



Market-based Autonomous Application and Resource Management in the Cloud

Stefania Costache, Samuel Kortas, Christine Morin, Nikos Parlavantzas

► To cite this version:

Stefania Costache, Samuel Kortas, Christine Morin, Nikos Parlavantzas. Market-based Autonomous Application and Resource Management in the Cloud. [Research Report] RR-8648, Inria. 2014, pp.35. hal-01091280v1

HAL Id: hal-01091280

<https://inria.hal.science/hal-01091280v1>

Submitted on 8 Dec 2014 (v1), last revised 8 Sep 2016 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Market-based Autonomous Application and Resource Management in the Cloud

Stefania Costache , Samuel Kortas , Christine Morin, Nikos
Parlavantzas

**RESEARCH
REPORT**

N° 8648

December 2014

Project-Team MYRIADS



Market-based Autonomous Application and Resource Management in the Cloud

Stefania Costache ^{*}, Samuel Kortas ^{*}, Christine Morin[†], Nikos Parlavantzas ^{‡ §}

Project-Team MYRIADS

Research Report n° 8648 — December 2014 — 32 pages

Abstract: Organizations owning High Performance Computing (HPC) infrastructures are facing difficulties in managing their resources. These difficulties come from the need to provide concurrent resource access to applications with different resource requirements while considering that users are selfish and might have different performance objectives, or Service Level Objectives (SLOs), when executing them. To address these challenges, this paper proposes Merkat, a market-based SLO-driven cloud platform. Merkat relies on a market-based model to allocate resources to applications while taking advantage of virtualization technologies and on-demand provisioning to maximize resource utilization. Merkat's resource market uses a combination of currency distribution and dynamic resource pricing to ensure proper resource distribution while decentralizing the resource control. In Merkat autonomous controllers apply adaptation policies to scale the application's resource demand according to user's SLO. 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 in simulation and on the Grid'5000 testbed. Results show that: (i) Merkat provides flexible support for different application types and different SLOs; (ii) it increases the resource utilization of the infrastructure; (iii) and is capable of providing good user satisfaction compared to existing centralized systems.

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

[§] The authors are in alphabetic order.

^{*} EDF R&D, Clamart, France

[†] INRIA Rennes-Bretagne Atlantique, Rennes, France

[‡] INSA, Rennes, France

**RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

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 concurrentement aux ressources tandis que les utilisateurs peuvent avoir des objectifs de performance (SLOs) variés. Pour résoudre ces difficultés, cet 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 sont régies par une économie de marché respectant au mieux des objectifs de niveau de service (SLO) tout en tirant parti de la flexibilité d'un nuage informatique et en maximisant l'utilisation 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 les objectifs (SLO) fixés par l'utilisateur. Les politiques d'adaptation peuvent : (i) adapter dynamiquement leur demande en terme de CPU et de mémoire pour les machines virtuelles en période de contention pour l'obtention de ressources (ii) et changer dynamiquement le nombre de machines virtuelles. Nous avons évalué cette plate-forme par simulation et sur l'infrastructure Grid'5000. Nos résultats ont montré que cette solution: (i) offre un support flexible aux applications de différents types ayant des demandes variés en terme de niveau de service; (ii) augmente l'utilisation des ressources de l'infrastructure; (iii) conduit à une meilleure satisfaction des utilisateurs par rapport aux solutions centralisées existantes.

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

1 Introduction

HPC infrastructures, composed of a large number of computers, are acquired and used by an increasing number of organizations. The efficient use of these infrastructures is a key challenge for many of these organizations, which strive to minimize the cost of maintaining the infrastructure while satisfying the constraints of applications running on it. Recently, to ease their infrastructure's management, some of these organizations choose to virtualize and transform them in "private clouds", managed by specialized frameworks [20, 38, 46, 32, 18]. The main difficulty which resource management frameworks need to address is allocating the resources needed for each application.

First, resource allocation has to be done with respect to the infrastructure's capacity limitations. Having enough capacity to meet all user requests in the highest demand periods is rarely the case, as expanding the resource pool is expensive. To differentiate between user requests in these periods, most of these frameworks [20, 38, 46, 18] rely on priority classes, i.e., users are given a priority class for their applications. However, in this case the users might abuse their rights and run less urgent applications with a high priority, taking resources from users who really need them.

Then, resource allocation has to be done by considering the application and user requirements. Some applications might have a resource demand that varies frequently, based on the load they need to process, e.g., web applications, while others can simply adapt their demand to the current resource availability, e.g., bags of tasks or MapReduce applications. At the same time, while some applications can achieve better performance by scaling horizontally, i.e., scaling the number of nodes, others might benefit from scaling their resource demand vertically, i.e., scaling the resource amount per node [4]. Disregarding these characteristics can lead to poor resource utilization and application performance. Finally, users might have different SLOs for their applications. Some users want the application results by a specific deadline, (e.g., a user needs to send her manager the output of a simulation by 7am the next day) or as soon as possible (e.g., a developer wants to test a newly developed algorithm). The cloud computing's "on-demand access" model to virtual resources is an attractive approach to deal efficiently with the variety of application resource models and user requirements. By provisioning virtual machines dynamically, advanced Platform-as-a-Service (PaaS) solutions [3, 6, 12, 34] can satisfy application performance requirements, but, while only some of them provide vertical and horizontal scaling, none of them properly address the case of contention periods.

In this paper we present the design and evaluation of a platform for application and resource management in private clouds, called Merkat. Merkat applies a unique approach to multiplex the limited infrastructure capacity between applications with different resource demand models while maximizing the infrastructure utilization and providing support to meet different SLOs. In Merkat each application runs in an environment composed of a set of virtual machines (VMs) customized to its needs. This environment, which we call a virtual platform, can autonomously adapt its resource demand to meet the user's SLO. The resource demand of the virtual platforms is regulated through the use of a proportional-share market: resources in terms of CPU and memory are allocated in a fine-grained manner to the provisioned VMs. The market's cur-

currency distribution policy ensures that users receive limited budgets and declare truthfully the execution of their applications. The combination of currency distribution and dynamic resource pricing ensures proper resource utilization in contention periods, by favoring users who need resources urgently. On top of the implemented market, the virtual platforms use two policies to adapt the application's resource demand to the current resource availability and price: (i) a vertical scaling policy that adapts the application resource demand per VM; (ii) a horizontal scaling policy that adapts the application resource demand in terms of number of VMs. These policies can also be combined for better application execution. Through the use of the market, each virtual platform can take resource demand adaptation decisions independently from the others. Thus, the platform becomes flexible and can support multiple application models and SLOs.

We evaluated Merkat in simulation and on a real testbed. We implemented the proportional-share market in CloudSim [8]. We evaluated the performance of the proportional-share market in terms of total user satisfaction when applications adapt their resource demands per VM to track a user-given SLO [14]. We also tested Merkat on the Grid'5000 [7] testbed with two application types: static MPI applications and malleable task processing frameworks [13]. Our results show that: (i) Merkat is flexible, allowing the co-habitation of different applications and policies on the infrastructure; (ii) Merkat increases the infrastructure resource utilization, through vertical and horizontal scaling of applications; (iii) Merkat has low performance degradation compared to a centralized system that supports a fixed SLO type; this degradation is caused by the decentralized nature and the application selfish behavior.

The remaining of this paper is organized as follows. Section 2 describes our approach. It introduces the principles behind it and the resource management process. Section 3 presents evaluation results and Section 4 discusses limitations and future directions of improvement. Section 5 describes the related work. Finally, Section 6 concludes the paper.

2 Merkat: A Market-based SLO-driven Cloud Platform

In this section we give an overview of Merkat. We describe its design principles, its architecture and the life-cycle of an application.

2.1 Overview

Merkat has been designed to manage the clusters owned by an organization running scientific workloads. We assume that the organization cannot use public cloud resources, due to security constraints, e.g., the data that needs to be processed is too sensitive.

Figure 1 gives an overview of Merkat. Merkat relies on: (i) a virtual machine market to allocate resources; (ii) and a set of virtual platforms to manage and elastically scale applications.

To run applications on the infrastructure, users are assigned budgets by an administrator in the form of a virtual currency. We call the currency unit a

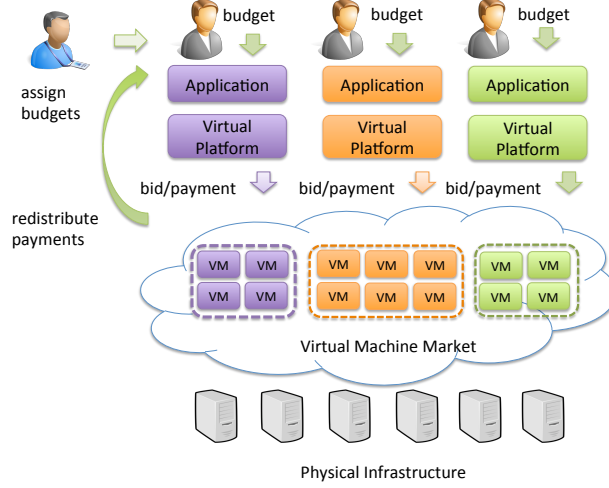


Figure 1: Overview of the Merkat system.

credit. Virtual currency is desirable when managing a private infrastructure; because no external currency is introduced, price inflation is bounded. In Merkat, the currency is re-chargeable, meaning that the charges incurred by the market return to user accounts after a while.

Users distribute amounts of credits, called budgets, to the virtual platforms in which their applications run, reflecting the maximum cost the user is willing to support, or the true priority, for running an application. Users have two options: (i) set these budgets to be renewed automatically; (ii) or renew them manually. Automatic budget renewal policies are used to minimize the risk of depleting the application’s budget in the middle of the application’s execution.

Virtual platforms acquire VMs from the virtual machine market by submitting payments for their resources. These payments, called bids, reflect the amount of resources the VMs receive, and they can be scaled based on user’s SLO and current budget.

The total amount of currency from the market is controlled by an infrastructure administrator, who can decide to adjust it based on price inflation and number of users. The infrastructure administrator sets up a total budget that is distributed among users and assigns to each user a weight based on which the user receives an amount from the total budget, i.e., proportional to the weight.

2.2 The Virtual Machine Market

We describe the mechanisms used to implement the virtual machine market.

2.2.1 Overview

To allow users to provision VMs with fine-grain resource allocations, e.g., 1 CPU core or 10% time on a CPU core, Merkat uses a proportional-share policy. Originally, this policy was used by the operating system schedulers to allocate CPU time to tasks proportional to a given weight, and inversely proportional

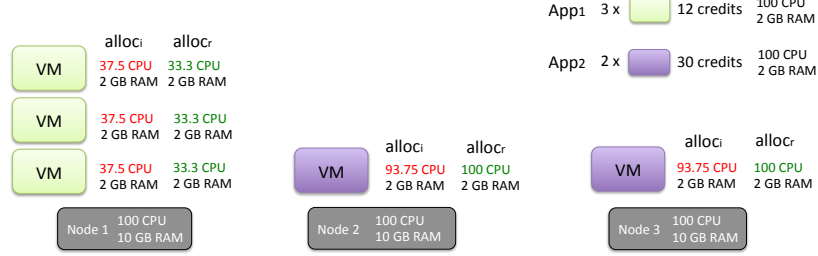


Figure 2: An example of resource fragmentation. 2 applications requesting 3 and respectively 2 VMs run on 3 nodes. If resources are allocated with the proportional-share policy from the entire infrastructure capacity, the CPU allocations for the first application’s VMs cannot be guaranteed. If resources are allocated with the proportional-share policy from the node capacity, allocations can be guaranteed and all available resources are distributed.

to the sum of all the other concurrent task weights [48]. Merkat uses a modified version of this policy to allocate resources to VMs based on their bids, representing the willingness to pay for their resources. In this modified policy, users, or the virtual platforms managing the VMs, bid for the resources of each provisioned VM, i.e., CPU and memory, and their VMs are allocated an amount of resource proportional to the bid and inversely proportional to the sum of other concurrent bids [24]. Users can change the value of their bid during the VM runtime and influence the resource allocation the VM receives.

This policy is advantageous for applications that can scale vertically, while applications that can scale horizontally can also be easily supported. Moreover, this policy is easy to understand and use as it avoids starvation and involuntary VM shutdown or preemption. Because each VM receives a resource amount, even in high price periods, policies can be designed to allow applications to decide whether to adapt to these small allocations or voluntarily shutdown some of their components.

In Merkat, the implementation of the proportional-share policy follows four steps: (i) VM bid submission; (ii) VM allocation computation; (iii) VM placement; (iv) price computation.

2.2.2 VM Bid Submission

To provision VMs, a bid, in the form of a vector $b = \langle b_{cpu}, b_{memory} \rangle$ needs to be submitted for their resources, CPU and memory, to a central entity, called VM Scheduler. This bid can be submitted directly by the user or the virtual platform that manages the VMs. The initial bid can be computed based on past price history and the user’s current budget. In our implementation we compute the VM bid based on the current resource price. The bid submitted for a VM is persistent: the user can specify it when the VM is started and the VM Scheduler will consider this value in its allocation decisions during the VM runtime. Nevertheless, the value of the bid can be further changed to cope with price fluctuations.

2.2.3 VM Resource Allocation

The VM Scheduler periodically computes resource allocations for the VMs for which a bid was submitted, by considering the value of their bid and a resource utilization cap, a_{max} , i.e., the maximum resource utilization of a VM during its lifetime. The VM Scheduler uses the a_{max} values to distribute free resources to other VMs needing them. We consider the user can estimate a_{max} for her VMs.

The resource allocation computation is performed in two steps: (i) to ensure maximum utilization, the VM Scheduler computes the VM allocation considering the entire infrastructure as a single physical node; (ii) to cope with the resource fragmentation, i.e., the infrastructure capacity is divided among nodes, the VM Scheduler corrects the allocations to fit to the node capacity.

When considering the entire infrastructure as a single physical node, the VM Scheduler computes the resource allocations as follows. For each resource, given a set of n resource bids $b_i(t)$, with $i \in \{1..n\}$, for a time interval t and a capacity of C units, the resource allocation for each bid $b_i(t)$ is equal to:

$$a_i(t) = \frac{b_i(t)}{\sum_i^n b_i(t)} \cdot C, \quad (1)$$

However, as the infrastructure capacity is partitioned between nodes, there are situations when resulted allocations cannot be enforced. This issue is illustrated in Figure 2. We consider 3 nodes with a capacity of 100 CPU units and 10 GB of RAM each. We also consider 5 VMs with a maximum resource utilization of 100 CPU units and 2 GB RAM each. The first 3 VMs receive a bid of 12 credits, and the last 2 VMs receive a bid of 30 credits. In this example, only the CPU resource is a bottleneck and each VM will receive the maximum amount of requested RAM. Using Equation 1, the VM Scheduler computes an allocation $alloc_{CPU}^j$ (depicted with red in Figure 2), $j \in 1..5$, as follows: 37.5 CPU units for the first 3 VMs and 93.75 CPU units for the last two VMs, where 1 CPU unit represents 1% of CPU time. Practically, these allocations cannot be enforced.

To solve this issue, the VM Scheduler corrects the allocations by recomputing them after placing the VMs on the nodes and using the capacity of the node in Equation 1. In the previous example, the resulted allocation $alloc_{CPU}^j$ (depicted with green in Figure 2), $j \in 1..5$, is: 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* and is defined as follows:

$$e_r = \frac{alloc^i - alloc^r}{alloc^i}, \quad (2)$$

As seen in the next section, this allocation error is used in placing the VMs on nodes; by minimizing the allocation error, the resource utilization is maximized.

2.2.4 VM Load Balancing

To minimize the VM allocation error, the VM Scheduler might migrate VMs between nodes. The process of migrating VMs among nodes is called load balancing. As having a high number of migrations leads to a performance degradation for the applications running in the VMs, the load balancing process tries to make a trade-off between the number of performed migrations and the VM

allocation error. For example, it won't make sense to migrate a VM when its allocation error is 1%. To select the VMs to be migrated at each scheduling period, the VM scheduler relies on an algorithm based on a tabu-search heuristic [17]. 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.

Algorithm 1 details the load balancing process. The algorithm receives the list of current nodes, *nodes*, the list of running VMs, *vms*, the list of VMs to be started at the current scheduling period, *newvms*, and three thresholds: (i) maximum number of iterations performed by the algorithm to obtain a better placement than the current one, N_{iter} ; (ii) maximum allocation error supported for the VMs in their current placement, E_{max} ; (iii) maximum number of migrations required to reach a better placement, M_{max} . Based on this information the algorithm computes the new placement of VMs on nodes and outputs a migration plan, composed of the VMs to be migrated, and a deployment plan, composed of the VMs to be started.

The algorithm starts from the current VM placement and tries to minimize the VM allocation error while keeping the number of VM migrations within the given limit M_{max} . If new VMs need to be created, they are placed on the least loaded nodes (Lines 6-8) before the VM placement is improved. Then, the VM placement is incrementally improved by placing the VM with the highest allocation error among the CPU and memory resources to the node that minimizes it (Lines 15-20). Note that, as each VM has two allocated resources, the maximum allocation error is the maximum among the computed error for each resource (Line 15). The VM allocation error computation is performed by a method called *ComputeErrors*. To avoid being stuck in a suboptimal solution, the algorithm uses a list that memorizes the last changes (Line 21).

The N_{iter} threshold is used to ensure the algorithm finishes: if there is no improvement in the last N_{iter} iterations, the algorithm stops.

2.2.5 Price Computation

In our approach we compute the resource price as the sum of all bids divided by the total infrastructure capacity. If this price is smaller than a predefined price, i.e., reserve price, then the reserve price is used. Users are charged based on their allocated resource amounts, which translates in the value of the VM bids.

2.3 Virtual Platforms

When running the application on the Merkat's virtual machine market, it is not sufficient just to acquire resources at the beginning of its execution. As the resource price and allocation are dynamic, guaranteeing the right amount of resources for the application is challenging for the user. If the price increases, the current user budget might not be enough to cover the cost of the acquired resources. If the price decreases, more resources could be provisioned, or the user could spend less for the already acquired resources. These issues are solved by running each application in a virtual platform, that adapts autonomously to the resource price and user requirements.

Algorithm 1 VM load balancing algorithm.

```

1: ComputePlacement ( $nodes, vms, newvms, N_{iter}, E_{max}, M_{max}$ )
2:  $migrationPlan \leftarrow \emptyset$  // migrations to be performed
3:  $deploymentPlan \leftarrow \emptyset$  // deployments to be performed
4:  $nIterations \leftarrow 0$  // number of iterations until an improvement
5:
6: for  $vm \in newvms$  do
7:    $node \leftarrow$  least loaded node from  $nodes$ 
8:    $node.vms \leftarrow node.vms \cup \{vm\}$ 
9:    $solution_{old} \leftarrow nodes$  // current placement of VMs
10:   $solution_{best} \leftarrow nodes$  // new placement of VMs
11:   $tabu\_list \leftarrow \emptyset$  // list of forbidden moves
12:   $e_{worse} \leftarrow \inf$ 
13:   $e = \text{ComputeErrors}(vms, nodes)$ 
14:  while  $nIterations < N_{iter}$  and  $e_{worse} > E_{max}$  do
15:     $(vm, e_{max}) \leftarrow vm$  with  $e_{max} = \max_{1 \leq i \leq n} \max\{e_{ic}, e_{im}\}, vm \notin tabu\_list$ 
16:     $source \leftarrow vm.node$ 
17:     $destination \leftarrow$  node which minimizes  $e_{max}$ ,  $node \notin tabu\_list$ 
18:     $vm.node \leftarrow destination$ 
19:     $source.vms \leftarrow source.vms - \{vm\}$ 
20:     $destination.vms \leftarrow destination.vms \cup \{vm\}$ 
21:     $tabu\_list \leftarrow tabu\_list \cup \{(vm, source)\}$ 
22:     $e = \text{ComputeErrors}(vms, nodes)$ 
23:     $e'_{max} \leftarrow \max_{1 \leq i \leq n} \max\{e_{ic}, e_{im}\}$ 
24:     $nMigrations =$  count number of migrations required to reach the new placement
25:    if  $e_{worse} - e'_{max} > 0$  and  $nMigrations < M_{max}$  then
26:       $solution_{best} = nodes$  // Keep the best solution so far
27:       $e_{worse} \leftarrow e'_{max}$ 
28:       $nIterations \leftarrow 0$ 
29:    else
30:       $nIterations \leftarrow nIterations + 1$ 
31:  for  $node \in solution_{old}$  do
32:    for  $vm \in node.vms$  do
33:      if  $vm.node \neq node$  and  $vm \notin newvms$  then
34:         $migrationPlan \leftarrow migrationPlan \cup (vm, vm.node)$ 
35:  for  $vm \in newvms$  do
36:     $deploymentPlan \leftarrow deploymentPlan \cup (vm, vm.node)$ 
37:
38: return ( $migrationPlan, deploymentPlan$ )

```

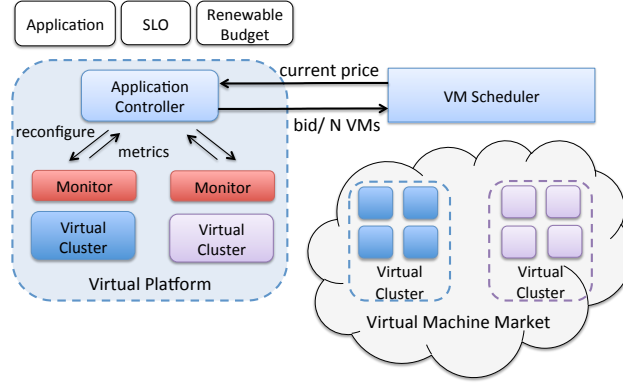


Figure 3: Virtual Platform overview.

2.3.1 Overview

Figure 3 illustrates the architecture of a virtual platform. A virtual platform is composed of one or more virtual clusters, based on how many components the application has, and an application controller, that manages them on behalf of the application. The application controller receives as input an application configuration file, containing the application description and adaptation policies. An application can have one or more components, requiring different software configurations. Thus, for each application component, the application controller deploys a virtual cluster, starts the application component in it and scales it according to user defined policies. We define a virtual cluster as a group of VMs that have the same resource configuration and use the same base VM image. Each virtual cluster has a monitor that sends application performance metrics to the application controller.

To meet the user's SLOs, the application controller periodically reads the application's performance metrics and based on them it adapts dynamically the virtual platform resource demand to the infrastructure resource prices and reconfigures the application. The user can interact with her application controller during its execution to modify her SLO, the budget for the application execution, or to retrieve statistics regarding the application.

In Merkat, virtual platforms use two policies to adapt the resource demand of an application: vertical and horizontal scaling. 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 SLOs.

Both policies address three cases:

- *Preserve budget when the SLO is met:* When the SLO can be met, the virtual platform reduces 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 requires more resources to meet its SLO, and its budget affords it, the virtual platform increases its resource demand.

Algorithm 2 Vertical scaling policy

```

1: VerticalAdaptation ( $bid, bid_{min}, last\_bid\_change, bid_{max}, alloc, alloc_{min},$ 
    $alloc_{max}, v, v_{ref}, v_{low}, v_{high}$ )
2:  $resources \leftarrow \{cpu, memory\}$ 
3:  $T \leftarrow \left\lfloor \frac{v_{ref} - v}{v} \right\rfloor$ 
4: for  $r \in resources$  do
5:   if ( $v < v_{low}$ ) and  $alloc[r] > alloc_{min}[r]$  then
6:      $bid_{tmp}[r] \leftarrow \max(bid[r] / \max(2, 1 + T), bid_{min})$ 
7:     if  $last\_bid\_change < 0$  then
8:        $\delta \leftarrow |bid[r] - bid_{tmp}[r]|$ 
9:       if  $\left| \frac{\delta - |last\_bid\_change|}{\delta} \right| < 0.1$  then
10:         $\delta \leftarrow \delta / 2$ 
11:         $bid_{tmp}[r] \leftarrow \max(bid_{tmp} - \delta, bid_{min})$ 
12:         $bid[r] \leftarrow bid_{tmp}[r]$ 
13:   if ( $v > v_{high}$  and  $alloc[r] < alloc_{max}[r]$ ) or ( $alloc[r] < alloc_{min}[r]$ ) then
14:      $bid_{tmp}[r] \leftarrow bid_{tmp}[r] \cdot \max(2, (1 + T))$ 
15:     if  $last\_bid\_change > 0$  then
16:        $\delta \leftarrow |bid[r] - bid_{tmp}[r]|$ 
17:       if  $\left| \frac{\delta - |last\_bid\_change|}{\delta} \right| < 0.1$  then
18:         $\delta \leftarrow \delta / 2$ 
19:         $bid_{tmp}[r] \leftarrow bid[r] + \delta$ 
20:         $bid[r] \leftarrow bid_{tmp}[r]$ 
21:   // bids are re-adjusted due to budget limitations
22:   if  $bid[memory] + bid[cpu] > bid_{max}$  then
23:      $w \leftarrow \emptyset$ 
24:     if  $alloc[r] \geq alloc_{max}[r], r \in resources$  then
25:        $w[r] \leftarrow 0$ 
26:        $bid_{max} \leftarrow bid_{max} - bid[r]$ 
27:     else
28:        $w[r] \leftarrow 1 - \frac{alloc[r]}{alloc_{max}[r]}, r \in resources$ 
29:     for  $r \in resources$  with  $w[r] \neq 0$  do
30:        $bid[r] \leftarrow \frac{bid_{max}}{\sum w[r]} \cdot w[r]$ 
31: return  $bid$ 

```

- *React when the SLO is not met due to budget limitations:* A last case that is considered is 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. In this case, we consider that the user has two options: (i) stop the execution of her application; (ii) or continue to run it with the same amount of resources.

The policies are run periodically and use two performance thresholds, upper and lower, as an alarm: when the application performance metric crosses the thresholds, the policy takes an action that changes the virtual platform resource demand.

2.3.2 Vertical Scaling Policy

The vertical scaling policy controls the amount of CPU and memory resources allocated to a VM as well the amount that is paid for them. This policy

computes periodically the resource bids for each VM based on the following information: (i) current, minimum and maximum VM resource allocation; (ii) current application performance metrics, v ; (iii) application reference performance, v_{ref} , upper and lower performance thresholds, v_{high} and v_{low} ; (iii) the current resource bids; (iv) the value of the last bid change, $last_bid_change$; (v) and the budget to be spent for the next time period, bid_{max} . Algorithm 2 describes this policy.

To preserve the user budget when the SLO can be met, the policy decreases the resource bids in two situations: (i) if the performance metric drops below the lower threshold (e.g., the remaining execution time of the application is less than 75% of remaining time to deadline); (ii) or if the allocation per VM for one resource reaches the maximum (Line 5).

To provide more resources when the SLO is not met, i.e., the performance metric is above the upper performance threshold, the policy increases the resource bids (Line 13).

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 (Lines 8-10 and 16-18).

If the current budget is not enough to meet the SLO, the policy decides between two options, based on the application and SLO type: (i) it suspends the application; (ii) it recomputes the bids to favor the resource with a small allocation. The first option is taken for batch applications, when the current price is too high to continue their execution, i.e., the VM allocation becomes less than $alloc_{min}$. In this case, the policy will resume the application when the price drops, e.g., VMs can receive 75% of their maximum allocation at the current budget. The second option is taken if the application cannot be suspended. In this case, when the price is too high increasing the bids with the amount T is not feasible due to the limited budget. Thus, the policy recomputes the bids as a part from the application budget that is proportional to the difference between the actual and maximum allocation of the VM (Lines 21-29).

2.3.3 Horizontal Scaling Policy

The horizontal scaling policy controls the number of VMs and their resource payments. 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, number of VMs and bids; (ii) the VM maximum allocation; (iii) application performance metric, upper and lower performance thresholds. Algorithm 3 details this policy.

The algorithm is composed of two steps: (i) first, a new number of VMs, $nvms$, is computed based on the application performance metric; (ii) then, this number is compared to and limited to a maximum number of VMs computed for the next time period so that all the VMs receive an amount of CPU and memory equal to their maximum resource utilization, a_{max} , at the current resource prices.

Algorithm 3 Horizontal scaling policy

```

1: HorizontalAdaptation ( $P, nvms_{old}, bid_{max}, alloc_{max}, v, v_{low}, v_{high}$ )
2: if  $v > v_{high}$  then
3:    $nvms \leftarrow nvms + 1$ 
4: if ( $v < v_{low}$  then
5:   pick vm to release
6:    $nvms \leftarrow nvms - 1$ 
7:  $(N, bid) \leftarrow \mathbf{GetUpperBound}(P, nvms, bid_{max}, alloc_{max})$ 
8: if  $N < nvms$  then
9:   release  $(nvms - N)$  VMs
10: else
11:   request new  $nvms - nvms_{old}$  VMs
12: return  $nvms$ 
13:
14: GetUpperBound( $P, nvms_{max}, bid_{max}, alloc_{max}$ )
15:  $resources \leftarrow \{cpu, memory\}$ 
16: find maximum value of  $N \in [1, nvms_{max}]$  for which
    
$$\sum_{r \in resources} \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}} < bid_{max}$$

17:  $nvms \leftarrow N$ 
18:  $bid[r] \leftarrow \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}}$ ,  $r \in resources$ 
19: return  $(nvms, bid)$ 

```

The first step, detailed in method *HorizontalAdaptation* (Lines 1-12), is used to preserve the budget when the SLO is met or to increase the resource demand when the SLO is not met. The second step, detailed in method *GetUpperBound* (Lines 14-19), is used to reduce the resource demand when the current budget is not enough to meet the SLO. For this last step, the bid value for each VM resource is easily computed 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 resource utilization 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.

2.3.4 Hybrid Scaling

When needed, the previous policies can be combined. For example, the application running under a deadline constraint, can first scale its resource demand vertically. Then, if the deadline is not met and the resource demand cannot be scaled vertically anymore, e.g., it reached the maximum resource utilization a_{max} of one VM, the algorithm switches to the horizontal scaling policy.

3 Validation

The goal of Merkat's validation is two-folded. First, we want to analyze the performance of Merkat in terms of total user satisfaction when applications adapt their resource demands to track a given SLO. Second, we want to show that Merkat is flexible enough to support different application types and SLOs. We have implemented a prototype of Merkat and validated it through simulation and on the Grid'5000 testbed [7]. Running a large workload gives a better

insight in the total satisfaction that the system can provide. However, due to time and resource limitations, running Merkat in a real environment for days, or possibly weeks, and with a large number of nodes, would have been unpractical. To achieve our validation goals, we have used real-world scenarios, which we encountered at EDF R&D (Electricité de France). EDF uses its own HPC clusters to run a variety of scientific applications. We discuss next the implementation of Merkat. Then we describe the results we obtained from measuring the total user satisfaction provided by Merkat in simulation and on a real-world testbed. Finally, we show that Merkat is flexible, allowing users to run different application types and SLOs.

3.1 Implementation

3.1.1 Simulation

To understand the behavior of our system we first implemented it in CloudSim [8], an event-driven simulator implemented in Java. In our case, we simulate the datacenter, the VM Scheduler and multiple applications, created dynamically during the simulation. Applications are created according to their submission times, taken from a workload trace, and are destroyed when they finish their execution. In our simulator there is no distinction between an application and its virtual platform. The application applies the resource demand adaptation policy by itself and interacts with the datacenter to change the bids for its VMs. As CloudSim does not model the cost of VM operations, we have previously implemented a model for several VM-related performance overheads: memory over-commit, VM boot/resume and VM migration [14].

3.1.2 Merkat Prototype and Testbed

To show the behavior of our system in a real environment, we implemented Merkat and evaluated it with several scientific applications. Merkat is composed of three main services, the Virtual Platform Manager, the VM Scheduler and the Virtual Currency Manager, which provide a XML-RPC interface to Merkat's users. Figure 4 gives an overview of them.

The *Virtual Platform Manager* acts as an entry point for running applications on our system. To start an application on the infrastructure, a user submits a request containing a virtual platform template to the Virtual Platform Manager. The virtual platform template contains information regarding the application controller, adaptation policy parameters and the configuration of the virtual clusters. To start the virtual platform, the Virtual Platform Manager checks any initial deployment conditions that a user has specified. For example, the user might not want to start running an application until the resource price drops below a threshold. If these conditions are not met, the deployment of the virtual platform is either postponed or canceled. For example, if the application needs to start its execution before a given deadline and the price is too high to allow it, the user might want to cancel the execution. If these conditions are met, the Virtual Platform Manager creates the virtual platform and registers it with the Virtual Currency Manager.

The *Virtual Currency Manager* distributes the virtual currency among users and manages the credits assigned by users to their running applications.

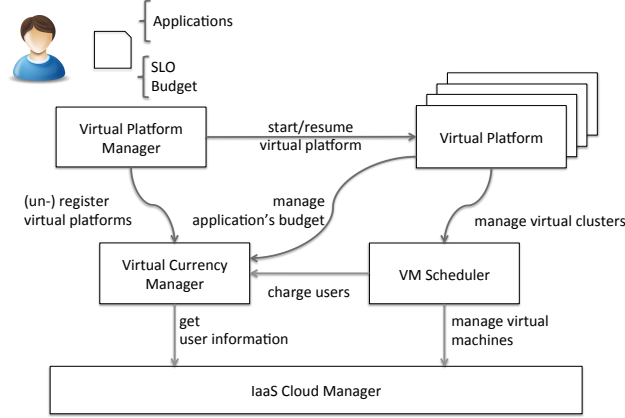


Figure 4: Prototype's components and their interactions.

The *VM Scheduler* is in charge of allocating VMs on the infrastructure and limiting the resource consumption in terms of memory and CPU for every VM according to infrastructure load and VM bid. The used algorithms are described in Section 2.2.

Merkat is implemented in Python and depends on the Twisted [45], paramiko [33] and ZeroMq [58] libraries. We used the Twisted framework 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. 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). Merkat's services store their state in a database storage for which we have used a MySQL server. The connections to the VMs are done through SSH in parallel, by using a thread pool with a size defined in the configuration file. As an IaaS Cloud Manager we use OpenNebula [41]. Merkat's services communicate with OpenNebula through its XML-RPC API. We also shut down OpenNebula's default scheduler, and replaced it with our VM Scheduler.

Merkat was deployed on the edel cluster of the Grenoble site from Grid'5000 testbed. Each cluster node has 2 Intel Xeon E5420 QC processors (4 cores) and 24GB of memory. The VM Scheduler assigns for each node an available capacity of 700 CPU units and 23 GB RAM (one core is reserved for the hypervisor/node's operating system) and it distributes it among the VMs running on the node. All nodes are connected through a Gigabit Ethernet and an Infiniband link. We use the Gigabit Ethernet link for VM communication and the Infiniband link for VM migration. Unless otherwise specified, all VMs have a maximum capacity of 100 CPU units and 900MB of RAM. Each VM is configured with Debian Squeeze 6.0.1 OS and KVM [22] as a hypervisor on each node. As an IaaS manager we used OpenNebula 4.6. The VM images are stored on a NFS server. To speed-up the VM deployment we use copy-on-write VM images. In all experiments, the VM Scheduler uses a scheduling period of 60 seconds and the application controllers read the application's performance metrics at 80 seconds. We wanted a scheduling period that is small enough to allow applica-

tions to adapt fast their bids, and large enough to allow the VM migrations to finish.

3.2 User Satisfaction

To see how the system performs in terms of user satisfaction when applications behave strategically and adapt their resource demand, we have tested Merkat's policies in simulation and on the real testbed. User satisfaction is an important metric regarding the performance of a resource management system. This satisfaction depends on the fact that the user's SLO is violated or not and on how much the user actually valued the execution of her application. We present in the following the SLO and user satisfaction modeling, together with the obtained results.

3.2.1 Experimental Setup

Application Model We consider batch applications composed of a fixed number of tasks running in parallel. To finish their execution, applications need to perform a certain computation amount (e.g. 1000 iterations). These applications have a relatively stable iteration execution time. The iteration execution time depends on the resource allocation received by each task. For example, if each task receives one full core, one iteration takes 1 second. If the resource allocation drops at half, the same iteration takes 2 seconds. We simulate these applications as sets of tasks, with each task requiring one CPU core and a specified amount of memory.

As a real-world application we use an open source software for fluid dynamics simulations, called Zephyr [57]. The implementation of Zephyr relies on MPI. Zephyr receives as input a configuration file and it simulates a volume filled by fluid for a specified simulated time, $t_{physical}$. The computation is done iteratively: at each iteration a set of equations is solved, e.g., Navier–Stokes, using a discretization step, τ . Zephyr periodically writes in a log file the following information: (i) the CPU time for each iteration, $t_{timestep}$; (ii) the current number of iterations, $N_{current}$; (iii) the total amount of time spent in I/O, t_{io} .

We used Zephyr to test Merkat on the Grid'5000 testbed. We modeled applications with different execution times, t_{exec}^i , by varying $t_{physical}$, starting from the same application and input data. The application's execution time is modeled as the following function:

$$t_{exec}^i = t_{io} + t_{avg} \cdot \frac{t_{physical}^i}{\tau} \quad (3)$$

where t_{avg} is the average CPU time per iteration. At each iteration, the application's progress is given by $N_{current}$ and the application's remaining execution time is computed as:

$$t_{remaining} = (N_{total} - N_{current}) \cdot t_{timestep} \quad (4)$$

We define N_{total} as the number of iterations performed by the application to obtain $t_{physical}$:

$$N_{total} = \frac{t_{physical}}{\tau} \quad (5)$$

User Model To measure the total user satisfaction, we model the user satisfaction as a utility function of the budget assigned by the user to its application and the application execution time. This function is defined as follows. The user values the application execution at her full budget if the application finishes before deadline. Otherwise, we express her dissatisfaction as a *penalty*, which is equal to the negative value of her budget.

Application Policy To meet the SLO of this user, we derive an application-specific policy from the vertical scaling policy, presented in Section 2.3. We call this policy *full-deadline*. To ensure the best chance to finish the application's execution before a deadline, the policy starts it as soon as the price drops enough so that application can afford a minimum allocation for each VM (e.g. 25% of CPU time). During their execution, applications adapt their bids to keep a low cost in low utilization periods and to use as much resource as their SLO allows in high utilization periods. If the application cannot pay for the resources needed to meet its SLO, it suspends its execution. In this way the application leaves resources for other applications and avoids wasting credits for nothing. The application resumes the application if the price drops enough to allow it to finish its execution within the deadline. If, during its execution, the application detects it cannot meet the deadline, it stops.

3.2.2 Simulation Experiment and Results

We measured the performance of our system in terms of total user satisfaction in different contention scenarios, compared to two traditional resource allocation policies: First-Come-First-Served (FCFS) and Earliest Deadline First (EDF). The first policy is usually applied by IaaS cloud managers to schedule VMs. It keeps the requests in a queue and schedules them when resources become available. The second policy is used to minimize the number of missed deadlines. The requests are ordered in the queue based on their remaining time to deadline and requests with the smallest remaining time to deadline are executed first. Cloud managers cannot practically apply this model without limiting their support to a predefined set of application goals (e.g., meeting deadlines). Nevertheless, we choose it for comparison purposes: we compare our system with a centralized system that targets a fixed type SLO.

To evaluate the system performance we use a real workload trace. Real workload traces are preferred to synthetic workloads as they reflect the user behavior in a real system. Such traces are archived and made publicly available [5]. As a workload trace, we chose the HPC2N file. This file contains information regarding applications submitted to a Linux cluster from Sweden. The cluster has 120 nodes with two 2 AMD Athlon MP2000+ processors each. We assigned to each node 2 GB of memory. The reason for choosing this trace is the detailed information regarding memory requirements of the applications. Nevertheless, this information was not specified for all the applications. Thus, for applications with missing memory requirements, we assigned a random amount of memory, between 10% and 50% of the node's memory capacity. We ran each experiment by considering the first 1000 jobs, which were submitted over a time period of 18 days. We scale the inter-arrival time with a factor between 0.1 and 1 and we obtain 10 traces with different contention levels. A factor of 0.1 gives a highly contended system while a factor of 1 gives a lightly loaded system.

We consider that all applications have a deadline and a re-chargeable budget. As we couldn't find any information regarding application deadlines, we assigned synthetic deadlines to applications, between 1.5 and 10 times the application execution time. We assume that the budget amount the user wants to pay depends on the application's deadline: a user with a less urgent application wants to pay less. Thus, the budgets assigned to applications are inversely proportional to the application's deadline factor, and computed from a base budget of 2000 credits per time period.

To see how the performance is influenced by the contention level, we measure the obtained satisfaction for each of the 10 obtained traces. We repeated each experiment four times and the results are presented as the mean plus standard deviation.

Figure 5 (a) describes the results of our comparison. For clarity, Figure 5 (b) describes the number of applications that successfully finished their execution until their deadline. We notice three aspects: (i) our system outperforms FCFS in all cases, as FCFS does not consider application valuation or SLO in its decisions. (ii) when the contention is not high, despite reaching almost the same number of finished applications as EDF, our system still outperforms it in terms of user satisfaction; (iii) however, when the system is highly loaded, its performance degradation increases. The performance gap between our mechanism and EDF can be explained as follows. When the contention is low, our system provides higher satisfaction than EDF, due to its fine-grained allocation policies. When the contention is high, the scheduling period is smaller and EDF is capable to take better scheduling decisions: thus more applications with smaller deadlines, and in the same time higher budgets, get to run on time. In the case of our system, applications do not take the best allocation decision, as they adapt independently with only limited information. This decentralization leads to a loss in performance, compared to EDF. The performance degradation is the "price" paid by the nature of our system, which allows applications to behave selfishly.

3.2.3 Testbed Experiment and Results

To measure the user satisfaction that Merkat provides on a real testbed, we set up a cloud managed first by Merkat and then by a batch scheduler, e.g., Maui [27]. Maui is often used by HPC organizations to manage their clusters. To run applications, Maui uses algorithms based on FCFS, with backfilling. We deployed both systems on a cloud of 10 compute nodes on the grenoble site of Grid'5000.

We run 160 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 Maui. In the later case, we deploy the VMs and add them to Maui as physical nodes at the beginning of the experiment. 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 [26] model to generate the workload. We chose this model as it is realistic. The generated application budgets were within the range of 1000 and 28700 credits per scheduling interval. In Merkat each application ran with a deadline-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-

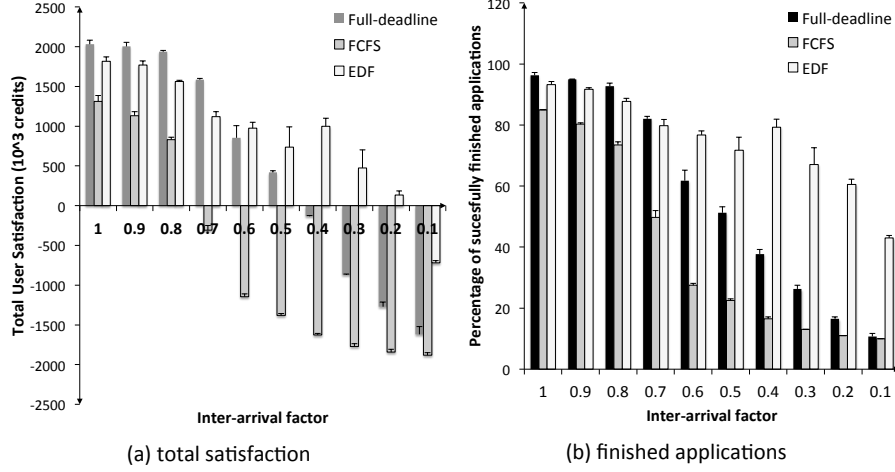


Figure 5: Proportional share market performance in different contention scenarios in terms of: total user satisfaction (a) and percentage of successfully finished applications (b). The contention increases from left to right.

Serve and backfilling to schedule applications to nodes.

We repeated the experiment 4 times and we averaged the number of successfully finished applications and the total satisfaction provided by Merkat and Maui. When Merkat is used to manage the cloud, 24.4% more applications finish until their deadline than when using Maui (in Merkat 82.5% of the applications finish until their deadlines compared to 58.1% when using Maui). Merkat also leads to more users being satisfied from running their applications on the cloud than traditional systems like Maui (in Merkat the total user satisfaction is 764330 credits, compared to a negative satisfaction of -178581 credits from Maui). This result provides evidence of the usefulness of a market-based autonomous system like Merkat.

3.3 Flexibility in Supporting Different Applications

We show that Merkat can be used to run different malleable frameworks with different SLOs, while improving the resource utilization of the infrastructure. This scenario is encountered in organizations that own a HPC infrastructure and want to share it between different application types and departments.

3.3.1 Experimental Setup

The support for malleable frameworks is illustrated through scaling two commonly used frameworks: Condor [25] and Torque [43]. Users submit applications to each framework. These frameworks are usually used by scientific organizations to manage their HPC clusters. Each framework has its own scheduler which puts applications in a queue and runs them on nodes when resources become available for them. We implemented a scaling policy for each framework

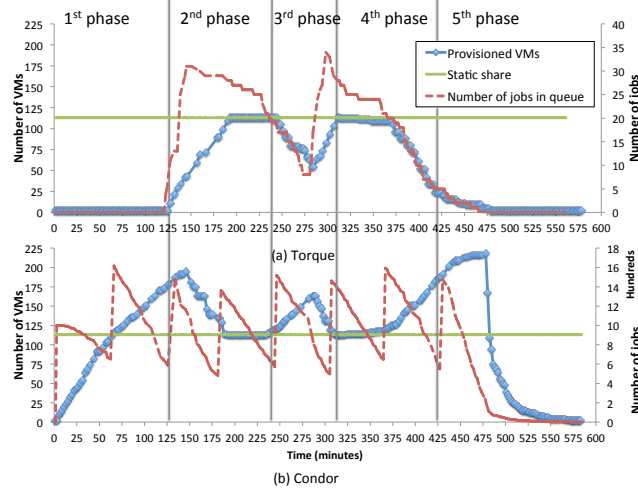


Figure 6: The variation of provisioned VMs versus queued jobs for each framework.

that uses a modified version of Algorithm 3 to provision VMs. For Torque applications, the policy minimizes the wait time in queue, while for the Condor applications the policy maximizes the throughput. In both scaling policies, the framework copes with fluctuations in price in two ways. First, it avoids requesting a large number of VMs per time period, as provisioning them is wasteful if the price increases in the next period. Second it avoids starting new VMs if some VMs were released due to a price increase in the previous period.

3.3.2 Testbed Experiment and Results

We deployed the Condor framework to process parameter sweep applications and the Torque framework to process MPI applications; both application types are commonly used at EDF. Then, we have studied how Merkat adapts the resource demand of each framework based on its workload.

We submitted 61 Zephyr applications to Torque with execution parameters taken from a trace generated using a Lublin model [26]: the number of processors was generated between 1 and 8 and the execution time had an average of 2479 seconds with a standard deviation of 1243.5. We submitted 8 parameter sweep applications composed of 1000 jobs of one processor each to Condor with an inter-arrival time of one hour. 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 execution time intervals generated with a Gaussian distribution. The average task execution time was 478 seconds, with a standard deviation of 363. Both frameworks were deployed on 32 nodes managed by Merkat and receive an equal budget.

Figure 6 shows the number of running and queued jobs in the Torque/Condor's queues and the number of VMs provisioned over time. If each framework can be assigned an equal share of the infrastructure, it would be able to provision a maximum of 112 VMs (green line from Figure 6).

We divided our experiment in five phases. In the first, third and fifth phase

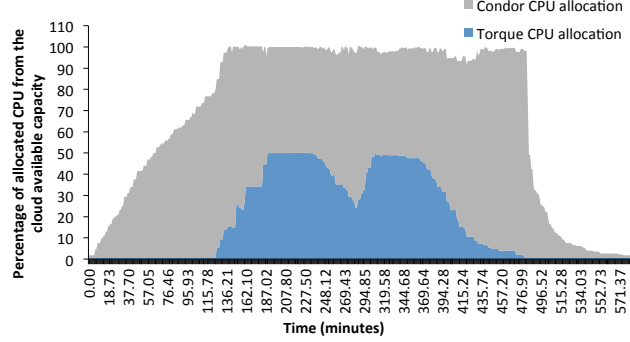


Figure 7: The variation of CPU allocation for both frameworks, as a percentage of the total available CPU capacity of the cloud.

Merkat’s controller is capable to take advantage of the under-utilization periods of the infrastructure and provision up to as many VMs as allowed by the infrastructure’s capacity for the Condor framework. In the second and fourth phase, both frameworks need more resources than their fair share to process their workload, thus each of them provisions an equal number of VMs. For clarity, Figure 7 shows the total CPU allocation for each framework, as a percentage from the total available cloud capacity. We have omitted the memory allocation, as it exhibits a similar trend. In contention periods, each framework receives half of the available CPU while in under-utilization periods, the Condor framework scales its CPU allocation according to its workload demand.

This experiment shows that Merkat allows applications to expand and shrink their resource demand according to the resource availability of the infrastructure, thus leading improving the resource utilization.

3.4 Flexibility to SLO Modifications

To show that Merkat provides the flexibility to dynamically change the application SLO, we also analyze the capacity of the application controller to react to a user imposed condition. For this, we ran a deadline-driven Zephyr application and changed the user-specified deadline during the application execution. We describe next the experimental setup and the results.

3.4.1 Experimental Setup

To show how an application controller can adapt to changes in its deadline, we ran a micro-benchmark using Zephyr applications: we submitted a deadline-driven application and four best-effort applications to Merkat. All the applications are started on the same node, each in a VM with 4 cores. The best-effort applications are started in the first few minutes after the start of the deadline-driven application. The application controllers of the best-effort submit a constant small bid for resources, while the application controller of the deadline-driven application uses the policy described in Section 3.2.1. The deadline-driven application has a budget of 60000 credits, from which it can spend as much as it wants, while the other best-effort applications start with a

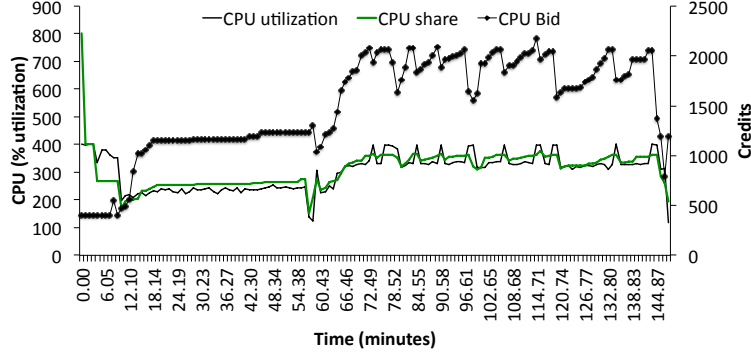


Figure 8: Application’s CPU utilization and bid variation due to adaptation to the new deadline.

bid of 100 CPU credits and 900 memory credits.

The deadline-driven application is started with an initial deadline of 12000 seconds. After 64 minutes from the application start the deadline is changed to 9000 seconds.

3.4.2 Results

The application controller’s bid adaptation and the allocation changes can be noticed in Figure 8. Figure 9 shows the application controller behavior and the variations in the estimated application execution time due to allocation and bid fluctuations. The additional submitted best-effort applications at the beginning of the experiment leads to a decrease in application allocation and thus an increase in its execution time. However, the application controller doesn’t react aggressively as its allocation, in this case 266 CPU units, is enough to meet the application deadline. When we decrease the application’s deadline (noticed in Figure 9 from the change in the reference time), however, the application controller also adjusts the bid aggressively, leading to an increased resource allocation, in this case close to 400 CPU units, which is the maximum VM allocation. This increase allows the application to keep its iteration execution time close to the reference, thus meeting its deadline.

4 Limitations and Future Work

In this section we discuss the remaining open issues of our approach.

Currency Policies Designing currency policies to decide how to give budgets to users remains a difficult task. Currently, to manage credits assigned to users, Merkat relies on an administrator, who decides the total credit amount in the system and assigns budgets to users proportional to some weights. Two issues are present with this policy: (i) deciding the total credit amount that circulates in the system; (ii) deciding the time period over which the user budgets are renewed.

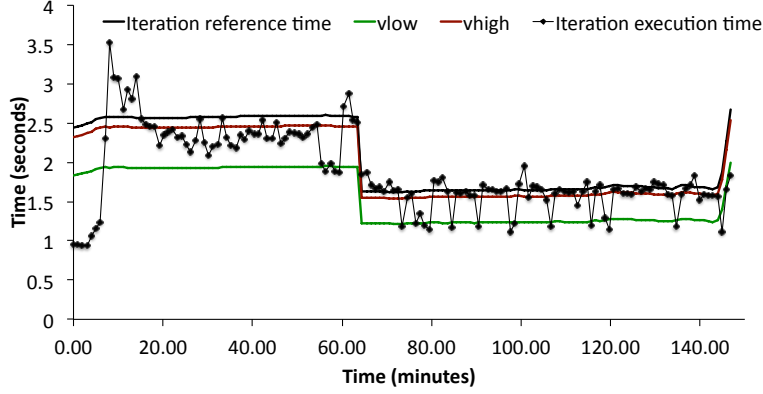


Figure 9: Application's iteration execution time variation due to adaptation to the new deadline. The reference iteration execution time and the two thresholds used in Algorithm 2 are also shown.

The value of the total credit amount influences the performance of the system. If the total credit amount is too large, the resource prices become too inflated and lead to poor user satisfaction. If the credit amount is small and new users arrive, current users might be funded at a rate too small to allow them to run highly urgent applications. Setting this amount can be done by the infrastructure administrator based on the number of users in the system, the price history and the capacity of the infrastructure. For example, if the administrator observes that the price increased considerably in the last period she can reduce the total credit amount. If new users arrive, she can increase this amount. Designing an automatic system to manage this amount remains an open issue. Such a system needs to adapt the amount based on past price fluctuations, infrastructure size and feedback from the users regarding their resource usage versus assigned budget.

The time period over which the user budgets are renewed influences the user behavior. If the time period is small, users will not have incentives to save budget, and thus, to judiciously use resources. If the time period is too large, users might starve while resources are left idle. We consider that such a period should be in terms of weeks or several months. However, deciding properly this time period remains an open issue as it requires running the system on a real-world infrastructure and getting feedback from users.

System Stability Poor adaptation policies, currency management policies or a too small scheduling interval might lead to system trashing, i.e., the system would spend too much time adapting, wasting resources and leading to poor application performance. For example, if application controllers adapt continuously the resources allocated to their applications, too many migrations might happen, wasting resources like network bandwidth, CPU, etc., and slowing down the application progress. In this case not only that applications will not receive the share they bid for, but they will also miss their SLOs. If application controllers suspend and resume applications too often, the application progress would be too slow and the application SLO will be missed. These operations

would also lead to price fluctuations, forcing other application controllers to adapt too, forming a vicious circle.

The system's stability can be improved in two ways: (i) increasing the VM scheduler's scheduling period, together with the application's controller's adaptation period; (ii) improving the adaptation policies. The first method will lead to a less reactive system, while the second method requires sophisticated algorithms for price and application performance prediction. Designing such prediction algorithms remains an open research question.

Scalability As the current scale on which Merkat was tested is quite limited, also due to limitations in the underlying third party software stack, we could not identify the upper scalability limit of Merkat when the scale increases. We believe that this upper limit is determined by two factors: (i) the performance of the VM Scheduler in allocating resources when it has to cope with adaptation requests from a large number of applications; (ii) and the performance of the IaaS manager. While the performance of the IaaS manager is outside the scope of this work, the performance of the VM Scheduler can be further improved by optimizing the VM allocation and placement algorithm.

Topology-awareness Co-location and migration might lead to performance degradation for running applications. Especially, in the case of HPC applications, care must be taken to the communication between application processes and the cluster topology. Currently, Merkat does not consider the application's or the cluster's topology. However, it can be further extended to allow users to express and pay for running their application processes on dedicated nodes as close as possible from each other.

I/O resources The current resource allocation algorithms could be extended to consider network and storage resources. Such resources can become a bottleneck when network or data intensive applications are executed. Thus, it is normal to make them available at a price that reflects the total resource demand. Regarding the network resource, software tools can be used to limit the bandwidth available to each virtual machine and ensure a proportional share. We envision that similar mechanisms could be applied for storage too.

High Availability To provide users with a production system, Merkat services require self-healing capabilities to make the platform highly available. State-of-the art solutions [21] can be used to achieve these goals.

5 Related Work

We classify the related work in three categories: (i) resource management systems for clusters; (ii) platform as a service systems for clouds; (iii) and resource management systems that apply a market.

5.1 Resource Management Systems for Clusters

Traditionally, HPC infrastructures are managed by batch schedulers [56, 43, 31, 29]. Batch schedulers are centralized resource management systems, which

employ a variety of scheduling algorithms derived from First-Come-First-Served (e.g., EASY [42] or Conservative Backfilling [42]). These systems provide poor SLO support to their users and, because they mostly rely on priority queues, do not properly distribute resources among users.

Several resource management systems were developed to support the execution of dynamic applications on shared infrastructures. To support different application types on the infrastructure, earlier systems focus on sharing the cluster or grid resources [9, 36, 23] between frameworks (e.g., Torque, SGE) using virtualization. However, in most cases, the policies to decide the resource allocations for each framework are missing. Mesos [20] is a system that allocates resources to frameworks based on the concept of "resource offer", i.e., a list of available resources on nodes. Mesos gives resource offers to frameworks while frameworks can filter the offers and decide what resources to accept. Omega [38] presents frameworks with a "replicated" shared view of the cluster, called a "cell state". Each framework keeps its own cell state, which is synchronized periodically to reflect allocation changes. Using their cell state, frameworks can select what resources to use, according to locality or resiliency constraints, even by preempting other framework's tasks. In Yarn [46] each application has its own application manager that requests resources from a global scheduler. The global scheduler allocates resources to them by considering specific application constraints. In contrast to Merkat, all these systems don't consider individual application performance or SLOs.

In the datacenter community there are different systems to allocate resources between different, usually web-based, applications [52, 60, 49, 19, 28]. They rely on the use of performance models to obtain estimations and predict the application resource allocation to minimize the number of used cluster nodes. In this case, resources are allocated to VMs in a fine-grained fashion as applications like web servers can have varying CPU and memory utilization. VM migrations are employed to move the VMs among nodes either reactively [51], or proactively [39]. These systems are designed specifically for one application type. The closest to our work is "Friendly virtual machines" [59]. In this work the resource control is fully decentralized: applications adapt autonomously their resource demand using a "feedback signal" received from the system. However, the authors assume that applications are altruistic. In Merkat, we consider that applications are selfish and application controllers adapt their application resource demand based on the current resource price and application budget.

Quasar [15] is a resource manager for clusters with support for multiple SLOs. Quasar uses classification techniques to determine the amount of resources required by each application and its dataset to meet a specified performance. In Quasar there are only two types of workloads: best-effort and SLO-driven. In contrast to Quasar, Merkat uses a market to distribute resources among users, leading to a more fine-grain differentiation between application priorities.

5.2 PaaS Systems

Many PaaS systems, both commercial [3, 6, 12] and research [34] provide runtime support for applications hiding from users the complexities of managing resources. These systems, however, provide typically closed environments, forcing users to run only specific application types (e.g., web, MapReduce). If new

programming frameworks appear, the PaaS provider needs to first develop the necessary support on the infrastructure, and then offer to the users the possibility to use it. In contrast to these systems, Merkat allows users to run new application types while distributing the resources among them based on their value.

Similar to Merkat, Meryn [16] is a PaaS that supports new application types through a decentralized resource control: resources are allocated among frameworks with consideration to the SLO and cost of the applications managed by them. Merkat is different from Meryn in two ways: (i) it allocates virtual environments per application while Meryn allocates them per framework; (ii) and it focuses on managing contention on a private infrastructure by implementing a virtual economy while Meryn uses cloud bursting to offload its workload in public clouds while optimizing the PaaS provider's profit.

5.3 Market-based Resource Management Systems

Using a market to manage the resources of a distributed infrastructure is a well-studied problem in the context of clusters and grids [53]. The reason why markets became so popular in the distributed systems community is the notion of *cost*, which makes users more aware of how many resources to acquire for their own use. By using a market, users have incentives to assign the right priorities to their applications. Thus, applications with urgent resource needs can get resources in time even in high load periods, while applications with less urgent resource needs run in low load periods.

There are two commonly used market models: commodity markets and auctions. These types of markets differ one from another in the way the market clearing price is computed. This market clearing price expresses the fact that the user demand matches the resource supply.

In commodity markets the resource price is established using demand and supply functions and both consumers and providers buy and respectively sell at this price. These market types rely on algorithms to adjust the resource price to reduce the excess demand close to zero, also called *tatonnement* algorithms [10]. These algorithms either use estimations of the excess demand, as in G-commerce [50], or they rely on the participants to send their demand as a function of price like in On-Call [30], which will raise scalability issues on large-scale systems.

One particular case is Libra [40], which brings a different model: it applies a proportional-share policy on each node to allocate CPU proportionally to the application's deadline while a global pricing mechanism, based on the infrastructure utilization, is used to balance supply with demand. Admission control ensures that the jobs accepted in the system do not lead to deadline misses for the other jobs. The use of the proportional-share policy maximizes the infrastructure utilization, as opposed to a case in which jobs are allocated exclusive-access to nodes. Aneka [55] implements a dynamic pricing model too by using advance reservations as a substrate resource allocation model. Unfortunately, as the price is set at the beginning of the execution, applications coming in the system in a low demand period will be charged with a small price, while urgent applications coming later, might not get all the resources they need for their execution.

Auctions establish the resource price based on how much users are willing to

pay for resources. Auctions clear the market faster than the price computation algorithms from the commodity markets, allowing users with the most urgent demand to get their resources with minimum delays. Multiple attempts were made to use auctions to schedule static MPI applications, on clusters [44, 54] or to run bag-of-tasks applications on grids [1]. In this case, users bid to execute their applications and the scheduler decides which application gets to run through the auction. An increased user satisfaction is obtained 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. Nevertheless, these systems address the case of applications with known execution times and static resource requirements.

A first step in providing more flexibility for dynamic applications was taken by Popcorn [35] and Spawn [47], in which applications composed of many tasks can shrink when the resource price is high and expand when the resource price is low. Systems like Tycoon [24] or REXEC [11] implement a proportional-share policy per node to allocate fractional amounts of CPU. Ginseng [2] is a cloud platform that uses an auction to allocate memory to VMs. The applications running in VMs have to decide the quantity of memory and the bid based on their performance. Although it represents a first step towards a market-based platform, further development is required to provide users with a complete system. A dynamic priority scheduler is proposed for Hadoop [37], to allocate map/reduce slots to jobs using proportional-share. Users are assigned a budget and they spend it to run jobs on the cluster, by specifying a spending rate, i.e. the user's willingness to pay for a slot and for a time period. The queue priority budget and the spending rate allow the user to control how many slots get assigned to her jobs over time, and thus, to control the job execution time. 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. In contrast to these systems, Merkat controllers can adapt applications in two different ways, vertical and horizontal, to meet user SLOs.

6 Conclusion

This paper introduced a platform for application and resource management in private clouds, called Merkat. The goal of Merkat is to maximize the resource utilization of the managed infrastructure while providing support for different application resource demand models and user SLOs. To meet this goal Merkat transforms the organization's infrastructure to a private cloud and relies on: (i) a proportional-share market to allocate fine-grained CPU and memory amounts to VMs and to make users aware of the cost of using resources; (ii) a set of autonomous application controllers that can scale the application's resource demand vertically and horizontally to meet the user's SLO under current price and user budget constraints. Merkat supports different applications and user SLOs by decentralizing the resource control and treats contention periods that might appear on the private infrastructure by using market mechanisms.

We evaluated Merkat in simulation and on the Grid'5000 testbed. The obtained results show that: (i) Merkat is flexible enough to support different applications and SLOs; (ii) Merkat can adapt the application resource demand to

the infrastructure load and application's SLO, maximizing resource utilization while leading to a better user satisfaction; (iii) the resource control decentralization has a reasonable performance impact compared to centralized resource management systems.

7 Acknowledgements

This work was done while the first author was a PhD student at INRIA and EDF R&D and was supported by ANRT through the CIFRE sponsorship No. 0332/2010. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see <https://www.grid5000.fr>).

References

- [1] 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*, 18(8):1061–1074, 2002.
- [2] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14*, pages 41–52, New York, NY, USA, 2014. ACM.
- [3] AmazonEBS. <http://aws.amazon.com/>.
- [4] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13*, pages 20:1–20:13, New York, NY, USA, 2013. ACM.
- [5] P. W. Archive. <http://www.cs.huji.ac.il/labs/parallel/workload/>.
- [6] M. Azure. <http://www.windowsazure.com/>.
- [7] 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.
- [8] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software Practice and Experience*, 41(1):23–50, 2011.
- [9] 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 12th IEEE International Symposium on High Performance Distributed Computing, 2003.*, pages 90–100. IEEE, 2003.

- [10] J. Q. Cheng and M. P. Wellman. The walras algorithm: A convergent distributed implementation of general equilibrium outcomes. *Computational Economics*, 12(1):1–24, 1998.
- [11] B. N. Chun and D. E. Culler. Rexec: A decentralized, secure remote execution environment for clusters. In *Proceedings of the 4th International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, CANPC '00, pages 1–14, London, UK, UK, 2000. Springer-Verlag.
- [12] CloudFoundry. <http://www.cloudfoundry.com/>.
- [13] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas. Merkat: A market-based slo-driven cloud platform. In *Proceedings of IEEE International Conference on Cloud Computing Technology and Science*, Cloud-Com'13, 2013.
- [14] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas. On the use of a proportional-share market for application slo support in clouds. In *Proceedings of the 19th International European Conference on Parallel and Distributed Computing*, Europar'13, 2013.
- [15] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, New York, NY, USA, 2014. ACM.
- [16] D. Dib, N. Parlavantzas, and C. Morin. SLA-based Profit Optimization in Cloud Bursting PaaS. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, États-Unis, 2014.
- [17] F. Glover, M. Laguna, et al. *Tabu search*, volume 22. Springer, 1997.
- [18] A. Gulati, G. Shanmuganathan, A. Holler, and I. Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, pages 3–3. USENIX Association, 2011.
- [19] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID'12, pages 644–651. IEEE, 2012.
- [20] 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 USENIX conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.
- [21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

- [22] A. Kivity, Y. Kamay, and D. Laor. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.
- [23] N. Kiyancilar, A. G. Koenig, and W. Yurcik. Maestro-vc: On-demand secure cluster computing using virtualization. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid Workshops*, 2006.
- [24] 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*, 1(3):169–182, 2005.
- [25] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, 1988.
- [26] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.
- [27] Maui. <http://www.nsc.liu.se/systems/retiredsystems/grendel/maui.html>.
- [28] M. Maurer, I. Brandic, and R. Sakellariou. Enacting slas in clouds using rules. In *Euro-Par 2011 Parallel Processing*, pages 455–466. Springer, 2011.
- [29] Moab. <http://www.clusterresources.com/products/moab-cluster-suite.php>.
- [30] J. Norris, K. Coleman, A. Fox, and G. Candea. Oncall: Defeating spikes with a free-market application cluster. In *Proceedings of the 2004 International Conference on Autonomic Computing*, ICAC’04, pages 198–205. IEEE, 2004.
- [31] OAR. oar.imag.fr/.
- [32] OpenStack. <http://www.openstack.org/>.
- [33] paramiko. www.lag.net/paramiko/.
- [34] G. Pierre and C. Stratan. Conpaas: a platform for hosting elastic cloud applications. *IEEE Internet Computing*, 2012.
- [35] O. Regev and N. Nisan. The popcorn market. online markets for computational resources. *Decision Support Systems*, 28(1):177–189, 2000.
- [36] P. Ruth, P. McGachey, and D. Xu. Viocluster: virtualization for dynamic computational domains. In *Proceedings of IEEE International Conference on Cluster Computing*, 2005.
- [37] T. Sandholm and K. Lai. Dynamic proportional share scheduling in hadoop. In *15th Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.
- [38] 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.

- [39] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11*, pages 5:1–5:14, New York, NY, USA, 2011. ACM.
- [40] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: a computational economy-based job scheduling system for clusters. *Software Practice and Experience*, 34:573–590, 2004.
- [41] 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*, 13(5):14–22, 2009.
- [42] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Proceedings of Parallel Processing Workshops*, pages 514–519. IEEE, 2002.
- [43] G. Staples. Torque resource manager. In *Proceedings of ACM/IEEE conference on Supercomputing*, 2006.
- [44] I. Stoica, H. Abdel-Wahab, and A. Pothen. A microeconomic scheduler for parallel computers. In *Job Scheduling Strategies for Parallel Processing*, pages 200–218. Springer, 1995.
- [45] Twisted. twistedmatrix.com/.
- [46] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 2013 ACM Symposium on Cloud Computing, SOCC'13*, 2013.
- [47] 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*, 18(2):103–117, 1992.
- [48] C. A. Waldspurger and W. E. Wehl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.
- [49] Z. Wang, Y. Chen, D. Gmach, S. Singhal, B. Watson, W. Rivera, X. Zhu, and C. Hyser. Appraise: application-level performance management in virtualized server environments. *IEEE Transactions on Network and Service Management*, 6(4):240–254, 2009.
- [50] R. Wolski, J. S. Plank, T. Bryan, and J. Brevik. G-commerce: Market formulations controlling resource allocation on the computational grid. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium.*, pages 8–pp. IEEE, 2001.
- [51] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.

- [52] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, 2008.
- [53] C. S. Yeo and R. Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Software: Practice and Experience*, 36(13):1381–1419, 2006.
- [54] C. S. Yeo and R. Buyya. Pricing for utility-driven resource management and allocation in clusters. *International Journal of High Performance Computing Applications*, 21(4):405–418, 2007.
- [55] C. S. Yeo, S. Venugopal, X. Chu, and R. Buyya. Autonomic metered pricing for a utility computing service. *Future Generation Computer Systems*, 26(8):1368–1380, 2010.
- [56] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.
- [57] Zephyr. <https://github.com/kortas/zephyr>.
- [58] ZeroMQ. <http://www.zeromq.org/>.
- [59] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 2–12, New York, NY, USA, 2005. ACM.
- [60] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. Mckee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12(1):45–57, 2009.



**RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

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

ISSN 0249-6399