Provisioning Policies for Elastic Computing Environments

Paul Marshall*, Henry Tufo*† and Kate Keahey‡

*Department of Computer Science, University of Colorado, Boulder, Colorado
†National Center for Atmospheric Research, Boulder, Colorado
‡Argonne National Laboratory, Argonne, Illinois
paul.marshall@colorado.edu

***Abstract*—Resources experience dynamic load as demand fluctuates. Therefore, resource providers must estimate the appropriate amount of resources to purchase in order to meet variable user demand. With the relatively recent introduction of infrastructure-as-a-service (IaaS) clouds (e.g. Amazon EC2) resource providers may choose to outsource demand as needed. As a result, a resource provider may decide to decrease his initial capital outlay and purchase a smaller resource that meets the needs of his users the majority of the time while budgeting for future outsourcing costs. When bursts in demand exceed the capacity of the resource, a resource provider can use elastic computing to outsource excess demand to IaaS clouds based on a defined budget.**

To create efficient elastic environments, existing services must be extended with elastic computing functionality and resource provisioning policies that match resource deployments with demand must be developed. In this paper we consider an elastic environment that extends a local cluster resource with IaaS resources. We present resource provisioning policies to dynamically match resource supply with demand. Our policies balance the requirements of users and administrators, such as minimizing the monetary cost of the IaaS deployment and reducing job queued time. We develop a discrete event simulator, the elastic cloud simulator (ECS), to evaluate our policies using scientific workloads. Our results demonstrate that by outsourcing on a flexible basis instead of simply provisioning the maximum number of instances preemptively, we reduce the average queued time by up to 58% and cost by 38%. Our results also demonstrate that our multi-variable policies provide more flexibility in balancing budget and time requirements than typical single-variable reference policies, giving resource providers controls to manage their elastic environments.

***Keywords*-Infrastructure-as-a-Service; cloud computing; elastic computing**

I. Introduction

Physical resources, such as a high-performance computing (HPC) cluster, provide static capacity yet experience dynamic load as demand fluctuates. As a result, resources may be under-utilized during periods of low demand, with idle cycles drawing power and costing the organization money, or they may be over-utilized during periods of high demand, resulting in increased queue wait times or crashing under the load. In typical HPC environments, Grid technologies [1] often employ meta-scheduling [2] strategies to distribute jobs across Grid resources of cooperating resource providers, shifting demand from highly loaded resources to under-utilized resources. Meta-scheduling strategies, however, have no mechanisms to deploy and configure additional resources when needed. Furthermore, because Grids are typically deployed on pre-configured resources, they limit the control that users have over remote resources and increase overhead by requiring that complex software stacks be installed and configured on each resource, possibly requiring assistance from the local Grid administrator.

Infrastructure-as-a-service (IaaS) clouds [3] offer an alternative resource provisioning mechanism to that of Grid computing; IaaS clouds offer on-demand virtual machines (VMs). The scientific community, in particular, has begun embracing IaaS clouds for their workflows [4], [5], [6], [7]. Because IaaS clouds provide virtual resources, users are given full control over the entire software stack, from the operating system upward, allowing them to install and configure the necessary operating system, libraries, or databases required by their applications. With on-demand access to resources, users are either granted or denied access to the resources in near-interactive time. On-demand provisioning is particularly advantageous for users working toward deadlines or responding to emergencies. Furthermore, on-demand provisioning enables the creation of elastic computing environments that adjust resource deployments based on variable demand.

In previous work [8] we presented a model of an "elastic site" and developed a prototype that extended a local Torque cluster with IaaS resources. We developed an elastic site resource manager that monitored the queue and responded to changes in the queue by launching or terminating IaaS cloud instances that integrated with the cluster. Our elastic site prototype successfully demonstrated the feasibility of an elastic computing environment which allowed a resource provider to outsource demand to IaaS clouds.

Because resource providers can use elastic computing techniques to dynamically extend their institutional resources with IaaS clouds, they may chose to decrease initial capital expenses by purchasing a smaller resource that meets the needs of their users the majority of the time and budget for future outsourcing costs. When demand exceeds the capacity of the resource, the resource provider can then

outsource some of the work to IaaS clouds based on their budget. Prior work developed a cloud charging model for HPC resources [9], allowing resource providers to compare their costs with cloud resources and determine whether to purchase additional physical resources or move portions of their workloads to IaaS clouds.

In this paper we consider the scenario where a resource provider extends existing resources with IaaS clouds on a fixed budget. Specifically, we consider the following use case: a research lab at a university with a small cluster may occasionally need more capacity than they purchased in capital equipment. They specify a fixed hourly budget (e.g. $5 per hour) that can be used to outsource excess demand to IaaS resources. This money may accumulate, so if, for example, they don't deploy any IaaS resources over a 3 hour period, they can then use $15 toward outsourcing their workloads to IaaS resources. The lab may use a wide variety of policies to select the appropriate number of resources to deploy on IaaS clouds. Perhaps the most straightforward policy is the relatively static case: deploy the maximum number of instances allowed by the budget. However, such a naive policy may not be the most efficient use of resources or money. In this work we explore different resource provisioning policies for outsourcing workloads to IaaS clouds in an attempt to respond effectively to changes in demand and limit monetary cost. We consider two basic policies that support flexible resource provisioning and respond directly to demand and we propose two advanced policies that balance various user- and administrator-defined metrics.

For evaluation we develop a discrete event simulator, the elastic cloud simulator (ECS), that simulates the entire system, from job submission to job execution on volatile infrastructures that consist of static local resources and dynamic cloud resources. We evaluate our policies using a real workload trace from Grid5000 [10], consisting primarily of single-core jobs, and a workload generated from the Feitelson workload model [11], containing a larger number of parallel jobs.

The remainder of the paper is organized as follows: in section II we discuss the general approach to create an elastic computing environment. In section III we present resource provisioning policies for elastic computing environments. Section IV discusses the design and implementation of our elastic cloud simulator. In section V we evaluate our policies using our simulator with a workload trace and the Feitelson workload model. We discuss related work in section VI. We propose directions for future work in section VII and conclude in section VIII.

II. Approach

Elastic computing offers the ability to dynamically match resource supply with demand. Elastic environments typically measure demand (e.g. number of jobs in a queue or CPU load) in a system and respond by either provisioning or relinquishing resources. Because elastic environments respond directly to demand, they eliminate many of the inefficiencies associated with static resource deployments. However, prior to deploying an elastic environment, a number of considerations must be addressed, in particular:

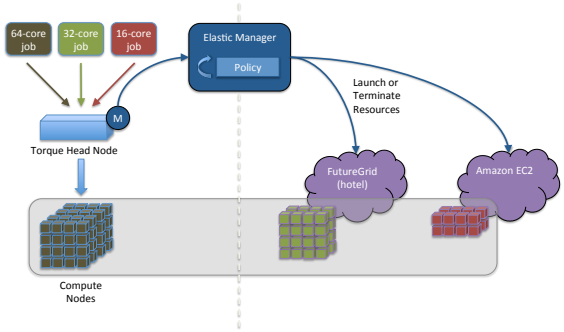


Figure 1. An example elastic environment that extends a local Torque cluster with IaaS cloud resources to process jobs. The IaaS cloud instances integrate directly with the Torque head node.

- *Appropriate workloads*: that experience variable demand need to be identified.
- *Computing paradigm*: how resources will be provisioned should be determined, for example, using infrastructure-as-a-service (IaaS) or platform-as-a-service (PaaS) computing paradigms.
- *Layer of implementation*: whether individual applications will be extended with elastic computing functionality or whether the functionality will be deployed as a separate service.
- *Support for existing resources*: whether the elastic environment will extend pre-existing and static resources, e.g. a local HPC cluster, with additional resources or whether the elastic environment will operate as a standalone entity.
- *Resource provisioning policies*: must be devised in order to minimize waste and ensure that elastic environments respond appropriately to changes in demand.

Given these considerations, we select a specific elastic environment configuration for this research. In this work we focus specifically on a batch-queued cluster running a scientific workload that consists of single-core and parallel jobs submitted by multiple users. We consider a "push" queue model where a central scheduler dispatches tasks to worker instances as opposed to a "pull" queue (e.g. BOINC [12]) where idle workers request tasks. (Condor [13] is a notable exception, and instead uses a match making process

to assign jobs to workers.) Perhaps the most common implementation of a "push" queue system is a cluster where a resource manager, e.g. Torque [14], dispatches jobs to available worker nodes.

As our computing paradigm we use IaaS clouds because of their on-demand and virtualized properties. Virtual cloud instances are configured to be cluster worker nodes that integrate directly with the cluster. We support deployments that span multiple cloud infrastructures, including private clouds, community clouds such as Argonne National Laboratory's Magellan cloud [15] or NSF's FutureGrid [16], and commercial clouds such as Amazon Elastic Compute Cloud (EC2) [17]. Finally, because cloud providers may charge for usage, often with real currency, we define an hourly allocation (i.e. budget), that may be used to acquire cloud instances. These allocation credits may accumulate and be used at a later time.

Elastic computing functionality is implemented as a separate service, called the elastic manager, that adjusts to demand by launching or terminating IaaS resources, as shown in Figure 1. The elastic manager loops regularly and gathers information about the environment, such as the number of queued jobs and the status of worker instances. Each loop iteration is referred to as a policy evaluation iteration. The elastic manager then executes a policy which evaluates this information and responds by launching additional IaaS resources, terminating IaaS resources, or leaving the environment unchanged.

In this context we define *users* as individuals who submit jobs to a queue. We define the *resource manager* to be the central scheduling service responsible for dispatching jobs to both local workers and cloud workers. The *elastic manager* is defined as the service that implements the elastic computing functionality and the *elastic environment* is defined as the combination of local resources integrated with IaaS cloud resources. Finally, we define the *elastic environment administrator* to be the individual responsible for managing the elastic environment. For our purposes the elastic environment administrator is also the administrator of the local resource.

We make a number of simplifying assumptions:

- A single cloud instance type is assumed. Measuring the price-performance ratio of different instance types is highly application dependent. Furthermore, determining an accurate price-performance ratio of different instance types can be very difficult, if not impossible, due to the multi-tenancy nature of VMs.
- Data and data transfer are not considered by the elastic manager due to the general lack of system-wide knowledge about the data requirements for individual jobs.
- The order in which to execute jobs has been determined previously by the resource manager scheduler. Combining job and VM scheduling has the potential to improve our solution. However, the current state of job schedulers makes this difficult to actually implement, cluster resource managers need better interfaces and APIs to allow job adjustments from a separate elastic manager service.
- Parallel jobs are scheduled across a single infrastructure to minimize excessive latency for tightly-coupled jobs.
- Job walltime is used to estimate the run time of jobs since it is readily accessible. This work does not attempt to model or exploit user-specific workflow patterns or application runtime characteristics.

The remainder of the paper discusses our work developing and evaluating resource provisioning policies that respond to changes in demand.

III. Resource Provisioning Policies

Policies that respond appropriately to changes in demand must balance the requirements of users and administrators, which often conflict. For example, launching additional IaaS instances to process queued jobs may decrease the queued time of the jobs, however, it will also increase the overall cost of the deployment.

For resource provisioning policies we first consider the relatively static policy where a resource provider launches the maximum number of allowable instances. This policy, *sustained max* (SM), immediately launches the maximum number of instances allowed by a cloud provider or the administrator-defined budget. If multiple clouds are available it launches the maximum number of instances (as determined by the budget or set by the cloud provider) on the least expensive cloud first before moving to the next cheapest cloud and so on. It leaves the instances running for the entire duration of the deployment. We use this policy as our base reference case for comparing our flexible resource provisioning policies.

A. Basic Policies for Flexible Resource Provisioning

Provisioning the maximum number of allowable instances may not be the most efficient use of resources or money, therefore, we also propose the following flexible resource provisioning policies.

- *On-demand* (OD) launches instances for all cores requested by jobs in the queued state. OD only attempts to launch instances for jobs until it has either launched enough instances for all jobs, depleted the allocation credits, or reached the maximum number of instances allowed by a cloud provider. Similar to SM, OD begins with the least expensive cloud before moving on to the next cheapest cloud. Instances are terminated when they are idle and there are no remaining jobs in the queued state.
- *On-demand*++ (OD++) is nearly identical to the on-demand policy: it launches an instance for every job in the queued state. The key difference between OD and OD++ is that OD++ only terminates idle instances

that will be "charged" before the next policy evaluation iteration.

Though the basic policies are undoubtedly preferable to users who are only concerned with reducing job queued time and are not personally responsible for the cost, such policies do not necessarily meet the needs of a shared cluster environment with limited resources (both monetary and computational). The elastic site administrator is not able to tailor the basic policies to meet his or her requirements. Therefore, we propose the following advanced policies that consider both user and administrator requirements:

B. Average Queued Time Policy

The *average queued time* policy (AQTP) launches instances for the first *n* queued jobs at each policy evaluation iteration. The elastic environment administrator defines the minimum number of jobs that AQTP may respond to as well as a maximum and starting value. Additionally, the elastic environment administrator defines a desired response, *r*, that is considered a reasonable average weighted queued time, *AWQT*, of the set of currently queued jobs, *Q*. *AWQT* is described below:

$$AWQT = \frac{\sum_{j}^{Q} j.num\_cores * j.queued\_time}{\sum_{j}^{Q} j.num\_cores}$$

The elastic environment administrator may also optionally define a threshold value, θ, for the desired response.

At each policy evaluation iteration, the policy subtracts one from the number of jobs that it responds to, until the minimum is reached, if the measured *AWQT* is less than $r - \theta$. If the measured *AWQT* is greater than $r + \theta$ then the policy adds one to the number of jobs that it will respond to, until the maximum is reached. For example, an administrator may determine that two hours is an appropriate desired response, *r*, with a threshold of 45 minutes for a particular elastic environment. Thus, when the policy measures an AWQT less than 1 hour and 15 minutes it will subtract 1 from the number of jobs that it responds to, and when it measures an AWQT greater than 2 hours and 45 minutes it will add one to the number of jobs that it responds to; if it measures an AWQT between those values it will respond to the same number of jobs that it did at the previous policy evaluation iteration.

After the policy has determined the number of jobs that it will launch instances for, $\hat{n}$, it then selects the maximum number of cloud providers it will consider by calculating:

$$NC = \lfloor \frac{AWQT}{r} \rfloor$$

If *NC* is less than 1, we set it to be 1. Once the number of jobs has been determined as well as the maximum number of clouds that may be used, the policy then proceeds to launch the instances.

Because cost is a particular concern with IaaS clouds, the policy first sorts the possible cloud providers by their cost, from least expensive to most expensive. It then determines the precise number of instances that each cloud can launch, as limited by the amount of allocation credits currently available as well as the maximum number of instances the particular cloud provider may allow. However, the number of instances that a cloud *can* launch may not be the optimal number of instances to launch. The policy only launches the appropriate number of instances as determined by the requested core counts for the $\hat{n}$ queued jobs. For example, the policy may determine that a cloud can launch 17 instances, based on the cost of the cloud provider and the current amount of allocation credits, however, if the policy is considering two 16-core jobs, then it should only launch 16 instances as the 17th instance will simply be wasted. The policy then proceeds to launch the optimal number of instances, as previously selected based on the $\hat{n}$ queued jobs, for *NC* clouds, beginning with the least expensive cloud.

Finally, the last action of the policy is to terminate any idle instances that will be charged before the next policy evaluation iteration, similar to the termination process of OD++.

C. Multi-Cloud Optimization Policy

The *multi-cloud optimization* policy (MCOP) attempts to find a elastic environment configuration that minimizes two conflicting objectives, job queued time and the cost of the deployment, making it a multi-objective optimization problem [18]. This policy is similar to the work in [19] where the authors use a genetic algorithm to identify the optimal schedule for jobs across static Grid slots. MCOP, however, schedules jobs across volatile cloud environments.

To find the optimal elastic environment configuration, we must search as many different configurations as possible. Searching all possible configurations may be impossible given the limited time during a single policy evaluation iteration. Of course, the number of configurations that must be searched depends on the number of cloud providers, the number of queued jobs, and the amount of allocation credits available. Because of the vast number of possible configurations, MCOP uses a genetic algorithm (GA) to explore a subset of configurations. Using a GA to find optimal solutions for multi-objective optimization problems has been widely accepted by the community [20]. Due to the time constraints in our case, the GA is only allowed to execute a set number of iterations. We do not allow the GA to run until it converges on the optimal set. Though not ideal, we believe that allowing the GA to explore a sufficient number of possible configurations will result in a reasonable configuration given the strict time constraints.

Similar to AQTP, MCOP only considers configurations that are relevant to the particular core counts of queued jobs. Therefore, alleles in the chromosome represent jobs in the queue. A 1 signifies that a job will be considered, whereas a 0 does not. The length of the chromosome is simply the maximum number of queued jobs for the particular, independent, policy evaluation iteration. Initially, a population of 30 individuals is randomly generated for each cloud (i.e. each cloud considers its own set of various combinations of all of the queued jobs). We use common values for the GA properties which are generally known to perform well [21], specifically, the GA iterates 20 times with a mutate probability of 0.031 and a crossover probability of 0.8. Each cloud provider under consideration evaluates its own set of 30 populations.

Execution of MCOP is similar to the standard execution of a GA. First, the initial populations for each cloud provider are randomly generated. In addition to the random individuals in the population, we also make sure to consider the extremes, specifically, all zeros (no jobs) and all ones (all jobs), at each policy evaluation iteration. The GA then iterates 20 times for each cloud provider's populations, performing crossover and mutation. The fitness of an individual is calculated as the weighted preference (specified by the administrator) of the estimated cost and job turn around time for the set of jobs; the individuals with the lowest estimated cost and job turn around time mate to produce offspring.

Once all iterations are complete, we estimate the cost and job queued time (for all jobs) of each possible elastic environment configuration, i.e., all final populations for each cloud provider compared against the final populations of all other cloud providers. Depending on the number of cloud providers, only a subset of final populations may be compared. The cost of each configuration is estimated by calculating the cost of launching instances on each cloud for the core counts of the jobs selected by the final population. The queued time of jobs for each configuration is estimated by building a schedule of jobs, executed in order, for the specific number of instances each cloud should launch, again, as determined by the final population.

After we have estimated the cost and job queued time for each elastic environment configuration, we then compare all configurations and generate the *Pareto optimal* set of solutions. We use domination [20] to compare environment configurations. Domination simply states that one environment configuration dominates another if both of the following conditions are true:

1) The cost of the first configuration is *less than or equal* to the cost of the second configuration **and** the total job queued time for the first configuration is *less than or equal* to the total job queued time for the second configuration.
2) The cost of the first configuration is *less than* the cost of the second configuration **or** the total queued job time is *less than* the cost of the second configuration.

All configurations that are not dominated by any other configurations are then added to the Pareto optimal set. In order to select the final solution from this set, the elastic environment administrator defines two weights that represent their preference for cost and job queued time. These weights are then multiplied by normalized values for the cost and job queued time of each configuration. The solution with the minimum value for the sum of these two items is selected. If there is more than one minimum, MCOP defaults to selecting the solution with the lowest cost. If two or more minimums have the same cost, the solution is chosen randomly. Similar to both OD++ and AQTP, MCOP terminates any idle instances that will be charged before the next iteration.

IV. Elastic Cloud Simulator

Effectively evaluating resource provisioning policies for IaaS clouds can require a substantial, if not prohibitive, investment time and money. Cost is especially a concern when using commercial cloud providers, such as Amazon EC2, that charge with real currency. Therefore, to evaluate our policies we developed a discrete event simulator, the elastic cloud simulator (ECS). ECS simulates all of the necessary components of the elastic environment including work submission, launching cloud instances, processing the workload, terminating instances, and accounting for allocation credits.

A. Measuring Cloud Variability

We consider two major sources of variability in IaaS clouds: instance launch times and instance termination times. To gather approximate estimates of launch and termination times for ECS, we measured launch and termination times of 60 Debian 5.0 (with a 10 GB image in S3) instances on EC2 east over the course of a day. After initiating the launch process for the image we began pinging it; we calculated the launch time as the time from when we sent the launch request to the time of the first successful ping. Measuring termination time followed a similar process, except we use the first failed ping as the time the VM terminated. We found that termination times were relatively consistent with an average time of 12.92 seconds and a standard deviation of 0.50. Launch times experienced much more variability. From our observations, launch times did not appear to assemble around a single average time, instead we observed that instance launches centered around one of three values. The majority of launches, 63 percent, averaged 50.86 seconds with a standard deviation of 1.91 seconds, 25 percent averaged 42.34 seconds with a standard deviation of 2.56 seconds, and 12 percent averaged 60.69 seconds with a standard deviation of 2.14 seconds.

B. Implementation

ECS uses a workload definition file to simulate job arrival and run times (both specified in seconds) as well as requested core counts. If a workload definition file is not used, ECS can automatically generate and format an appropriate workload based on the Feitelson workload model [11]. Jobs are processed in a first-in-first-out (FIFO) order, assigning jobs to the first-available instance in the order that they arrive. ECS can simulate both volatile cloud environments as well as static local resources, such as a compute cluster. Parallel jobs are executed on a single resource infrastructure; they are not allowed to span multiple infrastructures. Policies are implemented as individual Python modules and are completely interchangeable. We implemented the policies defined in section III. ECS is composed of a set of individual processes, each looping continually. The main ECS processes include a workload generator process, an elastic manager process, any number of instance processes (each representing individual cloud instances), a credit allocation process, and a trace output process.

V. Evaluation

For evaluation we examine the impact of elastic computing environments on scientific workloads. Our evaluation attempts to simulate the use case described in section I, that is, a university research lab extends a small local cluster with IaaS clouds on a fixed hourly budget. Therefore, our simulated evaluation environment consists of a small local cluster, a private cloud resource with limited scalability (representing a community cloud such as Magellan or FutureGrid), and a commercial cloud provider, e.g. Amazon EC2, that is able to launch as many instances as we request. We evaluate the policies presented in section III and compare them against the relatively static base case, SM, that preemptively launches the maximum number of allowable instances. To examine the policies we use a discrete event simulator, ECS, and a workload trace from Grid5000 [10] and the Feitelson workload model [11].

The local cluster contains 64 static single-core worker instances; no additional resources can be added or removed. Because the local cluster is "always on" we do not simulate the instances booting or terminating during the evaluation period. The private cloud resource contains up to 512 single-core instances. To simulate rejected responses on the private cloud we consider various rejection rates, i.e., requests are rejected a certain percentage of the time. We consider rejection rates of 10%, representing a lightly loaded private cloud and 90%, representing a heavily loaded private cloud. The private cloud does not have a direct (monetary) cost associated with it. For the commercial cloud provider, we do not define a maximum for the number of instances it can launch, nor do we use a rejection rate; it is always able to respond to an unlimited number of requests. However, the commercial cloud instances cost $0.085 per hour. Similar to Amazon EC2, partial hour charges are rounded up, e.g., an instance that runs for only 20 minutes still incurs the $0.085 hourly charge. Both the private cloud and the commercial cloud randomly generate their boot and shutdown times based on the times we gathered from Amazon EC2 in section IV. Parallel jobs are only executed when enough instances are available to execute them across a single infrastructure. Jobs are executed in order because we assume they have already been ordered by a separate scheduling process. For this evaluation we set a budget of $5 per hour, and unspent money from previous hours accumulates and may be spent later by the elastic manager. The elastic manager is configured with a policy delay iteration of 300 seconds and responds by launching or terminating instances.

As metrics, we define *cost* to be the total monetary cost of an elastic environment over the entire evaluation. We define the workload *makespan* to be the entire duration of the workload, from the time the first job is submitted until the time the last job completes. Finally, we define *average weighted response time* (AWRT) to be the average response time of all jobs, weighted by their requested number of cores, where response time is job completion time minus job submit time. Specifically, AWRT is defined as:

$$AWRT = \frac{\sum_{j}^{Q} j.num\_cores * j.response\_time}{\sum_{j}^{Q} j.num\_cores}$$

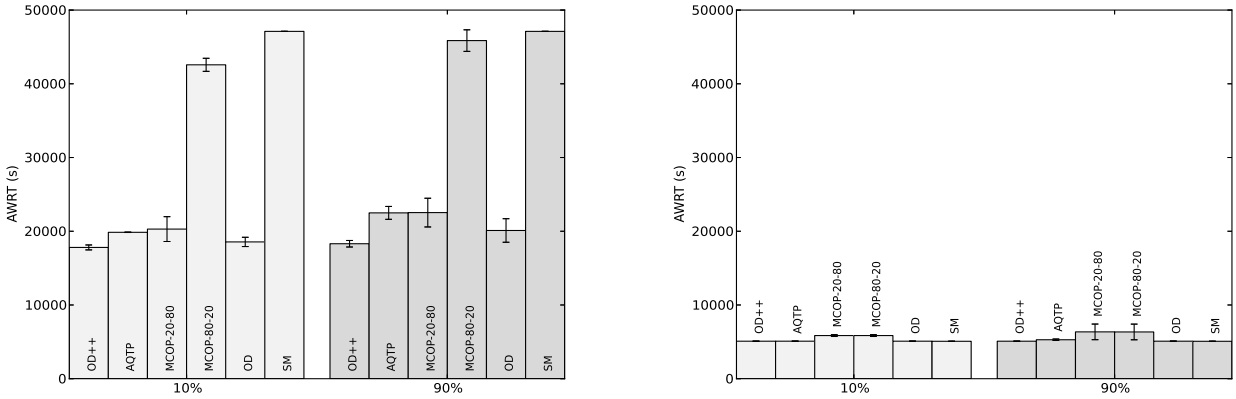
AWRT demonstrates the impact the policy has on users. Increasing AWRT suggests that jobs are remaining queued for longer times.

A. Workloads

For evaluation we select a workload trace and workload model that represent scientific workloads in multi-user cluster environments. A workload trace reveals interesting information about a single real workload while a workload model represents a broader class of workload characteristics.

The Grid5000 workload trace was obtained from the Grid Workload Archive [22]; we used a subset of this trace (approximately 10 days) for our evaluation. The subset contains 1061 jobs with a minimum run time of 0 seconds and a maximum run time of 36 hours. The average run time is 113.03 minutes with a standard deviation of 251.20. Jobs core counts range from 1 to 50, however, the majority, 733, are requests for a single core.

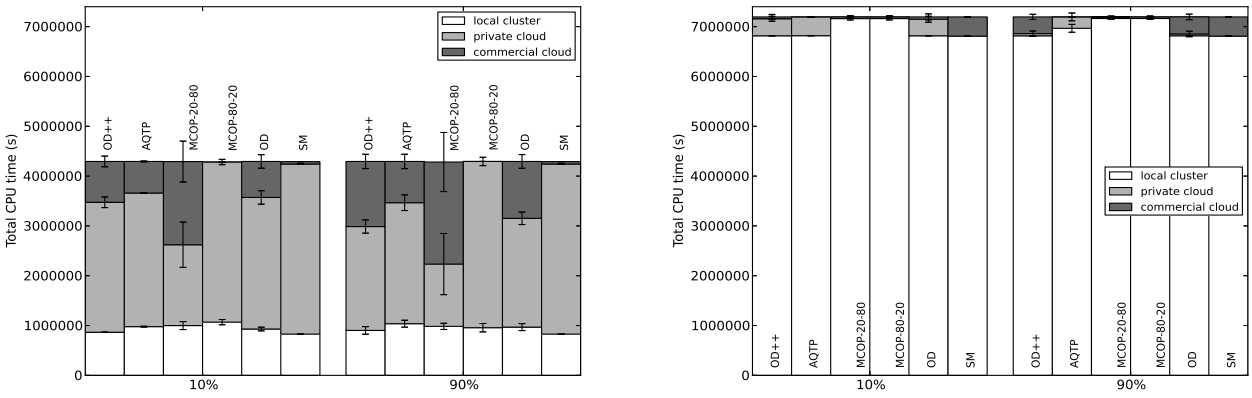
The Feitelson model [11] that we used generated 1,001 jobs that are submitted over approximately 6 days. The jobs range in size from 1 core to 64 cores. The minimum run time is 0.3123 seconds and the maximum is 23.58 hours with an average of 71.50 minutes and a standard deviation of 207.24. The Feitelson workload contains many more parallel jobs than the Grid5000 workload; it includes 146 8-core jobs, 32 32-core jobs, and 68 64-core jobs.



(a) Feitelson workload

(b) Grid5000 workload

Figure 2. Average Weighted Response Time with 10% and 90% rejection rates for the private cloud infrastructure.



(a) Feitelson workload

(b) Grid5000 workload

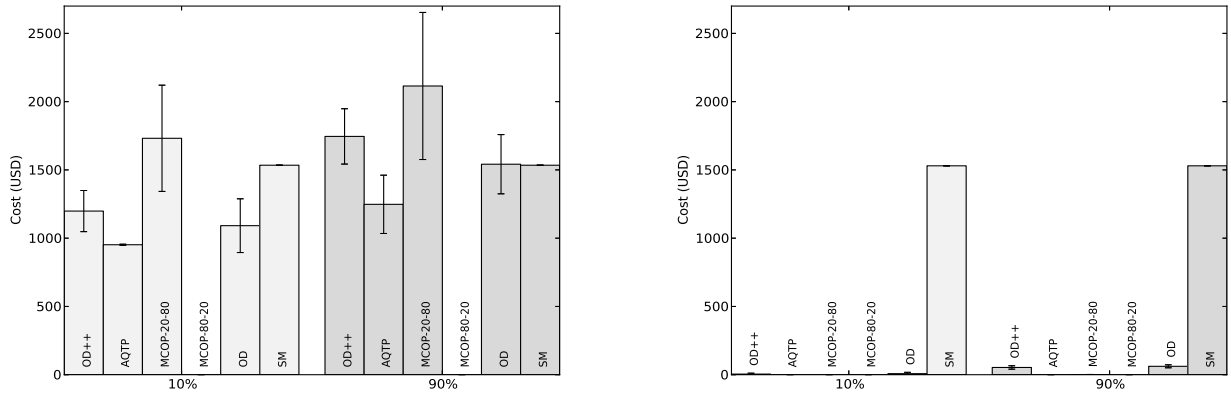
Figure 3. Total CPU time with 10% and 90% rejection rates for the private cloud infrastructure. CPU time is shown for individual resource infrastructures.

B. Understanding System Impact

To compare our policies we ran 30 iterations for each policy and each workload, as well as 10% and 90% rejection rates for the private cloud. For MCOP we specify a 20% preference for cost and 80% preference for job queued time (MCOP-20-80) and then an 80% preference for cost and 20% preference for job queued time (MCOP-80-20). MCOP has a tendency to experience wide variability in some cases; we believe this is due to its non-deterministic nature and the limited number of GA iterations we examine. Each simulation simulates 1,100,000 seconds (about 306 hours) to ensure that all jobs complete. The Feitelson workload has a makespan of approximately 601,000 seconds for all policies while the Grid5000 workload's makespan is approximately 947,000 seconds for all policies. Because there is almost no variability in the makespan, regardless of the policy, we omit the makespan graphs.

Increasing the cloud rejection rate results in a cost increase (Figure 4) because when the policies are unable to acquire the necessary instances on the private cloud they request extra instances on the commercial cloud. Unsurprisingly, we see in Figure 4 that the sustained max policy is generally one of the more expensive policies because it maintains the maximum number of instances on all clouds at all times, as allowed by the budget.

Figure 3 shows the total time each resource spends running jobs. As we would expect, the more time policies utilize the commercial cloud, the higher their expense (Figure 4). The one exception appears to be SM which has a high cost but doesn't utilize the commercial cloud extensively. The reason for this is due to the nature of the policy: SM launches the maximum number of instances on the commercial cloud and leaves them running for the entire



(a) Feitelson workload

(b) Grid5000 workload

Figure 4. Cost with 10% and 90% rejection rates for the private cloud infrastructure. The zero values are cases where the commercial cloud is not used, as the policy only selects local resources and the cost-free private cloud.

duration, regardless of whether or not the instances are in use. This results in the high cost of the SM policy. The Grid5000 workload primarily uses local resources (Figure 3(b)) because it has very few bursts that exceed the capacity of the local resources (its jobs are submitted over 10 days instead of the Feitelson workload which are submitted over a 6 day period) and it consists largely of single-core jobs which easily overlap on the local infrastructure.

One of the more interesting observations is the relatively high AWRT for Feitelson jobs and the SM policy in Figure 2(a). The expectation is that because the SM policy is running the maximum number of instances on both the private cloud (512 instances) and the commercial cloud (58-59 instances based on the $5 hourly budget and $0.085 instance cost), we should see a relatively low AWRT. Instead, we see that for the Feitelson workload, OD, OD++, and AQTP achieve lower AWRT because they deploy instances for each individual job. If demand is low enough then SM is able to process the jobs immediately, however, when demand bursts high enough, OD, OD++, and AQTP use money that has been saved from previous hours (and going into slight debt, if necessary) to deploy additional instances, resulting in higher overall costs (Figure 4(a)) but a lower AWRT (Figure 2(a)).

Figures 2 and 4 demonstrate the ability of the MCOP policy to balance the response time of jobs with cost. MCOP-20-80 achieves better AWRT for a greater cost while MCOP-80-20 sacrifices AWRT for cost. In Figure 4(b), we see that AQTP and both configurations of MCOP do not result in any cost because they only use the private cloud. OD and OD++ incur a slight cost because whenever they are rejected by the private cloud they immediately attempt to launch instances for jobs on the commercial cloud.

Of the policies we developed and evaluated AQTP appears to strike an ideal balance between cost and response time. It has a higher AWRT than OD and OD++ because it waits to adjust the deployment until the average queued time has reached a desired level. (An administrator can lower the desired response time to reduce AWRT.) However, the side effect of this delay is that it reduces the cost. In one particular case with the Feitelson workload, this results in an increase in AWRT of 18% while reducing the cost by approximately 40%. By adjusting based on the average queued time, AQTP gives the elastic environment administrator control over how quickly the environment should respond to changes in demand. If an administrator seeks fine-tune control over cost and job queued time, MCOP offers the administrator the ability to adjust as necessary. OD and OD++ both perform well with regard to the AWRT metric, as we would expect. However, neither policy offers an administrator any control over response time or cost, as they are more expensive than both AQTP and MCOP in numerous cases. For example, for the Feitelson workload with a private cloud with a rejection rate of 90%, OD++ costs approximately $1,811 more than MCOP-80-20 and its jobs experience an average weighted queued time of approximately 5 hours whereas MCOP-80-20 jobs experience an average weighted queued time of 12.5 hours. However, the entire workload completes in about the same amount of time for both policies. Undoubtedly, the question of which policy is "better" is strictly dependent upon the (monetary) resources available to the administrator, user requirements and their workloads, as well as any deadlines. Preemptively provisioning the maximum number of instances, based on budget and available resources, doesn't appear to offer many advantages over the other, more dynamic and flexible, policies. SM uses the entire budget regardless of demand, and as such, is unable to adjust appropriately for later demand (e.g.

by saving money when demand is low to process a burst in demand), resulting in an expensive environment that doesn't serve users efficiently.

VI. Related Work

Historically, job scheduling research [23] has focused on scheduling individual resources for a group of users in a fair manner that maximizes resource utilization. These systems are typically queue-based and employ techniques, such as backfilling, to ensure that resources are used efficiently. These scheduling strategies, however, are intended for individual systems with a static number of resources under direct control of the scheduler. More recent research focuses on energy saving techniques, also referred to as green computing, to achieve better resource efficiency [24]. A popular technique in green computing is to power off or suspend idle resources and reactivate them when they are needed. Green computing techniques may be a preferred option under certain conditions (assuming that they are supported by the underlying resource), especially in cases where resources are under-utilized. Both classical scheduling strategies and green computing techniques focus on improving the efficiency and utilization of individual resources. Market-based scheduling systems [25], [26] seek to maximize profits for the resource provider by scheduling requests from users that are willing to pay the most; they do not attempt to minimize costs from a user perspective.

Previous research in [19] provides models for efficient and cost-effective resource provisioning in Grid environments. The authors in [19] propose a solution that uses a genetic algorithm to find a resource provisioning plan that both minimizes the workflow makespan time as well as the cost. The work assumes, however, that there are a static number of remote Grid resources available where each resource schedules individual jobs using a queue-based system. The proposed model does not address resource provisioning in volatile computing environments with a changing number of resources. In [27] the authors extend two applications with the ability to provision resources using a feedback control mechanism. Because this work integrates directly with applications it measures CPU utilization and memory allocations to determine when to adjust the deployment. The proposed model and implementation are not intended for environments executing any number of applications, such as in a batch-queued cluster.

Our work differs from previous research by our emphasis on resource provisioning policies for volatile IaaS cloud computing environments. We also focus on batch-queued resources, thus our resource provisioning policies can be used to extend any batch-queue workloads. We leverage previous work in the field, specifically [19], and adjust it for elastic computing environments. Our policies balance user- and administrator-defined requirements, specifically minimizing cost and reducing job queued time, to create elastic environments that are both effective for users and useful for administrators.

VII. Future Work

In future work we will examine the impact and scalability of our policies in a real-world deployment across multiple cloud providers. In conjunction with this research we will consider additional policies that include workload data requirements. Data movement will undoubtedly impact individual job completion time as well as the overall workload time as input data has to be moved from storage to ephemeral compute resources and output data has to be moved back to a permanent storage location. In addition to including data in policies, combining job scheduling algorithms with resource provisioning policies may yield more optimal deployments than scheduling jobs and resources separately. Existing job scheduling algorithms only optimize static resource deployments since they are unaware of the ability to provision additional resources on-demand. Finally, we will consider the impact of high-throughput computing (HTC) [13] workloads where overall workload performance is preferred to optimizing individual jobs. In this context, we will explore the use of Amazon spot instances [28] and Nimbus backfill instances [29].

VIII. Conclusion

In this work we present flexible resource provisioning policies that create efficient and cost-effective elastic computing environments capable of outsourcing demand from local resources to IaaS clouds. We develop a discrete event simulator to evaluate our policies with a workload trace and a workload model.

Our results demonstrate that by outsourcing on a flexible basis instead of simply provisioning the maximum number of instances preemptively, we reduce the average queued time by up to 58% and cost by 38% for the entire workload. Our results also demonstrate that our multi-variable policies provide more flexibility in balancing budget and time requirements than typical single-variable reference policies, giving resource providers controls to manage their elastic computing environments.

Acknowledgments

We would like to thank Theron Voran and Matthew Woitaszek for their feedback on this work. We would also like to thank the Grid5000 team, specifically Dr. Franck Cappello and Dr. Olivier Richard, the OAR team, specifically Nicolas Capit, and the Grid Workload Archive for providing the Grid5000 trace.

References

[1] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, Fall 2001.

[2] S. Vadhiyar and J. Dongarra, "A metascheduler for the grid," in *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, 2002, pp. 343 – 351.

[3] M. Armbrust *et al.*, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep., February 2009.

[4] K. R. Jackson *et al.*, "Seeking supernovae in the clouds: a performance study," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 421–429.

[5] G. Juve *et al.*, "Data sharing options for scientific workflows on amazon ec2," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, Washington, DC, USA, 2010, pp. 1–9.

[6] J. Rehr *et al.*, "Scientific computing in the cloud," *Computing in Science Engineering*, vol. 12, no. 3, pp. 34 –43, may-june 2010.

[7] J. Wilkening *et al.*, "Using clouds for metagenomics: A case study," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 31 2009-sept. 4 2009, pp. 1 –6.

[8] P. Marshall, K. Keahey, and T. Freeman, "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*, ser. CCGRID '10, Washington, DC, USA, 2010, pp. 43–52.

[9] M. Woitaszek and H. M. Tufo, "Developing a cloud computing charging model for high-performance computing resources," in *10th IEEE International Conference on Computer and Information Technology*, Bradford, UK, June 2010.

[10] R. Bolze *et al.*, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, Winter 2006.

[11] D. Feitelson, "Packing schemes for gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin / Heidelberg, 1996, vol. 1162, pp. 89–110.

[12] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04, Washington, DC, USA, 2004, pp. 4–10.

[13] T. Tannenbaum *et al.*, "Beowulf cluster computing with linux." Cambridge, MA, USA: MIT Press, 2002, ch. Condor: a distributed job scheduler, pp. 307–350.

[14] R. Henderson, "Job scheduling under the portable batch system," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin / Heidelberg, 1995, vol. 949, pp. 279–294.

[15] Magellan cloud, Argonne National Laboratory. [Online]. Available: http://magellan.alcf.anl.gov/

[16] Future Grid. [Online]. Available: http://futuregrid.org/

[17] Amazon Elastic Compute Cloud (EC2), Amazon Inc. [Online]. Available: http://aws.amazon.com/ec2/

[18] K. Deb, "Multi-objective optimization," in *Search Methodologies*, E. K. Burke and G. Kendall, Eds. Springer US, 2005, pp. 273–316.

[19] G. Singh, C. Kesselman, and E. Deelman, "A provisioning model and its comparison with best-effort for performance-cost optimization in grids," in *Proceedings of the 16th international symposium on High performance distributed computing*, ser. HPDC '07. New York, NY, USA: ACM, 2007, pp. 117–126.

[20] K. Deb, *Multiobjective Optimization Using Evolutionary Algorithms*. Wiley, 2001.

[21] J. Grefenstette, "Optimization of control parameters for genetic algorithms," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 16, no. 1, pp. 122 –128, jan. 1986.

[22] Grid Workload Archive. [Online]. Available: http://gwa.ewi.tudelft.nl/

[23] D. Feitelson and L. Rudolph, "Parallel job scheduling: Issues and approaches," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin / Heidelberg, 1995, vol. 949, pp. 1–18.

[24] L. Lefèvre and A.-C. Orgerie, "Designing and evaluating an energy efficient cloud," *The Journal of Supercomputing*, vol. 51, pp. 352–373, 2010.

[25] C. Li, L. Li, and Z. Lu, "Utility driven dynamic resource allocation using competitive markets in computational grid," *Advances in Engineering Software*, vol. 36, no. 6, pp. 425 – 434, 2005.

[26] K. Lai *et al.*, "Tycoon: An implementation of a distributed, market-based resource allocation system," *Multiagent and Grid Systems*, vol. 1, no. 3, pp. 169–182, 01 2005.

[27] Q. Zhu and G. Agrawal, "Resource provisioning with budget constraints for adaptive applications in cloud environments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 304–307.

[28] Amazon Elastic Compute Cloud (EC2) Spot Instances, Amazon Inc. [Online]. Available: http://aws.amazon.com/ec2/spot-instances/

[29] P. Marshall, K. Keahey, and T. Freeman, "Improving utilization of infrastructure clouds," in *Proceedings of the 2011 11th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '11, 2011.