Merkat: Market-based Autonomous Application and Resource Management in the Cloud

Stefania Victoria Costache, Nikos Parlavantzas, Christine Morin, Samuel Kortas

Merkat: Market-based Autonomous Application and Resource Management in the Cloud

Stefania Costache , Nikos Parlavantzas , Christine Morin, Samuel Kortas

RESEARCH REPORT
N° 8343
August 2013
Project-Team MYRIADS

ISSN 0249-6399 ISRN INRIA/RR--8343--FR+ENG

Merkat: Market-based Autonomous Application and Resource Management in the Cloud

Stefania Costache [*], Nikos Parlavantzas [†], Christine Morin[‡], Samuel Kortas [*]

Project-Team MYRIADS

Research Report n° 8343 — August 2013 — 25 pages

Abstract: Organizations owning HPC infrastructures are facing difficulties in managing their infrastructures. These difficulties come from the need to provide concurrent resource access to applications with different resource requirements while considering that users might have different performance objectives, or Service Level Objectives (SLOs) for executing them. To address these challenges this paper proposes a market-based SLO-driven cloud platform. This platform relies on a market-based model to allocate resources to applications while taking advantage of cloud flexibility to maximize resource utilization. The combination of currency distribution and dynamic resource pricing ensures fair resource distribution. In the same time, autonomous controllers apply adaptation policies to scale the application resource demand according to user SLOs. The adaptation policies can: (i) dynamically tune the amount of CPU and memory provisioned for the virtual machines in contention periods; (ii) dynamically change the number of virtual machines. We evaluated this proposed platform on the Grid'5000 testbed. Results show that: (i) the platform provides flexible support for different application types and different SLOs; (ii) the platform is capable to provide good user satisfaction achieving acceptable performance degradation compared to existing centralized solutions.

Key-words: cloud computing, resource management, autonomous systems

[*] EDF R&D, Clamart, France
[†] INSA, Rennes, France
[‡] INRIA Rennes-Bretagne Atlantique, Rennes, France

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Merkat: Gestion autonome des ressources et des applications dans un nuage informatique selon une approche fondée sur un marché

Résumé : Les organisations qui possèdent des infrastructures de calcul à haute performance (HPC) font souvent face à certaines difficultés dans la gestion de leurs ressources. En particulier, ces difficultés peuvent provenir du fait que des applications de différents types doivent pouvoir accéder concurremment aux ressources tandis que les utilisateurs peuvent avoir des objectifs de performance (SLOs) variés. Pour atteindre ces difficultés, cette article propose un cadre générique et extensible pour la gestion autonome des applications et l'allocation dynamique des ressources. L'allocation des ressources et l'exécution des applications est régie par une économie de marché observant au mieux des objectifs de niveau de service (SLO) tout en tirant avantage de la flexibilité d'une nuage informatique et en maximisant l'utilisation de des ressources. Le marché fixe dynamiquement un prix aux ressources, ce qui, combiné avec une politique de distribution de monnaie entre les utilisateurs, en garantit une utilisation équitable. Simultanément, des contrôleurs autonomes mettent en œuvre des politiques d'adaptation pour faire évoluer la demande en ressource de leur application en accord avec la SLO requise par l'utilisateur. Les politiques d'adaptation peuvent : (i) adapter dynamiquement leur demande en terme de CPU et de mémoire demandés en période de contention de ressource aux machines virtuelles (ii) et changer dynamiquement le nombre de machines virtuelle. Nous avons évalué cette plateforme sur l'infrastructure Grid'5000. Nos résultats ont montré que cette solution: (i) offre un support plus flexible aux applications de type différent demandant divers niveaux de service; (ii) conduit à une bonne satisfaction des utilisateurs moyennant une dégradation acceptable des performances comparées aux solutions centralisées existantes.

Mots-clés : nuages informatiques, allocation de ressources, systemes autonomes

1 Introduction

Organizations owning HPC infrastructures are facing difficulties in managing their resources. An example of such an organization is Electricité de France (EDF), which performs many numerical simulations in the fields of neutronics, fluid dynamics or thermodynamics. EDF has different application types, requiring different software configurations and having different resource demands. For example, some simulations require specific frameworks (e.g., MPI, Hadoop, Condor) to run. Then, while some simulations are composed of a fixed set of processes, others are composed of thousands of independent tasks and can adapt their resource demand to the current resource availability. In the same time, users might also have different performance requirements, from simple ones, e.g., *stop my application when a certain performance condition is met* to Service Level Objectives (SLOs), e.g., *provide the results of my application by 7am next day*. As these studies need to be executed in a secure context, users are restricted at using EDF's clusters. Given this variety of requirements, an important concern is to manage the infrastructure's resources to satisfy the users while maximizing resource utilization.

Even if transforming the infrastructure in a private cloud is attractive, it does not completly address the previously mentioned problems. Using a private cloud has two main advantages. First, users gain control over the software environment in which their application runs in an easy-to-manage fashion for the infrastructure administrator. Then, the application performance and infrastructure utilization can be improved by provisioning dynamically virtual machines (VMs), avoiding under- and over-provisioning. In this context, a variety of solutions provide users with elastic environments for their applications, hiding from them the complexity of managing the infrastructure's resources [1, 2, 3, 4]. These systems face several issues. First, because they are typically closed environments, they force users to run specific application types. Second, they provide limited support to meet user's performance objectives: in the best case, users have access to poor APIs to define their own application management policies. Finally, they do not address corectly contention periods. These systems rely on the "on-demand" provisioning models provided by the Infrastructure-as-a-Service (IaaS) providers and user quotas to restrict the resource demand of each user. However, in this case users do not have incentives to limit their resource demand properly [5], leading to a poor user satisfaction. Given the limited infrastructure's capacity, it is important to ensure that users are satisfied from using it.

In this paper we present Merkat, a *private market-based cloud platform* that addresses these problems. The novelty of Merkat is the combination of three characteristics. First, Merkat provides support for per-application SLOs by running applications in autonomous virtual environments that can scale their resource demand according to application preferences. Second, Merkat provides fair maximized resource utilization. Merkat relies on a market to allocate resources to VMs, making users more responsible regarding their application execution. Merkat uses a proportional-share policy to compute the resource price and the CPU and memory amounts that each VM is entitled to, maximizing the used resources. Finally, Merkat provides two types of per-application resource scaling policies: horizontal and vertical scaling for CPU and memory under cost constraints. Thus, different applications types can run more efficiently on the

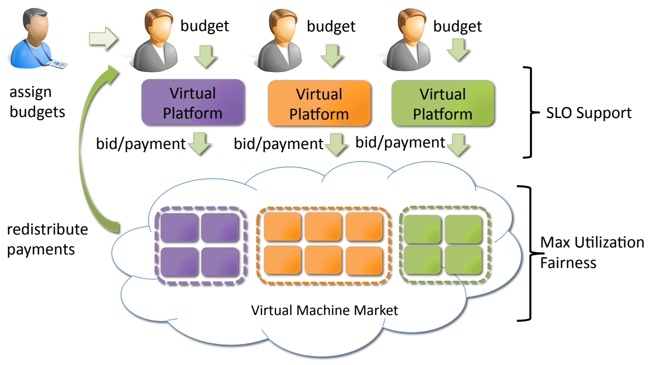


Figure 1: The Merkat eco-system.

infrastructure.

We evaluated Merkat on Grid'5000 [6], using applications encountered at EDF: static MPI applications and malleable task processing frameworks. We show that: (i) Merkat adapts the application resource demand to the infrastructure resource availability and user SLOs; (ii) Merkat can achieve better user satisfaction than traditional systems.

This paper is organized as follows. Section 2 gives an overview of Merkat and surveys related work. Section 4 describes its architecture and Section **??** outlines the resource management model used in Merkat. We present experimental results in Section 5. Section 7 concludes the paper.

2 Merkat

Figure 1 gives an overview of Merkat and its main design principles. We describe these design principles next.

2.1 Flexible SLO support

To provide SLO support to its users, Merkat runs each application in an autonomous virtual platform. A virtual platform is represented by the virtual environment in which the application runs complemented with a decision logic to scale the application resource demand. The architecture of a virtual platform is illustrated in Figure 2. A virtual platform is composed of one or multiple virtual clusters, a set of monitors and an application controller. A *virtual cluster* is a group of VMs that are deployed using the same disk image, and host the same application components. The *application controller* manages the application life-cycle and provisions VMs to run the user's application, which may have SLOs, such as deadlines. VM provisioning is done based on current application performance metrics, retreived from *monitors* deployed in virtual clusters. By running each application in its own virtual platform, Merkat supports different software configurations on the same physical resources with minimal interference from the administrator. Each virtual platform adapts individually, without

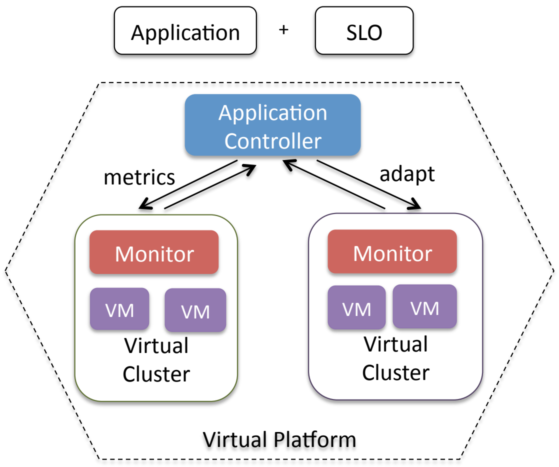


Figure 2: The architecture of a virtual platform.

knowledge regarding other participants. Thus, users can run different applications and express a variety of performance goals.

2.2 Fair resource utilization

Merkat implements a market to allocate resources to users. We use a *credit* as a currency unit. This currency is internal to the organization and is controlled by an administrator. To run applications on the infrastructure, users receive budgets from a bank, i.e., amounts of credits. Users distribute amounts from these budgets to the virtual platforms running their applications, reflecting the maximum cost the user is willing to support for the execution of her application. In the same time, the CPU and memory resources have a price set through market policies, e.g., auctions, which fluctuates based on the resource demand. As users are limited by their available currency and the resource price, they also have incentives to assign a budget for their application execution that expresses their valuation for resources. Thus, the system can ensure a fair resource utilization.

2.3 Maximized resource utilization

The market implemented in Merkat is based on a proportional-share policy to allocate CPU and memory to VMs. We chose the proportional-share policy, as, compared to other auctions, it can be used to allocate fractional resource amounts to VMs at a low implementation complexity cost, leading to a maximized resource utilization. Fractional resource allocation allows the leveraging of different application resource utilization patterns, i.e., some applications might have varying resource demand per VM, while other applications can run with less resources per VM. As resources are allocated to VMs dynamically, both application types can be efficiently supported on the infrastructure while avoiding over-provisioning.

Proportional-share was used by operating system schedulers to share resources between a dynamic number of tasks. In our market, resources are distributed among running VMs, instead of running tasks, as follows. The virtual platforms submit bids for the CPU and memory allocated to each of their VMs. Then, resources are allocated with the following rule: given a set of n VM bids $b_i(t)$ for a time interval t and a resource with a capacity of C units, a scheduler allocates a resource amount a_i for each VM i equal to:

$$a_i(t) = \frac{b_i(t)}{p(t)}, p(t) = \frac{\sum_{i=1}^{n} b_i(t)}{C} \tag{1}$$

where $p(t)$ is the price for a unit of that resource. VMs are billed based on their bids.

On top of this market, application controllers can scale the resource demand of their applications in two ways: *vertically*, by adjusting their bid per VM, and *horizontally*, by provisioning/releasing VMs. Vertical scaling is attractive for applications that cannot be modified to take advantage of a dynamic number of virtual machines. Horizontal scaling is attractive for malleable applications as they can provision a dynamic number of VMs during their execution.

3 Resource Management Model

In this section we describe the resource model applied in Merkat. We detail how resources are allocated to VMs and how applications can adapt their resource demand.

3.1 VM provisioning

To request a VM, a user, or her application controller, submits a set of bids (a bid for each of the VM's allocated resources) together with a maximum resource allocation. A scheduler uses this information to place VMs on nodes and allocate resources to them such that all the infrastructure capacity is distributed while considering node capacity constraints. As the system load or VM bids can vary in time, VMs are also migrated among nodes, process called "load balancing". To minimize the impact of VM migrations on the application's performance, load balancing is performed in two cases: (i) if the maximum *allocation error* that each VM has for its resources is above a given value; (ii) and if the number of migrations performed at the given time period is below a given value. We explain next what is the allocation error and how load balancing is done.

Allocation error The allocation error for a resource is the difference between the allocation computed from the capacity of a node and the allocation computed from the infrastructure's capacity. To ensure that all the infrastructure capacity is distributed, a straightforward solution is to apply Equation 1 by considering the infrastructure as a huge host. However, as the infrastructure capacity is partitioned between nodes, resulted allocations cannot be enforced all the time. This issue is illustrated through the following example, in which, for simplicity, we focus only on CPU allocation. We consider 3 nodes with a capacity of 100 CPU units each, 3 VMs, each receiving a bid of 12 CPU units, and 2 VMs, each receiving a bid of 30 CPU units. Using Equation 1, the market scheduler computes an allocation of 37.5 CPU units for the first 3 VMs and an allocation of 93.75 CPU units for the last two VMs. Intuitively, these allocations cannot be enforced.

To solve this issue, we compute the resource allocations after placing the VMs on the nodes, by using the capacity of the node. To use the previous example, the resulted allocations would be: 33.3 CPU units for the first 3 VMs and 100 CPU units for the last 2 VMs. The resulted allocation difference is called *allocation error*. This error can also appear from changes in the VM bids or system load.

Load balancing To minimize the allocation error of the VMs while considering the cost of migration on the application performance, the market scheduler migrates them among the nodes. To select the VMs to be migrated at each scheduling period, the market scheduler relies on an algorithm based on a tabu-search heuristic [7]. Tabu-search is a local-search method for finding the optimal solutions of a problem by starting from a potential solution and applying incremental changes to it. At each iteration, the algorithm tries to improve the solution by moving the VM with the maximum allocation error among its resources that is not in the tabu list to the physical node that minimizes it. The move is recorded in the tabu list to avoid reversing it in the next iterations.

3.2 Application resource demand adaptation

To provide SLO support on top of the virtual machine market, Merkat uses controllers that supervize the applications and apply resource demand adaptation policies. These policies use the dynamic resource price as a feedback signal regarding the resource contention and respond by adapting the application resource demand given the user's SLO and budget. We designed two policies: (i) vertical scaling; (ii) and horizontal scaling. Both policies address three cases:

- *Preserve budget when the SLO is met:* When the user SLO can be met, the application can reduce its resource demand and thus its execution cost. The remaining budget can be used afterwards to run other applications.
- *Provide more resources when the SLO is not met:* When the application workload changes, the application might need to increase its resource demand to meet the SLO.
- *React when the SLO is not met due to budget limitations:* When the application cannot meet its SLO, because the current resource price is too high and the application budget is too limited to ensure the desired resource allocation, the application might reduce its resource demand, to minimize the cost of using resources.

The policies are run periodically and use two thresholds, upper and lower, as an alarm: when the application performance metric traverses the thresholds, the policy takes an action that changes the application resource demand. The vertical scaling policy adapts the resource demand of each VM by tunning its bid for resources and uses suspend/resume mechanisms to avoid high price periods. This policy can be applied to optimize the budget when running static MPI

Algorithm 1 Vertical scaling adaptation policy

```
Input: $bid, bid_{min}, bid_{max}, alloc, alloc_{max}, alloc_{min}, share, v, v_{ref},\ v_{low},\ v_{high},\ history$    // $bid_{min}$ is the minimum limit accepted by the infrastructure
    // $bid_{max}$ - the budget to be spent for the next time period
    // $history = (bid - bid_{new})$ - the previous bid change
    // $alloc, alloc_{min}, alloc_{max}$ - the actual, minimum and maximum allocation
    // share - the current share the application receives
    // $v, v_{reference}$ - the application current and reference performance values
    // $v_{low}, v_{high}$ - the lower and upper bounds of the performance values
Output: bid    // the bids submitted for the next time period
 1: resources = [cpu, memory]
 2: $T = \| \frac{v_{ref} - v}{v} \|$
 3: direction = 0
 4: if $v > v_{high}$ then
 5:     direction = -1
 6: if $v < v_{low}$ then
 7:     direction = 1
 8: for $r \in resources$ do
 9:     if ($direction > 0$) and $alloc[r] > alloc_{min}[r]$ or ($share > alloc_{max}$) then
10:         if history < 0 then
11:             bid[r] = decrease bid[r] with max(2, (1 + T)/2) until $bid_{min}$    // the bid was increased
12:         else
13:             bid[r] = decrease bid[r] with max(2, 1 + T) until $bid_{min}$
14:     if ($direction < 0$ and $alloc[r] < alloc_{max}[r]$) or ($alloc[r] < alloc_{min}[r]$) then
15:         if history > 0 then
16:             bid[r] = increase bid[r] with max(2, (1 + T)/2)    // the bid was decreased
17:         else
18:             bid[r] = increase bid[r] with max(2, 1 + T)
    // bids are re-adjusted due to budget limitations
19: if $bid[memory] + bid[cpu] > bid_{max}$ then
20:     $w = \emptyset$
21:     if $alloc[r] \geq alloc_{max}[r],\ r \in resources$ then
22:         bid[$i \in resources - r$] = $bid_{max} - bid[r]$
23:     else
24:         $w[r] = 1 - \frac{alloc[r]}{alloc_{max}[r]},\ r \in resources$
25:         $bid[r] = \frac{bid_{max}}{w[r]}$
```

applications under user-given time constraints[1]. The horizontal scaling policy adapts the number of VMs and their bids. This policy can be applied to optimize the budget when running malleable applications. We describe each policy next.

Vertical Scaling Policy

Algorithm 1 describes the vertical scaling policy. The policy outputs the resource bids for the next period based on the following information: (i) current VM allocation; (ii) current application performance metrics, v; (iii) application reference performance, v_{ref} (iii) the current resource bids (iv) the value of the last bid change; (v) and the budget to be spent for the next time period.

The policy preserves the user budget when the SLO can be met by decreasing the resource bids in two situations: (i) if the performance metric drops below the lower threshold (e.g., 75% of remaining time to deadline); (ii) or if the allocation per VM for one resource reaches the maximum.

To provide more resources when the SLO is not met, i.e., the performance metric is above the higher threshold, the policy increases the resource bids.

If the current budget is not enough to meet the SLO, the policy applies two decisions, based on the user SLO type: (i) if it is advantageous to suspend the

[1]Note that when the infrastructure is underutilized, the policy doesn't shrink the application resource demand as the application still receives a maximum allocation. Nevertheless, its execution cost is reduced

Algorithm 2 Horizontal scaling adaptation policy

```
Input: $nvms_{old}, bid_{max}, alloc_{max}, v, v_{low}, v_{high}$
        // $nvms_{old}$ number of VMs from the previous period
        // $bid_{max}$ is the budget to be spent for the next time period
        // $alloc_{max}$ are the actual, minimum and maximum allocation
        // $v, v_{low}, v_{high}$ are the application current, lower and upper bound performance value
Output: nvms, bid    // the bids submitted for the next time period
 1: direction = 0
 2: if $v > v_{high}$ then
 3:     direction = -1
 4: if $v < v_{low}$ then
 5:     direction = 1
 6: if $direction < 0$ then
 7:     $nvms = nvms + 1$
 8: if ($direction > 0$ then
 9:     pick vm to release
10:     $nvms = nvms - 1$
11: (N, bid) = compute VM number upper bound ($bid_{max}$, $alloc_{max}$, nvms)
12: if $N < nvms$ then
13:     release (nvms - N) VMs
14: else
15:     request new $nvms - nvms_{old}$ VMs
```

Algorithm 3 VM upper bound computation

```
Input: $nvms_{max}, bid_{max}, alloc_{max}$
Output: nvms, bid
 1: resources = [cpu, memory]
 2: find maximum value of N $\in [1, nvms_{max}]$ for which
 3: $\sum_{r \in resources} \cdot \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}} < bid_{max}$
 4: nvms = N
 5: bid[r] = $\frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}}$, $r \in resources$
```

application execution, e.g., in the case of best-effort or deadline-driven batch applications, it suspends the application, resuming it when the price drops, e.g., VMs can receive 75% of the maximum allocation at the current budget; (ii) if it is not advantageous to suspend the application execution, e.g., interactive applications, or applications for which the user might be satisfied with partial results, i.e., 90% of the computation, until a deadline, it recomputes the bids in a way that favors the resource with a small allocation. The bids are increased using a proportional-share policy, where the resource "weight", is the difference between the actual and maximum allocation of the VM.

The value with which the bid changes is given by the "gap" between the current performance value and the performance reference value: $T = |(v_{ref} - v)/v|$. A large gap allows the application to reach its reference performance fast. To avoid too many bid oscillations the policy uses the value of the past bid change in its bid computation process. For example, if the bid was previously increased and at the current time period the bid needs to be decreased with a similar value, the bid oscillates indefinitely. Thus, the algorithm decreases the current bid with half of its value.

Horizontal Scaling Policy

The horizontal scaling policy outputs the number of VMs and the bids for the next time period based on the following information: (i) current resource prices; (ii) the user budget for the next time period; (iii) and the VM maximum allocation.

These results are computed in two steps: (i) it changes the current number of VMs, *nvms*, based on the application performance metric; (ii) it computes the maximum number of VMs for the next time period so that all the VMs receive a maximum resource allocation at the current resource prices.

The first step is used to preserve the budget when the SLO is met or to increase the resource demand when the SLO is not met. Algorithm 2 describes this step. The algorithm provisions VMs as long as a reference value (e.g., execution time, number of tasks) is above a given threshold and releases them otherwise. To ensure that the VMs receive a maximum allocation given the user's budget constraints, the number of VMs is limited to a given value, computed in the second step of the algorithm.

The second step is used to reduce the resource demand when the SLO is not met due to budget limitations. For clarity, Algorithm 3 describes this step. The main idea of this algorithm is to compute the bid value for each VM resource from the Equation 1 by replacing a_i with $N \cdot a_{max}$, where N is the number of VMs and a_{max} is the maximum allocation per VM. To find the upper bound on the number of VMs, the algorithm performs a binary search between 1 and *nvms* by checking at each iteration if the sum of bids the controller needs to submit is less than its budget.

In cases in which applications can scale their demand both horizontally and vertically, a combination of the Algorithm 1 and Algorithm 2 is desirable. For example, an application can scale its resource demand first vertically. Then, when the allocation of its VMs is already at maximum, it can increase its VM number incrementally until the upper bound is reached or the application performance value drops.

4 The Architecture of Merkat

Merkat is designed to be generic and extensible to support a variety of workloads and resource provisioning policies. In this section we describe the services of Merkat and we illustrate the application life-cycle in Merkat and how users can adapt Merkat to their specific needs.

4.1 Services

Figure 3 gives an overview of Merkat's services and the interactions between them. Merkat is composed of three services: the applications manager, the market scheduler, and the credit manager. These services are built on top of an IaaS cloud manager, which provides basic functionalities for managing VMs and users. Users and administrators interact with these services through a set of CLI tools or by using the XML-RPC protocol.

4.1.1 The Applications Manager

The applications manager acts as the entry point for deploying applications. When it receives a new deployment request from a user, it creates a new virtual platform, which manages the user's application.

Users specify their deployment information in a file, called virtual platform template. Such information can be budgeting, the type of application controller

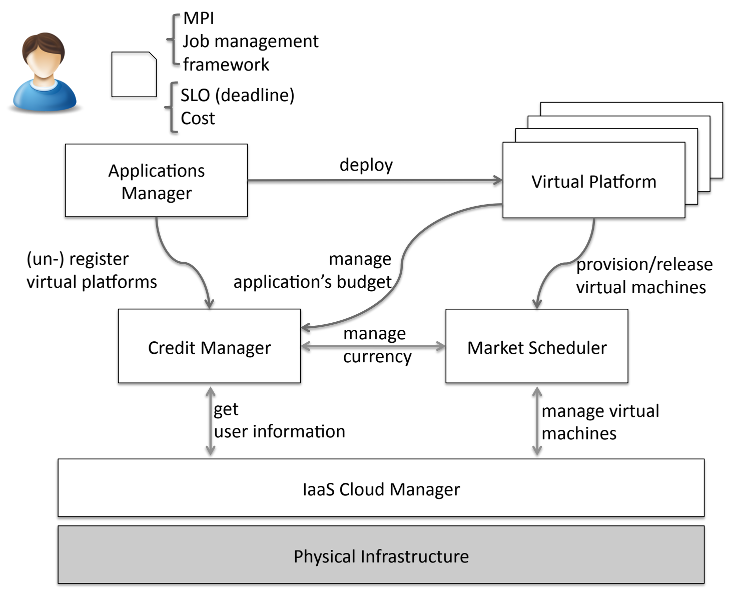


Figure 3: The architecture of Merkat.

and resource scaling policies to load and the virtual cluster configuration. A virtual cluster configuration includes information on commands to manage application components, files, copied in/from the VMs before/after the application is started/stopped, initial user bids, and VM configurations. To support multiple SLOs users can also specify required parameters for the development of their resource scaling policies. Users can customize the management of their applications by implementing generic callbacks. Such callbacks are provided to monitor the application, compute its new resource demand and reconfigure it when VMs are removed or added.

4.1.2 The Market Scheduler

The market scheduler performs all following functions related to VM provisioning: (i) it collects data regarding node and VM resource utilization; (ii) it deploys the VMs on the physical nodes; (iii) it enforces the VM resource allocations computed for the current time period; (iv) and it migrates VMs between the nodes.

4.1.3 The Credit Manager

The credit manager manages the virtual currency from the system. Each virtual platform has an account registered with the credit manager, which stores the user's defined currency amount. The controller account is charged by the market scheduler for all the application resources. To avoid situations in which applications run out of credits in the middle of their execution, a transfer is made from the user account to the controller account at a global predefined time interval.

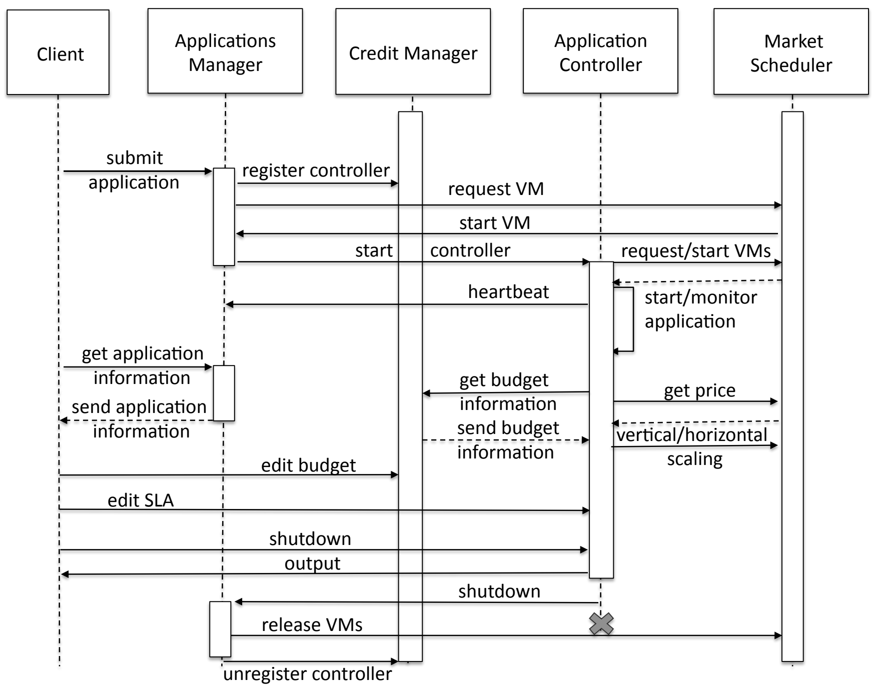


Figure 4: Typical application execution flow.

4.2 Application Life-cycle

Figure 4 illustrates how users and Merkat services interact in order to manage an application during its life-time.

4.2.1 Application Deployment

Users specify all needed information to deploy the application in an application description file and start its associated (application) controller. Figure 5 illustrates the contents of such a description file. This example defines a set of compute nodes used to run an MPI application, Code_Saturne [8]. Users can specify diverse information like budgeting, the type of application controller to load, commands to manage the application, the virtual cluster configuration and the application SLO policies.

An application controller can manage multiple virtual clusters with dependencies between them that establish the order in which they need to be started.[2] For example, some resource management frameworks (e.g. Condor [9], Torque [10]) require starting first the master node and then the worker nodes by communicating to them the master hostname and IP address. As another example, in the case of an MPI application, the MPI master process needs to know the IP addresses of the VMs to start application processes in them. A virtual cluster configuration includes information on commands to manage applica-

[2]Currently we support only a one-level dependency: a "slave" virtual cluster can depend on one "master" virtual cluster.

```
<platform>
<name>CodeSaturne</name>
<controller>MPIMal</controller>
<keepidle>600</keepidle>
<budget>80000</budget>
<renew>80000</renew>
<group name='saturne-0' poll='yes' controller='yes' process=‘mpirun’ depends='saturne-1'>
        ……
</group>
<group name='saturne-1' poll=’no' controller=’no’ depends='saturne-1'>
        ……
</group>
<policy>
        <type>deadline</type>
        <reference>10800</reference>
        <params>
        <treconf>180</treconf>
        <profile>/profiles/profile_saturne</profile>
        </params>
</policy>
<poll>60</poll>
</platform>
```

Figure 5: Application configuration example.

tion components, files, copied in/from the VMs before/after the application is started/stopped, initial user bids, and VM configurations.

As each virtual cluster can have different application components installed, the way the information is processed also differs. Thus, each virtual cluster can have its own monitor which parses the information from the running application components. Each monitor periodically sends afterwards this information to the application controller, which aggregates and processes the application performance value at a period specified in the configuration file.

To support multiple SLOs users can also specify in the configuration file any parameters needed in the development of their resource demand adaptation policies. In the example given in Figure 5 , the application controller uses a policy for an MPI application to scale the application resource demand to meet a given deadline. As additional parameters, the application reconfiguration time is given (i.e., the time required to restart the application with a different number of processors).

4.2.2 Application Bootstrap

To start their applications, users submit a virtual platform template to the applications manager, which puts it in an application queue. The requests are taken from the queue periodically (e.g., at a 10 seconds interval) and a start condition is checked. At this point the applications manager invokes an application controller start policy. If the virtual platform cannot start, e.g., the price is too high, it is put back in the queue. When the start condition is validated, the applications manager requests one initial VM from the market scheduler, copies the controller code in it and starts the controller. After the controller provisions the remaining VMs, it performs three steps: (i) it runs a health check on them, restarting any failed or timeout ones; (ii) it configures them, e.g., mounts a shared file-system, sets their hostnames; (iii) and start the application components by running commands specified in the virtual platform template.

4.2.3 Monitoring and Elastic Scaling

During the application runtime the controller receives performance metrics (e.g., number of tasks, progress) from monitors running in virtual clusters and retrieves the current price and budget. Based on this information it may scale horizontally and vertically the application resource demand. Heartbeats are sent periodically to the applications manager, to detect possible failures.

4.2.4 Shutdown

There are several cases when the application shuts down: (i) when it finishes its execution; (ii) when the user requests a shutdown; (iii) when there is not enough budget to keep the application running at the current price. In the first case the controller asks the applications manager to release all provisioned VMs while in the other cases, the controller also shuts down the application components.

4.3 Extending Merkat for User-specific Needs

There are two ways in which users can automate the deployment and management of their applications: (i) by using predefined controllers and writing plugins for application-specific operations (e.g., start, suspend, resume, monitor, stop) and specifying them in the XML description file; (ii) by writing custom application controllers based on the APIs provided by Merkat services, possibly reusing existing controller code. The second way is necessary if policies for more complex applications are to be developed. For example, custom controllers might include prediction, profiling and learning modules to determine the allocation that the application needs for its execution and use this information in changing the number of provisioned VMs.

Algorithm 4 illustrates the main application controller loop, highlighting the main callbacks that a user should implement for her own specific policies. For example, if a user wants to execute a fluid dynamics simulation (e.g., the flow of water in the reactor of a nuclear plant) and stop it as soon as it reaches a stable state (i.e., the execution time per simulation time step drops considerably), she needs to implement the *ProcessMonitorMessage* callback and add a simple comparison of the read performance value with a threshold. If the performance value drops below the threshold it instructs the applications manager to shutdown the application.

5 Evaluation

In this section we present an experimental evaluation of Merkat. We deployed Merkat on the Grid'5000 testbed and tested it with several scientific applications. The goal of this evaluation is to show that: (i) Merkat is flexible by allowing applications to scale their resource demand vertically and horizontally;

Algorithm 4 Application controller main loop

```
 1: vmgroups = start application virtual clusters
 2: if application needs resume then
 3:     ResumeApplication()
 4: for vmgroup in vmgroups do
 5:     if vmgroup['poll'] == 'yes' then
 6:         StartMonitor(vmgroup)
 7: while not stopped do
 8:     message = get message from monitor queue
 9:     ProcessMonitorMessage(message)
10:     if application needs reconfiguration then
11:         ReconfigureApplication()
12:     if NeedsSuspend() then
13:         SuspendApplication()    // if any callback is implemented
14:         send suspend message to applications manager
15: transfer files from virtual clusters
16: send shutdown message to applications manager
```

(ii) and Merkat provides better user satisfaction than traditional resource management systems, by differentiating fairly between users. We describe next our prototype, the testbed and the applications used to validate it. Finally, we discuss the results of our experiments.

5.1 Implementation

We implemented Merkat in Python. Merkat depends on the Twisted [11], paramiko [12] and ZeroMq [13] libraries. The Twisted framework is used to develop the XML-RPC services. Paramiko is used for the VM connection: the application management system needs to test and configure the VMs in which the application runs and it does so through SSH. The VM SSH connections are done using a thread pool with a configurable size. ZeroMq is used for the internal communication between various components of Merkat (e.g., the applications manager and application controllers, or the application controllers and virtual cluster monitors). As an IaaS Cloud Manager we use OpenNebula [14]. Merkat's services communicate with OpenNebula through its XML-RPC API. We also replace OpenNebula's default scheduler with our market scheduler. As a hypervisor we use KVM and the market scheduler controls the resource allocations through *cgroups*. For better application performance, the VM VCPUs are pinned to the physical cores. Each Merkat service stores its persistent information in a database storage. Such information includes the controller identifiers, budget and virtual cluster information (the VM bid and provisioned VMs). Each service has its own database stored on a MySql server.

5.2 Experimental Setup

Merkat was evaluated on a cloud deployed on the Grid'5000 testbed [6]. The cloud is composed of multiple compute nodes, one node dedicated to the OpenNebula frontend and the Merkat services and another node configured as an NFS server for storing the VM images. All the nodes have a Gigabit Ethernet interface. Each VM is configured with a Debian Squeeze 6.0.1 OS. To speed-up the VM deployment we use copy-on-write VM images. The cloud configuration for each experiment is described in Table 1. In all experiments, our market scheduler uses a scheduling period of 40 seconds. This period is chosen to allow controllers to adapt fast and in the same time to allow VM operations, e.g., migration and boot, to finish before the start of the next scheduling period.

Experiment	Compute nodes	VM CPU	VM Memory	Max VM capacity
Vertical scaling illustration	1	4 cores	900 MB	2
Horizontal scaling illustration	20	1 core	900 MB	160
User satisfaction	4	1 core	900 MB	32

Table 1: Infrastructure characteristics.

5.3 Evaluated Applications

Merkat currently supports the automatic execution and scaling of two types of applications: static and malleable. These are application types executed on the EDF's infrastructure. For each type we implemented a controller to scale the resource demand according to a user objective. For static applications the objective is to execute the application before a given deadline while for malleable applications the objective is to finish the processing performed by the application as fast as possible.

5.3.1 Static applications

To illustrate the support for static applications, we test Merkat by running multiple instances of Zephyr [15]. Zephyr is a parallel application used for fluid dynamic simulations. Zephyr receives as input a configuration file and simulates the move of fluid over a specified time. The computation is iterative: at each iteration a set of equations is solved, e.g., the Navier–Stokes equations. Zephyr writes in a log file the CPU time for each iteration, i.e., the iteration execution time, together with the total elapsed time, i.e., the time from the execution start. To run Zephyr we designed an application controller for MPI applications. This application controller can run Zephyr in two modes: best-effort and SLO-driven. In best-effort mode, the application controller starts the application when the price is low enough and then does not react at all to changes in the application's performance or infrastructure price. In SLO-driven, i.e., deadline-driven, mode, the application controller uses the vertical scaling algorithm, described in Section **??**, to scale the VM bid using as a performance metric the application iteration execution time and as a reference metric the remaining time to deadline distributed over the remaining number of iterations. The performance metrics are sent to the application controller by a monitor running in the same VM as the MPI main process that periodically reads Zephyr's log file.

5.3.2 Maleable applications

To illustrate the support for malleable applications, we implemented controllers that scale horizontally two resource management frameworks: Condor [16] and Torque [10]. These frameworks use a master-worker architecture: a master receives tasks from users and it distributes them to the running workers. The frameworks can scale their resource demand by increasing or decreasing their number of workers. Users submit their applications to these frameworks and Merkat scales their resource demand proportional to their budget and their current workload. The framework performance metrics are retrieved through commands to get the number of running and queued tasks, i.e., *qstat* for Torque and *condor_q* for Condor. The controller uses the horizontal scaling algorithm described in Section **??** to keep the total wait time of Torque tasks or the number of queued Condor tasks is below a threshold. To cope with bursts in workloads, VMs are kept idle for a specified time period, e.g., 10 minutes.

5.4 Results

We show the capability of Merkat to scale vertically and horizontally the application resource demand with the goal to keep the application's SLO in a dynamic environment. To show how differentiating between the different values users might have for their applications leads to a better user satisfaction than traditional schedulers, we quantify and measure the total user satisfaction obtained with our system and a traditional scheduler.

5.4.1 Vertical Scaling Illustration

To show how applications can scale vertically their resource demand we ran a micro-benchmark using Zephyr applications: we started an SLO-driven application and four best-effort applications, each in one VM. The SLO-driven application was started at the begining of the experiment, while the other four applications were started during its execution. The SLO-driven application has an ideal execution time of 77.5 minutes. To test the limitation of the system, we ran the application with three different deadlines: 200, 150 and 100 minutes. We repeated each experiment three times and computed the average of the obtained values.

Figure 6 describes the SLO-driven controller behavior from the start of the experiment and until the application deadline.

Figure 6(a) shows the bid variation for the different deadlines. The bid stabilizes after all the applications were submitted. The application with the smallest deadline demands the maximum requested allocation for the VM, and thus, the submitted bid is also much higher than in the other two cases.

Figure 6(b) shows the resource utilization, i.e., how much the application inside the VM consumes, for the different VMs. After all the best-effort applications started running, the SLO-driven controller keeps a reduced resource allocation, depending on the application deadline. In the case of the 100-deadline application, the application controller keeps maximum allocation.

Figure 6(c) shows the iteration execution time variation. We notice that after all the best-effort applications started running, the application controller manages fairly well to keep the iteration execution time close to the reference execution time. However, there are cases in which the iteration execution time oscillates. We think that one cause for this variation is the performance intereference from the other VM processes. A case in which the deadline cannot be met by our system is the 6000-deadline application. In this case, obtaining the maximum allocation does not help the application to meet its deadline, due to performance interference of other applications.

This experiment shows that Merkat allows applications to use just enough resources to meet their SLOs. One issue with this adaptation policy is the performance interference from the incoming workloads. For this case, the user might require VMs with a capacity equal to that of a node. Then, the application controller could adjust the bid at a high enough value, leading to the migration of the additional workload to other nodes.

5.4.2 Horizontal Scaling Illustration

To show how Merkat can be used by malleable applications too, we tested it by allocating the infrastructure between a Condor and a Torque framework. We deployed the Condor framework to process parameter sweep applications and the Torque framework to process MPI applications. We assume that each framework is owned by an infrastructure administrator, who wants to share the infrastructure equally among them. This administrator assigns an equal budget to each framework, which allows it to provision as many VMs as to fill the infrastructure at a reserve price, i.e., 160 VMs. When both frameworks have a maximum resource demand, they get half of the infrastructure. We submitted 33 Zephyr applications to Torque with execution parameters taken from a trace generated using a Lublin model [17] and a number of processors between 1 and 8. We submitted 5000 jobs to Condor, simulating a parameter sweep application. As we did not have access to a real parameter sweep application, we used the *stress* benchmark, which ran a CPU intensive worker for different time intervals, generated with a Gaussian distribution.

Figure 7 shows the number of running and queued jobs in the Torque/Condor's queue and the number of VMs provisioned over time. It can be noticed how both application controllers adapt the framework's resource demand to the variations in the number of queued jobs. If each framework would be assigned an equal share of the infrastructure, it would be able to provision a maximum of 80 VMs. However, in our case, the Condor framework is capable to take advantage of the under-utilization periods of the infrastructure (the first and the last phase of the experiment).

We noticed a significant delay in provisioning VMs, as seen in Figure 7. During the experiment runtime, we had issues with many VMs which were booting slow, or the sshd daemon failed to start, making the SSH connection to them to fail, after a timeout period.

5.4.3 User Satisfaction

To measure the satisfaction our system can provide to users, we compare Merkat with a traditional batch scheduler, i.e., Maui [18]. For this, we considered the following user behavior. First, to reflect the user's valuation for her application, users assign budgets to their applications inversely proportional to their deadlines. Second, users value their application execution at the assigned budget if the application finishes before deadline and at the negative budget value if the application doesn't meet its deadline. The satisfaction provided by the system is the sum of the user valuations.

We run 30 Zephyr applications, with 1 to 8 processes, each process in one VM, in two situations: (i) when the cloud is managed by Merkat; (ii) and then when it is managed by Torque and Maui. In the later case, we deploy

VMs on each node and add them to Torque as physical nodes. As using a real workload trace would have lead to longer experiment execution times, which would not have been possible on Grid'5000, we used a Lublin [17] model to generate the workload. We chose this model as it is realistic and easy to tune. In Merkat each application ran with a SLO-driven controller while in Maui applications were submitted to the scheduler's queue. Maui doesn't consider the user's value for resources and uses First-Come-First-Serve and backfilling to schedule applications to nodes.

We measured the total satisfaction and we obtained that Merkat provides 2.79 times more value to users than Maui. This result is a promising step in proving the usefulness of a market-based autonomous system like Merkat.

6 Related Work

In this section we describe the solutions related to Merkat. We focus first on resource management solutions that share resources among applications dynamically. Then, we detail solutions which use virtualization and rely on the cloud computing paradigm to manage applications. Finally, we describe notable solutions in the area of market-based distributed resource management.

6.1 Dynamic bare-metal resource management

Usually, infrastructures are managed through batch schedulers. Batch schedulers are centralized resource management systems, which employ a variety of scheduling algorithms derived from First-Come-First-Served (e.g., EASY [19] or Conservative Backfilling [19]). Their provided interface is generic enough to be used by a variety of applications. Currently, batch schedulers like SLURM, Torque and Moab, provide support for dynamic applications by shrinking and expanding the resources allocated during their execution. SLURM provides partial support as the change is initiated by the user and, to expand its resource allocation, the user needs to submit a new "job" and to merge the acquired resources with the original "job". In Torque and Moab the application provides a script called afterwards by the batch scheduler during the application execution.

Other solutions, like CooRM [20, 21], ReSHAPE [22] or Omega [23] share the infrastructure dynamically among applications. While CooRM and Omega employ a decentralized resource model, in which applications can decide how many resources they should receive, ReSHAPE computes the resource allocations in a centralized way, by knowing the performance models of all running applications. All these solutions don't use virtualization, leading to the difficulty of administrating the infrastructure.

6.2 Virtualized resource management

Some of the Merkat's principles were introduced in Themis, a system that simulated application adaptation policies on top of the proportional-share market [24]. However, Themis addressed only CPU allocation. This paper improves the proposed solution with the following contributions: (i) new VM load balancing and resource demand algorithms for multiple resources (CPU and memory); (ii) an implemented and evaluated system.

Earlier solutions focus on sharing the cluster or grid resources [25, 26, 27] between frameworks (e.g., Torque, SGE) using virtualization to support more application types on the infrastructure. More recent solutions, like Mesos [28] gives resource offers, i.e., lists of available resources on nodes, to frameworks while frameworks can filter the offers and decide what resources to accept. However, these solutions don't consider individual application performance or SLO.

6.3 PaaS

There are many PaaS systems, both commercial [1, 2, 3] and research [4]. These systems provide runtime support for applications hiding from users the complexities of managing resources. However, these systems are typically closed environments, forcing users to run specific applications (e.g., web, MapReduce), while they don't differentiate between the user valuations when contention appears. Opposed to these solutions, Merkat allows users to run new application types while distributing the resources among them based on their value.

6.4 Market-based resource management

Market-based resource allocation has been well explored by previous works in the context of grids and shared clusters. Multiple attempts were made to use auctions to schedule static MPI applications, on clusters [29, 30] or to run bag-of-tasks applications on grids [31]. In this case, users bid to execute their applications and the scheduler decides which application gets to run by applying an auction. The output of the auction is an increased overall user satisfaction. This comes from the fact that the bids assigned by the users reflect the valuation the application execution has for them. An auction, which assigns resources to the highest bidders, ensures that the most valuable applications are running on the infrastructure. Popcorn [32] and Spawn [33] were designed for applications composed of many tasks which can shrink when the resource price is high and expand when the resource price is low. Here, the application is assigned a budget from which it funds each task to run on a node. Each node has a resource manager that applies an auction to decide which task to run. Systems like Tycoon [34] or REXEC [35] implement a proportional-share policy per node to allocate fractional amounts of CPU. A dynamic priority scheduler is proposed for Hadoop [36], to allocate map/reduce slots to jobs. However, these systems do not provide support for different SLOs, as, more specifically, they do not monitor and adapt the application resource demand on the market. Opposed to these systems, Merkat controllers can adapt applications in two different ways to meet user SLOs.

7 Conclusion

In this paper we presented the implementation of Merkat, a market-based platform for application and resource management in private clouds. Merkat implements a proportional-share market to allocate fine-grained CPU and memory amounts to VMs, thus ensuring a maximum resource utilization. In the same time, Merkat provides incentives to users to use just as many resources as needed, thus ensuring fair resource utilization. To ease the user's task of executing different application types, Merkat offers tools for application management and automatic vertical and horizontal scaling. We deployed Merkat on Grid'5000 and tested it with real applications. We showed how Merkat can enable the cohabitation of different resource usage policies on the infrastructure, providing better user satisfaction than traditional systems.

As future steps, we plan to perform larger scale experiments and analyze the impact of VM operations and Merkat's scalability. Also, Merkat can be improved in several ways. First, mechanisms can be integrated to ensure that services remain available despite failures. Second, better application performance guarantees can be provided through leveraging price prediction algorithms [37]. Finally, other, more cooperative, market-based allocation policies remain to be investigated.

References

[1] AmazonEBS, "http://aws.amazon.com/."

[2] M. Azure, "http://www.windowsazure.com/."

[3] CloudFoundry, "http://www.cloudfoundry.com/."

[4] G. Pierre and C. Stratan, "Conpaas: a platform for hosting elastic cloud applications," *IEEE Internet Computing*, 2012.

[5] K. Lai, "Markets are dead, long live markets," *SIGecom Exch.*, vol. 5, pp. 1–10, July 2005.

[6] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, 2006.

[7] F. Glover, M. Laguna, *et al.*, *Tabu search*, vol. 22. Springer, 1997.

[8] CodeSaturne, "Codesaturne : a finite volume code for the computation of turbulent incompressible flows - industrial applications," *International Journal on Finite Volumes*, 2004.

[9] M. Litzkow, M. Livny, and M. Mutka, "Condor-a hunter of idle workstations," in *8th International Conference on Distributed Computing Systems*, 1988.

[10] G. Staples, "Torque resource manager," in *Proceedings of SC'06*, 2006.

[11] Twisted, "twistedmatrix.com/."

[12] paramiko, "www.lag.net/paramiko/."

[13] ZeroMq, "http://www.zeromq.org/."

[14] B. Sotomayor, R. Montero, I. Llorente, and I. Foster, "An Open Source Solution for Virtual Infrastructure Management in Private and Hybrid Clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, 2009.

[15] Zephyr, "https://github.com/kortas/zephyr."

[16] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience.," *Concurrency Practice and Experience*, vol. 17, no. 2-4, 2005.

[17] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: modeling the characteristics of rigid jobs," *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1105–1122, 2003.

[18] Maui, "http://www.nsc.liu.se/systems/retiredsystems/grendel/maui.html."

[19] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," in *Proceedings of Parallel Processing Workshops*, pp. 514–519, IEEE, 2002.

[20] C. Klein and C. Perez, "An rms architecture for efficiently supporting complex-moldable applications," in *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications*, HPCC '11, 2011.

[21] C. Klein and C. Pérez, "An rms for non-predictably evolving applications," in *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 326–334, IEEE, 2011.

[22] R. Sudarsan and C. J. Ribbens, "Reshape: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment," in *Proceedings of the 2007 International Conference on Parallel Processing*, pp. 44–44, IEEE, 2007.

[23] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European conference on Computer systems*, Eurosys'13, 2013.

[24] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas, "Themis: Economy-based automatic resource scaling for cloud systems," in *Proceedings of HPCC'12*, 2012.

[25] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle, "Dynamic virtual clusters in a grid site manager," in *Proceedings of HPDC'03*, pp. 90–100, IEEE, 2003.

[26] P. Ruth, P. McGachey, and D. Xu, "Viocluster: virtualization for dynamic computational domains," in *Proceedings of Cluster'05*, 2005.

[27] N. Kiyanclar, A. G. Koenig, and W. Yurcik, "Maestro-vc: On-demand secure cluster computing using virtualization," in *Proceedings of CCGrid Workshops*, 2006.

[28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of NSDI'11*, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2011.

[29] I. Stoica, H. Abdel-Wahab, and A. Pothen, "A microeconomic scheduler for parallel computers," in *JSSPP'95*, pp. 200–218, Springer, 1995.

[30] C. S. Yeo and R. Buyya, "Pricing for utility-driven resource management and allocation in clusters," *International Journal of High Performance Computing Applications*, vol. 21, no. 4, pp. 405–418, 2007.

[31] D. Abramson, R. Buyya, and J. Giddy, "A computational economy for grid computing and its implementation in the nimrod-g resource broker," *Future Generation Computer Systems*, vol. 18, no. 8, pp. 1061–1074, 2002.

[32] O. Regev and N. Nisan, "The popcorn market. online markets for computational resources," *Decision Support Systems*, vol. 28, no. 1, pp. 177–189, 2000.

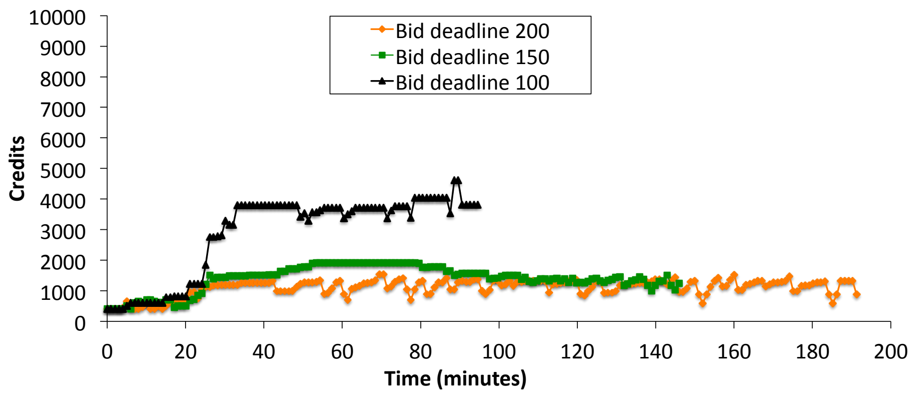
[33] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, "Spawn: A distributed computational economy," *IEEE Transactions on Software Engineering*, vol. 18, no. 2, pp. 103–117, 1992.

[34] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. Huberman, "Tycoon: An implementation of a distributed, market-based resource allocation system," *Multiagent and Grid Systems*, vol. 1, no. 3, pp. 169–182, 2005.

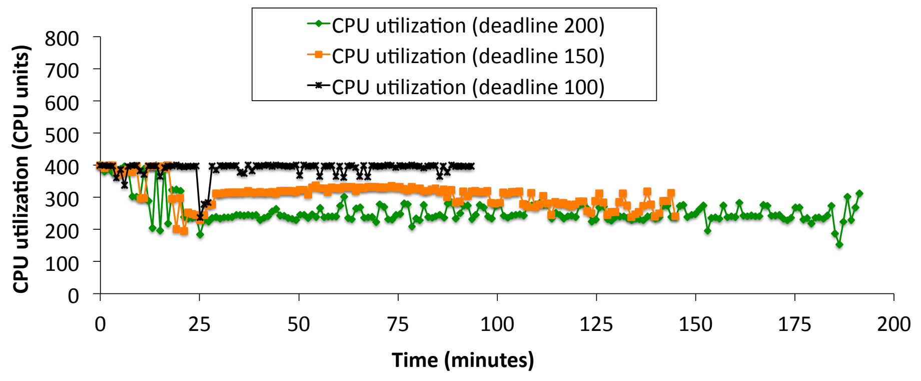
[35] B. N. Chun and D. E. Culler, "Rexec: A decentralized, secure remote execution environment for clusters," in *Proceedings of the CANPC'00*, (London, UK, UK), pp. 1–14, Springer-Verlag, 2000.

[36] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *JSSPP'10*, 2010.

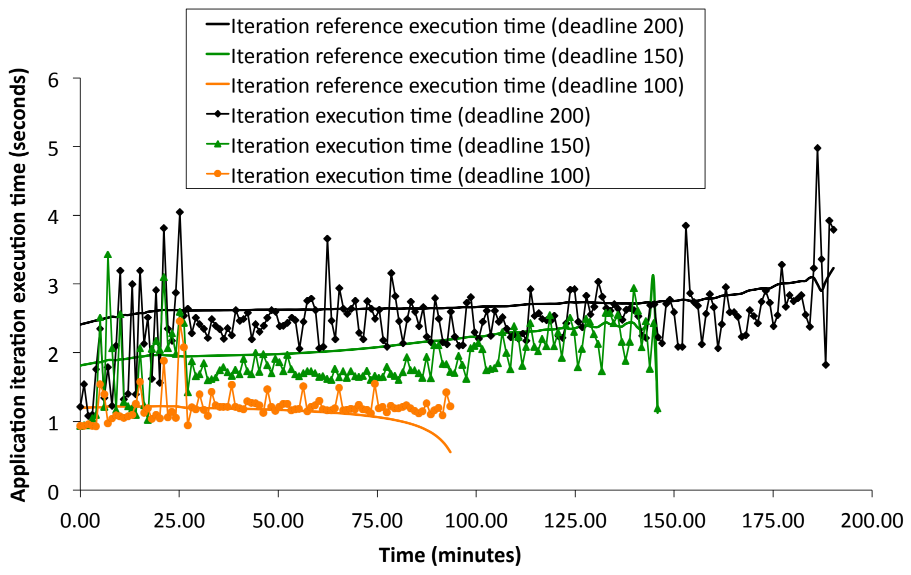
[37] T. Sandholm and K. Lai, "A statistical approach to risk mitigation in computational markets," in *Proceedings of HPDC'07*, vol. 25, pp. 85–96, Citeseer, 2007.



(a)



(b)



(c)

Figure 6: Controller adaptation for different deadlines: (a) bid adaptation, (b) the resulting CPU utilization and (c) execution time variation.

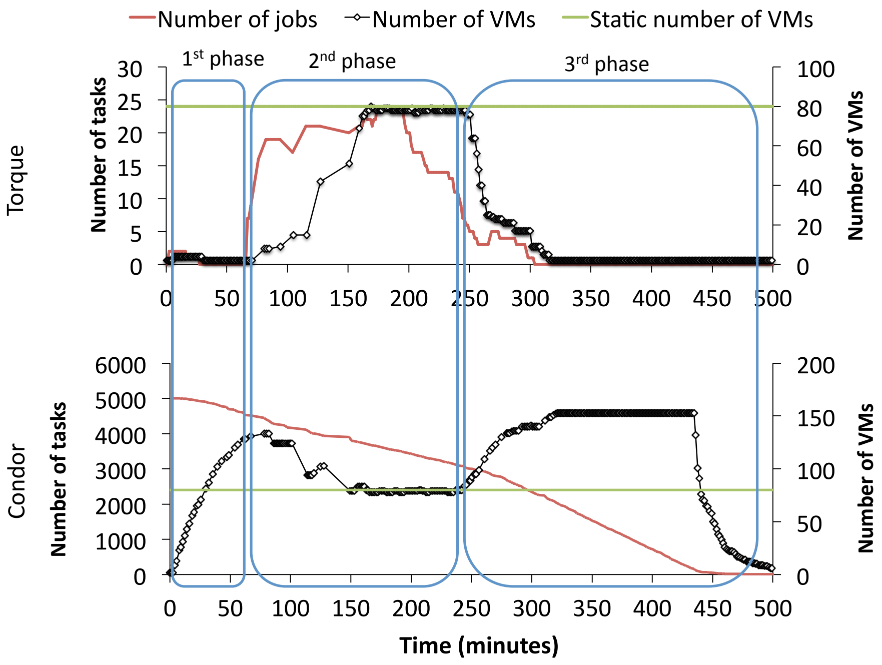


Figure 7: Provisioned VMs versus queued tasks for both frameworks.

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399