Infrastructure Outsourcing in Multi-Cloud Environment

Kate Keahey
Argonne National Laboratory
keahey@mcs.anl.gov

Patrick Armstrong
University of Chicago
oldpatricka@uchicago.edu

John Bresnahan
Argonne National Laboratory
bresnaha@mcs.anl.gov

David LaBissoniere
University of Chicago
labisso@uchicago.edu

Pierre Riteau
University of Chicago
priteau@uchicago.edu

ABSTRACT

Infrastructure clouds created ideal conditions for users to outsource their infrastructure needs by offering on-demand, short-term access, pay-as-you-go business model, the use of virtualization technologies which provide a safe and cost-effective way for users to manage and customize their environments, and sheer convenience, as users and institutions no longer have to have specialized IT departments and can focus on their core mission instead. These key innovations however also bring challenges which include high levels of failure; lack of interoperability between cloud providers, which puts significant lock-in pressure on the user, and lack of tools that allow users to leverage the on-demand growing and shrinking of infrastructure. All these factors prevent users from capitalizing on the infrastructure cloud opportunity. In this paper we propose a multi-cloud auto-scaling service that enables the user to leverage "computational power on tap" provided by infrastructure clouds, i.e., allows the user to easily deploy resources across multiple private, community, and commercial clouds; provides high availability in that it allows users to replace failed resources; and scales to demand. The policies governing scaling are customizable based on system and application-specific indicators. We will describe the service architecture and implementation and discuss results obtained in the sustained deployment and management of thousands of virtual machines on EC2.

General Terms

Management, Design, Experimentation.

Keywords

Cloud computing, Infrastructure-as-a-Service, Platform-as-a-Service, Nimbus.

1. INTRODUCTION

Outsourcing and sharing resources has many potential benefits to scientific projects. First, it provides access to more sophisticated resources -- in terms of size, cutting-edge technology or architectural diversity -- that is often beyond the means of a single institution to acquire. The resource can also be used with greater flexibility: e.g., a small "slice" of resource over long time or a larger slice occasionally. Further, it creates potential for access to economies of scale via consolidation and thus provides a better amortization of the original investment. And, last but not least, it eliminates of the overhead of system acquisition and operation for an institution via outsourcing computing. This is valuable as it allows scientific institutions to focus on delivering results in the form of scientific breakthroughs that are its mission in the first place. All those reasons are powerful motivators for outsourcing where a suitable outsourcing paradigm can be found.

Workshop on Cloud Computing Federation, September 21st, 2012, San Jose, CA

Infrastructure-as-a-Service (IaaS) clouds [1] (also called infrastructure clouds) created ideal conditions for users to outsource their infrastructure needs. A typical infrastructure cloud offers (1) on-demand, short-term access, which allows users to flexibly manage peaks in demand, (2) pay-as-you-go model, which helps save costs for bursty usage patterns (i.e., helps manage "valleys" in demand), (3) access via virtualization technologies which provides a safe and cost-effective way for users to manage and customize their own environments, and (4) sheer convenience, as users and institutions no longer have to have specialized IT departments and can focus on their core mission instead. The flexibility of the approach allows users to also outsource as much or as little of their infrastructure procurement as their needs justify: they can keep a resident amount of infrastructure in-house while outsourcing only at times of increased demand and they can outsource to a variety of providers choosing the best service levels for the price the market has to offer.

However, as well as all these advantages the cloud environment also has its challenges. The clouds existing today have levels of failure that are higher than traditional in-house machines; this is due not only to service levels of specific providers, but also to the fact that infrastructure cloud resources are typically accessed over the Internet with high potential for network failures or delays. Interoperability between providers is in its infancy, making efficient markets infeasible and putting significant lock-in pressure on the user (i.e., the barrier for moving from provider to provider is significant and the effort to scale it is borne entirely by the user). And last but not least, currently the methods of leveraging the on-demand growing and shrinking of infrastructure are crude to say the least: users typically are reduced to doing it manually or setting up primitive infrastructures to manage part of the solution space for them. This prevents them from capitalizing on the infrastructure cloud opportunity.

In this paper, we describe infrastructure which automates outsourcing computation to the cloud. We propose a multi-cloud auto-scaling service that provides an easy way for the user to leverage "computational power on tap", i.e., allows the user to easily deploy resources across multiple private, community, and commercial clouds. Our emphasis is on enabling users to build highly scalable services and also provide high availability, to allow users to replace failed resources ensuring continuous presence on the network. Further, the system carries support for adapting policies governing scaling based on system and application-specific indicators. We will describe the service architecture and implementation and discuss results obtained in

the sustained deployment and management of thousands of virtual machines on Amazon Web Services' EC2 service [2].

This paper is structured as follows. Section 3 describes the model our services implement. Section 4 and 2 describe respectively the architecture and the design principles that led to the development of this architecture. Section 5 describes challenges in scalability and their resolution when implementing scaling of thousands of VM instances on Amazon's EC2.

2. DESIGN PRINCIPLES

Below we summarize the principles behind our design:

Any Scale. To ensure "computational power on tap" it is necessary to allow users scale the resources over which their computation is deployed, up and down, easily and automatically – much as electrical devices have the ability to draw as much or as little power from the grid as they need. We should provide the ability to auto-scale via dynamically provisioning resources in the cloud in reaction to system-specific or application-specific sensors, which the user can pick from a library of ready-made, pre-defined sensors, but also give the user the ability to define and add his or her own sensors that can eventually be published and shared with other users. The sensors should include a large variety of events -- including operator-driven/manual events. This implies information presentation consistent with manual control: the ability to "zoom in" on the state and composition of any collection of resources, as well as the ability to use simple visual cues such as e.g., up and down arrow keys to regulate resource size.

High Availability (HA). Infrastructure clouds today are often unreliable [3]. This is due not only to service levels of specific providers, but also to the fact that infrastructure cloud resources are typically accessed over the Internet with high potential for network failures or delays – and as such may only improve in special cases in the foreseeable future. For this reason, it is essential that an infrastructure aiming to deliver "uninterrupted power supply" address this problem. We develop systems with the assumption that any VM can die and be replaced quickly to preserve the system from a prolonged downtime or service level deterioration. For this reason, we design the system to achieve minimum time to repair (TTR).

MultiCloud. Any system that provisions resources from only one provider exposes itself to the same kind of failure that threatens electricity consumer taking power from just one source. From the perspective of achieving "uninterrupted power supply" working with multiple providers isolates the consumer from technical and business failures any one provider might be experiencing and thus prevents vendor lock-in. It also provides the underpinnings for the creation of markets and thus conveying the best price to the consumer via competition. For this reason our infrastructure should allow users to integrate resources from multiple infrastructure clouds: from private clouds to community and commercial clouds as a "continuum" of available infrastructure resources.

Your Policies, Our Enactment. Different applications and services want to use cloud resources differently. Our focus is on providing infrastructure, i.e., solid and robust enactment that can carry out user policies taking into accont resource availability and status. Providing policies should be as broad as possible, e.g. it should be possible to extend the system by building customized policy plugins. The system will provide the mechanism to collect all the sensor inputs that could be relevant for the user to develop relevant policies and also allow the user to add custom sensors.

3. MODEL

We define a worker VM as a single, ephemeral and replacable compute resource that can be provisioned on-demand and capable of carrying out computation of a specific kind. Worker VMs are ephemeral by which we mean that they may become unavailable at any time, due to resource failure, temporary or persistent network failure, or other issues. A worker unit also needs to be replaceable, i.e. it should be possible to replace it with a worker unit of the same kind such that the new unit can automatically rejoin the computation carried out by the worker units. The worker units are independent of each other.

We define a domain as a pool of homogeneous worker VMs that are contextualized (in terms of both security and configuration) to work with domain-specific entities and whose size is governed by domain-specific policies. There could be potentially many distinct domains composed of VMs of the same type, belonging to different clients or serving different purposes for those clients, just like there are many potential instantiations of a service.

The policies governing the size of the domain may take into account multiple factors including system-specific metrics, such as number of individual worker units or load on individual worker units, application-specific metrics, such as the size and makeup of the workload queue or the number of network connections open to a specific resource, or a variety of other events including e.g., console events or an explicit request by a domain stakeholder (such as a scheduler for example). At any given time, due to a variety of conditions (such as changes in policy, worker node failures or inaccessibility), a domain has an intended size and an actual size – the purpose of a domain manager is to ensure that the intended size is equal to the actual size.

3.1 Using Domains

Our model is based on the assumption that the work of a service can be performed on multiple VMs independent of each other, that can be scaled up or down according to demand. This means that an application or service leveraging this model has to be capable of absorbing new processing capability, i.e., an application process that is newly strated can be automatically integrated into the application. The concept of high availability is additionally reliant on the fact that any process may die without affecting the application as a whole. This means that the work carried out on the VMs is such that can be easily interrupted and easily taken over by a replacement unit should a resource failure occur.

The most common way to leverage the concept of a domain is via applications with a well structured, ordered, workload. Such workload can be represented by an AMQP message queue, where the messages represent work units, a scheduler queue, where jobs represent work units, a workflow of tasks to be executed on a domain, or a list of data transfers to be serviced in a data transfer service. Workload is a logical concept and does not necessarily imply a durable queue. In practice, a domain could be used to support e.g., web servers that work with DNS round-robin. Howeer, in order to provide high availability we have to rely on a durable queue, i.e., with the property that a work unit persists on the queue until its receipt has been acknowledged. A workload can represent a "push queue", which directs units of work to specific worker units in a domain (typically implemented by e.g., batch schedulers or workflow systems) or a "pull queue" which relies on the worker units to claim the work units themselves (implemented by e.g. systems such as BOINC [4] or Condor [5]).

In our model, a service is realized by conveying the workload from the service clients. On the interface side, the

clients add their work units to the workload queue and get notified of the acceptance of the request and eventually a result. On the execution side, the workload is distributed to worker units of a domain hosting the execution of the service (either by the workers accessing the workload queue themselves in a pull queue, or by the work units being distributed to the workers). The domain is monitored and can dynamically grow or shrink as dictated by policies governing that domain. Results of the execution are represented in ways defined by the service layer (e.g., they may or may not be returned to the client directly).

Domains can be used with systems supporting loosely-coupled preemptible processes such as Condor [5], BOINC [4], or Swift [6]. Examples of how an application can dynamically provision resources based on the state of its scheduler queue have been described in [7, 8]. Alternatively, they can also be used with applications requiring stronger synchronization using primitives such as leader election [9] as described in Section 4.2 when discussing our system design.

4. APPROACH

The Domain Manager service implements the concept of a domain, i.e., ensures that it is properly deployed and contextualized, and the number of worker units within a domain scales up and down according to domain policies. The rest of this section describes the architecture of the Domain Manager.

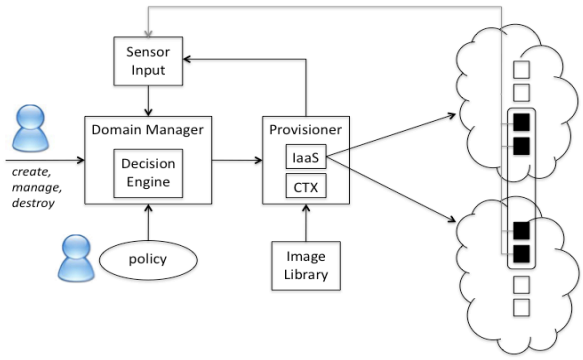

Figure 1: Domain Manager

Figure 1 above shows a domains being managed by the Domain Manager. The domain is distributed over two clouds; with two instances on each. The Domain Manager regulates the size of a domain to reflect sensor input and the associated policies. The Domain Manager relies on sensor information to monitor the deployed VMs, assess their health, and deploy, terminate or redeploy them as needed. The policies are enforced in the context of heartbeat information, informing the Domain Manager of a worker VM health state as well as Provisioner information, informing the Domain Manager of a worker VM lifecycle. Other sources of information can be added. Based on evaluating the information against policies the Domain Manager makes a decision to either add or terminate some of the worker VMs. The Domain Manager decision is carried out by the Provisioner that acts by either deploying or terminating VMs and contextualizing them after deployment them to work as part of the right domain.

To ensure high availability of a domain, all the major components of the system need to be highly available. Furthermore, the system depcted in Figure 1 itself is bootstrapped and monitored by cloudinit.d [10] which ensures repeatable deployment and ongoing monitoring. Note that if any of the system components die, it will not affect the execution of the domain; it will merely affect how quickly it responds to user policies and sensor inputs.

Section 4.1. provides an overview of the interacting components. The design for high availability is descrbed in Section 4.2. Section 4.3 explains implemtation details.

4.1 Components

The *Domain Manager* Service allows clients to create, destroy, and manage domains. Each domain is associated with a Decision Engine that takes two inputs: (1) policy (which can be dynamically modified by a client) and (2) dynamically provided sensor information that can be both system-specific (e.g., information about VM lifecycle and health state) and application-specific (e.g., size of workload for an application). The Decision Engine is a component of the Domain Manager that evaluates the policies against current sensor input and issues commands to the Provisioner to start or stop VM instances. The output of the Domain Manager is a specific directive to deploy specific numbers of VM instances on specific resources. The Domain Manager is implemented to be highly available as per the design described in Section 4.2.

The *Provisioner* provides an adaptation layer for IaaS sites, and contextualizes deployed VMs. Each launch has a unique identifier, supplied by the client, i.e., the Domain Manager, to provide support for idempotency so that a specific request can be retried without launching an additional VM as a result (this functionality is mirrored by IaaS). A launch request also contains a VM type obtained from the Image Library and scheduling constraints such as instance size, the targeted IaaS provider, availability zones, etc. Optionally, a launch request also contains an attribute bag of name/value pairs that can be injected into the VM type to turn it into a customized VM image. The Provisioner is implemented to be highly available as per the design described in Section 4.2.

The *Image Library* allows the Provisioner to dereference a VM type identifier into the necessary VM image compatible with a particular IaaS site. It also currently does the work of interpolating the attribute bag referred to above into a configuration template specific to the deployable type. The Image Library is implemented to be highly available.

Sensors. To function correctly, the system relies on input from a variety of system-specific and application-specific sensors. The default sensors consist of a VM lifecycle sensor and VM agent. The lifecycle sensor is implemented by the Provisioner; it continually queries IaaS and creates notifications of any VM lifecycles stage change (i.e., whether it has been deployed or terminated). In addition the VM agent runs on every VM instance launched via Domain Manager, watches processes and reports heartbeats to the Domain Manager. The lack of heartbeats for too long of a period causes the Domain Manager to consider the VM instance unhealthy (it may in reality be healthy but suffering from a network partition). The VM agent is bootstrapped by the contextualization process [] executed on VM boot.

4.2 Leveraging the Domain Model

To ensure high availability of the overall system, each of the critical components shown in Figure 1, in particular the Domain Manager, Provisioner and Image Library, has to be implemented as a highly available component, capable of scaling up and down and absorbing failure. The challenge of such implementation was that these services do not lend themselves easily to such an implementation: they have a need for internal synchronization, critical sections, and specialized roles. In this section we will describe how we used the domain model to implement those services and at the same time give an example of how the domain

model could be leveraged by other applications seeking to leverage the domain model. We assume that each service is implemented by a set of processes and for simplicity we also assume that each VM has one process, implementing a service function, that is started on boot.

As per the assumptions about worker units (which can appear or disappear at any moment), leveraging the domain model requires us to decompose the function of the service into independent processes that can die or become unavailable at any moment. When they die, they need to be replaced by new service processes that can join the computation automatically. The challenge of the design is therefore the same as the challenge of adapting applications described in Section 3.1; we need to make sure that (1) any given service process can die without affecting the service function and (2) a service process that is newly started to compensate can automatically replace it.

To implement this, we analyzed the functions of services used in our approach and decomposed them into smaller units of work, isolating critical sections. We discovered that "work units" fall roughly into two categories: work that can be executed by reactors, i.e., multiple service process working at the same time (e.g., processing query requests) and work that can be carried out only by actors, i.e., exactly one service process at a time (e.g., critical sections). We then devised a role relay system that allowed us to weave work of the service components back together. According to the characterization above, for each service we defined several roles: one role for the reactor processes, and one for each of the unique work units. The reactors work on units contained in an internal queue whereas the actors work both on reactor tasks and on their specialized function.

We then implemented the service processes such that they can take on any role required by the service. Whenever one of the reactors dies, its work is taken over by the other reactors (and a new reactor is eventually added). Whenever one of the actors dies, we use the requirement that any service process can take on any role, stage a leader election among the reactor processes, and designate a new actor from amongst those processes (a new reactor is eventually added). In this way we ensure that critical service processes are quickly replaced on failure.

4.3 Implementation

Each component service is made up of multiple worker processes which share a common AMQP message queue. Each worker functions as a reactor, pulling messages from the queue and executing them. Messages are service calls specifying a service operation and a set of parameters. The reactor executes the operation and may return a result to the caller. Each operation is idempotent or has known retry semantics.

All service workers are bootstrapped in the same way with the same configuration. Actors are determined by leader elections implemented via Apache ZooKeeper [9]. These elections are participated in by all workers and determine exactly one worker to be promoted to the actor role. If this elected actor later dies, or is partitioned away, it will lose its actor role and another worker will be elected within a predictable bound of time.

This model is used for each of the component services. The Domain Manager has two actor roles. The Decider actor performs regular invocations of the decision engine for each domain. The Doctor monitors the heartbeats received from each deployed node and determines when to mark nodes as unhealthy.

The Provisioner has a single Leader actor that performs queries of IaaS sites and contextualization services. It adjusts node lifecycle states based on the information determined by these queries.

The Image Library has no actor roles; all workers serve as reactors only.

5. Evaluating Time to Scale

To evaluate how reactive our solution is, we tested the time to scale (TTS) of our service. Our focus was on a scenario which represents the worst-case scenario for our system: where potentially multiple clients request modifications for potentially multiple domains by the deployment of a VM; our TTS measured how quickly those aggregate requests can be satisfied. In our experiments this situation was approximated by using one client and single domain. We tested the system on EC2 micro instances. Initially, we create an empty domain and reconfigure it to be composed of 1,850 instances. Each of these instances are requested independently to the IaaS provider with a VM image containing a pre-installed deployment of our VM agent. Each VM is contextualized at boot time to configure the information unique to this domain.

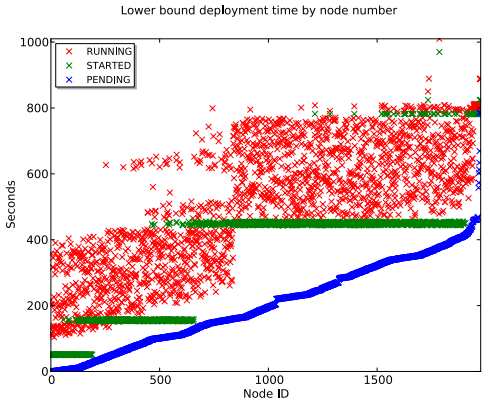

Figure 2: Deployment time by node number

Figure 2 shows that 1,850 instances can be deployed in around 13 minutes. It also shows the various components of the deployment process.

The blue points shows when request to deploy has been acknowledged by Amazon. In our experiments, we found that issuing requests to the IaaS provider can very quickly become a bottleneck. To compensate for that, we deployed a multi-Provisioner, implemented based on the domain model described above; the data shown in the figure was deployed using 10 Provisioner processes. Further, we found that in addition to instance limits (which are explicit and specific to accounts), Amazon also has request limits per time unit which are not explicitly defined and hard to track; however, once such a limit is exceeded, the requests are rejected and the deployment of those VMs needs to be re-requested. One possible way of dealing with it is implementing exponential backoff. Another issue we discovered is that Amazon EC2 micro instances can be unreliable, with a small fraction of instances never reaching a successful VM boot. The strategy we implemented here is overprovisioning, i.e., requesting a certain percentage of VMs above the desired number (in our case it was ~10%) so that a desired number may eventually be obtained.

The green points shows when Amazon deployed the VM, i.e., transferred the image and started it booting. While measuring this quality we discovered that Amazon sometimes does not report transition to this state in a timely fashion; sometimes the reporting is so far from reality in fact that it looks as if the VM has died. For

this reason we approximated the moment when the VM reaches this state by the time it starts contextualization (as reported by the contextualization agent). The pattern visible in the green straight lines in the Figure 2 shows when groups of VMs reached that state and is an artifact of sampling frequency; more frequent sampling of this information would produce more leaning lines.

Finally, the red points show when contextualization for the specific instances was finished. This is the point when the VM is ready for use, i.e., the operating system has booted and all additional configuration work has finished. This group of points has the highest variance as system boot and contextualization take time and phenomena such as noisy neighbour introduce a significant level of variability to how fast processes in the VMs can execute. Another potential issue is sequential context queries, which make some VMs wait.

6. RELATED WORK

Several commercial tools [11-14] provide capabilities covering some subset of the work described here but are either tied to specific commercial provider, specialized for a particular mode of usage not consistent with scientific requirements, or closed proprietary solutions that cannot be studied or adapted to scientific resources. Our purpose is to build a highly adaptable system capable of executing in a multi-cloud environment.

The University of Victoria's Cloud Scheduler project [15, 16] also demonstrated the viability of similar concepts when applied to the scientific community, but with emphasis on sharing resources provisioned in the cloud between different communities using batch queue systems, rather than more general need-based scaling. Cloud Scheduler monitors a Condor queue for new jobs, and provisions resources across Nimbus clouds and Amazon EC2.

Several projects [17-21] evaluate different policies in the context of auto-scaling systems such as the one presented here. Our objectives are focused on the design and implementation of the system itself rather than the policies described in those works.

7. SUMMARY

We described a system that allows users to automatically outsource their computational needs to infrastructure cloud providers. We describe the properties of scalability and availability that we seek to provide in our design as well as the model for applications to use in order to leverage them. Our EC2 evaluation highlights several deployment challenges at infrastructure cloud providers that arise when implementing a system of this type to work at large scales and provides insight into how they were overcome.

8. ACKNOWLEDGMENTS

This material is based on work supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed.", in part by the OOI Cyberinfrastructure program funded through the JOI Subaward, JSA 7-11, which is in turn funded by the NSF contract OCE-0418967 with the Consortium for Ocean Leadership, Inc., and in part by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

9. REFERENCES

[1] Armbrust, M., et al., Above the Clouds: A Berkeley View of Cloud Computing. 2009, University of California at Berkeley.

[2] Amazon Elastic Compute Cloud (Amazon EC2) http://www.amazon.com/ec2.

[3] Jackson, K., 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 Amazon Web Services Cloud. in CloudCom. pp.159-168, Nov. 30 2010-Dec. 3 2010, Indianapolis, IN.

[4] Berkeley Open Infrastructure for Network Computing. 2002; Available from: http://boinc.berkeley.edu.

[5] Litzkow, M.J., M. Livny, and M.W. Mutka, Condor - A Hunter of Idle Workstations, in 8th International Conference on Distributed Computing Systems. 1988. p. 104-111.

[6] The Swift Parallel Scripting Language: http://www.ci.uchicago.edu/swift/main/.

[7] Harutyunyan, A., P. Buncic, T. Freeman, and K. Keahey, Dynamic virtual AliEn Grid sites on Nimbus with CernVM. Journal of Physics: Conference Series, 2010. 219(7).

[8] Marshall, P., K. Keahey, and T. Freeman, Elastic Site: Using Clouds to Elastically Extend Site Resources. CCGrid 2010, 2010.

[9] Hunt, P., M. Konar, F.P. Junqueira, and B. Reed, ZooKeeper: Wait-free Coordination for Internet-Scale Systems, in USENIX Annual Technology Conference 2010.

[10] Bresnahan, J., T. Freeman, D. LaBissoniere, and K. Keahey, Managing Appliance Launches in Infrastructure Clouds. TeraGrid Conference, 2011.

[11] Amazon Web Services: Auto Scaling: http://aws.amazon.com/autoscaling/.

[12] Cloud Foundry. 2011: http://www.cloudfoundry.com/.

[13] RightScale: www.rightscale.org.

[14] SCALR: http://scalr.com/.

[15] Armstrong, P., et al., Cloud Scheduler: a Resource Manager for a Distributed Compute Cloud. 2010.

[16] Sobie, R., et al., Data Intensive High Energy Physics Analysis in Distributed Cloud, in High Performance Computing Symposium. 2011: Montreal, Canada.

[17] Mao, M. and M. Humphrey. Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. in SC11. 2011.

[18] Marshall, P., H. Tufo, K. Keahey, D. LaBissoniere, and H.M. Woitaszek. Architecting a Large-Scale Elastic Environment - Recontextualization and Adaptive Cloud Services for Scientific Computing. in ICSOFT. 2012. Rome, Italy.

[19] Marshall, P., H. Tufo, and K. Keahey. Provisioning Policies for Elastic Computing Environments. in 9th High-Performance Grid and Cloud Computing Workshop and the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS). 2012.

[20] Ruth, P., P. McGachey, and D. Xu, VioCluster: Virtualizationfor Dynamic Computational Domains. IEE International Conference on Cluster Computing, 2005.

[21] Assunacao, M.D., A.D. Constanzo, and R. Buyya. Evaluating the Cost-Benefit of Using Cloud Computing to Extend the Capacity of Clusters. in 18th ACM Symposium on High Performance Distributed Computing. 2009.