Rebalancing in a Multi-Cloud Environment

Dmitry Duplyakin
University of Colorado
dmitry.duplyakin@colorado.edu

Paul Marshall
University of Colorado
paul.marshall@colorado.edu

Kate Keahey
Argonne National Laboratory
University of Chicago
keahey@mcs.anl.gov

Henry Tufo
University of Colorado
henry.tufo@colorado.edu

Ali Alzabarah
University of Colorado
ali.alzabarah@colorado.edu

ABSTRACT

With the proliferation of infrastructure clouds it is now possible to consider developing applications capable of leveraging multi-cloud environments. Such environments provide users a unique opportunity to tune their deployments to meet specific needs (e.g., cost, reliability, performance, etc.). Open source multi-cloud scaling tools, such as Nimbus Phantom, allow users to develop such multi-cloud workflows easily. However, user preferences cannot always be achieved at the outset for a variety of reasons (e.g., availability limitations, financial constraints, technical obstacles, etc.). Despite this, it is possible that many preferences can be met at a later time due to the elastic nature of infrastructure clouds. Rebalancing policies, which replace instances in lower-preferred clouds with instances in higher-preferred clouds, are needed to meet these preferences.

We present an environment that manages multi-cloud deployment rebalancing by terminating instances, in lower-preferred clouds and launching replacement instances in higher-preferred clouds to satisfy user preferences. In particular, users define a preferred cloud ratio, e.g., 75% instances on one cloud and 25% instances on another, which we attempt to achieve using rebalancing policies. We consider three rebalancing policies: 1) only idle excess instances are terminated, 2) excess instances are terminated gracefully, and 3) worker instances are aggressively terminated, even if they are running user jobs. To gauge the effectiveness of our rebalancing strategy, we evaluate these policies in a master-worker environment deployed across multiple NSF FutureGrid clouds and examine the ability of the policies to rebalance multi-cloud deployments appropriately, and analyze trade-offs.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems; K.6.2 [**Installation Management**]: Computing Equipment Management

General Terms

Management, Design, Experimentation

Keywords

Cloud computing, Infrastructure-as-a-Service, Rebalancing, Policies

1. INTRODUCTION

Multi-cloud environments leverage infrastructure-as-a-service (IaaS) clouds to integrate resources from multiple cloud infrastructures. These deployments allow users to take advantage of differences in various clouds, including price, performance, capability, and availability differences. As an example, a user may leverage multiple clouds, including community clouds, such as FutureGrid [14], and for-pay public clouds, such as Amazon EC2 [18]. A community cloud, for example, often is the user's preferred cloud because it is provided at a reduced monetary cost, or possibly even free. However, it may offer significantly fewer resources than larger for-pay public cloud providers. Users are then faced with a dilemma of choosing where to deploy their instances. In such a scenario, a user may choose to delay deployment, and thus delay processing his workload, until his preferred cloud is available. However, another option is to define an explicit preference for the clouds (e.g., specifying that the less expensive clouds should be used whenever available) but immediately deploy instances wherever possible and then rebalance the deployment at a later time as needed. For example, because community clouds with limited resources are not always available, especially if demand is high, the environment can launch instances on lower-preferred clouds, such as public cloud providers. As clouds with a higher-preference become available, the environment should rebalance, automatically downscaling, that is, terminating instances, in lower-preferred clouds, and upscaling, launching instances in higher-preferred clouds.

Upscaling is typically automated by auto-scaling services, such as Nimbus Phantom [9] or Amazon's Auto Scaling Service [13], which allow users to define the number of instances that should be deployed in each cloud. Auto-scaling services then launch the instances and monitor them, replacing them if they crash or are terminated prematurely. However, downscaling isn't typically automated and presents a number of unique challenges that must be addressed. For instance, users must be able to clearly define their preferences for different clouds and policies must be created. These policies should identify which clouds to terminate instances in, which instances to terminate, and how the instances should be terminated (e.g., wait until instances are idle or immediately terminate instances with running jobs, thus causing jobs to be rescheduled). Rebalancing policies should balance user requirements as well as workload and environment characteristics to avoid introducing excessive workload overhead. To accomplish this, rebalancing implementations must integrate with existing workload resource managers and provide the functionality required by the policies, such as identifying the state of instances (e.g., busy or idle) or marking workers as "offline".

In this paper, we examine factors that influence rebalancing decisions. Specifically, we consider master-worker environments where resources are dedicated to processing user workloads. In this context, we attempt to identify whether, and if so under what conditions, we may be justified in causing jobs to be killed by terminating running instances for rebalancing purposes. We propose three rebalancing policies that use job and instance

Science Cloud'13, June 17, 2013, New York, NY, USA.

information to determine whether or not an instance should be terminated. The first policy waits until instances are idle before they are terminated. The second policy forcibly marks instances offline, allowing them to finish running jobs but preventing them from accepting new jobs, and then terminates the instances once they become idle. The third policy uses a "progress threshold" to decide whether or not to terminate the instances. For example, if the threshold is set to 25% and a job is expected to run for two hours, a node that has been running the job is only eligible for termination during the first 30 minutes of execution. We also develop a multi-cloud architecture that incorporates these policies with master-worker environments.

For experimental evaluation, we deploy our solution using multiple NSF FutureGrid clouds and use Nimbus Phantom for our auto-scaling service. Our evaluation examines the benefits and trade-offs associated with each policy. Less aggressive policies are able to provide zero-overhead rebalancing at the expense of leaving the deployment in a non-desired state for longer periods of time. More aggressive policies, on the other hand, rebalance the environment quickly but introduce workload overhead and delay overall workload execution. The most aggressive policy, however, appears to strike the best balance by rebalancing the environment quickly, reducing cost by up to a factor of 3, while only increasing workload execution time by up to 14.7%.

The remainder of the paper is organized as follows. In Section 2 we examine the general approach of rebalancing in multi-cloud environments and describe our implementation and policies. In Section 3, we evaluate the policies and identify the trade-offs associated with each policy. In Section 4, we discuss the related work and in Section 5 we propose directions for future work. We conclude in Section 6.

2. APPROACH

2.1 Models and Assumptions

We propose a multi-cloud environment that is capable of processing user demand and distributing work to resources deployed across multiple clouds. For example, such an environment might consists of a pool of web servers, distributed between different cloud data centers, responding to user requests for a single website. Another example is an HTCondor pool with workers distributed between multiple clouds, all pulling jobs from a single, central queue. When deploying such an environment, a user may define a desired request for how many resources to deploy across different clouds throughout the environment. These requests can be expressed in two forms: 1) in terms of absolute numbers of instances needed in selected clouds, e.g., R={32 instances in cloud A, 8 instances in cloud B}; 2) in terms of total numbers of instances and preferred ratios, e.g., R={40 instances total; 80% in cloud A, 20% in cloud B}. However, such requests may lead to situations where a deployment cannot be satisfied, at least initially. For example, instead of matching the request R={32 instances in cloud A, 8 instances in cloud B}, we may have 24 instances in cloud A (which may not be able to launch additional instances) and, thus, end up with 16 instances in cloud B. Therefore, as the environment adapts, additional instances should be launched in cloud A whenever possible and instances in cloud B should be terminated until the users' preferences are met.

We assume that the multi-cloud deployment is deployed and managed by a central auto-scaling service. In particular, we use the open source Phantom auto-scaling service, which is responsible for servicing requests to deploy instances across multiple clouds and monitoring those instances, replacing them if they crash or are terminated prematurely. We also assume that the environment uses a master-worker paradigm to process demand, such as an HTCondor pool, using a "pull" queue model. That is, workers distributed across multiple clouds request jobs from a central queue when they are available to execute jobs. The central scheduler reschedules jobs when workers fail or are terminated for rebalancing. The job scheduler must also be able to: 1) provide status information about workers, including job state (e.g., busy or idle), jobs running on the workers, and up-to-date job runtimes; 2) provide information about the queue, including a list of running and queued jobs; 3) add an instance to the worker pool; and 4) remove an instance from the worker pool, either gracefully by allowing it to finish running its job or immediately by preemptively terminating the worker and its jobs. Many modern job schedulers, including Torque and HTCondor, provide these capabilities. Because we use a "pull" queue model, only embarrassingly parallel workflows are considered. In such workflows, jobs can be terminated, rescheduled, and re-executed out of order without consideration for other jobs in the set.

In this context, we define the following terms:

- *Multi-cloud request*: a request that specifies the configuration of a multi-cloud deployment either using absolute numbers (e.g., 32 instances in cloud A and 8 instances in cloud B) or as a ratio (e.g., 40 instances with 80% in cloud A and 20% in cloud B). It is specified by the user and typically represents his desired preferences for various clouds.

- *Desired state*: when the multi-cloud deployment matches the specified multi-cloud request. For example, if the user specifies a multi-cloud request with 32 instances in cloud A and 8 instances in cloud B and the running deployment matches this request, then it is in the desired state.

- *Rebalancing*: the process of transitioning the deployment in an attempt to reach the desired state. This occurs when the deployment is not in the desired state so instances are terminated in some clouds and replacement instances are launched in other clouds in order to reach the desired state. Even after the desired state is reached, the system must maintain it. However, it is possible for the system to return to an undesired state. For example, if failed instances cannot be replaced on the preferred cloud and are instead deployed on a less-preferred cloud, then the system will again attempt to reach the desired state.

- *Downscaling*: the termination of running instances that occurs during rebalancing. Depending on the difference between the current deployment and the request, there may be a need to terminate many instances (e.g., terminate all instances in cloud A) or only a particular one (e.g., terminate an instance in cloud A with a particular ID). Selecting the appropriate instance to terminate is a non-trivial task. This process has two main components: 1) real-time information about all instances is required, such as jobs currently running on those instances, and 2) specific instances need to be identified for termination according to the policy. For example, a policy may select an instance for termination based on the progress of the job it is executing. An instance that is close to completing its job is considered to be more valuable than an instance that has just started execution of its job. Additionally, if an instance is idle (i.e., it's not currently executing a job) then it may be terminated immediately.

- *Upscaling*: the process of deploying instances throughout the multi-cloud environment to ensure that the total number of

instances specified in a multi-cloud request is satisfied. Upscaling attempts to launch instances on clouds with a higher preference, however, if such clouds are unavailable then upscaling will deploy instances on clouds with lower preferences.

- *Excess instances*: instances in a particular cloud that exceed the desired amount. For example, if a multi-cloud request specifies that 15 instances should be deployed with 10 instances in cloud A and 5 instances in cloud B, but at the given time all 15 instances can only be deployed in cloud B, then cloud B would have 10 excess instances.

While there may be cases where it is crucial for multi-cloud deployments to satisfy user preferences before job execution begins, we believe stalling job execution until the environment is in the desired state is not necessary. Instances that are already deployed in less desirable clouds can yield partial results while rebalancing occurs instead of delaying deployment and providing no results at all. A multi-cloud deployment that is running jobs should continue to process jobs in less desirable clouds and rely on rebalancing to achieve the desired state.

2.2 Architecture

The multi-cloud architecture that we designed extends the architecture presented in [15]. It deploys workers across several cloud resource infrastructures and dynamically balances this deployment based on user-defined preferences. The architecture is depicted in Figure 1 and consists of four main components: (1) a workload management system (including a job scheduler and workers), (2) sensors to monitor demand, (3) policies to scale the number of deployed instances up or down, and (4) an auto-scaling service to enforce the chosen policy.

In this environment, the master node hosts the job scheduler for the workload management system that accepts user jobs and schedules them to run on worker nodes deployed across multiple clouds (see Figure 1). The sensor periodically queries the job scheduler and gathers information about the queued jobs and their runtimes as well as worker status. The sensor provides this information to the decision engine, which then executes a policy that decides how to adjust the deployment, potentially downscaling it on one cloud and upscaling on another, to satisfy user preferences.

Users define their preferences for different clouds in their multi-cloud requests, e.g., specifying that they want 20 instances, with half deployed on one cloud and half on another cloud. The decision engine also includes per-cloud timers and predefined time-outs to avoid undesired scenarios where upscaling and downscaling are performed on the same cloud within a short period of time.

To enact upscaling or downscaling, the auto-scaling decision engine instructs the service to deploy, maintain, or terminate workers across the different clouds. The job master and workers operate relatively independently of the multi-cloud architecture, integrating new workers that are deployed, processing jobs on available instances, and rescheduling any jobs that are terminated prematurely. However, in some cases, the decision engine may need to make specific requests of the workload management system. For example, policies that elect to mark workers offline, allowing them to finish their jobs and preventing them from accepting new jobs, must be able to communicate with the workload management system. In our architecture this is accomplished by communicating the request through the sensor, which then performs the operation on the master node.

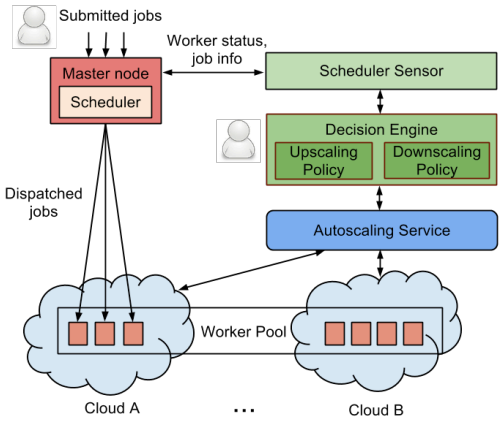

Figure 1. Multi-cloud architecture with workload and management components where rebalancing is guided by upscaling and downscaling policies.

2.3 Rebalancing Policies

Rebalancing alters multi-cloud deployments and may be disruptive from the workload's perspective. For example, terminating an instance may cause a running job to be terminated prematurely, causing the job to be rescheduled for execution on a new worker. The workload's total execution time may increase as a result. On the other hand, executing a user's workload in a multi-cloud environment that is not in the desired state might lead to other problems, including overall performance degradation, unexpected expenses, additional data transfers, etc. Therefore, rebalancing policies are needed for multi-cloud environments; their objective is to achieve the desired state, if possible, and only minimally impact user workloads.

In this work, the existing auto-scaling decision engine performs upscaling and tries to maintain the total number of instances as specified in the multi-cloud request. For downscaling, we propose the following policies:

- *Opportunistic-Idle* (OI): waits until excess instances in less desired clouds are idle (i.e., not running jobs according to information from the sensor) and then terminates them. This policy continues to terminate excess idle instances until the deployment reaches the desired state. It begins with clouds that have the most excess instances before proceeding to the clouds with fewer.
- *Force-Offline* (FO): is similar to OI but excess instances are terminated gracefully, that is, jobs are allowed to complete before the instances are terminated. To terminate instances gracefully, the auto-scaling service notifies the jobs scheduler which instances are to be terminated, and the job scheduler then marks them "offline". Graceful termination allows those instances to complete currently running jobs. Once the jobs complete, the workers do not accept any new jobs and can be terminated. This policy requires that the job scheduler support the ability to "offline" workers. FO does not terminate instances once the desired number of instances is reached in each cloud.
- *Aggressive Policy* (AP): sacrifices work cycles and discards partially completed jobs in order to satisfy requests in the shortest amount of time possible. This policy terminates excess instances even if those instances are currently running jobs. To minimize overhead associated with job re-execution, this policy

proceeds in termination from instances with jobs that have been running for the least amount of time to instances with jobs that have been running for a longer time. Additionally, since the amount of work to discard may vary, we include a tunable parameter, work threshold (measured in percent). Work threshold specifies how much progress on its current job with respect to the job's walltime (i.e., expected job's runtime) an instance has to make before this instance may no longer be terminated. For example, if we choose the 25% threshold and one of the instances executes a two-hour job, the policy is only allowed to terminate it for up to 30 minutes after beginning of execution. While job runtimes may be predictable, we avoid relying on the accuracy of such predictions and consider walltimes that are explicitly provided. Job walltimes are typically limited by the cluster administrator (e.g., to 24 hours), and prevent users from specifying excessive walltime requests.

2.4 Implementation

We leverage a number of existing technologies for our multi-cloud deployment. Specifically, we use infrastructure clouds, such as Nimbus [17] and Amazon EC2 [18], which provide on-demand resource provisioning. Our environment integrates with the open source Phantom service for auto-scaling. Phantom [9] is responsible for servicing multi-cloud requests and deploying the instances across the specified clouds. It also continually monitors the deployment and maintains the requested number of instances. We also rely on a master-worker workload management system, specifically HTCondor [19], which monitors and manages a pool of workers across distributed resources. This includes the ability to submit jobs to a central queue and schedule jobs across distributed workers. HTCondor also includes job resubmission and migration capabilities, which guarantee that every job eventually completes even if some workers are terminated prematurely. Lastly, HTCondor provides the required information about workload execution, specifically, which jobs are queued or running, as well as the amount of time they have been running.

To integrate with the leveraged technologies, we develop two additional components for the implementation: the sensor and the rebalancing policy. The sensor, written in Python, communicates with HTCondor master nodes and obtains necessary job and worker information for the policies using the condor_status and condor_q commands. This provides three pieces of information: 1) the number of HTCondor workers running in each cloud (e.g., condor_q -run), 2) information required to identify idle workers (e.g., condor_status), and 3) a list of workers and their current runtimes and walltimes. However, some policies require that the sensor also communicates with the workload management system (e.g., when marking nodes offline). Therefore, the sensor is both able to send information to the policy and receive instructions from it. When the policy instructs the sensor to remove a worker from the pool, the sensor issues the condor_off command. To terminate an instance gracefully (e.g., when using FO), the sensor executes the following command: condor_off –peaceful <hostname>. AP removes instances from the pool instantly using: condor_off –fast <hostname>.

In addition to workload information, the system must also query the auto-scaling service, Phantom, and the individual IaaS clouds to identify all of the instances in the auto-scale group as well as their distribution across the clouds. The instance IDs reported by Phantom also need to be compared with the instance IDs reported by individual clouds in order to identify the specific clouds auto-scale instances are running on. This information is used by the policy, along with workload information, to guide rebalancing decisions. The policies, written in Python, collect the necessary information and then attempt to match the deployment with the user's multi-cloud request, downscaling on clouds with excess instances and upscaling on high-preferred clouds when needed. To accomplish this, the downscaling component of the policy communicates with the auto-scaling service, using the Python boto library [7], and makes adjustments to the configuration in order to satisfy the request. Phantom then provides an ordered cloud list and capacity parameters that can be adjusted for this purpose. The ordered cloud list specifies the maximum number of instances in each cloud and can be set to match the request for a specific cloud, while the capacity parameter controls the total number of instances across all clouds and can be set independently from the ordered list.

Upscaling is performed by an existing Phantom decision engine, which tries to maintain a total number of instances deployed across the preferred clouds. Phantom implements its own n_preserving policy, which allows it to maintain the requested number of running instances, replacing failed instances when needed. When Phantom replaces failed or terminated instances, it does so according to the ordered cloud list, meaning that if instances are terminated in a lower-preferred cloud, it first attempts to deploy replacement instances in a higher-preferred cloud. It only resorts to lower-preference clouds if it is unable to deploy instances in higher-preferred clouds (e.g., due to unavailability).

3. Evaluation

To evaluate proposed rebalancing policies, we examine the ability of the environment to rebalance the deployment in order to satisfy multi-cloud requests and reach the desired state. That is, the policies attempt to adjust the deployment to match user preferences as quickly as possible while avoiding excessive workload overhead. Specifically, we consider the scenario where cloud availability changes over time due to external factors. For example, when other users terminate instances it may be possible to deploy additional instances in higher-preferred clouds and downscale in lower-preferred clouds. We choose not to simulate this type of unexpected change in availability directly; instead we focus on the behavior of our policies once such changes in availability occur. Therefore, in each of our experimental evaluations, we assume that the evaluation begins with the deployment in an undesired state but that higher-preferred clouds now have additional capacity available, allowing the policies to attempt to reach the desired state.

For our environment, we use NSF FutureGrid [14] and two workload traces from the University of Notre Dame's Condor Log Analyzer [8]. HTCondor is used as the workload management system. We leverage its ability to reschedule jobs that are terminated prematurely. On FutureGrid we use the Hotel cloud at the University of Chicago (UC) and Sierra at the San Diego Supercomputer Center (SDSC). Both clouds use Nimbus as the IaaS toolkit and Xen for virtualization. The master node and all the worker nodes run Debian Lenny images, approximately 1 GB compressed, with HTCondor 7.8.0 as the workload manager. The master node VM has two 2.93GHz Xeon cores and 2 GB of RAM. Workers have one 2.93GHz Xeon core and 2 GB of RAM. Instances are contextualized as the master or a worker automatically at boot. In particular, the IaaS userdata field is used to provide the hostname of the master, in which case, the node configures itself as a worker and attempts to join the master. If the field is empty, the instance configures itself as the master.

Figure 2. OI terminates 16 instances after most jobs complete.

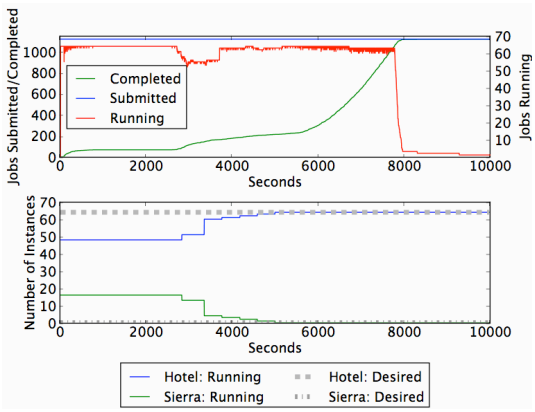

Figure 3. Gradual downscaling with FO.

Figure 4. Considerate aggressive downscaling with AP-25.

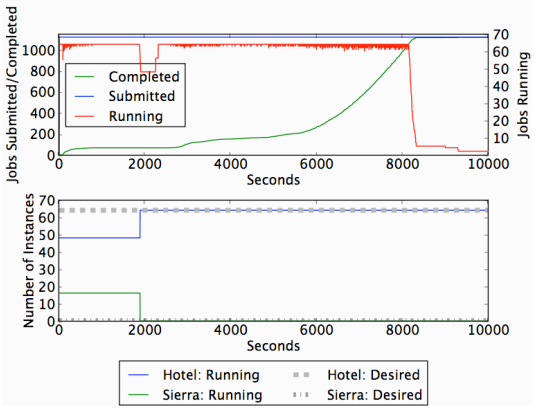

Figure 5. Fast aggressive downscaling with AP-100.

In this evaluation, we differentiate between Hotel and Sierra by specifying different costs for their instances. Specifically, we assume instances on Hotel have 1 unit of cost and instances on Sierra have 2 units of cost. We also specify a multi-cloud request for a total of 64 instances with 100% of the instances on Hotel and 0% on Sierra, representing the case where a user desires that all of his instances be deployed in the less expensive cloud. However, for each experiment, we initialize the environment to have 75% of the instances in Hotel (48 instances) and 25% of the instances in Sierra (16 instances), requiring rebalancing to occur in order to reach the desired state.

For a job trace, we combine two traces from the HTCondor Log Analyzer [8], one consisting primarily of smaller jobs and another that contains longer running jobs. The traces are combined into a single workload that is submitted immediately at the beginning of the evaluation by selecting jobs randomly from each trace. With this approach we consider a workload that consists of a variety of job runtimes. Individual jobs are submitted as sleep jobs, which sleep for the runtime specified in the trace. The combined workload consists of 1120 jobs, with a minimum runtime of 54 seconds and a maximum runtime of 119 minutes. The median runtime is 106 seconds and the mean is 443 seconds.

The system can be configured to execute the policy at any user-defined interval. For this evaluation, we configure the policy to execute every 30 minutes, beginning 30 minutes after the initial job submission, in order to rebalance the deployment regularly while still allowing for some jobs to complete between rebalancing intervals. Exploring different policy execution intervals and their impact is left for future work. We expect overly aggressive intervals (i.e., a short policy execution interval) and relatively passive intervals (i.e., a long policy execution interval) to negatively impact the environment by either rebalancing too frequently and preventing jobs from finishing or not rebalancing often enough, causing high excess cost. However, further experimentation is needed to identify the appropriate balance between the policy execution interval and workload characteristics, including the rate that jobs are submitted and the duration they execute.

For experimental evaluation, we define the following metrics:

- *Workload execution time*: the amount of time required for the entire workload to complete, that is, the amount of time from when the first job is submitted until the time the last job completes.
- *Workload overhead percent*: the total percent of time jobs run before being terminated prematurely. For example, if a two-hour job runs for 30 minutes before it is terminated, causing it to be re-queued and needing to be rerun, then the job experiences 25% overhead (assuming that the second run finishes).
- *Convergence time*: the amount of time it takes for the deployment to rebalance from a non-desired state to the desired state once the evaluation begins. For example, if the system is operating in an undesired state at the moment the first job starts execution, convergence time is the time from that moment until the system reaches the desired state (i.e., the specified multi-cloud preference is satisfied).

- *Excess cost* (EC): the total user-defined cost associated with running excess instances. EC is described as:

$$EC = \sum_{r}^{R} \sum_{i}^{Ir} c_r * p_r(t_i - l_i)$$

The variables are defined as:
R: set of all cloud resources used for the deployment,
I_r: set of all excess instances for every cloud resource, r,
c_r: user-defined instance cost for cloud resource, r,
p_r: time accounting function for cloud resource, r,
t_i: termination time for instance, i,
l_i: launch time for instance, i.

R consists of all clouds that have running instances. I_r is a set containing all of the excess instances for cloud resource r. c_r represents the user's definition of cost associated for an excess instance in cloud resource, r, for one unit of time. Depending on the cloud resource, r, p_r may differ to represent various time accounting methods, such as per-second usage or rounding up to the nearest hour, etc. t_i and l_i are the specific instance termination and launch times. Intuitively, EC is intended to represent the cost of running excess instances, that is, using instances on clouds beyond the amount specified in the multi-cloud request. EC offers a fine-grained metric compared to convergence time, which only provides a course representation of when the environment finally reaches the desired state.

In addition to these metrics, we also include a set of job traces that show the number of instances running as well as the number of jobs submitted, running, and complete. As described earlier, we specify a multi-cloud request with a preference for 64 workers in Hotel and 0 in Sierra, but the environment is initialized in an undesired state with 48 workers running in Hotel and 16 workers in Sierra. Therefore, the rebalancing policies attempt to terminate all workers in Sierra, while Phantom replaces the instances in Hotel until it has 64 running worker instances.

3.1 Understanding Deployment Transformations

All policies pursue the same goal: downscaling 16 instances on Sierra, the lower-preferred cloud, and upscaling on Hotel, the higher preferred cloud. Traces are included for the four different policies, OI (Figure 2), FO (Figure 3), AP-25 (Figure 4), and AP-100 (Figure 5). AP-25 uses a 25% threshold and AP-100 uses a 100% threshold. The traces show job information (jobs completed, submitted and running), as well as the distribution of worker instances across Hotel and Sierra.

In the experiment shown in Figure 2, OI only attempts to terminate idle instances. OI is first able to perform downscaling approximately 9000 seconds after initial job submission, that is, when all 16 instances in Sierra are idle and can be terminated after the entire workload has been processed. This downscaling has no effect on the workload since the few jobs that are still running at the time of downscaling occupy several of Hotel's instances (where downscaling doesn't occur) and continue to run unaffected. This downscaling policy may be valuable if the user continues to use the same deployment and executes another workload at a later time. Then, the deployment has already been adjusted and, if no failures happen, all new jobs are executed in the rebalanced environment. In general, OI is only useful in situations where user jobs are submitted in a series of batches that are interleaved with periods of idle time when rebalancing can occur (i.e., OI will not work on a fully utilized system).

Figure 3 illustrates an experiment where FO terminates instances gracefully. This policy marks all 16 excess instances "offline" during its first evaluation at 1800 seconds after initial job submission. Instances that are executing jobs at that time must wait for jobs to finish before the instances can be removed from the worker pool and terminated. Thus, after 2300 seconds, 3 instances are terminated, after 2800 seconds 8 additional instances are terminated, etc. Rebalancing continues until there are no remaining instances in Sierra, which occurs when the last instance is finally terminated after 4600 seconds. Every termination causes a noticeable drop in the trace showing the number of running jobs. These drops indicate temporary reductions in the worker pool that follow downscaling actions and last until replacement instances boot and join the worker pool. Since there are less than 64 running instances at certain periods of time, it takes longer for the workload to complete in this experiment than for the OI experiment, which rebalances after the workload completes. However, we do observe a faster convergence time for FO than OI; all 64 instances are running on Hotel after 4600 seconds. FO demonstrates an average execution time increase of 12.6% and an average convergence time decrease of 48.4% with respect to OI's time characteristics (Figure 6).

We evaluate two variations of AP: AP-25 and AP-100. The first, AP-25, terminates excess instances only if jobs running on those instances have been running for less than 25% of their requested walltime. We choose to use the 25% threshold to demonstrate a considerate implementation of AP, which attempts to achieve low overhead rather than fast convergence. In contrast, AP-100, having no threshold, terminates all excess instances without consideration of job progress (specifically, AP-100 terminates instances with jobs that have been running for less than 100% of their requested walltime). This is a special case when the user prefers to rebalance as fast as possible regardless of the amount of work that is discarded.

Figure 4 shows that AP-25 cannot perform the necessary adjustments all on the first try, and thus, rebalancing is a gradual process, similar to FO. A 25% threshold yields premature termination of 16 jobs that are running on Sierra's instances, which individually have been running for 267 seconds on average. The maximum runtime among those jobs is 933 seconds. In this experiment the total amount of discarded work cycles is about 71 minutes, which is only 0.9% of the total CPU time used by all 64 instances throughout the evaluation.

Figure 5 shows AP-100, which converges to the desired state the fastest. This policy terminates all 16 instances in Sierra during its first evaluation after 1800 seconds. At that time, 16 jobs are terminated prematurely after running for an average of 28 minutes. The maximum runtime is over 30 minutes. 458 minutes of discarded work result in 5.2% workload overhead.

3.2 Understanding Trade-offs

Figure 6 shows workload execution time, convergence time, percent workload overhead, and excess cost for all four policies. We calculate mean values and standard deviations for these metrics using a series of experiments with three iterations for each policy. OI provides the shortest workload execution time, while other policies introduce a noticeable increase in the execution time. This is because OI does not affect the workload, as described earlier, and waits for the entire workload to complete before rebalancing occurs. Switching from OI to FO, AP-25 and AP-100 introduces an average workload execution increase of 12.6%, 12.9% and 14.7%, respectively. FO, AP-25 and AP-100 all appear to have comparable execution times, while OI consistently provides the lowest execution time with negligible variance (indicated by no error bar in the graph).

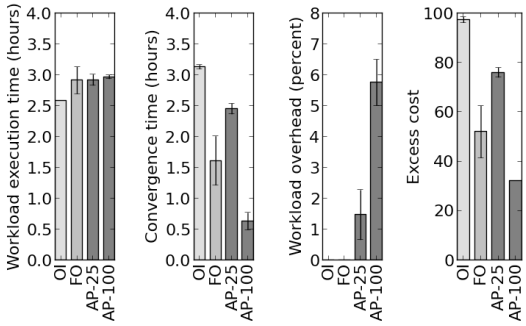

Figure 6. Mean workload execution time (hours), convergence time (hours), workload overhead (percent), and excess cost. Three iterations are run for each policy and no error bar indicates very little deviation. OI and FO also have 0% workload overhead.

OI's convergence, shown in Figure 6, happens after workload completion, specifically, at 3.1 hours. FO's gradual downscaling completes after 1.6 hours. AP-25's convergence completes after 2.5 hours. As expected, AP-100 provides the fastest convergence time, which is approximately 0.6 hours.

For workload overhead (that is, the amount of discarded work), OI and FO have none (indicated with zero-height bars in the rightmost graph). By design, these policies never attempt to terminate instances when it leads to premature job termination. AP-25 and AP-100, however, are designed to accomplish downscaling via premature termination, demonstrate average workload overheads of 1.5% and 5.7%, respectively. The highest observed overheads are 2.4% for AP-25 and 6.6% for AP-100.

Finally, we also consider the excess cost, EC, of the experiments (Figure 6). As described earlier, we assume c_r for Hotel to be 1 unit of cost and c_r for Sierra to be 2 units of cost. However, it should be noted that this cost may differ for users and could instead correspond to a user's weighted preference or a dollar amount for instances. We consider a p_r function that rounds up instance usage to the nearest hour and obtain t_i and l_i times for instances from our experiment logs. In Figure 6, OI has the highest EC at 97.3 units of cost on average, since rebalancing doesn't occur until after the workload completes and, thus, requires excess instances to run for the majority of the experiment. Other policies incur lower EC than OI because they perform rebalancing earlier. FO has an average EC of 52.0 units of cost, AP-25 has an average EC of 76.0 units of cost and AP-100 has an average EC of 32.0 units of cost since it rebalances completely at the first policy execution, 30 minutes into the experiment.

OI, FO, and AP-100 each have different advantages and trade-offs, for example, FO converges faster than OI and both have no workload overhead but OI has the shortest workload execution time. AP-25, on the other hand, doesn't offer significant advantages; it has comparable workload execution time to FO but higher convergence time, workload overhead, and EC. AP-100, however, appears to strike the best balance between quick rebalancing and minimizing excess workload overhead and execution time. Specifically, AP-100 reduces EC by a factor of 3 over OI while introducing only 6.6% workload overhead and 14.7% workload execution time.

4. RELATED WORK

Much work has been done on leveraging elastic capabilities of infrastructure clouds based on user preferences, performance, and scalability. To upscale or downscale cloud deployments, researchers have proposed systems that predict workload computational needs during execution [1], [3], [4]. These systems take two approaches: model-based and rule-based [2]. Our purpose was not to predict workload execution, but to monitor the workload execution through HTCondor sensors and rebalance the environment effectively based on our policies. Typically, however, rebalancing is motivated by cost, where the environment terminates idle instances to avoid excessive charges. As another example, some policies govern rebalancing by triggering live migration from one cloud to another [5]. Our work is more general than such live migration approaches; in our research, workload migration is based on predefined user preferences for different clouds and aims to satisfy these preferences before the workload completes. Existing policy frameworks that execute auto-scaling policies can be adapted to include our policies [9]. Our framework relies on similar services to guide the deployment toward the user's desired state. There are also projects that examine different policies for efficient demand outsourcing from local resources to IaaS clouds [10], [11], but our work focuses on policies governing downscaling behavior in multi-cloud environments, gradually achieving the user's desired state.

5. FUTURE WORK

In future work, we will investigate mechanisms to improve rebalancing in multi-cloud deployments. In particular, we will develop a model for multi-cloud requests that include multiple objectives, for example, cost and performance. This will allow users to specify increasingly complex multi-cloud requests for their workflows. We will also investigate the relationship between cloud availability, the policy execution interval, and workload characteristics. This will provide a foundation for automated workflow-aware rebalancing in multi-cloud environments. As part of this work, we will consider additional factors to guide rebalancing processes, such as, the requested number of CPUs per job and the amount of communication between parallel jobs. Tightly coupled applications (i.e., those that have high communication to computation ratio), should execute on instances within a single cloud as much as possible and the rebalancing policies should accommodate that. In cases when only a single cloud, perhaps not the most desired one, is capable of providing a large number of CPUs requested by queued jobs, the policies should avoid downscaling in that cloud past the minimum number of instances required by the jobs. Policies should also be adapted to support workloads with dependencies. Rebalancing should be avoided or postponed if it causes rescheduling jobs and delaying many jobs in dependency chains.

6. CONCLUSIONS

We configure a multi-cloud environment capable of processing user demand where worker instances are distributed across multiple cloud infrastructures and work collaboratively to process queued tasks. We use an auto-scaling service, Phantom, to launch and monitor instances across multiple clouds. Phantom replaces instances if they crash and terminates them based on rebalancing policies. We also propose several rebalancing policies that guide these deployments towards requested multi-cloud configurations while having minimal impact on the workload, if possible.

To evaluate our policies in this environment, we use a workload that consists of traces from the HTCondor Log Analyzer and monitor the policies while they rebalance deployments that consist of 64 instances running across two clouds and processing workload jobs. Our experimental evaluation shows that the policies adjust the multi-cloud deployments properly to reach the desired state. Our opportunistic policy is able to rebalance the

deployment without introducing workload overhead, however, it requires a high excess cost. Another policy, force-offline, not only terminates idle instances but also marks them "offline," preventing them from accepting new jobs before being terminated. Finally, our aggressive policy provides the fastest convergence time and the lowest excess cost, reducing it by a factor of 3 over the opportunistic policy while only introducing 6.6% workload overhead due to premature job termination required for immediate rebalancing.

7. ACKNOWLEDGMENTS

This material is based on work supported in part by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

We would like to thank Nimbus team, specifically Pierre Riteau and Patrick Armstrong, who provided helpful information about Phantom and helped to quickly resolve issues that we encountered.

8. REFERENCES

[1] Roy, N., Dubey, A., and Gokhale, A. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing* (CLOUD '11). Washington, DC, USA, 500-507.

[2] Ghanbari, H., Simmons, B., Litoiu, M., and Iszlai, G. Exploring Alternative Approaches to Implement an Elasticity Policy. *In Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing* (CLOUD '11). IEEE Computer Society, Washington, DC, USA, 716-723.

[3] Shen, Z., Subbiah, S., Gu, X., and Wilkes, J. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (SOCC '11). ACM, New York, NY, USA, Article 5, 14 pages.

[4] Yang, J., Yu, T., Jian, L. R., Qiu, J., and Li, Y. An extreme automation framework for scaling cloud applications. *IBM Journal of Research and Development*, 55(6), 8-1.

[5] Simarro, L., Moreno-Vozmediano, R., Montero, R.S., and Llorente, I.M. Dynamic placement of virtual machines for cost optimization in multi-cloud environments. *High Performance Computing and Simulation* (HPCS), 2011 International Conference on , vol., no., pp.1-7, 4-8 July 2011.

[6] Lim, H.C., Babu, S., and Chase, J.S. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing* (ICAC '10). ACM, New York, NY, USA, 1-10.

[7] Boto: A Python Interface to Amazon Web Services. [Online]. Retrieved February 24, 2013, from: http://boto.readthedocs.org

[8] Condor Log Analyzer. University of Notre Dame. [Online]. Retrieved February 24, 2013, from: http://condorlog.cse.nd.edu

[9] Keahey, K., Armstrong, P., Bresnahan, J., LaBissoniere, D., and Riteau, P. Infrastructure outsourcing in multi-cloud environment. In *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit* (FederatedClouds '12). ACM, New York, NY, USA, 33-38.

[10] Marshall, P., Tufo, H.M., and Keahey, K. 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.

[11] Marshall, P., Tufo, H.M., et al. Architecting a Large-Scale Elastic Environment - Recontextualization and Adaptive Cloud Services for Scientific Computing. *In ICSOFT*. 2012. Rome, Italy.

[12] Mao, M. and Humphrey, M. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '11). ACM, New York, NY, USA.

[13] Amazon Web Services: Auto Scaling. [Online]. Retrieved February 25, 2013, from: http://aws.amazon.com/autoscaling

[14] FutureGrid. [Online]. Retreived February 25, 2013, from: http://futuregrid.org

[15] Marshall, P., Keahey, K., and Freeman, T. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (CCGRID '10). IEEE Computer Society, Washington, DC, USA, 43-52.

[16] TORQUE Resource Manager. Adaptive Computing. [Online]. February 25, 2013, from: http://www.adaptivecomputing.com/products/open-source/torque

[17] Nimbus: Cloud Computing for Science. [Online]. Retrieved April 19, 2013, from: http://www.nimbusproject.org

[18] Amazon Elastic Compute Cloud. Amazon Web Services. [Online]. Retrieved April 19, 2013, from: http://aws.amazon.com/ec2/

[19] HTCondor. High Throughput Computing. [Online]. Retrieved April 24, 2013, from: http://research.cs.wisc.edu/htcondor/