Adapting Market-Oriented Scheduling Policies for Cloud Computing

Mohsen Amini Salehi and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory,
Department of Computer Science and Software Engineering,
The University of Melbourne, Australia
{mohsena,raj}@csse.unimelb.edu.au

Abstract. Provisioning extra resources is necessary when the local resources are not sufficient to meet the user requirements. Commercial Cloud providers offer the extra resources to users in an on demand manner and in exchange of a fee. Therefore, scheduling policies are required that consider resources' prices as well as user's available budget and deadline. Such scheduling policies are known as market-oriented scheduling policies. However, existing market-oriented scheduling policies cannot be applied for Cloud providers because of the difference in the way Cloud providers charge users. In this work, we propose two market-oriented scheduling policies that aim at satisfying the application deadline by extending the computational capacity of local resources via hiring resource from Cloud providers. The policies do not have any prior knowledge about the application execution time. The proposed policies are implemented in Gridbus broker as a user-level broker. Results of the experiments achieved in real environments prove the usefulness of the proposed policies.

1 Introduction

In High Performance Computing (HPC), providing adequate resources for user applications is crucial. For instance, a computing center that a user has access to cannot handle the user applications with short deadlines due to limited computing infrastructure in the center [2]. Therefore, to get the application completed by the deadline, users usually try to get access to several computing centers (resources). However, managing several resources, potentially with different architectures, is difficult for users. Another difficulty is optimally scheduling applications in such environment.

User-level brokers work on behalf of users and provide access to diverse resources with different interfaces. Additionally, existing brokers such as Gridway and Gridbus broker [10] optimally schedule user application on the available resources. User-level brokers consider user constraints (such as deadline and budget) and user preferences (such as minimizing time or cost) in their scheduling [10].

Recently, commercial Cloud providers offer computational power to users in an on-demand manner and in exchange of a fee. These Cloud providers are

also known as Infrastructure as a Service (IaaS) providers and charge users in a pay-as-you-go fashion. For instance, in Amazon Elastic Compute Cloud (Amazon EC2) [1], which is a popular IaaS provider, users are charged in an hourly basis for computational resources. In this paper, we term this charging period as "charging cycle".

The computational power offered by IaaS providers can compensate for the limited computational capacity of non-commercial local resources when they are not enough to meet the user deadline. However, as mentioned earlier, getting access to this extra computational power incurs cost for the user. In fact, there is a trade-off between spending budget to get resources from IaaS providers and running the application on local resources.

Therefore, the problem we are dealing with is how scheduling policies inside the broker can benefit from resources supplied by the IaaS providers in addition to the local resources to get the user application completed by the requested deadline and provided budget. Furthermore, we assume that the end user does not have any knowledge about the application execution time. The problem is more complicated when we consider the user preference in terms of time minimization or cost minimization in addition to the budget and deadline limitations. Such scheduling policies are broadly termed market-oriented scheduling policies [3].

Buyya et al. [3] propose scheduling policies to address the time minimization and cost minimization problem in the context of Grid computing. They term their proposed policies DBC (Deadline Budget Constraint) scheduling policies and define them as follows:

- Time Optimization: minimizing time, within time and budget constraints.
- Cost Optimization: minimizing cost, within time and budget constraints.

However, Buyya et al. do not consider the mixture of non-commercial and commercial resources. Moreover, there are some differences in hiring resources from IaaS providers and assumptions in mentioned DBC policies. One difference is that in the policies proposed by Buyya et al., the user is charged when a job is submitted to a resource. Nevertheless, IaaS providers charge users as soon as a resource is allocated to the user regardless of being deployed by the user or not [9]. Another difference is that current IaaS providers charge users in an hourly basis, whereas in these policies [3] user is charged in cpu-per-second basis. These differences raise the need to propose new DBC scheduling policies to meet the user constrains by hiring resources from IaaS providers.

In this work, we propose two scheduling policies (namely Cost Optimization and Time Optimization) to increase the computational power of the local resources and complete the application by the given deadline and budget.

In summary, our work makes the following contributions:

- We propose the Cost Optimization and the Time Optimization scheduling policies that increase the computational capacity of the local resources by hiring resources from IaaS providers to meet the application deadline within a budget.

- We extend Gridbus broker (as a user-level broker) to hire resources from Amazon EC2 (as an IaaS provider).
- We evaluate the proposed policies in a real environment by considering different performance metrics.

The rest of this paper is organized as follows. In Section 2, related works in the area are introduced. Proposed scheduling policies are described in Section 3. Details of implementation are described in Section 4. Then, in Section 5, we describe the experiments for evaluating the efficiency of the new policies. Finally, conclusion and future works are provided in Section 6.

2   Related Work

A number of research projects have been undertaken over the last few years on provisioning resources based on IaaS providers. The approach taken in these research projects typically consists of deploying resources offered by IaaS providers in two levels. One approach is using resources offered by IaaS providers at resource provisioning level, the other approach deploys resources offered by the IaaS provider at broker (user) level. In this section, a review of these works is provided.

2.1   Deploying Cloud resources at resource provisioning level

The common feature of these systems is that they do not consider user constrains such as deadline or budget in hiring resources from IaaS providers. In other words, in these works resources offered by IaaS providers are used in order to satisfy system level criteria such as handling peak load.

Table 1. Comparing different aspects of resource provisioning mechanisms from IaaS providers

Proposed Policy	Use Non-Cloud Resources	User constrains	User Transparency	Scheduling Level	Goal
OpenNebula [5]	Local	No	Yes	System-level	Handling peak load
Llorenete et al. [6]	Local and grid(Globus enabled)	No	Yes	System-level	Provision extra resources to handle peak load
Assunção et al. [4]	Local	Yes (Budget)	Yes	System level	Handling peak load
Vazquez et al. [2]	Local and grid(Globus enabled)	No	No	User level	Federating several providers from Grid and Cloud
Silva et al. [8]	No	Yes (Budget)	No	User level	Run Bag-Of-Task application on Cloud
Time and Cost Optimization (this paper)	Local	Yes (Budget and Deadline)	Yes	User level	Minimizing completion time and incurred cost within a deadline and budget

OpenNebula [5] is a system that can manage several virtualization platforms, such as Xen, inside a cluster. OpenNebula is able to hire resources from IaaS providers, such as Amazon EC2, in an on-demand manner. In OpenNebula hiring resources from Amazon happens when the capacity of local resources is being

saturated. Llorente et al. [6] extend OpenNebula to provision excess resources for high performance applications in a cluster or grid to handle peak load.

Assunção et al. [4] evaluate the cost-benefit of deploying different scheduling policies, such as conservative backfilling and aggressive backfilling for an organization to extend its computing infrastructure by allocating resources from an IaaS provider. However, our work proposes cost and deadline aware scheduling policies for user application.

2.2   Deploying Cloud resources at broker (user) level

Vazquez et al. [2] propose dynamic provisioning mechanism by federating grid resources from different domains with different interfaces along with resources from IaaS providers. The federation is achieved through GridWay.

In this solution all the resources have to support Globus Toolkit (GT). Even in the case of resources from IaaS providers, Gridway can just awake resources with the Globus adapter. Since this work takes advantage of resources from IaaS providers in the user-level broker, it is similar to our work. However, GridWay neither considers the user constraint in terms of budget nor the user preferences in terms of time or cost minimization. In the mentioned work [2], it is stated that investigating cost-aware scheduling policies for resources from an IaaS provider is required and, in fact, our work addresses this requirement.

Silva et al. [8] propose a mechanism for creating optimal number of resources on Cloud based on the trade-off between budget and speedup. This work considers Bag-of-Tasks applications where the run times are not known in advance. In fact, heuristics proposed by Silva et al. [8] focus on predicting the workload. Nonetheless, our work focuses on providing scheduling policies to satisfy user preferences and we do not deal with workload prediction issues.

In Table 1 different systems that provision resources from IaaS providers are compared from different aspects. This table also reveals the differences of our proposed policies with similar works in the area.

3   Proposed Policy

Scheduling applications is complex when a user places constraints such as execution time and computation cost. Satisfying such requirements is challenging specifically when the local resources are limited in computational capacity and the execution time of the application is not known in advance. In this situation, scheduling policies need to adapt to the changing load in order to meet the deadline and cost constraints. Moreover, the scheduling policy should consider the user preference in terms of time or cost minimization.

To cope with the challenge, our solution relies on supplying more resources from IaaS providers. Therefore, we propose two scheduling policies namely, Time Optimization and Cost Optimization. In this section, details of these policies are described.

3.1 Time Optimization Policy

The Time Optimization policy, as mentioned before, aims at completing the application as quickly as possible, within the available budget. Therefore, according to the pseudo code presented in Algorithm 1, the scheduler spends the whole available budget for hiring resources from the IaaS provider (steps 1 to 3). *Available budget* is defined according to equation (1) and indicates the amount of money the scheduler can spend per hour. However, the number of hired resources should not be more than the number of remaining tasks.

There is a delay between the time the request is sent to the IaaS provider and the time resources become accessible[1]. After getting accessible, hired resources are added to the list of available servers (step 4) and the scheduler can dispatch tasks to them (steps 6, 7). *AddAsServer()* is a thread that keeps track of the request sent to the IaaS provider to get accessible. To attain the minimum execution time, hired resources are kept up to the end of execution. At the end of execution, all resources from the IaaS provider are terminated (step 8).

$$BudgetPerHour = \frac{TotalBudget}{\lceil Deadline - CurrenrtTime \rceil} \tag{1}$$

Algorithm 1: Time Optimization Scheduling Policy.

   Input: deadline, totalBudget,resourceCost

1 budgetPerHour =totalBudget / (deadline −currentTime);
2 reqCounter =budgetPerHour / resourceCost;
3 RequestResource(reqCounter);
4 availResources $+=$ AddAsServer();
5 DoAccounting();
6 **while** TaskRemained() $=$ *True* **do**
7 | SubmitTask(availResources);
8 Terminate(reqCounter);

3.2 Cost Optimization Policy

The Cost Optimization policy completes the application as economically as possible within the deadline. According to the pseudo code presented in Algorithm 2, In each scheduling iteration, a set of tasks are submitted to available resources (step 4). Available resources refer to local resources plus resources hired from IaaS provider so far. Then, in step 5 the scheduler estimates the completion time of the remaining tasks based on the time taken for the tasks that have got completed so far.

---

[1] The delay is actually because of the time taken to make (boot) a virtual machine from machine image. For more details see [9].

However, since we are not dealing with the workload prediction issues in this work, we assume that all tasks of the high performance application have the same execution time. Therefore, *EstimateCompletionTime()* in step 5 is a function that estimates the completion time based on equation (2).

$$Estimation = TasksRemained * TaskRunTime \qquad (2)$$

If there is not any available resource (step 6), then a default initial estimation is assumed (step 7) to make the policy hire one resource from the IaaS provider.

In each scheduling iteration, if it is realized that the deadline could not be met and there is enough budget available (steps 9, 10), then just one resource is hired from the IaaS provider. *AddAsServer()* add the hired resource to available resources as soon as it becomes accessible.

Thus, in the Cost Optimization policy resources are requested from the IaaS provider over time. This results in spending more time to get access to hired resources rather than the Time Optimization policy. We investigate the impact of this issue in the experiments in more details.

Termination of the hired resources happens when the estimated completion time is smaller enough than the deadline (steps 15, 16). In fact, to maximize the chance that the deadline can still be met after terminating one resource, termination is only done if the estimated completion time is smaller than a fraction of the deadline ($estimation < (deadline * \alpha)$ where $\alpha < 1$). In the current implementation of Cost Optimization policy, we consider α as a constant coefficient (0.7 in our experiments). However, as a future work, we plan to investigate an adaptive value for α.

DoAccounting(), in both Time Optimization and Cost Optimization policies, takes care of budgeting for hired resources and decreases the available budget base on the price of the hired resources per hour. If there is not enough budget, then *DoAccounting()* terminates each hired resource before it starts a new charging cycle.

Note that the implementation of the above policies contains extra steps to keep track of ordered resources to get accessible, accounting, and terminating hired resources. All of these processes are done in separate threads to have the minimum impact on the scheduling performance.

Time Optimization and Cost optimization policies are implemented in Gridbus broker. Moreover, Gridbus broker is extended to be able to interact with Amazon EC2 as an IaaS provider. In the next section, details of extending the broker to Amazon EC2 are discussed.

4   System Implementation

Gridbus broker mediates access to distributed resources running diverse middleware. The broker is able to interface with various middleware services, such as Condor, and SGE [10]. In this work, we extend Gridbus broker to interact with Amazon EC2 as an IaaS provider. Then, we incorporate the aforementioned scheduling policies into the broker.

Algorithm 2: Cost Optimization Scheduling Policy.

Input: deadline, totalBudget,resourceCost

1 `SetAvailBudget(totalBudget);`

2 **while** `TaskRemained()` = *True* **do**

3   **if** availResources > 0 **then**

4   `SubmitTask(availResources);`

5   estimation = `EstimateCompletionTime();`

6   **else**

7   estimation = deadline + 1;

8   **if** estimation > deadline **then**

9   availBudget = `GetAvailBudget();`

10   **if** availBudget ≥ resourceCost **then**

11   `RequestResource(1);`

12   availResources += `AddAsServer();`

13   `DoAccounting();`

14   **else**

15   **if** estimation < (deadline ∗ α) **then**

16   `Terminate(1);`

In our implemented architecturen (Fig. 1), *EC2ComputesManager* has a key role in managing resources from Amazon EC2. *EC2ComputesManager* initiates a thread that keeps track of resources requested by scheduling policy on Amazon EC2. When a resource gets accessible, the resource is added to the available resources as an *EC2ComputeServer* object and scheduler can submit task to it. *EC2ComputeServer* also deals with the pricing model of the Amazon EC2 by overriding the payment method in *isPaid()* method. Finally, *EC2ComputeServer* terminates resources from IaaS provider when *terminate()* method is invoked by the scheduler. *EC2Instance* contains all related attributes and relevant methods for managing resources from the IaaS provider. Particularly, *isReadytoUse()* method determines if a requested resource from the Amazon EC2 is accessible or not.

Results of our evaluation on the Time Optimization and the Cost optimization policies in Gridbus broker are presented in the next section.

5 Performance Evaluation

5.1 Experiment Setup

The specification of the resources and applications used in the experiments are described in this section. We use a cluster (Snowball.cs.gsu.edu) as the local resource. The cluster has 8 Pentium III (XEON 1.9 GHZ) CPU, 1GB RAM, and Linux operating system. We also use Amazon EC2 as the IaaS provider.
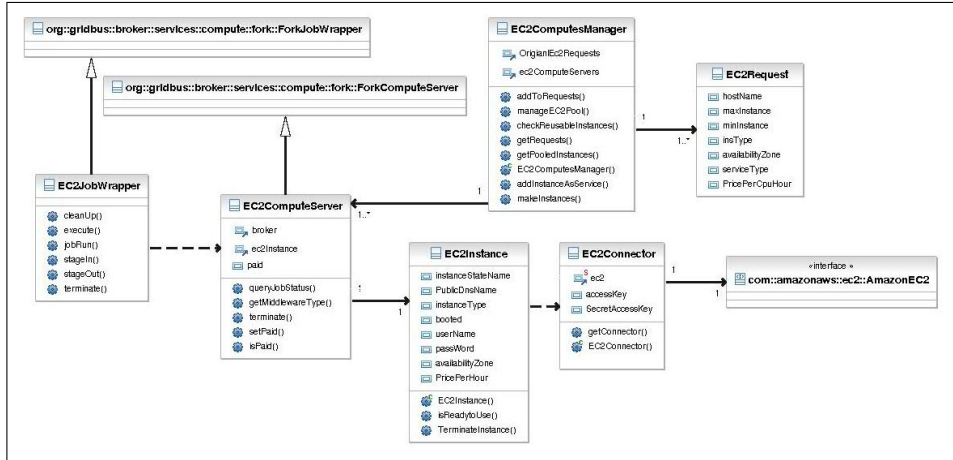
Fig. 1. Class diagram describing the implementation details of extending Gridbus roker to Amazon EC2 as an IaaS provider.

Amazon EC2 offers resources with different computational power. In the experiments we use the cheapest resource type which is known as *small* computational unit (we call it small instance). Another reason for using small instances is that, in terms of hardware specification, small instances are the most similar resource types to our local resources in the cluster. Each small instance is equivalent to 1.2 GHZ XEON CPU, 1.7GB RAM, Linux operating system, and costs 10 cents per hour.

We use a Parameter Sweep Application (PSA) in the experiments. A PSA typically is a parameterized application which has to be executed independently with different parameter values (each one is called a task). Pov-Ray [7] is a popular parameter sweep application in image rendering and it is widely used in testing distributed systems. In the experiments, we configure Pov-Ray to render images with the same size. Therefore, we ensure that the execution time is almost the same for all the tasks.

5.2 Experiment Results

The Impact of Budget Spent on Completion Time. In the first experiment, we measure how the application completion time is affected based on the different budget allocated by the user.

For this purpose, we consider the situation that the user wants to run Pov-Ray to render 144 images in two hours (120 minutes) as the deadline. The overall execution time when just Snowball.cs.gsu.edu cluster is used is 138 minutes and since no cost is assigned to the cluster, the overall execution time does not vary by increasing budgets (Baseline in Figure 2). Then, two proposed policies (Time Optimization and Cost Optimization) are tested in the same situation.
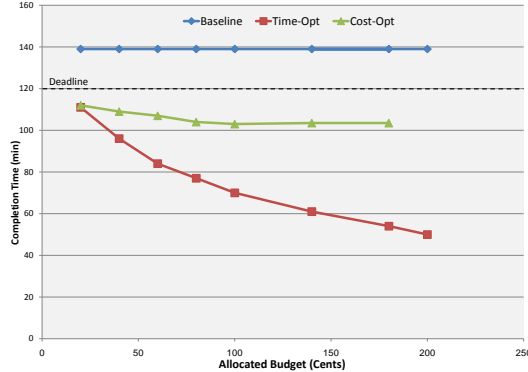
Fig. 2. The impact of allocating more budget on the application completion time without resources from the IaaS provider (Baseline), with Time optimization policy (Time-Opt) and with Cost Optimization policy (Cost-Opt).

As it can be understood from Figure 2, in the Time Optimization policy the application completion time decreases by available budget almost linearly. However, in the Cost Optimization the completion time does not improve anymore after a certain budget (100 cents in this case). In fact, this is the point that the policy does not spend any more money to request more resources from the IaaS provider even if there is some budget available. Moreover, for the budgets less than 100 cents, Cost Optimization policy takes more time to complete rather than Time Optimization with the same budget allocated. This is mainly because the Cost Optimization policy does not spend all of the allocated budget. This issue is discussed more in the next experiment in which the efficiency of the two policies is illustrated. Another reason is that, resources in the Cost Optimization are added over the time and terminated as soon as the scheduler realizes that the deadline can be met. However, in the Time Optimization all resources are requested in the very first moments and kept up to the end of execution.

Efficiency of the Time Optimization and Cost Optimization Policy. In this experiment, the efficiency of the Time Optimization and the Cost Optimization scheduling policies are measured for different amount of allocated budget. We define the efficiency as follows:

$$efficiency = \frac{(\alpha - \beta)}{\delta} \tag{3}$$

Where α is the time taken to complete the application just by deploying Snowball.cs.gsu.edu cluster. β is the completion time by using both Snowball.cs.gsu.edu cluster and resources from the IaaS provider. Finally, δ indicates the budget spent to hire resources from IaaS providers. Other experiment parameters are the same as experiment 5.2.
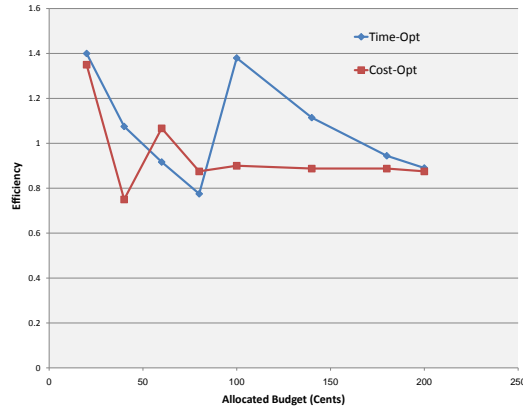
Fig. 3. Efficiency of Time Optimization and Cost Optimization policies with different budget allocated.

According to Figure 3, in the Time Optimization policy, the decrease in efficiency (from 1.4 to 0.77 and from 1.38 to 0.89) happens because of the rapid increase in spent budget. However, there is a sharp rise in efficiency (from 0.77 to 1.38) when the allocated budget increases to 100. This is mainly because of the decrease in spent budget. In other words, by hiring five resources from the IaaS provider, the application gets completed before another charging cycle for resources from IaaS provider starts. Therefore, the spent budget decreases sharply (from 80 to 50 cents) and the efficiency increases. We expect more similar sudden rises in the Time Optimization policy, specifically when the deadline is long (several hours or days).

Similar behavior happens in Cost Optimization, when the allocated budget is increased from 40 to 60 cents. In this case, again more resources (three resources for one hour) are kept for less time instead of fewer resources for more time (two resources for two hours when allocated budget is 40).

The Impact of the Time Optimization and Cost Optimization Scheduling Policies on the Completion Time of Different Workload Types. In this experiment, we investigate how the Time Optimization and the Cost Optimization policies behave for different workload types. Doing this experiment, we consider five workload types. According to Table 2, the number of tasks increase exponentially whereas the run time for each task decreases at the same rate from type 1 to type 5. All of these workloads have the same completion time (150 minutes) when just Snowball.cs.gsu.edu cluster is used. For all of these workloads, we use the same deadline and budget (120 minutes and 100 cents respectively) during the experiment.

This experiment demonstrates the applicability of the proposed policies for different kinds of workloads. As it is illustrated in Figure 4, both policies can get the application completed before the deadline. The only deadline violation
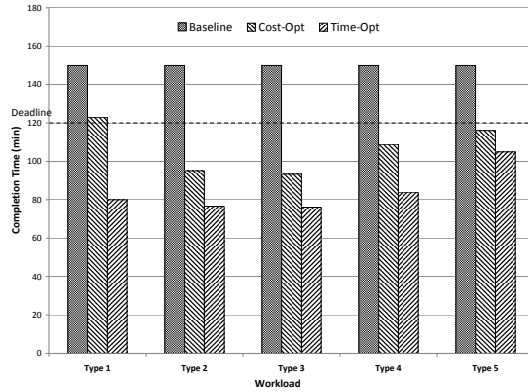
Fig. 4. Application completion time for different workload types with the cluster as local resource (Baseline) and with the resources from the IaaS provider in Time Optimization (Time-opt) and Cost Optimization (Cost-opt) policies.

is in Cost Optimization policy for workload type 1. The reason is that there is not enough scheduling iteration in which Cost Optimization policy can request more resources from the IaaS provider, thus that workload get completed just by two resources ordered from the IaaS provider.

The minimum difference in completion time between Time Optimization and Cost Optimization is in the workload type 5. The reason is that the execution time for each task is short (2.34 minutes according to Table 2). This results in more frequent scheduling iterations. Hence, extra resources from the IaaS provider are requested in the very first scheduling iteration and these additional resources can contribute more for running tasks. However, the reason for dif-

Table 2. Different workload types used in the experiment 5.2.

Workload	No of Tasks	Task Time (minutes)
Type 1	32	38
Type 2	64	18.75
Type 3	128	9.37
Type 4	256	4.65
Type 5	512	2.34

ference in completion time is that in the Cost Optimization resources are not retained up to the end of execution. Another reason for difference in completion time is the time taken by the IaaS provider to make the resources accessible. Since in the Cost Optimization policy resources are requested over time, the overhead related to preparing resource by the IaaS provider is longer than the Time Optimization policy in which all resources from the IaaS provider are requested at the same time.

6 Conclusion and Future Work

In this paper, two market-oriented scheduling policies are proposed to increase the computational capacity of the local resources by hiring resources from an IaaS provider. Both policies consider user provided deadline and budget in their scheduling. Time Optimization scheduling policy minimizes the application completion time. On the other hand, Cost Optimization scheduling policy minimizes the cost incurred for running the application. We evaluate these policies in real environment using Gridbus broker as a user-level broker. We observed that in the Time Optimization policy, completion time reduces almost linearly by increasing the budget. However, in the Cost Optimization the completion time does not improve after a certain budget (100 cents in our experiments). We can also conclude that the efficiency of the Time Optimization and Cost Optimization policies can potentially increase by increasing the budget. Finally, we observed that different workload types can get completed before the deadline and within the budget using the proposed policies.

As a future work we plan to extend the current work to a situation that there are several IaaS providers with different prices for their resources.

References

1. Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2.
2. C.V. Blanco, E. Huedo, R.S. Montero, and I.M. Llorente. Dynamic provision of computing resources from grid infrastructures and cloud providers. *Grid and Pervasive Computing Conference*, 0:113–120, 2009.
3. R. Buyya, M.M. Murshed, D. Abramson, and S. Venugopal. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. *Software Practice and Experience*, 35(5):491–512, 2005.
4. M.D. de Assunção, A. di Costanzo, and R. Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 141–150. ACM New York, NY, USA, 2009.
5. J. Fontán, T. Vázquez, L. Gonzalez, RS Montero, and IM Llorente. OpenNEbula: The open source virtual machine manager for cluster computing. In *Open Source Grid and Cluster Software Conference*, 2008.
6. I. Llorente, R. Moreno-Vozmediano, and R. Montero. Cloud computing for on-demand grid resource provisioning. *Advances in Parallel Computing*, 2009.
7. The Persistence of Vision Raytracer, http://www.povray.org.
8. J.N. Silva, L. Veiga, and P. Ferreira. Heuristic for resources allocation on utility computing infrastructures. In *Proceedings of the 6th international workshop on Middleware for grid computing*, pages 9–17, 2008.
9. B. Sotomayor, K. Keahey, and I.T. Foster. Combining batch execution and leasing using virtual machines. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 87–96, 2008.
10. S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling e-science applications on global data grids. *Concurrency and Computation: Practice & Experience*, 18(6):685–699, 2006.