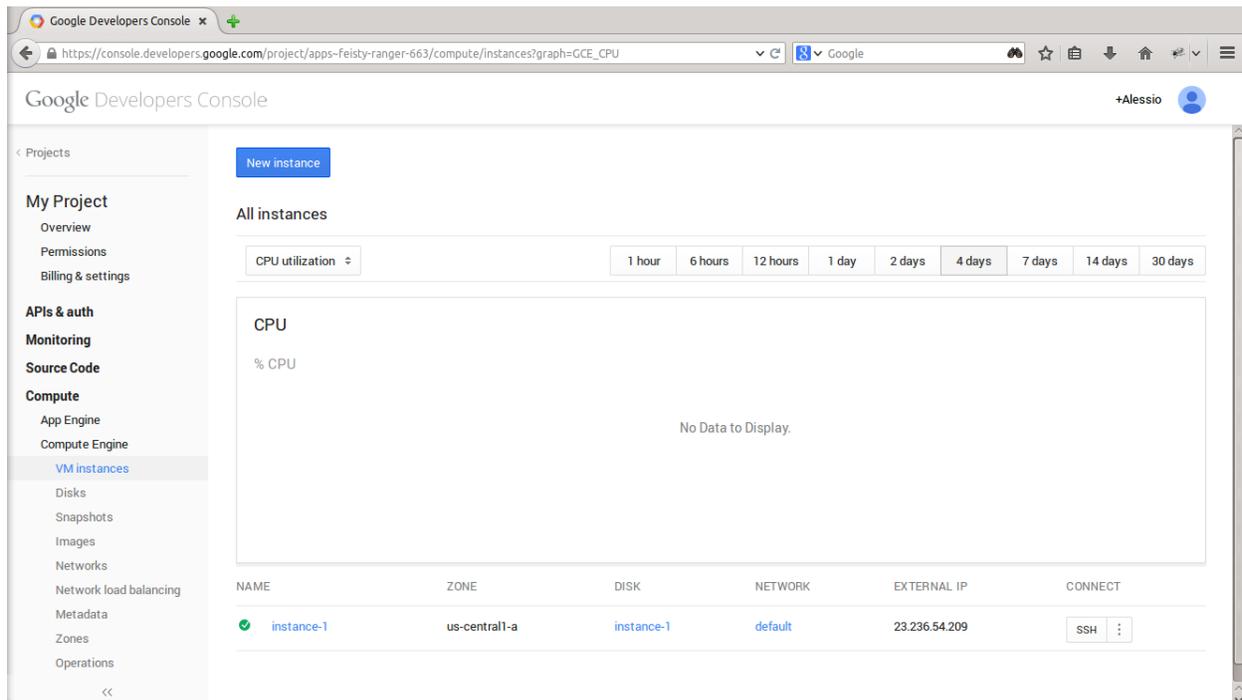
Then create the image.

In the bottom of the page, it is possible to convert the parameter chosen with the webpage to commands that can be used to perform the same action with gcloud or REST APIs.





After a few seconds, the new instance will be shown and will be accessible soon.



Instance Image Upload and Instance Creation by Command Line

Google Cloud allows the user to upload Virtual Machine images (compressed raw images) that can be run.

It is possible to set up a Virtual Machine image compatible with Google Cloud by following the instructions described here <https://developers.google.com/compute/docs/images> or, simply, by using the tool <https://github.com/GoogleCloudPlatform/compute-image-packages/tree/master/gcimagebundle>, that automatically creates an image of the Google Cloud disk, so that it can be customized and uploaded back.

Once the image is ready, the following steps will be required to upload it and put it in execution.

0) Setup the gcloud tools:

```
`curl https://sdk.cloud.google.com | bash`  
and restart the shell.
```

1) Authenticate:

With gcloud, the first step is to add a Google Cloud account.

```
`gcloud auth login <account name>`
```

This authentication method requires a web browser, so is simple and straightforward for a desktop user.

2) Create a bucket:

The first step for uploading a VM disk image is to create the space in Google Cloud Storage.

The bucket ownership has to be verifiable, so the bucket name must be equal to the DNS name of an owned domain.

In my case, the domain name of my website was "www.welovebinary.it", so I used

```
"gs://www.welovebinary.it/":
```

```
`gsutil mb gs://DOMAIN`
```

3) Upload the tar.gz:

Once you have a working image, it is possible to upload it:

```
`gsutil cp raw.image.tar.gz gs://DOMAIN`
```

4) Create the image from tar.gz:

```
`gcloud compute images create my-image-test --source-uri gs://DOMAIN`
```

If all worked, then the image should appear in the list

```
`gcloud compute images list`
```

5) Run a new VM instance:

```
`gcloud compute instances create my-instance-test --image my-image-test --zone us-central1-a`
```

6) Log into the image:

```
`gcloud compute ssh my-instance-test`
```

Microsoft Windows Azure Custom Data

As I discovered, there's no user metadata feature in Azure, equivalent to AWS or GCE, yet, but Azure has a similar feature for passing parameters to VMs, referred to as "custom data".

This mechanism basically passes the parameters to the VM by constructing a file (in initialization phase) in the VM's filesystem.

This file contains a base-64 encoded string and can be retrieved (in linux) in

```
"/var/lib/waagent/ovf-env.xml"
```

and one copy of this file is also created (documentations says, but in reality it is not) in

```
"/var/lib/waagent/CustomData"
```

One way to set the content of the file is through the command-line tool (`azure vm create`) by adding the "-d/--custom-data <custom-data-file>" parameter.

The ovf-env.xml file has the following format:

```
-----
```

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<Environment xmlns="http://schemas.dmtf.org/ovf/environment/1"
```

```
xmlns:oe="http://schemas.dmtf.org/ovf/environment/1"
```

```
xmlns:wa="http://schemas.microsoft.com/windowsazure"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
<wa:ProvisioningSection><wa:Version>1.0</wa:Version><LinuxProvisioningConfigurationSet  
t xmlns="http://schemas.microsoft.com/windowsazure"
```

```
xmlns:i="http://www.w3.org/2001/XMLSchema-
```

```
instance"><ConfigurationSetType>LinuxProvisioningConfiguration</ConfigurationSetType>
```

```
<HostName>myImage1</HostName><UserName>azureuser</UserName><UserPassword>alsCSF##1293
```

```
87</UserPassword><DisableSshPasswordAuthentication>false</DisableSshPasswordAuthentic
```

```
ation><CustomData>cHlvdnEgY3VzdC9tI CRhdGEgZnl sZQo=</CustomData></LinuxProvisioningCon  
figurationSet></wa:ProvisioningSection>
```

```
<wa:PlatformSettingsSection><wa:Version>1.0</wa:Version><PlatformSettings
```

```
xmlns="http://schemas.microsoft.com/windowsazure"
```

```
xmlns:i="http://www.w3.org/2001/XMLSchema-
```

```
instance"><KmsServerHostName>kms.core.windows.net</KmsServerHostName><ProvisionGuestA
```

```
gent>true</ ProvisionGuest Agent ><Guest Agent PackageName>Wn7_Wn8_IaaS_rd_art_stable_14
0703-
0050_Guest Agent Package. zip</ Guest Agent PackageName></ Plat formSet t i ngs></ wa: Plat formSet
t i ngs Sect i on>

</ Envi ronment >
```

From that file, it is possible to get back the "custom data" encapsulated in the <CustomData> tag and obtain the original text by base-64 decoding it.

The ONLY limitation on the "custom data" is the size (the maximum length of the binary array is 65535 bytes).

Google Cloud: Google Compute Engine in Detail

Accessing the REST API

I discovered an issue with the GCE's OAuth2 authentication method through the REST API. The issue has been accepted, but not yet solved, so, the only way of using the REST API with GCE is by authenticating with other approaches.

In the next steps I will use the Python API to authenticate, obtaining the access token needed by the REST API.

Prerequisites

Python 2.5, 2.6, or 2.7 is required.

Install the pip tool:

Download <https://bootstrap.pypa.io/get-pip.py>

And execute it, as superuser.

Install Google API Python client

```
# pip install --upgrade google-api-python-client
```

(<http://github.com/google/google-api-python-client>)

Install OAuth2 Python client

```
pip install --upgrade oauth2client
```

(<https://github.com/google/oauth2client>)

Authentication

As described in "<https://developers.google.com/compute/docs/api/python-guide>", it is possible to get authenticated with the Python API.

First of all, from the Google Cloud Console (<https://console.developers.google.com>), note your project id (next, will be called "MY_PROJECT_ID") access the project and, on the left column, reach

"APIs & auth -> Credentials"

create a new OAuth Client ID (Installed application -> Other).

Download the JSON file associated to the just created "Client ID for native application".

Save the JSON file as "client_secrets.json" in the folder you'll place the next script.

The following Python script returns the access token needed by the GCE REST API.

```
#!/usr/bin/env python
```

```
import json
```

```
from pprint import pprint
```

```
import logging
```

```
import sys
```

```
import argparse
```

```
import httplib2
```

```
from oauth2client.client import flow_from_clientsecrets
```

```
from oauth2client.file import Storage
```

```
from oauth2client import tools
```

```
from oauth2client.tools import run_flow
```

```

from api client . discovery import build

DEFAULT_ZONE = ' us - cent ral 1 - a '

API_VERSION = ' v1 '

GCE_URL = ' ht t ps : // www . googl eapi s . com / comput e / %s / pr oj ect s / ' % ( API_VERSION )

PROJECT_ID = ' MY_PROJECT_ID

CLIENT_SECRETS = ' client _ secrets . json '

OAUTH2_STORAGE = ' oaut h2aut h . dat '

GCE_SCOPE = ' ht t ps : // www . googl eapi s . com / aut h / comput e '

def main( argv ):

    logging . basicConf ig( level = logging . INFO )

    parser = argparse . Argument Parser (

        description = __doc __ ,

        formatter_class = argparse . RawDescriptionHelpFormatter ,

        parents = [ tools . argparse ] )

    # Parse the command-line flags .

    flags = parser . parse_args( argv[ 1 : ] )

    # Perform OAuth 2.0 authorization .

    flow = flow_from_client_secrets( CLIENT_SECRETS , scope = GCE_SCOPE )

```

```

storage = Storage(OAUTH2_STORAGE)
credentials = storage.get()

if credentials is None or credentials.invalid:
    credentials = run_flow(flow, storage, flags)

http = httplib2.Http()
auth_http = credentials.authorize(http)

storage_data = open(OAUTH2_STORAGE)
storageData = json.load(storage_data)
storage_data.close()

print storageData["access_token"]

if __name__ == '__main__':
    main(sys.argv)

```

Rest API

A simple REST call that returns the project's instances info contains is a GET request to:

```
"https://www.googleapis.com/compute/v1/projects/MY_PROJECT_ID/zones/us-central1-a/instances"
```

That contains in the header the following access token:

```
"Authorization: Bearer ACCESS_TOKEN_RETURNED_BY_PYTHON_SCRIPT"
```

In Curl, it's the following:

```
$ curl -H "Authorization: Bearer ACCESS_TOKEN_RETURNED_BY_PYTHON_SCRIPT" -X GET
"https://www.googleapis.com/compute/v1/projects/MY_PROJECT_ID/zones/us-central1-a/instances"
```

Replica Pools and Autoscaler

The Google Compute Engine Replica Pool provides the tools for managing a set of equivalent (homogeneous) virtual machine instances, all generated starting from a single configuration.

By combining the Replica Pool with Autoscaler, it is possible to dynamically adjust the pool size, based on specific pool metrics, like average CPU load, network transfers, I/O requests, etc..

Replica Pool is currently under development, so, the only way to use that is by being authorized to join the Limited Preview.

After joining, enable the Google Compute Engine and Autoscaler APIs from the Google Developer Console (APIs & auth -> APIs).

Replica Pools management

Because of the limited preview, also the gcloud tool instructions for managing the Replica Pools are hidden. Activate them with the following command:

```
$ gcloud components update preview
```

Now the tool is ready and keep in mind that for any command invocation, the following syntax has to be used:

```
$ gcloud preview replica-pools --zone ZONE COMMAND
```

Let's start creating a Replica Pool. The first step is to create a JSON template from which each replica will be generated. Let's call it example-template.json and fill it with

```
"template": {  
  "vmParameters": {  
    "machineType": "n1-standard-1",  
    "baseInstanceName": "my-replica",  
    "disksToCreate": [{  
      "boot": "true",  
      "initializeParameters": {  
        "sourceImage": "https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-7-wheezy-v20140828",  
        "diskSizeGb": "100"  
      }  
    }  
  }  
}
```

```

    }],
    "networkInterfaces": [ {
      "network": "default",
      "accessConfigs": [ {
        "type": "ONE_TO_ONE_NAT",
        "name": "External NAT"
      } ]
    } ]
  }
}

```

Other important settings that can be inserted in the JSON file are user metadata and explicit instructions that will be launched at instances' boot.

To create the Replica Pool, it is possible to use the following command

```

$ gcloud preview replica-pools --zone us-central1-a create example-pool \
    --size 3 --template example-template.json

```

To verify the correct creation of the Replica Pool the following command can be launched:

```

$ gcloud preview replica-pools --zone us-central1-a list

```

And, to see the list of replicas inside the pool, launch:

```

$ gcloud preview replica-pools --zone us-central1-a replicas --pool example-pool list

```

To delete the pool, simply launch:

```

$ gcloud preview replica-pools --zone us-central1-a delete example-pool

```

This command deletes all the instances of the pool. It is possible to choose the instances that should not be automatically deleted by writing their names at the bottom of the command:

```

$ gcloud preview replica-pools --zone ZONE delete example-pool my-replica-387m my-replica-1120e

```

Autoscaler management

Let's add a more interesting feature to the replica pool: the Autoscaler.

The Autoscaler basically handles a set of requirements and manages the number of replicas in the Replica Pool. The number of replicas is chosen as the minimum that respects the requirements.

Let's suppose we want to add an Autoscaler that:

- keeps the overall CPU load of the instances of the pool below the 80%;
- keeps the number of instances in the interval [1, 20];
- checks the metrics every 15 seconds.

Take the address of the replica pool you want to associate the Autoscaler:

```
$ gcloud preview replica-pools --zone us-central1-a list
```

Use that address as "--target" argument of the call:

```
$ gcloud preview autoscaler --zone us-central1-a create example-autoscaler --cool-down-period 15 \
    --max-num-replicas 20 --min-num-replicas 1 --target-cpu-utilization 0.8
    --target https://www.googleapis.com/replicapool/v1beta1/projects/feisty-ranger-663/zones/us-
central1-a/pools/example-pool
```

To delete the Autoscaler, just launch:

```
$ gcloud preview autoscaler --zone us-central1-a delete example-autoscaler
```

Metadata Management

Google Compute Engine allows to set user metadata non only at instances creation, but also when instances are running.

This is an easy way of sending parameters to virtual machines.

Setting Metadata

There are two possibilities for setting metadata:

- adding a metadata header to a file being uploaded to Google Cloud bucket (in the following examples, the bucket will be addressable as "gs://MYDOMAIN/");
- setting metadata to an already existing object.

Let's follow the step for accomplishing the first step. Let's suppose that we want to upload `my_file`, with some metadata. To add the simple metadata header, it's possible to launch the command

```
$ gsutil -h "Content-Type:text/html" -h "Cache-Control:public, max-age=3600" cp -r my_file
gs://DOMAIN
```

In the second case, for example, to set "`my_file`"'s metadata, it is possible to use the "`setmeta`" command:

```
$ gsutil setmeta -h "Content-Type:text/html" -h "Cache-Control:public, max-age=3600" \  
    -h "Content-Disposition" gs://MYDOMAIN/my_file
```

To read the uploaded file, an easy way is to launch

```
`$ gsutil cat gs://MYDOMAIN/my_file`
```

Getting Metadata

To get the header metadata associated to "my_file", can be launched the following command:

```
$ gsutil ls -L gs://MYDOMAIN/my_file
```

Only from the virtual machine instance, is possible to get the metadata associated to the instance itself. It is done by querying the link "<http://metadata.google.internal/computeMetadata/v1/XXX>", and adding "Metadata-Flavor: Google" to the request header.

The address that ends with '/' is a directory, otherwise is an entry point.

The command for getting metadata will be similar to the following:

```
$ curl "http://metadata.google.internal/computeMetadata/v1/instance/disks/" \  
    -H "Metadata-Flavor: Google"
```

The CUSTOM metadata, in particular, is stored in "attributes" folder:

```
"http://metadata.google.internal/computeMetadata/v1/<instance|project>/attributes/"
```

Ceph Project at FNAL

Firstname Lastname

October 16, 2014

1 INTRODUCTION TO CEPH

Ceph is a open source storage platform designed to present object, block, and file storage from a single distributed computer cluster. Ceph's main goals are to be completely distributed without a single point of failure, scalable to the exabyte level, and freely-available. The data is replicated, making it fault tolerant. Ceph software runs on commodity hardware. The system is designed to be both self-healing and self-managing and strives to reduce both administrator and budget overhead.

Ceph's software libraries provide client applications with direct access to the reliable autonomic distributed object store (RADOS) object-based storage system, and also provide a foundation for some of Ceph's features, including RADOS Block Device (RBD), RADOS Gateway, and the Ceph File System.

1.1 CEPH STORAGE CLUSTER

A Ceph Storage Cluster requires at least one Ceph Monitor and at least two Ceph OSD Daemons. The Ceph Metadata Server is essential when running Ceph Filesystem clients.

- Ceph OSDs: A Ceph OSD Deamon stores data, handles data replication, recovery, backfilling, rebalancing, and provides some monitoring information to Ceph Monitors by checking other Ceph OSD Daemons for a heartbeat. A Ceph Storage Cluster requires at least two Ceph OSD Daemons to achieve an active + clean state when the cluster makes two copies of your data.
- Monitors: A Ceph Monitor maintains maps of the cluster state, including the monitor map, the OSD map, the Placement Group (PG) map, and the CRUSH map. Ceph maintains a history (called an "epoch") of each state change in the Ceph Monitors, Ceph OSD Daemons, and PGs.
- MDSs: A Ceph Metadata Server (MDS) stores metadata on behalf of the Ceph Filesystem (i.e., Ceph Block Devices and Ceph Object Storage do not use MDS). Ceph Metadata Servers

make it feasible for POSIX file system users to execute basic commands like ls, find, etc. without placing an enormous burden on the Ceph Storage Cluster.

2 CEPH STORAGE CLUSTER AT FNAL

The Ceph Storage Cluster at FNAL consists of 3 Monitors, 8 OSDs and 1 MDS. There are four hosts in the cluster, each of them has 8 cores, 16GB RAM, 15TB HDD, and they are connected by 10GB Ethernet. As shown in figure 1, in each host machine, there reside 1 MON(MDS) and two OSDs. Since MON(MDS) require little cpu or storage resources, they won't affect the performance of OSD that reside in the same host.

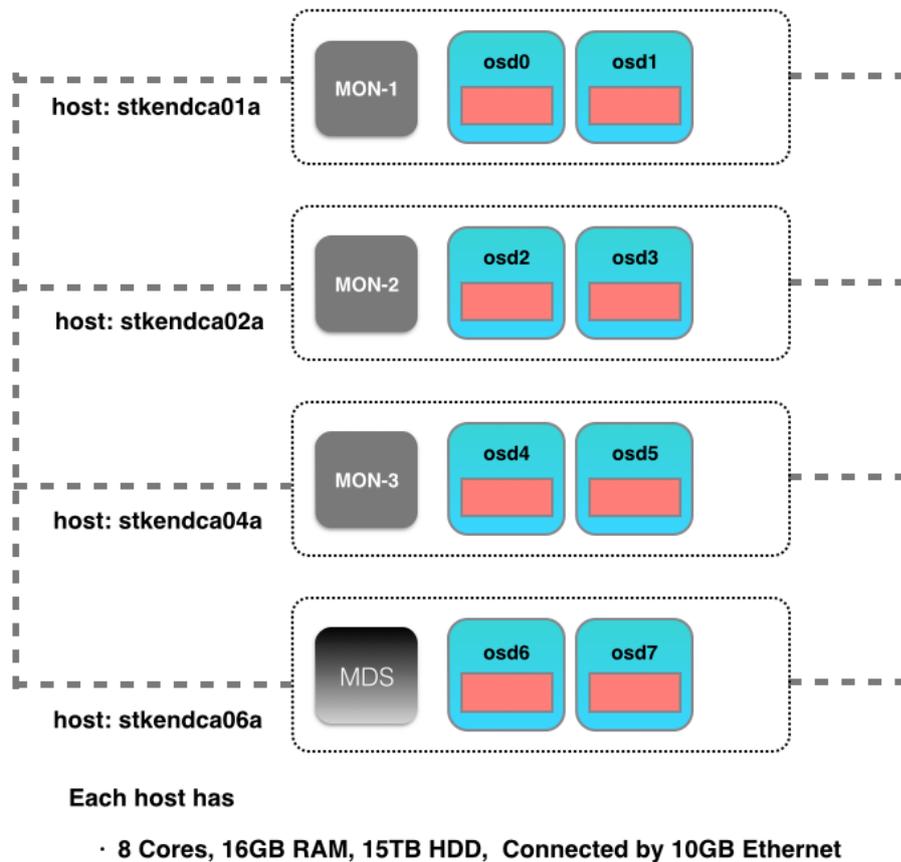


Figure 1: Ceph Storage Cluster at FNAL–Stage 1. Each OSD contributes about 7TB storage capacity to the ceph storage cluster.

3 CEPH INSTALLATION AND OPERATION AT FNAL

The Ceph storage cluster at FNAL is implemented and operated by using an automatic tool **ceph-deploy**, which can also automatically add OSD, monitor node, metadata server node and

Table 1: iozone test results of 10 VMs

| I/O rate (kB/sec) | Max | Min | Avg | Total |
|-------------------|---------|---------|----------|-----------|
| Writer | 1208.32 | 1404.13 | 1321.477 | 66073.85 |
| Re-writer | 1288.06 | 1470.5 | 1363.449 | 68172.45 |
| Reader | 2886.46 | 4126.75 | 3672.29 | 183614.5 |
| Re-reader | 3868.67 | 5290.6 | 4386.161 | 219308.05 |

ceph-client node. **ceph-deploy** is installed on host **stkendca01a**. It only need to be installed on one machine. To use **ceph-deploy**, user need to log in **stkendca01a** as root and work under the directory */home/ceph/ceph-cluster*.

4 EVALUATION RESULTS

We have conducted three sets of experiments to evaluate the performance of Ceph storage cluster. In our first experiment, we use 10 VMs as ceph-client to create file system using rbd block device image. Then we conduct iozone tests with these 10 VMs to analysis the I/O performance of Ceph Storage Cluster. In our second experiment, we replace the 10 VMs with 10 physical machines to see the best I/O performance we can get from Ceph Storage Cluster without the virtualization layer. In the third experiment, we try to export 3 copies of rbd block device images on each physical machine simultaneously.

4.1 IOZONE TEST WITH VMs

In this experiment, we use 10 VMs that been added in the Ceph storage cluster as clients. Each of them creates a 60GB rbd block device image and map the image to its block device. Then, the VM will use the block device by creating a file system and mount the file system on itself.

The iozone test start with 10 VMs and each VM initiates 5 process perform 10GB I/O operations write/re-write, read/re-read to the Ceph storage cluster simultaneously. The results are presented in table 1.

4.2 IOZONE TEST WITH PHYSICAL MACHINES

The prepare work is the same as in 4.1. The iozone test start with 10 physical machines and each initiates 5 process perform 10GB I/O operations write/re-write, read/re-read to the Ceph storage cluster simultaneously. The results are presented in table 2.

Table 2: iozone test results of 10 physical machines

| I/O rate (kB/sec) | Max | Min | Avg | Total |
|-------------------|---------|---------|----------|-----------|
| Writer | 1244.13 | 1489.29 | 1387.963 | 62582.35 |
| Re-writer | 1246.1 | 1636.93 | 1378.424 | 60938.1 |
| Reader | 1980.19 | 2407.94 | 2196.304 | 99310.25 |
| Re-reader | 2234.9 | 3172.75 | 2553.519 | 116336.95 |

4.3 RBD IMAGE EXPORT TEST WITH PHYSICAL MACHINES

The rbd image export tests are conducted by making 10 physical machines export 3 copies of their own rbd image simultaneously. The average bandwidth performance of in this test is 139.78MB/s.

Understanding the Performance and Potential of Cloud Computing for Scientific Applications

Iman Sadooghi, Jesús Hernández Martín, Tonglin Li, Kevin Brandstatter, Ketan Maheshwari, Tiago Pais Pitta de Lacerda Ruivo, Gabriele Garzoglio, Steven Timm, Yong Zhao, Ioan Raicu

Abstract— Commercial clouds bring a great opportunity to the scientific computing area. Scientific applications usually require significant resources, however not all scientists have access to sufficient high-end computing systems, many of which can be found in the Top500 list. Cloud Computing has gained the attention of scientists as a competitive resource to run HPC applications at a potentially lower cost. But as a different infrastructure, it is unclear whether clouds are capable of running scientific applications with a reasonable performance per money spent. This work studies the performance of public clouds and places this performance in context to price. We evaluate the raw performance of different services of AWS cloud in terms of the basic resources, such as compute, memory, network and I/O. We also evaluate the performance of the scientific applications running in the cloud. This paper aims to assess the ability of the cloud to perform well, as well as to evaluate the cost of the cloud running scientific applications. We developed a full set of metrics and conducted a comprehensive performance evaluation over the Amazon cloud. We evaluated EC2, S3, EBS and DynamoDB among the many Amazon AWS services. We evaluated the memory sub-system performance with *CacheBench*, the network performance with *iperf*, processor and network performance with the HPL benchmark application, and shared storage with NFS and PVFS in addition to S3. We also evaluated a real scientific computing application through the Swift parallel scripting system at scale. Armed with both detailed benchmarks to gauge expected performance and a detailed monetary cost analysis, we expect this paper will be a recipe cookbook for scientists to help them decide where to deploy and run their scientific applications between public clouds, private clouds, or hybrid clouds.

Index Terms— Cloud computing, Amazon AWS, performance, cloud costs, scientific computing



1 INTRODUCTION

THE idea of using clouds for scientific applications has been around for several years, but it has not gained traction primarily due to many issues such as lower network bandwidth or poor and unstable performance. Scientific applications often rely on access to large legacy data sets and pre-tuned application software libraries. These applications today run in HPC environments with low latency interconnect and rely on parallel file systems. They often require high performance systems that have high I/O and network bandwidth. Using commercial clouds gives scientists opportunity to use the larger resources on-demand. However, there is an uncertainty about the capability and performance of clouds to run scientific applications because of their different nature. Clouds have a heterogeneous infrastructure compared

with homogenous high-end computing systems (e.g. supercomputers). The design goal of the clouds was to provide shared resources to multi-tenants and optimize the cost and efficiency. On the other hand, supercomputers are designed to optimize the performance and minimize latency.

However, clouds have some benefits over supercomputers. They offer more flexibility in their environment. Scientific applications often have dependencies on unique libraries and platforms. It is difficult to run these applications on supercomputers that have shared resources with pre-determined software stack and platform, while cloud environments also have the ability to set up a customized virtual machine image with specific platform and user libraries. This makes it very easy for legacy applications that require certain specifications to be able to run. Setting up cloud environments is significantly easier compared to supercomputers, as users often only need to set up a virtual machine once and deploy it on multiple instances. Furthermore, with virtual machines, users have no issues with custom kernels and root permissions (within the virtual machine), both significant issues in non-virtualized high-end computing systems.

- I. Sadooghi, J. H. Martin, T. P. P. de Lacerda Ruivo, T. Li, K. Brandstatter and I. Raicu are with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: isadoogh@iit.edu, lher-na22@hawk.iit.edu, tonglin@iit.edu, kbrandst@iit.edu, iraicu@cs.iit.edu
- Y. Zhao is with School of Computer Science and Engineering, Univ. of Electronic Science and Technology of China, Chengdu, China E-mail: yongzh04@gmail.com
- K. Maheshwari is with the Argonne National Laboratory Lemont, IL 60439. E-mail: ketan@mcs.anl.gov
- G. Garzoglio and S. Timm are with the Fermi National Accelerator Laboratory, PO Box 500, Batavia, IL 60510. E-mail: timmm@fnal.gov, garzogli@fnal.gov

There are some other issues with clouds that make them challenging to be used for scientific computing. The network bandwidth in commercial clouds is significantly lower (and less predictable) than what is available in supercomputers. Network bandwidth and latency are two of the major issues that cloud environments have for high-performance computing. Most of the cloud resources use commodity network with significantly lower bandwidth than supercomputers [13].

The virtualization overhead is also another issue that leads to variable compute and memory performance. I/O is yet another factor that has been one of the main issues on application performance. Over the last decade the compute performance of cutting edge systems has improved in much faster speed than their storage and I/O performance. I/O on parallel computers has always been slow compared with computation and communication. This remains to be an issue for the cloud environment as well.

Finally, the performance of parallel systems including networked storage systems such as Amazon S3 needs to be evaluated in order to verify if they are capable of running scientific applications [3]. All of the above mentioned issues raise uncertainty for the ability of clouds to effectively support HPC applications. Thus it is important to study the capability and performance of clouds in support of scientific applications. Although there have been early endeavors in this aspect [10][14][16][20][23], we develop a more comprehensive set of evaluation and metrics. In some of these works, the experiments were mostly run on limited types and number of instances [14][16][17]. Only a few of the researches have used the new Amazon EC2 cluster instances that we have tested [10][20]. However the performance metrics in those papers are very limited. This paper covers a thorough evaluation covering major performance metrics and compares a much larger set of EC2 instance types and the commonly used Amazon Cloud Services. Most of the aforementioned above mentioned works lack the cost evaluation and analysis of the cloud. Our work analyses the cost of the cloud on different instance types.

The main goal of this research is to evaluate the performance of the Amazon public cloud as the most popular commercial cloud available, as well as to offer some context for comparison against a private cloud solution. We run micro benchmarks and real applications on Amazon EC2 and S3 to evaluate its performance on critical metrics including throughput, bandwidth and latency of processor, network, memory and storage [2]. Then, we evaluate the performance of HPC applications on EC2 and compare it with a private cloud solution (FermiCloud 0). We also identify the weaknesses and advantages of the cloud environment in the scientific computing area.

Finally, this work performs a detailed price/cost analysis of cloud instances to better understand the upper and lower bounds of cloud costs. Armed with both detailed benchmarks to gauge expected performance and a detailed monetary cost analysis, we expect *this paper will be a recipe cookbook for scientists to help them decide*

where to deploy and run their scientific applications between public clouds, private clouds, or hybrid clouds.

This paper is organized as follows: Section 2 provides the evaluation of the EC2, S3 and DynamoDB performance on different service alternatives of Amazon AWS. We provide an evaluation methodology. Then we present the benchmarking tools and the environment settings of the testbed in this project. Section 2.4 presents the benchmarking results and analyzes the performance. On 2.5 we compare the performance of EC2 with FermiCloud on HPL application. Section 3 analyzes the cost of the EC2 cloud based on its performance on different aspects. In section 4, we review the related work in this area. Section 5 draws conclusion and discusses future work.

2 PERFORMANCE EVALUATION

In this section we provide a comprehensive evaluation of the Amazon AWS technologies. We evaluate the performance of Amazon EC2 and storage services such as S3 and EBS. We also compare the Amazon AWS public cloud to the FermiCloud private cloud.

2.1 Methodology

We design a performance evaluation method to measure the capability of different instance types of Amazon EC2 cloud and to evaluate the cost of cloud computing for scientific computing. As mentioned, the goal is to evaluate the performance of the EC2 on scientific applications. To achieve this goal, we first measure the raw performance of EC2. We run micro benchmarks to measure the raw performance of different instance types, compared with the theoretical performance peak claimed by the resource provider. We also compare the actual performance with a typical non-virtualized system to better understand the effect of virtualization. Having the raw performance we will be able to predict the performance of different applications based on their requirements on different metrics. Then we compare the performance of a virtual cluster of multiple instances running HPL application on both Amazon EC2 and the FermiCloud. Comparing the performance of EC2, which we don't have much information about its underlying resources with the FermiCloud, which we know the details about, we will be able to come up with a better conclusion about the weaknesses of the EC2. On the following sections we try to evaluate the performance of the other popular services of Amazon AWS by comparing them to the similar open source services.

Finally, we analyze the cost of the cloud computing based on different performance metrics from the previous part. Using the actual performance results provides more accurate analysis of the cost of cloud computing while being used in different scenarios and for different purposes.

The performance metrics for the experiments are based on the critical requirements of scientific applications. Different scientific applications have different priorities. We need to know about the compute performance of the instances in case of running compute intensive applications.

We also need to measure the memory performance, as memory is usually being heavily used by scientific applications. We also measure the network performance which is an important factor on the performance of scientific applications.

2.2 Benchmarking tools and applications

It is important for us to use wide-spread benchmarking tools that are used by the scientific community. Specifically in Cloud Computing area, the benchmarks should have the ability to run over multiple machines and provide accurate aggregate results.

For memory we use CacheBench. We perform read and write benchmarks on single instances. For network bandwidth, we use Iperf [4]. For network latency and hop distance between the instances, we use ping and traceroute. For CPU benchmarking we have chosen HPL benchmark [5]. It provides the results in floating-point operations per second (FLOPS).

In order to benchmark S3, we had to develop our own benchmark suite, since none of the widespread benchmarking tools can be used to test storage like this. We have also developed a tool for configuring a fully working virtual cluster with support for some specific file systems.

2.3 Parameter space and testbed

In order to better show the capability of Amazon EC2 on running scientific applications we have used two different cloud infrastructures: (1) Amazon AWS Cloud, and (2) FermiCloud. Amazon AWS is a public cloud with many datacenters all around the world. FermiCloud is a private Cloud which is used for internal use in Fermi National Laboratory.

In order to compare the virtualization effect on the performance we have also included two local systems on our tests: (1) A 6-core CPU and 16 Gigabytes of memory system (DataSys), and (2) a 48-cores and 256 Gigabytes memory system (Fusion).

2.3.1 Amazon EC2

The experiments were executed on three Amazon cloud data centers: US East (Northern Virginia), US West (Oregon) and US West (Northern California). We cover all of the different instance types in our evaluations.

The operating system on all of the US West instances and the local systems is a 64bits distribution of Ubuntu. The US East instances use 64 bits CentOS operating system. The US West instances use Para-virtualization technique on their hypervisor. But the HPC instances on the US East cloud center use Hardware-Assisted Virtualization (HVM) [7]. HVM techniques use the features of the new hardware to avoid handling all of the virtualization tasks like context switching or providing direct access to different devices at the software level. Using HVM, Virtual Machines can have direct access to hardware with the minimal overhead.

There is no information about the underlying architecture and technologies of Amazon AWS publicly available.

2.3.2 FermiCloud

FermiCloud is a private cloud providing Infrastructure-as-a-Service services internal use. It manages dynamically allocated services for both interactive and batch processing. As part of a national laboratory, one of the main goals FermiCloud is being able to run scientific applications and models. FermiCloud uses OpenNebula Cloud Manager for the purpose of managing and launching the Virtual Machines. It uses KVM hypervisor that uses both paravirtualization and full virtualization techniques. The FermiCloud Infrastructure is enabled with 4X DDR Infiniband network adapters. The main challenge to overcome in the deployment of the network is introduced when virtualizing the hardware of a machine to be used (and shared) by the VMs. This overhead slows drastically the data rate reducing the efficiency of using a faster technology like Infiniband. To overcome the virtualization overhead they use a technique called Single Root Input/Output Virtualization (SRIOV) that achieves device virtualization without using device emulation by enabling a device to be shared by multiple virtual machines. The technique involves with modifications to the Linux's Hypervisor as well as the OpenNebula manager.

Each server is enabled with a 4x (4 links) Infiniband card with a DDR data rate for a total theoretical speed of up to 20 Gb/s and after the 8b/10b codification 16 Gb/s. Network latency is 1 μ s when used with MPI [6]. Each card has 8 virtual lanes that can create 1 physical function and 7 virtual functions via SR-IOV. The servers are enabled with 2 quad core 2.66 GHz Intel processors, 48Gb of RAM and 600Gb of SAS Disk, 12TB of SATA, and 8 port RAID Controller.

2.4 Performance Evaluation of AWS

2.4.1 Memory hierarchy performance

This section presents the memory benchmark results. We sufficed to run read and write benchmarks. The experiments for each instance were repeated three times.

Memory bandwidth is a critical factor in scientific applications performance. Many Scientific applications like GAMESS, IMPACT-T and MILC are very sensitive to memory bandwidth [8]. Amazon has not included the memory bandwidth of the instances. It has only listed their memory size. We also measure the memory bandwidth of each instance.

Fig. 1 shows the system memory read bandwidth in different memory hierarchy levels. The vertical axis shows the cache size. The bandwidth is very stable up to a certain cache size. The bandwidth starts to drop after a certain size. The reason for the drop off is surpassing the memory cache size at a certain hierarchy level.

As shown in the figure, the memory performance of the m1.small instance is significantly lower than other instances. The low memory bandwidth cannot be only attributed to the virtualization overhead. We believe that the main reason for that is memory throttling imposed by EC2 based on the SLA of those instances.

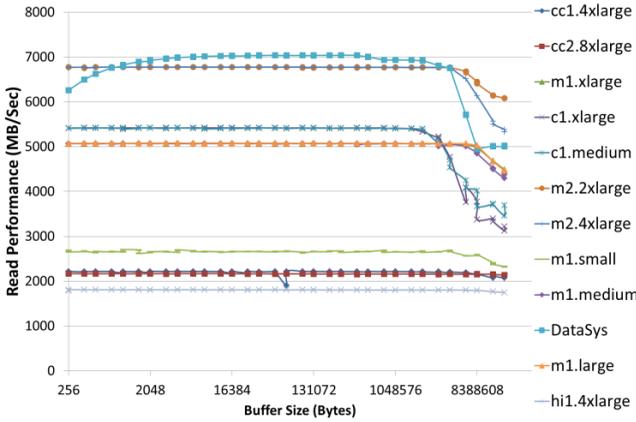


Fig. 1. CacheBench Read benchmark results, one benchmark process per instance

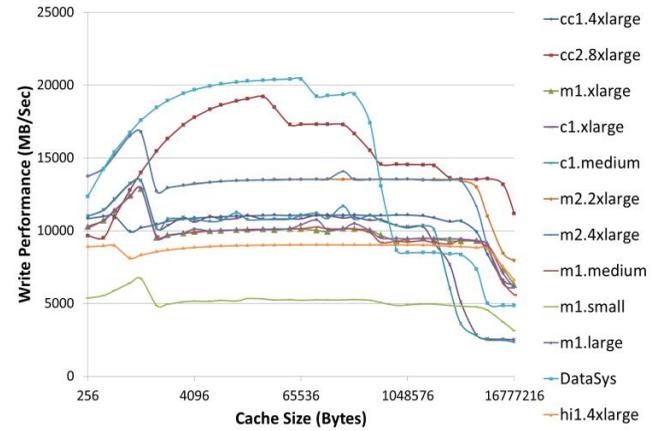


Fig. 2. CacheBench write benchmark results, one benchmark process per instance

Another noticeable point is the low bandwidth of the cc1.4xlarge, cc2.8xlarge and hi1.4xlarge. These instances have similar performance that is much lower than normal instances. A reason for that can be the result of the different virtual memory allocation on the VMs by HVM virtualization on these instances. We have however observed an effect in large hardware-assisted virtual machines such as those on FermiCloud that it will take a while for the system to balloon the memory out to its full size at the first launch of the VM.

After all, the results show that the memory bandwidth for read operation in the larger instances is close to the local non-virtualized system. *We can conclude that the virtualization effect on the memory is low, which is a good sign for scientific applications that are mostly sensitive to the memory performance.*

Fig. 2 shows the write performance of different cloud instances and the local system. The write performance shows different results from the read benchmark. As in write, the cc2.8xlarge instance has the best performance next to the non-virtualized local system.

For each instance we can notice two or three major drop-offs in bandwidth. These drop-offs show different memory hierarchies. For example on the cc2.8xlarge instance we can notice that the memory bandwidth drops at 24 Kbytes. We can also observe that the write throughputs for different memory hierarchies are different. These data points likely represent the different caches on the processor (e.g. L1, L2, L3 caches).

Comparing the cluster instance with the local system, we observe that on smaller buffer sizes, the local system performs better. But cloud instance outperforms the local system on larger cache sizes. The reason for that could be the cloud instances residing on more powerful physical nodes with higher bandwidths. We can observe that the write bandwidth on the cloud instances drops off at certain buffer sizes. That shows the memory hierarchy effects on the write operation.

Users can choose the best transfer size for write operation based on the performance peaks of each instance type to get the best performance. This would optimize a scientific application write bandwidth.

2.4.2 Network performance

We have run many experiments on network performance of Amazon cloud. The experiments test the network performance including bandwidth and latency. We also test wide area network bandwidth of the instances.

We first test the local network bandwidth between the same types of instances. Fig. 3 shows the network performance of different types of nodes. In each case both of the instances were inside the same datacenter. The network bandwidth for most of the instances were as expected except for two instances.

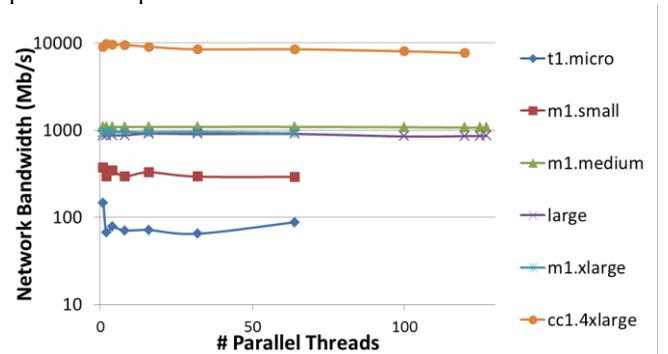


Fig. 3. iPerf benchmark results. Network bandwidth in a single client and server connection, internal network.

The lowest performance belongs to the t1.micro and m1.small instances. These two instances use the same 1 Gb/s network cards used by other instances. But they have much lower bandwidth. We believe that the reason is sharing the CPU cores and not having a dedicated core. This can affect network performance significantly as the CPU is shared and many network requests cannot be handled while the instance is on its idle time. During the idle time of the instance, the virtual system calls to the VMM will not be processed and will be saved in the queue until the idle time is over. The network performance is highly affected by processor sharing techniques. Other works had the same observations and conclusions about the network performance in these two instance types [9]. Another reason for the low performance of the m1.small and t1.micro instances could be throttling the

network bandwidth by EC2. The Xen hypervisor has the ability of network throttling if needed.

Among the instances that use the slower network cards the m1.medium instance has the best performance. We did not find a technical reason for that. The m1.medium instances use the same network card as other instances do and don't have any advantage on system configuration over other instance types. We assume the reason for that is the administrative decision on hypervisor level due to their popularity among different instance types.

Another odd result is for m1.medium instance. The bandwidth in medium instance exceeds 1 Gb/Sec, which is the specified network bandwidth of these. m1.medium instance bandwidth achieves up to 1.09 Gb/sec. That is theoretically not possible for a connection between two physical nodes with 1 Gb/s network cards. We believe the reason is that both of the VMs reside in the same physical node or the same cluster. In case of residing on the same node, the packets stay in the memory. Therefore the connection bandwidth is not limited to the network bandwidth. We can also assume that not necessarily the instances have 1 Gb/s network cards. In fact the nodes that run medium instances may have more powerful network cards in order to provide better network performance for these popular instances.

The HPC instances have the best network bandwidth among the instances. They use 10 Gb/sec network switches. The results show that the network virtualization overhead in these instances is very low. The performance gets as high as 95% of ideal performance.

We also measure the network connection latency and the hop distance between instances inside the Oregon datacenter of Amazon EC2. We run this experiment to find out about the correlation of connection latency and the hop distance. We also want to find the connection latency range inside a datacenter. We measure the latency and the hop distance on 1225 combinations of m1.small instances. Fig. 4 shows the network latency distribution of EC2 m1.small instances. It also plots the hop distance of two instances. The network latency in this experiment varies between 0.006 ms and 394 ms, an arguably very large variation.

We can observe from the results that: (1) 99% of the instances which have the transmission latency of 0.24 to 0.99 ms are 4 or 6 hops far from each other. So we can claim that if the latency is between 0.24 to 0.99 ms the distance between the instances is 4 to 6 hops with the probability of 99%. (2) More than 94% of the allocated instances to a user are 4-6 percent far from each other. In other words the hop distance is 4-6 instances with the probability of more than 94%.

We can predict the connection latency based on the hop distance of instances. We have run the latency test for other instance types. The results do not seem to be dependent on instance type for the instances with the same network interconnect. *The latency variance of Amazon instances is much higher than the variance in a HPC system. The high latency variance is not desirable for scientific applications. In case of HPC instances which have the 10 Gigabit Ethernet*

cards, the latency ranges from 0.19ms to 0.255ms which shows a smaller variance and more stable network performance.

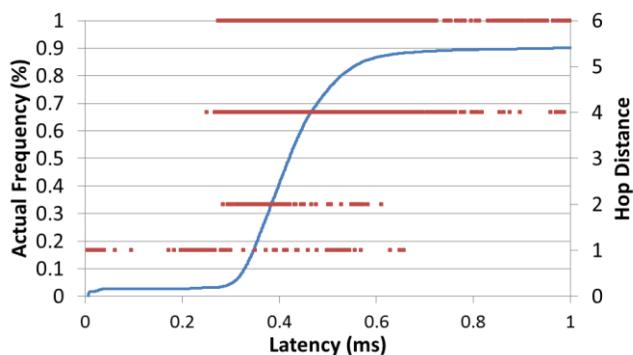


Fig. 4. Cumulative Distribution Function and Hop distance of connection latency between instances inside a datacenter.

Other researches have compared the latency of EC2 HPC instances with HPC systems. The latency of the EC2 HPC instance is reported to be 3 to 40 times higher than a HPC system with 23 Gb/s network cards [10]. The latency variance is also much higher.

2.4.3 Compute Performance

In this section we evaluate the compute performance of EC2 instances. Fig. 5 shows the compute performance of each instance using HPL as well as the ideal performance claimed by Amazon. It also shows the performance variance of instances.

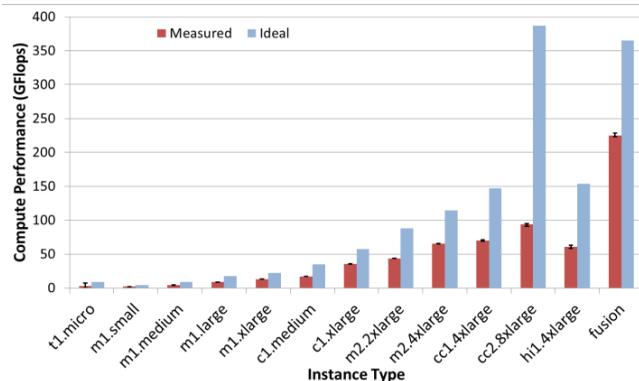


Fig. 5. HPL benchmark results: compute performance of single instances comparing with their ideal performance.

Among the Amazon instances, the cc2.8xlarge has the best compute performance. The t1.micro instance shows the lowest performance. The figure also shows the performance variance for each instance. The performance variance of the instances is low in most of the instance types. Providing a consistent performance is an advantage for cloud instances.

Among all of the instances and local nodes, the best efficiency belongs to the non-virtualized system. Overall we can observe that the efficiency of the instances is relatively low. Other papers have suggested the low performance of HPL application while running on virtualized environments [11][14]. Although the cc2.8xlarge instance has the largest compute capacity among the instances, it is the most inefficient instance. The reason for that lies behind the number of the cores in this instance. cc2.8xlarge has 16 cores. The expected performance is the aggregate perfor-

mance of all of the cores of the instance. But the real performance is lower because of the communication overhead of 16 cores which is caused by the MPI application. Other papers have also reported the poor MPI performance on EC2 cloud [15][16]. Other papers have also reported the poor MPI performance on EC2 cloud [15][16].

2.4.4 I/O Performance

In this section we evaluate the I/O performance of the EBS volume and local storage of each instance. The following charts show the results obtained after running IOR on the local storage and EBS volume storage of each of the instances with different transfer sizes and storage devices. Fig. 6 shows the performance of POSIX read operation on different instances. Except for the hi1.4xlarge, which is equipped with SSDs, the throughput among other instances does not vary greatly from one another. For most of the instances the throughput is close to a non-virtualized system with a normal spinning HDD.

Fig. 7 shows the maximum write and read throughput on each instance on both EBS volumes and local storage devices. Comparing with local storage, EBS volumes show a very poor performance, which is the result of the remote access delay over the network.

Finally, to complete these micro-benchmarks, we set up a software RAID-0 with EBS volumes, varying the number of volumes from 1 to 8. We ran the same benchmark on a c1.medium instance. Fig. 8 shows the write performance on RAID0 on different number of EBS volumes.

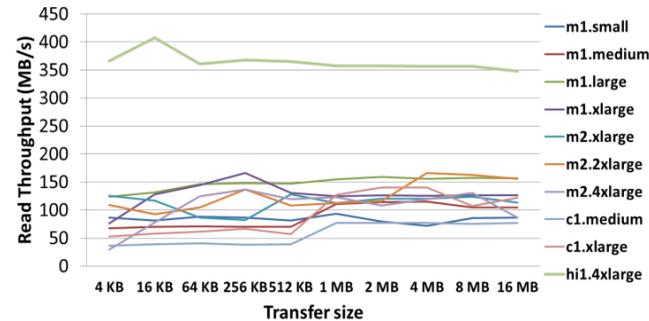


Fig. 6. Local POSIX read benchmark results on all instances

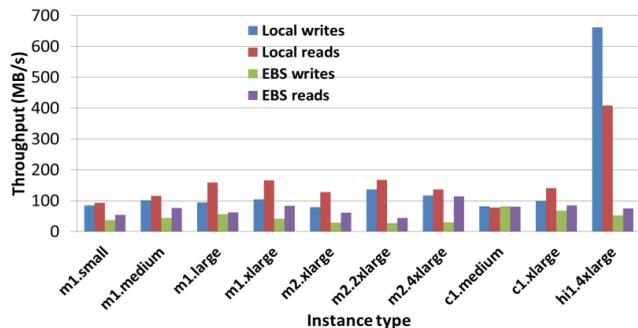


Fig. 7. Maximum write/read throughput on different instances

Looking at the write throughput, we can observe that the throughput does not vary a lot and is almost constant as the transfer size increases. That shows a stable write throughput on EBS drives. The write throughput on the RAID 0 increases with the number of drives. The reason

for that is that the data will be spread among the drives and is written in parallel to all of the drives. That increases the write throughput because of having parallel write instead of serial write. Oddly, the performance does not improve as the number of drives increases from 1 to 2 drives. The reason for that is moving from the local writes to network. Therefore the throughput stays the same. For 4 EBS volumes, we can observe a 4x increase on the throughput. In case of 8 EBS volumes we expect a 2x speed up comparing with the 4 EBS experiment. However the write throughput can not scale better because of the limitation of the network bandwidth. The maximum achievable throughput is around 120MB/s, which is bound to the network bandwidth of the instances that is 1 Gb/s. so we can conclude that the RAID throughput will not exceed 120 MB/s if we add more EBS volumes.

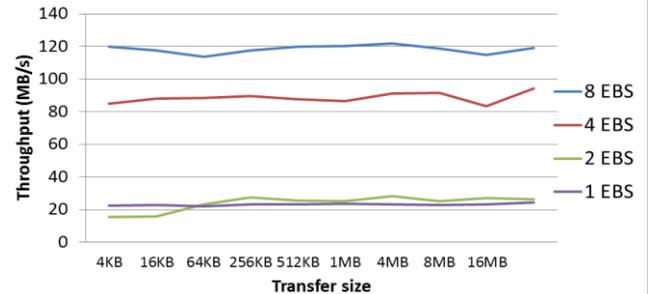


Fig. 8. RAID0 Setup benchmark for different transfer sizes - write

2.4.5 S3 and PVFS Performance

In this section we evaluate and compare the performance of S3 and PVFS. S3 is a highly scalable storage service from Amazon that could be used on multinode applications. Also, a very important requirement for most of the scientific applications is a parallel file system shared among all of the computing nodes. We have also included the NFS as a centralized file system to show how it performs on smaller scales.

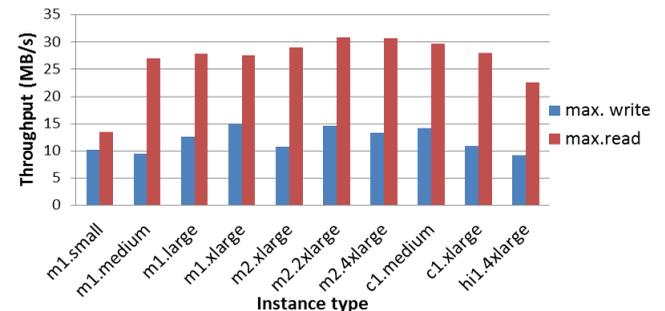


Fig. 9. S3 performance, maximum read and write throughput

First we evaluate the s3 performance on read and write operations. Fig. 9 shows the maximum read and write throughput on S3 accessed by different instance types. Leaving aside the small instances, there is not much difference between the maximum read/write throughput across instances. The reason is that these values are implicitly limited by either the network capabilities or S3 itself.

Next, We compare the performance of the S3 and PVFS as two possible options to use for scientific applications.

PVFS is commonly used in scientific applications on HPC environments. On the other hand, S3 is commonly used on the multinode applications that run on cloud environment. We have only included the read performance in this paper. The experiment runs on m1.medium instances. Fig. 10 shows that the read throughput of the S3 is much lower compared to PVFS on small scales. This results from the fact that the S3 is a remote network storage while PVFS is installed and is spread over each instance. As The number of the instances increase, PVFS cannot scale as well as the S3 and the performance of the two systems get closer to each other up to a scale that S3 slightly performs better than the PVFS. Therefore it is better to choose S3 if we are using more than 96 instances for the application.

Next, we evaluate the performance of PVFS2 for the scales of 1 to 64 as we found out that it performs better than S3 in smaller scales. To benchmark PVFS2 for the following experiments we use the MPIIO interface instead of POSIX. In the configuration that we used, every node in the cluster serves both as an I/O and metadata server.

Fig. 11 shows the read operation throughput of PVFS2 on local storage with different number of instances and variable transfer size. The effect of having a small transfer size is significant, where we see that the throughput increases as we make the transfer size bigger. Again, this fact is due to the overhead added by the I/O transaction.

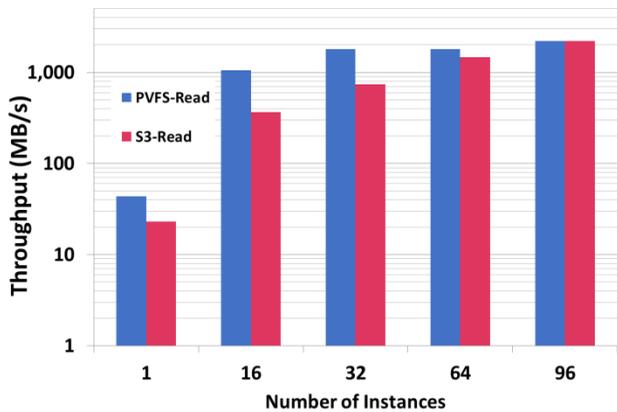


Fig. 10. Comparing the read throughput of S3 and PVFS on different scales

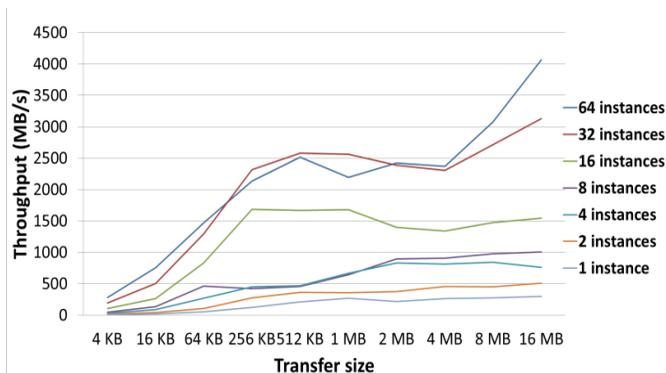


Fig. 11. PVFS read with different transfer sizes over on instance storage

Finally, Fig. 12, shows the performance of PVFS2 and NFS on memory through the POSIX interface. The results show that the NFS cluster does not scale very well and the throughput does not increase as we increase the number of nodes. It basically bottlenecks at the 1Gb/s which is the network bandwidth of a single instance. PVFS2 performs better as it can scale very well on 64 nodes on memory. But as we have shown above, it will not scale on larger scales.

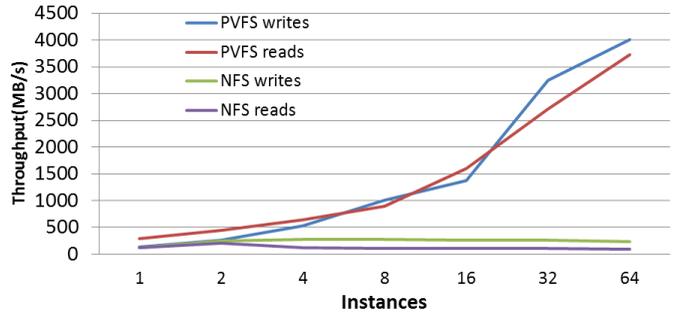


Fig. 12. Scalability of PVFS2 and NFS in read/write throughput using memory as storage

2.4.6 DynamoDB performance

In this section we are evaluating the performance of Amazon DynamoDB. DynamoDB is a commonly used NoSql database used by commercial and scientific applications. We conduct micro benchmarks to measure the throughput and latency of insert and look up calls scaling from 1 to 96 instances with total number of calls scaling from 10000 to 960000 calls. We conduct the benchmarks on both m1.medium and cc2.8xlarge instances. The provision capacity for the benchmarks is 10K operations/s which is the maximum default capacity available. There is no information released about how many nodes are used to offer a specific throughput. We have observed that the latency of DynamoDB doesn't change much with scales, and the value is around 10ms. This shows that DynamoDB is highly scalable. Fig. 13 shows the latency of look up and insert calls made from 96 cc2.8xlarge instances. The average latency for insert and look up are respectively 10 ms and 8.7 ms. %90 of the calls had a latency of less than 12 ms for insert and 10.5 ms for look up.

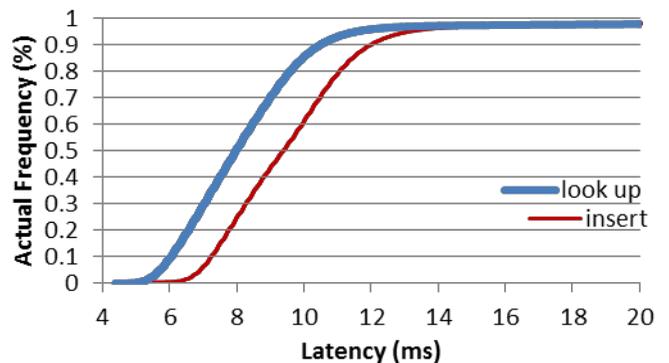


Fig. 13. CDF plot for insert and look up latency on 96 cc2.8xlarge instances

We compare the throughput of DynamoDB with ZHT 0 on EC2. ZHT is an open source consistent NoSql

database providing a service which is comparable to DynamoDB in functionality. We conduct this experiment to better understand the available options for having a scalable key-value store. We use both m1.medium and cc2.8xlarge instances to run ZHT. On 96 nodes scale with 2cc.8xlarge instance type, ZHT offers 1215.0 K ops/s while DynamoDB failed the test since it saturated the capacity. The maximum measured throughput of DynamoDB was 11.5K ops/s which is found at 64 cc2.8xlarge instance scale. For a fair comparison, both DynamoDB and ZHT have 8 clients per node.

Fig. 14 shows that the throughput of ZHT on m1.medium and cc2.8xlarge instances are respectively 59x and 559x higher than DynamoDB on 1 instance scale. On the 96 instance scale they are 20x and 134x higher than the DynamoDB. We can conclude that the ZHT has a significantly higher throughput than DynamoDB up to 96 instance scale and is a better option than DynamoDB for normal AWS users. In the Cost Analysis section we will compare the costs of running workloads over DynamoDB and ZHT.

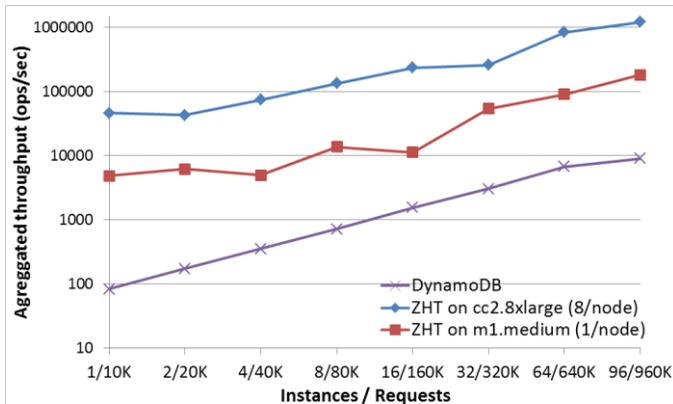


Fig. 14. Throughput comparison of DynamoDB with ZHT running on m1.medium and cc2.8xlarge instances on different scales.

2.4.7 Workflow Application Performance

In this section we analyze the performance of a complex scientific computing application on the Amazon EC2 cloud. The application investigated is Power Locational Marginal Price Simulation (LMPS), and it is coordinated and run through the Swift parallel programming system [12]. Optimal power flow studies are crucial in understanding the flow and price patterns in electricity under different demand and network conditions. A big computational challenge arising in power grid analysis is that simulations need to be run at high time resolutions in order to capture effect occurring at multiple time scales. For instance, power flows tend to be more constrained at certain times of the day and of the year, and these need to be identified.

The power flow simulation application under study analyzes historical conditions in the Illinois grid to simulate instant power prices on an hourly basis. The application runs linear programming solvers invoked via an AMPL (A Mathematical Programming Language) 0 model representation and collects flow, generation, and price data with attached geographical coordinates. A typical application consists of running the model in 8760 inde-

pendent executions corresponding to each hour of the year. Each application task execution spans in the range between 25 and 80 seconds as shown in the application tasks time distribution graph in Fig. 15.

A snapshot of one such result prices plotted over the map of Illinois is shown in Fig. 16. The prices are in US dollars per megaWatt-hour shown as interpolated contour plots across the areas connected by transmission lines and generation stations shown as lines and circles respectively. A series of such plots could be post processed to give an animated visualization for further analysis in trends etc.

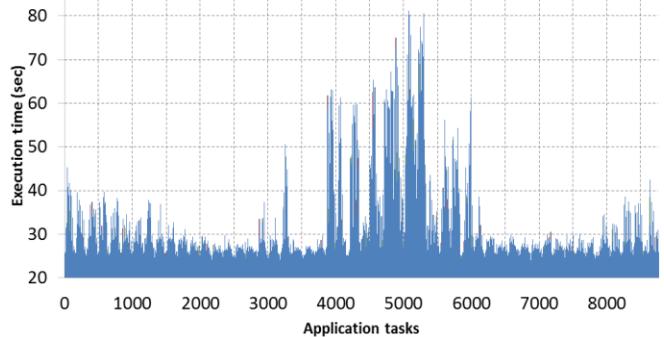


Fig. 15. The LMPS application tasks time distributions.

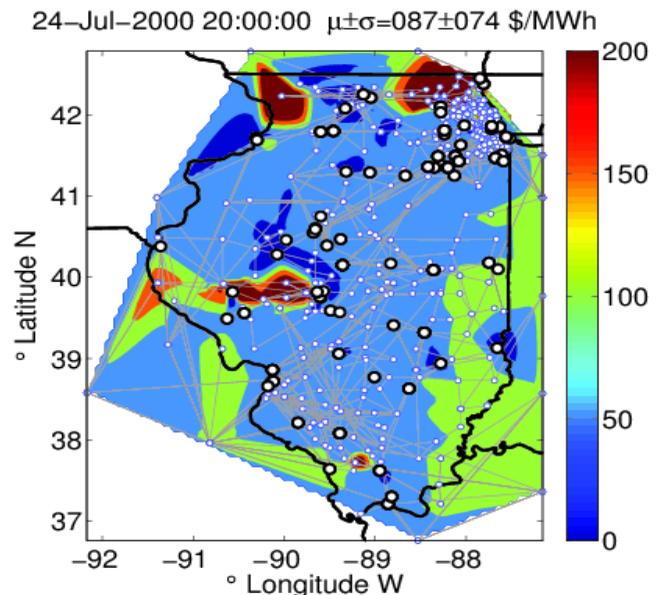


Fig. 16. A contour plot snapshot of the power prices in \$/MWh across the state of Illinois for an instance in July 2000

The execution of the application was performed on an increasing number of m1.large instances (see Fig. 17). For data storage, we use S3. Given that the application scales well to 80 instances, but not beyond that. The performance saturation is a salient point that comes out of Fig. 17. With S3 object store being remote, at 100 VMs it takes long enough to fetch the data that its dominating execution time. More scalable distributed storage subsystem should be investigated that is geared towards scientific computing, such as PVFS, Lustre, or GPFS.

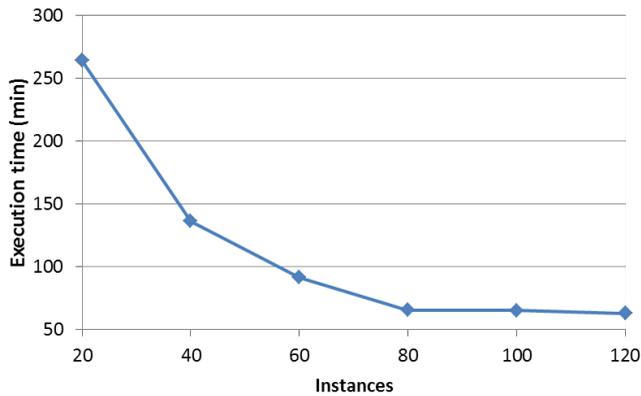


Fig. 17. The runtime of LMPS on m1.large instances in different scales.

2.5 Performance Comparison of EC2 vs. FermiCloud

In this section we want to compare the performance of the EC2 as a public cloud with FermiCloud as a private cloud on HPL benchmark which is a real HPC application. Before comparing the performance of Amazon on real Applications, we need to compare the raw performance of the two resources.

2.5.1 Raw performance comparison

Before comparing the performance of the two infrastructures on real applications like HPL, we need to compare their raw performance on the essential metrics in order to find the root causes of their performance difference. The most effective factors on HPL performance are compute power and Network. We need to compare these factors on the instances with similar functionalities.

On both of the Clouds, we chose the instances that can achieve the highest performance on HPL applications. On EC2, we use cc1.4xlarge instances that are enabled with an 8 core 2.6 GHz Intel processors and a 10 Gigabit network adapter. On FermiCloud, each server machine is enabled with with 2 quad core 2.66 GHz Intel processors, and 8 port RAID Controller.

The CPU efficiency is defined as the performance of the VM running HPL on a single VM with no network connectivity, divided by the the theoretical peak performance of the CPU.

Fig. 18 compares the raw performance of the Amazon EC2 with FermiCloud on CPU and network performance.

The results show that the virtualization overhead on FermiCloud instances are significantly lower than the EC2 instances. This would be an effective factor while running applications on simultaneously on multiple nodes.

The FermiCloud instances are enabled with infiniband network adapters and are able to provide higher performance compared to the EC2 instances that have 10 Gigabit network cards. The efficiency of both of the systems on network throughput is high. The network throughput efficiency is defined as the VM network performance divided by the theoretical peak of the device. FermiCloud and EC2 network adapters respectively achieve %97.9 and %97.4 efficiency.

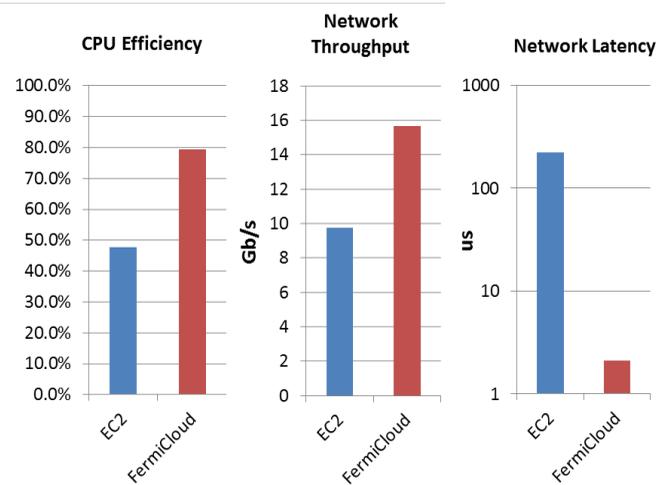


Fig. 18. Raw performance comparison overview of EC2 vs. FermiCloud

There is a huge gap between the network latency of the two clouds. The latency of the FermiCloud instance is 2.2 us as compared to the latency of EC2 instance which is 222 us. Another important factor is the latency variance. The latency variance on both systems is within %20 which is stable. HPL application uses MPI for communication among the nodes. The network latency can decrease the performance of the application by affecting the MPI performance.

2.5.2 HPL performance comparison

In this section we evaluate the performance of HPL application on both on a virtual cluster on both FermiCloud and EC2. The main difference on the two infrastructures is on their virtualization layer and the network performance. FermiCloud uses KVM and is enabled with infiniband network adapters. EC2 uses its own type of virtualization which is based on Xen hypervisor and has 10 Gigabit network adapters.

The best way to measure the efficiency of a virtual cluster on a cloud environment is defining it as the performance of the VM which include the virtualization overhead divided by the host performance that doesn't include virtualization overhead. We can measure the efficiency as defined for FermiCloud since we have access to the host machines. But that is not possible for EC2 since we don't have access to the host machines. Therefore we compare the scalability efficiency of the two clouds which is defined as the overhead of the application performance as we scale up the number of cloud instances.

Fig. 19 compares the efficiency of EC2 and FermiCloud running HPL application on a virtual cluster. Due to budget limitations we run the experiment up to 32 instances scale.

The results show that the efficiency is majorly dependent on the network performance. On the 2 instances scale, both cloud show good efficiency and only lose %10 efficiency that is due to the MPI communications latency added between the instances. Since both of the clouds have powerful network adapters, the communication overhead is still not a bottleneck on 2 instances scale. As the number of instances increase, the applications pro-

cesses make more MPI calls to each other and start saturating the network bandwidth. Having infiniband network, the FermiCloud loses less efficiency than the EC2. The efficiency of EC2 drops to %78 as the FermiCloud efficiency drops to %87. We can notice that the efficiency of the EC2 decreases significantly on 8 instances scales. The reason for that is that the network gets saturated due to too many MPI communications.

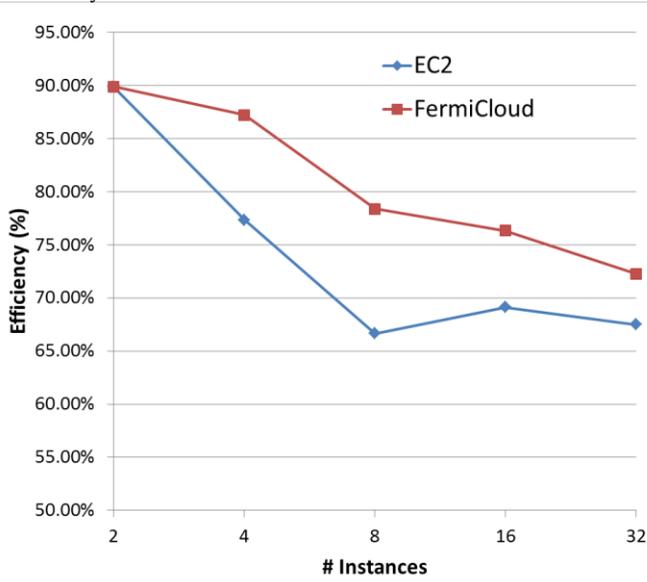


Fig. 19. Efficiency comparison of EC2 and FermiCloud running HPL application on a virtual cluster.

3 COST ANALYSIS

In this section we analyze the cost of the Amazon EC2 cloud from different aspects. We analyze the cost of instances for compute intensive applications as well as for data intensive applications. Our analysis provides suggestions to different cloud users to find the instance type that fits best for certain application with specific requirements. Next section compares the instances based on their memory capacity and performance.

3.1 Memory Cost

This section compares the cost of the memory on Amazon EC2 instances. Fig. 20 compares the cost of instances based on their memory capacity and bandwidth.

The GB/Dollar metric on the left hand side shows the capacity cost effectiveness of the instances. The most cost effective instances for memory capacity are the high memory (m2.2xlarge & m2.4xlarge) instances. But looking at the cost of the memory bandwidth, we can observe that these instances don't have the best memory bandwidth efficiency. The most cost effective instances based on the memory bandwidth efficiency are the m1.small and m1.medium instances.

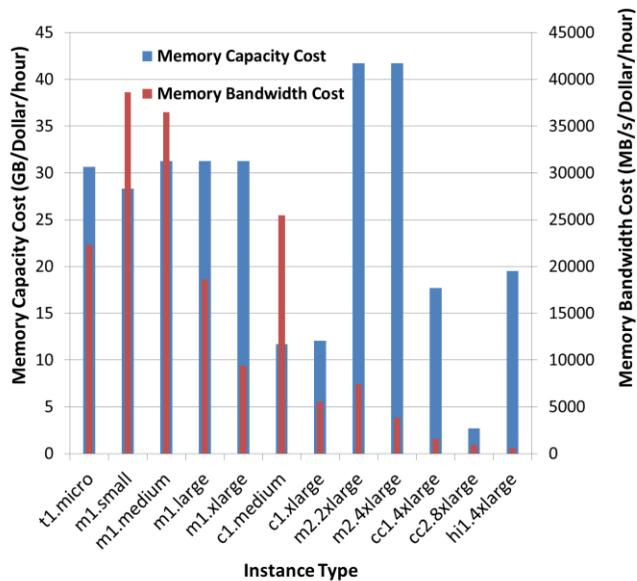


Fig. 20. Memory capacity and memory bandwidth cost.

3.2 CPU Cost

In this section we analyze the cost-effectiveness of instances based on the performance of the instances while running compute intensive applications. The metric for our analysis is GFLOPS/Dollar.

Fig. 21 compares the ideal performance cost of the instances based on Amazon claims with their actual performance while running HPL benchmark. The results show that although the cc2.8xlarge is expected to be the most cost-effective instance, the best compute type is c1.medium. This instance is listed as a High CPU instance.

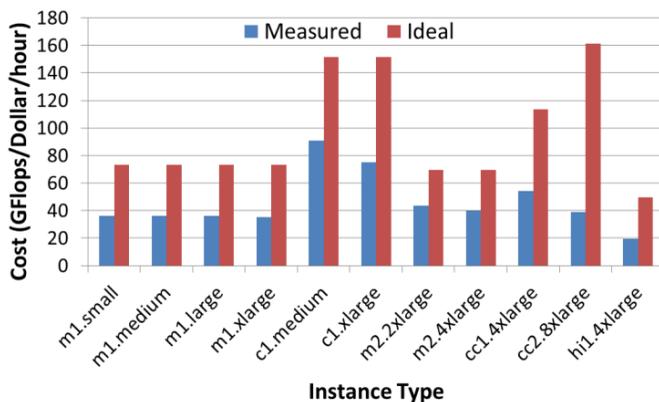


Fig. 21. CPU performance cost of instances

3.3 Cluster Cost

We analyze the cost of the virtual clusters set up by m1.medium and cc1.4xlarge instances in different sizes. Fig. 22 compares the cost of the virtual clusters based on their compute performance.

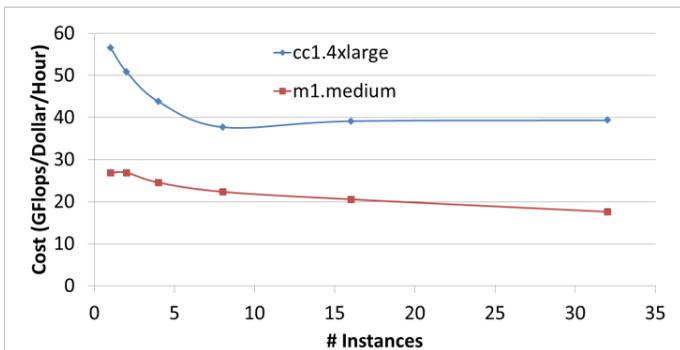


Fig. 22. Cost of virtual cluster of m1.medium and cc1.4xlarge.

3.4 DynamoDB Cost

Finally in this section we evaluate the cost of DynamoDB. In order to better understand the value of offered service, we compare the cost with the cost of running ZHT on EC2 on different instance types.

Fig. 23 shows the hourly cost of 1000 ops/s capacity offered by DynamoDB compared to the equal capacity provided by ZHT from the user point of view.

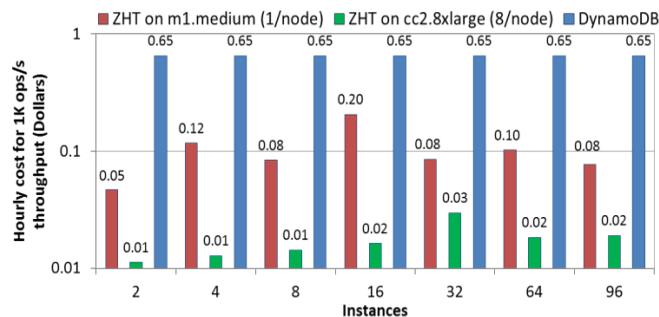


Fig. 23 Cost Comparison of DynamoDB with ZHT

We are comparing the two different scenarios of cost of using a free application on rented EC2 instances versus getting the service from DynamoDB. In case of DynamoDB, since the users pays for the capacity that they get, the number of instances doesn't affect the cost. That's why the cost of DynamoDB is always constant. For ZHT, the system efficiency and performance varies on different scales hence the variation in costs for ZHT at different scales. Since the cc2.8xlarge instances provide much better performance per money spent, the cost per operation is as good as 65X lower than DynamoDB. However, the better costs come at the complexity of managing a virtual cluster of machines to operate ZHT. It is likely that for low loads including sporadic requirements for DynamoDB, it makes financial sense to run on Amazon AWS services, but for higher performance requirements it is much more beneficial to simply operate a dedicated ZHT system over EC2 resources.

3.5 Performance and Cost Summary

This section summarizes the performance and the cost efficiency of Amazon EC2 and other services of AWS. Table 1 shows the performance overview of the different instance types on EC2. The performance results of the instances mostly match with the prediction based on the claims of Amazon. There have been anomalies in some of the specific instance types. Instances like m1.xlarge have

average performance while m1.medium instance has shown a performance that was higher than expected.

TABLE 1: Performance summary of EC2 instances

| | CPU bw | Mem. bw | Net. bw | Disk I/O |
|-----------|--------|---------|---------|----------|
| m1.small | Low | Low | Low | Low |
| m1.med | Low | Avg | Avg | Low |
| m1.lrg | Avg | Avg | Avg | Avg |
| m1.xlrg | Avg | Avg | Avg | Avg |
| c1.med | Avg | Avg | Avg | Low |
| c1.xlrg | Avg | High | Avg | Avg |
| m2.2xlrg | High | High | Avg | Avg |
| cc1.4xlrg | High | High | High | Avg |
| cc2.8xlrg | High | High | High | Avg |
| hi1.lrg | High | Avg | High | High |

Table 2 summarizes the cost-efficiency of instance types of EC2. As it is noticeable from the table, the cost efficiency of the high end instances that are better fits for HPC and scientific applications is lower than the small instances. Finally table 3 summarizes the performance of S3 and DynamoDB.

TABLE 2: Cost-efficiency summary of EC2 instances

| | CPU bw | Mem. Cap. | Mem. bw | Net. bw |
|-----------|--------|-----------|---------|---------|
| m1.small | Avg | Avg | High | High |
| m1.med | Avg | Avg | High | High |
| m1.lrg | Avg | Avg | Avg | Avg |
| m1.xlrg | Avg | Avg | Low | Low |
| c1.med | High | Low | High | Low |
| c1.xlrg | High | Low | Low | Low |
| m2.2xlrg | Low | High | Low | Low |
| cc1.4xlrg | Avg | Low | Low | Low |
| cc2.8xlrg | Low | Low | Low | Low |
| hi1.lrg | Low | Low | Low | Low |

TABLE 3: Performance and Cost-efficiency summary of AWS services

| | Scalability | Cost-efficiency | Data Granularity |
|----------|-------------|-----------------|------------------|
| S3 | High | High | Large data |
| DynamoDB | High | Low | Small data |

4 RELATED WORK

There have been many efforts to investigate the usefulness of cloud computing and virtualization for scientific applications. Researchers have tried to evaluate the performance of clouds in order to understand the weaknesses and benefits of them when used for scientific applications.

There have been many researches that have tried to evaluate the performance of Amazon EC2 cloud [14][16][17]. However the experiments were mostly run on limited types and number of instances. Therefore they lack the generality in their results and conclusions, as they have not covered all instance types. Unlike these

previous works, we cover all instance types in order to give a general view of the instances and enable users to choose the best instances for different use case scenarios.

Ostermann et al. have evaluated Amazon EC2 using micro-benchmarks in different performance metrics. However their experiments do not include the more high-end instances that are more competitive to HPC systems. Moreover, the Amazon performance has improved since then and more features have been added to make it useful for HPC applications [14]. In addition to the experiments scope of that paper, our work provides the evaluations of the raw performance of a variety of the instances including the high-end instances, as well as the performance of the real applications.

He et al. have deployed a NASA climate prediction application into major public clouds, and compared the results with dedicated HPC systems results. They have run micro-benchmarks and real applications [15]. However they only run their experiments on small number of VMs. We have evaluated the performance of EC2 on larger scales.

Jackson has deployed a full application that performs massive file operations and data transfer on Amazon EC2 [18]. The research mostly focuses on different storage options on Amazon. Our work covers the storage services performance both on micro-benchmarks as well as the performance while being used by data-intensive applications.

Only a few of the researches that measure the applicability of clouds for scientific applications have used the new Amazon EC2 cluster instances that we have tested [10][20]. Mehrotra compares the performances of Amazon EC2 HPC instances to that of NASA's Pleiades supercomputer [10]. However the performance metrics in that paper is very limited. They have not evaluated different performance metrics of the HPC instances. Ramakrishnan have measured the performance of the HPCC benchmarks [20]. They have also applied two real applications of PARATEC and MILC. They have compared the performance of Amazon EC2 with Magellan cloud while running the same applications.

Juve investigates different options of data management of the workflows on EC2 0. The paper evaluates the runtime of different workflows with different underlying storage options. It uses a limited number of instance types. It also evaluates the cost of running workflow applications. The aforementioned works have not provided a comprehensive evaluation of the HPC instances. Their experiments are limited to a few metrics. Among the works that have looked at the new HPC instances, our work is the only one that has evaluated all of the critical performance metrics such as memory, compute, and network performance. Our paper has evaluated the performance of the new HPC instances and also has shown the compute performance of a virtual cluster that is using MPI and is made of such instances. This experiment is very useful in that it shows the performance of EC2 while running scientific applications using MPI.

Many works have covered the performance of public clouds without having an idea about the host perfor-

mance of the nodes without virtualization head [14][15][16]. Younge has evaluated the performance of different virtualization techniques on FutureGrid private cloud [11]. The focus of that work is on the virtualization layer rather than the cloud infrastructure. Our work compares the performance of the public cloud and a private cloud on different aspects running both micro-benchmarks and real scientific applications. Being able to measure the virtualization overhead on the FermiCloud private cloud, we could provide a better comparison of the two cloud environments.

Many papers have analyzed the cost of the cloud as an alternative resource to dedicated HPC resources [18][19]. However this paper is the only work that compares the cost of different instances based on major performance factors in order to find the best use case for different instances of Amazon EC2.

5 CONCLUSION

In this paper, we present a quantitative study to evaluate the performance of the Amazon EC2 for the goal of running scientific applications. We evaluate the performance of various instance types by running micro benchmarks on memory, compute, network and storage. In most of the cases, the actual performance of the instances is lower than the expected performance or what Amazon claims. Most of the instances have stable memory bandwidth, which is comparable with non-virtualized systems. The compute performance of the instances is affected by virtualization overhead on the larger instances. We also run different types of network benchmarking. The results show stable internal network performance of single client-server connections. However we notice the poor performance and scalability in wide area connections between datacenters. The network latency is higher and less stable than what is available on the supercomputers.

We also compare the performance of EC2 as a commonly used public cloud with FermiCloud, which is a higher end private cloud that is tailored for scientific for scientific computing. We compare the raw performance as well as the performance of the real applications on virtual clusters with multiple HPC instances. The results show that the performance of the MPI applications is highly dependent on network performance of the infrastructure. In this case, FermiCloud is able to achieve higher performance and efficiency due to having infiniband network cards. We can conclude that the cloud infrastructures with more powerful network capacity are more suitable to run scientific applications.

We evaluated the I/O performance of Amazon instances and storage services like EBS and S3. The I/O performance of the instances is lower than performance of dedicated resources. The only instance type that shows promising results is the high-IO instances that have SSD drives on them. The performance of different parallel file systems is lower than performance of them on dedicated clusters. The read and write throughput of S3 is lower than a local storage. Therefore it could not be a suitable option for scientific applications. However it shows

promising scalability that makes it a better option on larger scale computations. The performance of PVFS2 over EC2 is convincing for using in scientific applications that require a parallel file system.

Amazon EC2 provides powerful instances that are capable of running HPC applications. However, the performance a major portion of the HPC applications are heavily dependent on network bandwidth, and the network performance of Amazon EC2 instances cannot keep up with their compute performance while running HPC applications and become a major bottleneck. Moreover, having the TCP network protocol as the main network protocol, all of the MPI calls on HPC applications are made on top of TCP protocol. That would add a significant overhead to the network performance. Although the new HPC instances have higher network bandwidth, they are still not on par with the non-virtualized HPC systems with high-end network topologies. The cloud instances have shown to be performing very well, while running embarrassingly parallel programs that have minimal interaction between the nodes [10]. The performance of embarrassingly parallel application with minimal communication on Amazon EC2 instances is reported to be comparable with non-virtualized environments [21][22].

Armed with both detailed benchmarks to gauge expected performance and a detailed price/cost analysis, we expect that this paper will be a recipe cookbook for scientists to help them decide between dedicated resources, cloud resources, or some combination, for their particular scientific computing workload.

ACKNOWLEDGEMENT

This work was supported in part by the National Science Foundation grant NSF-1054974. It is also supported by the US Department of Energy under contract number DE-AC02-07CH11359 and by KISTI under a joint Cooperative Research and Development Agreement. CRADA-FRA 2013-0001/KISTI-C13013. This work was also possible in part due to the Amazon AWS Research Grants. We thank V. Zavala of ANL for power grid application.

REFERENCES

- [1] Amazon EC2 Instance Types, Amazon Web Services, [online] 2013, <http://aws.amazon.com/ec2/instance-types/> (Accessed: 2 November 2013)
- [2] Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, <http://aws.amazon.com/ec2/> (Accessed: 2 November 2013)
- [3] Amazon Simple Storage Service (Amazon S3), Amazon Web Services, [online] 2013, <http://aws.amazon.com/s3/> (Accessed: 2 November 2013)
- [4] Iperf, Souceforge, [online] June 2011, <http://sourceforge.net/projects/iperf/> (Accessed: 2 November 2013)
- [5] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary. "HPL", (netlib.org), [online] September 2008, <http://www.netlib.org/benchmark/hpl/> (Accessed: 2 November 2013)
- [6] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the MPI standard," Tech. Rep. CS-95-274, University of Tennessee, Jan. 1995
- [7] Release: Amazon EC2 on 2007-07-12, Amazon Web Services, [online] 2013, <http://aws.amazon.com/releasenotes/Amazon-EC2/3964> (Accessed: 1 November 2013)
- [8] K. Yelick, S. Coghlan, B. Draney, and R. S. Canon, "The Magellan report on cloud computing for science," U.S. Department of Energy, Tech. Rep., 2011
- [9] L. Ramakrishnan, R. S. Canon, K. Muriki, I. Sakrejda, and N. J. Wright. "Evaluating Interconnect and virtualization performance for high performance computing", ACM Performance Evaluation Review, 2012
- [10] P. Mehrotra, et al. 2012. "Performance evaluation of Amazon EC2 for NASA HPC applications" In *Proceedings of the 3rd workshop on Scientific Cloud Computing* (ScienceCloud '12). ACM, New York, NY, USA, pp. 41-50
- [11] A. J. Younge, R. Henschel, J. T. Brown, G. von Laszewski, J. Qiu, and G. C. Fox, "Analysis of virtualization technologies for high performance computing environments," International Conference on Cloud Computing, 2011
- [12] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. "Swift: Fast, reliable, loosely coupled parallel computation", IEEE Int. Workshop on Scientific Workflows, pages 199-206, 2007
- [13] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. "Towards Loosely-Coupled Programming on Petascale Systems", IEEE/ACM Supercomputing 2008
- [14] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. "A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing". In *Cloudcomp*, 2009
- [15] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. "Case study for running HPC applications in public clouds," In *Proc. of ACM Symposium on High Performance Distributed Computing*, 2010
- [16] G. Wang and T. S. Eugene Ng. "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center". In *IEEE INFOCOM*, 2010
- [17] S. L. Garfinkel, "An evaluation of amazon's grid computing services: Ec2, s3 and sqs," Computer Science Group, Harvard University, Technical Report, 2007, tR-08-07
- [18] K. R. Jackson et al. "Performance and cost analysis of the supernova factory on the amazon aws cloud". *Scientific Programming*, 19(2-3):107-119, 2011
- [19] J.-S. Vockler, G. Juve, E. Deelman, M. Rynge, and G.B. Berri-man, "Experiences Using Cloud Computing for A Scientific Workflow Application," 2nd Workshop on Scientific Cloud Computing (ScienceCloud), 2011
- [20] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu. "Magellan: experiences from a science cloud". In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 49-58, San Jose, USA, 2011
- [21] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright, "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud," in *2nd IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2010, pp. 159-168
- [22] J.-S. Vockler, G. Juve, E. Deelman, M. Rynge, and G.B. Berri-man, "Experiences Using Cloud Computing for A Scientific Workflow Application," 2nd Workshop on Scientific Cloud Computing (ScienceCloud), 2011
- [23] J. Lange, K. Pedretti, P. Dinda, P. Bridges, C. Bae, P. Soltero, A. Merritt, "Minimal Overhead Virtualization of a Large Scale Su-

percomputer," In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011)*, 2011

- [24] G. Juve, E. Deelman, G.B. Berriman, B.P. Berman, and P. Maechling, 2012, "An Evaluation of the Cost and Performance of Scientific Workflows on Amazon EC2", *Journal of Grid Computing*, v. 10, n. 1 (mar.), p. 5–21
- [25] R. Fourer, D. M. Gay, and B. Kernighan, "Algorithms and model formulations in mathematical programming," Ed. New York, NY, USA: Springer-Verlag New York, Inc., 1989, ch. AMPL: a mathematical programming language, pp. 150–151
- [26] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu. "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table", *IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2013*
- [27] FermiCloud, Fermilab Computing Sector, [online], <http://fclweb.fnal.gov/> (Accessed: 25 April 2014)
- [28] R. Moreno-vozmendiario, S. Montero, I. Llorente." IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures: Digital Forensics", Computer (Long Beach, CA)
- [29] K. Hwang, J. Dongarra, and G. C. Fox, *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Morgan Kaufmann, 2011
- [30] W. Voegels. "Amazon DynamoDB—a fast and scalable NoSQL database service designed for Internet-scale applications." <http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html>, January 18, 2012
- [31] I. Sadooghi et al. "Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing." In *Proc. 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'14)*, 2014
- [32] T. Lacerda Ruivo, G. Bernabeu Altayo et al. "Exploring Infiniband Hardware Virtualization in OpenNebula towards Efficient High-Performance Computing." *CCGrid'14*, 2014
- [33] Macleod, D. (2013). OpenNebula KVM SR-IOV driver.
- [34] K. Maheshwari et al. "Evaluating Cloud Computing Techniques for Smart Power Grid Design using Parallel Scripting" In *Proc. 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13)*, 2013



Iman Sadooghi is a 3rd year PhD student in the department of the Computer Science of Illinois Institute of Technology and a member of DataSys laboratory. He received his MS degree in Software Engineering from San Jose State University. Iman's research interest is Distributed Systems and Cloud Computing. He has two publications in the area of Cloud Computing. Iman is also a member of the IEEE and the IEEE Computer Society.



Jesus Hernandez is a data engineer at Lyst. He obtained his MS degree in Computer Science from Illinois Institute of Technology in 2012. He also received a BS + MS degree in Computer Engineering from Technical University of Madrid during the same year. His interests range from distributed systems, data processing, and analytics to high performance client-server architectures.



Tonglin Li is a 5th year PhD student working in Datasys Lab, computer science department at Illinois Institute of Technology, Chicago. His research interests are in Distributed systems, Cloud computing, Data-intensive computing, Supercomputing, and Data management. He received his B.E and M.E in 2003 and 2009 from Xi'an Shiyu University, China. Tonglin Li is an IEEE and ACM

member.



Kevin Brandstatter Kevin Brandstatter is a 4th year undergraduate student in the department of Computer Science at Illinois Institute of Technology. He has worked as a Research Assistant in the Datasys lab under Dr. Raicu for the past 3 years. His research focus is primarily related to distributed storage systems such as distributed file systems and distributed key value storage.



Ketan Maheshwari is a postdoctoral researcher in the Mathematics and Computer Science Division at Argonne National Laboratory. His research is focused on applications for parallel scripting on distributed and High Performance resources. His main activities in recent years involve design, implementation and execution of parallel applications on clouds, XSEDE, Cray XE6 and IBM supercomputers at University of Chicago. The applications include massive protein docking, weather and soil modeling, earthquake simulations, and power grid modeling funded by DOE and NIH.



Yong Zhao is professor at the University of Electronic Science and Technology of China. His research interests are in big data, cloud computing, grid computing, data intensive computing, extreme large scale computing, and cloud workflow. He has published more than 40 papers in international computer books, journals and conferences, which are referenced more than 4000 times; He has chaired/co-chaired ACM/IEEE Workshop on Many Task Computing on Grids Clouds and Supercomputers, Workshop on Data Intensive Computing in the Clouds, and IEEE International Workshop on CloudFlow 2012, 2013.



Tiago Pais is a MS graduate in Computer Science by the Illinois Institute of Technology and former Graduate Research Intern in the Fermi National Accelerator Laboratory. His research interests are in network virtualization and mobile technologies.



Gabriele Garzogio is the head of the Grid and Cloud Services Department of the Scientific Computing Division at Fermilab and he is deeply involved in the project management of the Open Science Grid. He oversees the operations of the Grid services at Fermilab. Gabriele has a Laura degree in Physics from University of Genova, Italy, and a PhD in Computer Science from DePaul University.



Steven Timm is an Associate Department Head for Cloud Computing in the Grid and Cloud Services Department of the Scientific Computing Division at Fermi National Accelerator Laboratory. He received his Ph.D in Physics from Carnegie Mellon. Since 2000 he has held various positions on the staff of Fermilab relating to Grid and Cloud Computing. He has been the lead of the FermiCloud project since its inception in 2009.



Ioan Raicu is an assistant professor in the Department of Computer Science at Illinois Institute of Technology, as well as a guest research faculty in the Math and Computer Science Division at Argonne National Laboratory. He is also the founder and director of the Data-Intensive Distributed Systems Laboratory at IIT. His research work and interests are in the general area of distributed systems. His work focuses on a relatively new paradigm of Many-Task Computing (MTC), which aims to bridge the gap between two predominant paradigms from distributed systems, HTC and HPC. His work has focused on defining and exploring both the theory and practical aspects of realizing MTC across a wide range of large-scale distributed systems. He is particularly interested in resource management in large scale distributed systems with a focus on many-task computing, data intensive computing, cloud computing, grid computing, and many-core computing.