Market-based Autonomous Resource and Application Management in Private Clouds

Stefania Costache, Samuel Kortas, Christine Morin, Nikos Parlavantzas

▶ **To cite this version:**

Stefania Costache, Samuel Kortas, Christine Morin, Nikos Parlavantzas. Market-based Autonomous Resource and Application Management in Private Clouds. Journal of Parallel and Distributed Computing, 2017, 100, pp.85-102. 10.1016/j.jpdc.2016.10.003 . hal-01378536

HAL Id: hal-01378536

https://hal.science/hal-01378536

Submitted on 9 Jun 2017

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 Resource and Application Management in Private Clouds

Stefania Costache[a,1], Samuel Kortas[a,2], Christine Morin[b], Nikos Parlavantzas[c]

[a]*EDF R&D, 1 av. du General De Gaule, BP 408, 92141 Clamart cedex, France*
[b]*INRIA, Campus de Beaulieu, 35042 Rennes cedex, France*
[c]*IRISA, Campus de Beaulieu, 35042 Rennes cedex, France*

Abstract

High Performance Computing (HPC) clouds need to be efficiently shared between selfish tenants having applications with different resource requirements and Service Level Objectives (SLOs). The main difficulty relies on providing concurrent resource access to such tenants while maximizing the resource utilization. To overcome this challenge, we propose Merkat, a market-based SLO-driven cloud platform. Merkat relies on a market-based model specifically designed for on-demand fine-grain resource allocation to maximize resource utilization and it uses a combination of currency distribution and dynamic resource pricing to ensure proper resource distribution among tenants. To meet the tenant's SLO, Merkat uses autonomous controllers, which apply adaptation policies that: (i) dynamically tune the application's provisioned CPU and memory per virtual machine in contention periods, or (ii) dynamically change the number of virtual machines. Our evaluation with simulation and on the Grid'5000 testbed shows that Merkat provides flexible support for different application types and SLOs and good tenant satisfaction compared to existing centralized systems, while the infrastructure resource utilization is improved.

Keywords: resource management, cloud computing, elastic scaling, market mechanisms, Service Level Objective, HPC

1. Introduction

The emergence of new workload types, like data analytics, raises multiple issues regarding how resources of HPC clouds should be managed. Currently, these infrastructures need to be shared between multiple tenants, each requiring different frameworks, and having various application performance constraints, e.g., high availability, throughput, latency. Satisfying all these constraints while minimizing the cost of maintaining the infrastructure is a key challenge for many organizations. To ease their infrastructure's management, some of these organizations choose to transform the infrastructure in a "private cloud", managed by a specialized system [1, 2, 3, 4, 5].

Email address: s.costache@vu.nl, samuel.kortas@kaust.edu.sa, christine.morin@inria.fr, nikos.parlavantzas@irisa.fr (Nikos Parlavantzas)

[1]The author is currently working at VU Amsterdam, De Boelelaan 1081A 1081 HV Amsterdam, The Netherlands

[2]The author is currently working at KAUST, 4700-Thuwal, Jeddah Zip Code 23955-6900, Saudi Arabia

The main difficulty that this system needs to address is allocating the resources needed for each application. This resource allocation must fulfill the following requirements. First, the resource allocation has to respect the infrastructure's capacity limitations. Having enough capacity to meet all tenant requests in the highest demand periods is rarely the case, as expanding the resource pool is expensive. To differentiate between tenant requests in these periods, most of the state of the art systems [1, 2, 3, 5] rely on priority classes, i.e., tenants are given a priority class for their applications, or quotas, i.e., tenants have limited resource amounts to provision. However, in this case the tenants might abuse their rights and run less urgent applications, taking resources from tenants who really need them. Second, the resource allocation has to consider the application characteristics and tenant Service Level Objectives (SLOs). Some applications might have a resource demand that varies frequently due to the variability in the number of requests they have to process, e.g., web applications, while others can simply scale and use all currently available resources, 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 also benefit from scaling their resource demand vertically, i.e., scaling the resource amount per node [6]. Statically allocating resources is inefficient as it leads to under- or over- provisioning, thus either poor application performance or poor utilization. Then, tenants might have different SLOs for their applications. Many state-of-the art systems allocate resources to meet tenant SLOs by using offline and online profiling and specialized policies designed only for some application types, mostly batch data analytics [7, 8, 9] or web applications [10, 11, 12, 13]. These systems focus on specific SLOs and provide poor tenant differentiation policies, by separating the workloads in classes, e.g., silver, gold [14] or best-effort and latency-sensitive [8, 7, 9, 15].

In this paper we present the design and evaluation of Merkat, a HPC cloud platform. *The novelty of our proposed system comes from the decentralized resource allocation model, in which applications provision resources autonomously to manage their tenants' SLOs by controlling the amount they pay for resources; the resources are allocated to them through a proportional-share market that provides fine-grained resource allocation and a dynamic pricing model which varies with the total infrastructure demand.* The contributions of Merkat are the following:

- **A proportional-share model for dynamic virtualized resource allocation** Merkat's market relies on an algorithm which allocates virtualized resources in terms of proportional shares of CPU and memory to virtual machines (VMs), and performs load balancing between physical nodes. This fine-grained distribution scheme allows applications to scale their resource demand both horizontally and vertically. The market's currency distribution policy and dynamic price *ensure proper resource utilization in contention periods*, by favoring tenants who get the most value from the resources.

- **Two SLO-driven resource demand adaptation policies** We implemented in Merkat two resource demand adaptation policies to be used by SLO-driven applications running on the proportional-share market: (i) a policy that adapts the application resource demand per VM to the current resource availability and resource price; (ii) a policy that adapts the application resource demand in terms of number of VMs to the current

resource availability and resource price. These policies can also be combined to allow applications to satisfy better their SLOs.

- **A generic platform for application and resource management** We designed Merkat to provide tenants with generic automatic application management mechanisms, allowing them to implement their own resource demand policies. The platform is extensible: it allows new scheduling algorithms to be developed and new application types to be integrated on the infrastructure.
- **An evaluation of Merkat in simulation and on a real testbed** We evaluated the performance of Merkat's proportional-share market in terms of total tenant satisfaction when applications adapt their resource demands per VM to track a tenant-given SLO [16] using CloudSim [17]. We also tested Merkat on the Grid'5000 [18] testbed with two application types: static MPI applications and malleable task processing frameworks [19]. 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; (ii) 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 describes the vertical and horizontal policies implemented in Merkat. We illustrate the use of the policies with examples validated on the Grid'5000 testbed. Section 4 presents evaluation results, with simulation, showing how Merkat's algorithms behave on large-scale system, and how the Merkat prototype behaves on a small real testbed. Section 5 discusses limitations and future directions of improvement while Section 6 describes the related work. Finally, Section 7 concludes the paper.

2. Merkat

Merkat has been designed to manage the clusters owned by an organization that needs to run HPC workloads but cannot use public cloud resources due to security constraints, e.g., the data that needs to be processed is too sensitive. This is often the case of organizations carrying out research activities in computational sciences. An example of such organization is Electricite de France (EDF), which relies on HPC simulations to optimize the day-to-day electricity production or to choose the safest and most effective configurations for nuclear refueling. The clusters are shared among a multitude of tenants (e.g., scientific researchers) which might come from different departments (e.g., production, development) and might need to use specific frameworks or libraries to run their applications. These tenants not only have different SLOs for their applications, e.g., getting computation results until a specific deadline, or executing the application as fast as possible, but they might also want to assign different importance degrees to their requests.

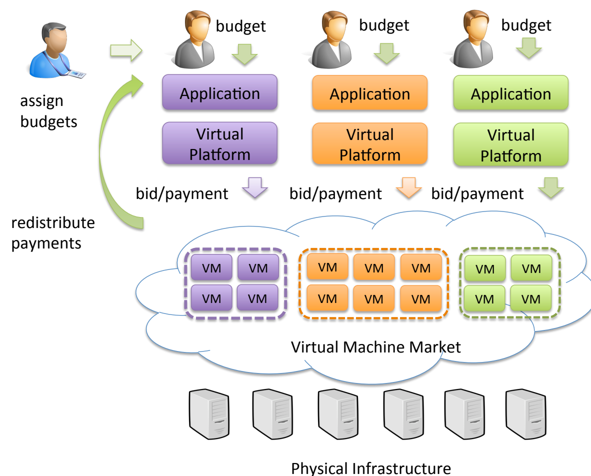


Figure 1: Overview of the Merkat system.

Merkat meets the tenant demands, while also allowing the organization to get the best out of its infrastructure, by maximizing its resource utilization. Figure 1 gives an overview of Merkat. Merkat relies on: (i) a virtual machine market to allocate resources and internal currency to manage the priority of tenants' demands and (ii) a set of virtual platforms that manage and elastically scale applications to meet tenant SLOs.

To be able to run applications on the infrastructure, tenants are assigned budgets by an administrator in the form of a virtual currency. Merkat disposes of a total amount of currency, which is distributed among uses based on defined weights. We call the currency unit a *credit*. A tenant will receive a budget of credits proportional to its weight in the system. It is the task of the administrator to add/remove tenants to/from the system and set up and adjust the total amount of currency and the tenants' weights. Virtual currency is desirable when managing a private infrastructure; because no external currency is introduced, price inflation is bounded.

When a tenant wants to run an application, she assigns it a replenishable amount of credits, called application budget, and sends a request to Merkat to start a virtual platform for it. This application budget reflects the maximum cost the tenant is willing to support, or the true priority, for running an application. These budgets are replenished automatically at a system-wide interval defined by the administrator. The replenishment is defined as transferring a tenant-defined amount from her account to the application's budget. Merkat ensures that the amount with which the budget is replenished never leads to a total budget that exceeds the initial budget amount. Replenishable budgets are used to minimize the risk of depleting the application's budget in the middle of the application's execution.

A virtual platform runs the tenant's application by acquiring VMs from the virtual machine market. The design of the virtual platform is specific to the application type the tenant wants to run. VMs are acquired by submitting payments for their resources, also called bids. Bids can be scaled up or down during the VM runtime. This mechanism has several advantages. First, it provides *flexibility for designing a variety of policies* to adjust the resource allocation

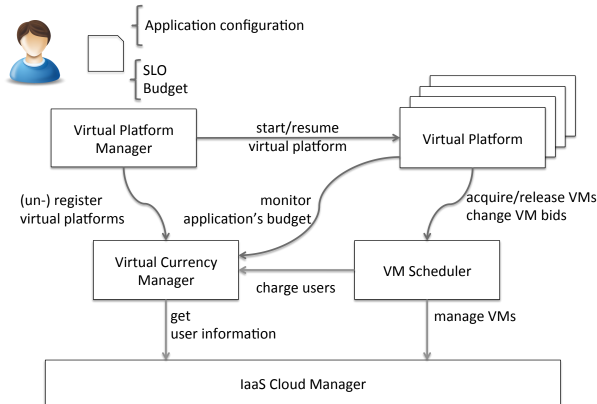


Figure 2: Prototype's components and their interactions.

of the application in a selfish way, with regard only to the tenant's SLO and application budget. This selfish behavior is natural, as tenants care only about the performance of their applications. In this context, currency management allows some control over this selfish behavior and gives tenants incentives to use better the infrastructure, while also allowing flexibility in meeting their SLOs. Second, allowing a bid per resource per VM leads to *efficient resource utilization* for the organization, as there is a strong incentive for tenants to design policies that avoid oversubscription by requesting just as much as the application uses from the capacity of a VM. Lastly, communicating resource demands through the use of bids leads to *a generic system*, capable of supporting any kind of application and SLO.

2.1. Architecture

Merkat is composed of three main services, the VM Scheduler, the Virtual Currency Manager and the Virtual Platform Manager. Figure 2 gives an overview of them. The *Virtual Currency Manager* applies virtual currency distribution policies and manages tenant and application budgets. The *VM Scheduler* is in charge of allocating resources to running VMs and computing their node placement. The used algorithms are described in Section 2.2. The *Virtual Platform Manager* acts as an entry point for tenants to run their applications on our system. To start an application on the infrastructure, a tenant 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, including desired tenant SLO, and the configuration of the VMs, e.g., VM disk image. We define the SLO as the performance objective the tenant wants for her application, e.g., a specific execution time or throughput. To start the virtual platform, the Virtual Platform Manager checks any initial deployment conditions that a tenant has specified. For example, the tenant might want to start running an application only if the resource price is below a threshold. If these conditions are not met, the deployment of the virtual platform is either postponed or canceled. For example, the deployment is canceled if the application needs to start its execution before a given deadline (the tenant's defined SLO) and the price is too high to allow it. If these conditions are met, the Virtual Platform Manager creates the

virtual platform and registers it with the Virtual Currency Manager. The Virtual Platform Manager is also in charge of resuming any virtual platforms that might suspend their VMs to avoid executing applications in high price periods.

To manage the tenants and the VMs, Merkat interacts with an IaaS Cloud Manager [20]. The IaaS Cloud Manager provides interfaces to start, delete and migrate VMs, manage their storage, network, and moreover, to keep information about the infrastructure's tenants.

2.2. *The Virtual Machine Market*

To allocate resources to VMs based on the value of the submitted bids, Merkat uses a proportional-share policy. Originally, this policy was used by the operating system schedulers to allocate CPU time to tasks proportionally to a given weight, and inversely proportionally to the sum of all the other concurrent task weights [21]. Merkat uses a modified version of this policy, in which each provisioned VM has an associated bid for its resources, i.e., CPU and memory, and it receives an amount of resource proportional to the bid and inversely proportional to the sum of all concurrent bids [22]. The value of the bid of a VM can be changed during the VM runtime.

This policy is advantageous as it allows provisioning VMs with arbitrary resource allocations at a small time complexity (O(M), where M is the number of VMs in the system), aspect which becomes important with the increasing scale of the infrastructures. 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 voluntary shutdown some of their components.

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

2.2.1. *VM Bid Submission*

To provision VMs, a bid, in the form of a vector $b = < b_{cpu}, b_{memory} >$ 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 tenant or the virtual platform that manages the VMs. The initial bid can be computed based on past price history and the tenant'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 tenant 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.

2.2.2. *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.

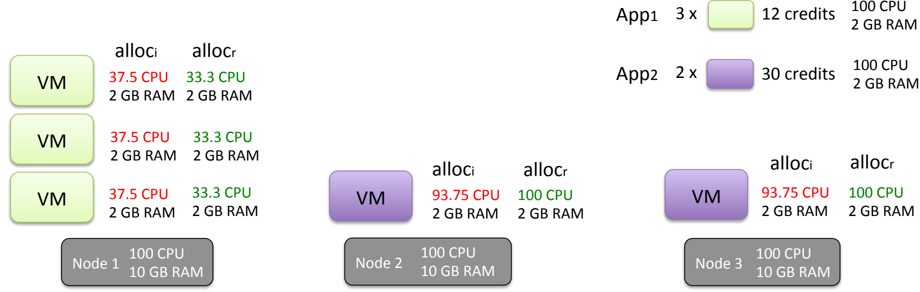


Figure 3: An example of resource fragmentation. 2 applications requesting 3 and respectively 2 VMs run on 3 nodes. If resources are allocated with the proportional-share policy from the entire infrastructure capacity (allocated amounts are noted with $alloc_i$), the CPU allocations for the first application's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

We consider that the tenant can estimate a_{max} for her VMs. For example, on a cluster with 8-core nodes the tenant knows that her VM cannot use more than 8 cores.

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 $b_j(t)$, with $j \in \{1..n\}$, for a time interval t and a capacity of C units, the resource allocation for each bid $b_j(t)$ is equal to:

$$alloc_j(t) = \frac{b_j(t)}{\sum_j^n b_j(t)} \cdot C, \quad (1)$$

However, as the infrastructure capacity is partitioned between nodes, there are situations when resulted allocations cannot be enforced. This issue is illustrated in Figure 3. 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 *initial allocation* for VM j, $alloc_{iCPU\,j}$ (depicted with red in Figure 3), $j \in 1..5$, 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 *resulting allocation* for the VM j, $alloc_{rCPU\,j}$ (depicted with green in Figure 3), $j \in 1..5$, is: 33.3 CPU units for the first 3 VMs and 100 CPU units for

the last 2 VMs. The resulting allocation difference is called *allocation error* and is defined as follows:

$$e_{\{CPU,mem\}} = \frac{\left|alloc_{i\{CPU,mem\}} - alloc_{r\{CPU,mem\}}\right|}{alloc_{i\{CPU,mem\}}}, \tag{2}$$

As seen in the next section, this allocation error is used in placing the VMs on nodes.

We note that the time complexity for this algorithm is O(M), where M is the total number of VMs from the system.

2.2.3. VM Load Balancing

When VM requests are received the VM Scheduler places them initially on the nodes with the lowest resource utilization. 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 [23]. Tabu-search is a local-search method for finding the good 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 maximum VM allocation error, noted with e_{max} in Algorithm 1 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 (the allocation error for the CPU/memory resource of the VM i is noted with e_{ic} and, respectively, e_{im}) 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.

The time complexity for the load balancing algorithm is given by: (i) the time complexity of the $ComputeErrors$ function (Line 12 and Line 22); (ii) the VM selection algorithm (Line 15); (iii) the node selection algorithm (Line

17); (iv) the computation of the required number of migrations (Lines 25-28); (v) and the update of the migration plan (Lines 35-40). The time complexity of the *ComputeErrors* function is O(M) , where M is the total number of VMs in the system, as it is given by the time complexity of the resource allocation algorithm presented in Section 2.2.2. The VM selection algorithm has O(M) time complexity while the node selection algorithm has O(N) time complexity, where N is the total number of nodes in the system. The computation of the required number of migrations and the update of the migration plan has O(M) time complexity.

2.2.4. 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. Tenants are charged for their running applications at each scheduling interval with a credit amount equal to the product between the resource price and the allocated resource amounts for all running VMs.

2.3. Virtual Platforms

When running the application on the Merkat's virtual machine market, it is not sufficient for the virtual platform logic just to acquire a number of VMs and compute their bids at the beginning of the application execution. As the resource price and allocation are dynamic, this logic will not guarantee that the application receives the right amount of resources to efficiently meet its SLO. If the price increases, the current tenant budget might not be enough to cover the cost of the needed resources. Thus, the application will run with less resources and not only will the SLO not be met, but also the budget will be wasted. If the price decreases, more resources could be provisioned, or the tenant could spend less for the already acquired resources. These issues are solved by the virtual platform, which monitors the application performance and adapts autonomously to the resource price and tenant requirements.

2.3.1. Virtual Platform Architecture

Figure 4 illustrates the architecture of a virtual platform. A virtual platform is composed of one or more virtual clusters, and an application controller, that manages them on behalf of the application. The application controller receives as input a virtual platform template, 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 and starts the application component in it. We define a virtual cluster as a group of VMs that have the same resource configuration and use the same base VM image. As the application's components might have different performance metrics, a different virtual cluster monitor running tenant-specific monitoring policies can be started in each virtual cluster.

During the application runtime, the application controller checks the application's performance metrics and adapts dynamically the virtual platform resource demand to the infrastructure resource prices and reconfigures the application. The tenant 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.

Algorithm 1 VM load balancing algorithm.

```
1:  ComputePlacement (nodes, vms, newvms, N_iter, E_max, M_max)
2:  migrationPlan ← ∅     // migrations to be performed
3:  deploymentPlan ← ∅     // deployments to be performed
4:  nIterations ← 0     // number of iterations until an improvement
5:
6:  for vm ∈ newvms do
7:      node ← least loaded node from nodes
8:      node.vms ← node.vms ∪ {vm}
9:  solution_old ← nodes     // current placement of VMs
10: solution_best ← nodes     // new placement of VMs
11: tabu_list ← ∅     // list of forbidden moves
12: e_worse ← inf
13: e = ComputeErrors(vms, nodes)
14: while nIterations < N_iter and e_worse > E_max do
15:     (vm, e_max) ← vm with e_max = max_{1≤i≤n} max{e_ic, e_im}, vm ∉ tabu_list
16:     source ← vm.node
17:     destination ← node which minimizes e_max, node ∉ tabu_list
18:     vm.node ← destination
19:     source.vms ← source.vms − {vm}
20:     destination.vms ← destination.vms ∪ {vm}
21:     tabu_list ← tabu_list ∪ {(vm, source)}
22:     e = ComputeErrors(vms, nodes)
23:     e'_max ← max_{1≤i≤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 ← e'_max
28:         nIterations ← 0
29:     else
30:         nIterations ← nIterations + 1
31: for node ∈ solution_old do
32:     for vm ∈ node.vms do
33:         if vm.node ≠ node and vm ∉ newvms then
34:             migrationPlan ← migrationPlan ∪ (vm, vm.node)
35: for vm ∈ newvms do
36:     deploymentPlan ← deploymentPlan ∪ (vm, vm.node)
37:
38: return (migrationPlan, deploymentPlan)
```

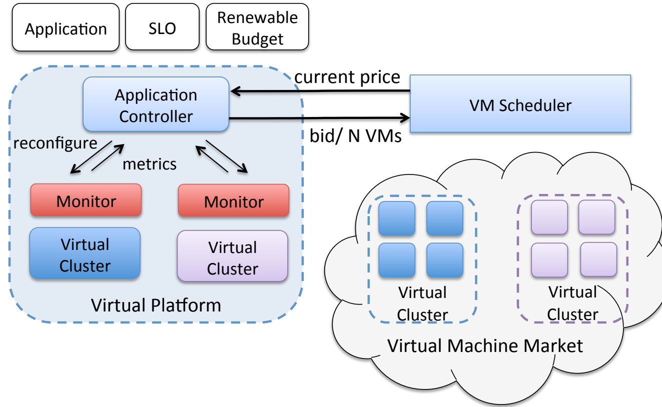


Figure 4: Virtual Platform overview.

3. Application Adaptation Policies and Use Cases

We demonstrate the case of virtual platforms in Merkat by proposing and applying 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 tenant'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.
- *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 tenant 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 a trigger: when the application performance metric crosses the thresholds, the policy takes an action that changes the virtual platform resource demand.

3.1. A 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 payed for them. This policy computes periodically the resource bids for each VM based on the following information: (i) current, minimum ($alloc_{min}$) and maximum ($alloc_{max}$) VM resource allocation; (ii) current

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: $T \leftarrow \left| \frac{v_{ref} - v}{v} \right|$

4: **for** $r \in resources$ **do**

5:     **if** ($v < v_{low}$ and $alloc[r] > alloc_{min}[r]$) or ($alloc[r] == alloc_{max}[r]$) **then**

6:         $bid_{tmp}[r] \leftarrow max(bid[r]/max(2, 1+T), bid_{min})$

7:         **if** $last_bid_change < 0$ **then**

8:             $\delta \leftarrow \left| bid[r] - bid_{tmp}[r] \right|$

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** $last_bid_change[r] > 0$ **then**

16:             $\delta \leftarrow \left| bid[r] - bid_{tmp}[r] \right|$

17:             **if** $\left| \frac{\delta - |last_bid_change[r]|}{\delta} \right| < 0.1$ **then**

18:                 $\delta \leftarrow \delta/2$

19:                 $bid_{tmp}[r] \leftarrow bid[r] + \delta$

20:             $bid[r] \leftarrow bid_{tmp}[r]$

    // bids are re-adjusted due to budget limitations

21: **if** $bid[memory] + bid[cpu] > bid_{max}$ **then**

22:     $w \leftarrow \emptyset$

23:     **if** $alloc[r] \geq alloc_{max}[r]$, $r \in resources$ **then**

24:         $w[r] \leftarrow 0$

25:         $bid_{max} \leftarrow bid_{max} - bid[r]$

26:     **else**

27:         $w[r] \leftarrow 1 - \frac{alloc[r]}{alloc_{max}[r]}$, $r \in resources$

28:     **for** $r \in$ resources with w[r]≠ 0 **do**

29:         $bid[r] \leftarrow \frac{bid_{max}}{\Sigma w[r]} \cdot w[r]$

30: return bid

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, $last_bid_change$, representing the difference between the current bid value and the previous one; (v) and the budget to be spent for the next time period, bid_{max}.

The parameters $alloc_{max}$ and $alloc_{min}$ can be determined by the tenant or the system administrator. For example, the tenant could know that a VM will never use more than 8 cores, thus $alloc_{max}$ can be set at 800 CPU units. Also, $alloc_{min}$ can be set by considering the minimum resource requirements of the VM, e.g., if the VM receives an allocation of less than 10

Algorithm 2 describes this policy. The policy works as follows. The given thresholds, together with the application allocation act as an alarm: when the application performance metric traverses them, the algorithm takes an action regarding the VM bid. For example, for an application which needs to finish its execution until a given deadline, the performance metric can be the (estimated) remaining application execution time while the upper threshold can be defined as v_{high} = 95% of the remaining time to deadline and the lower threshold as $v_{low} = 75\%$ of the remaining time to deadline. In this case, when the performance value is above the upper threshold, meaning that the application execution could exceed the deadline, or when the application receives an allocation below its specified minimum, the resource bids are increased. To optimize the cost per VM, when the allocation for one resource reaches the maximum, the bid increase for that resource stops. When the performance value drops below the lower threshold, meaning that the application could finish sooner than its deadline, the resource bids are decreased. To avoid cases in which the application receives a resource amount that is too small to allow it to make progress in its computation, its resource allocation is kept above a minimum value even when the performance value drops below the lower threshold. [3].

The value with which the bid changes is given by T (line 2), and it represents the "gap" between the current performance value and the performance reference value. A large gap allows the application to reach its reference performance fast. To avoid too many bid oscillations the algorithm uses the value of the past bid change, i.e., $last_bid_change$ 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 10, 18).

When the current budget is not enough to meet the performance reference value, bids are recomputed in a way that favors the resource (i.e., CPU or memory) with a small allocation, i.e., the resource with a small allocation represents a bottleneck in the application's progress. We consider that having lower application performance is still useful for the tenant, e.g., a scientific application finishing 90% of its computation before its deadline. Thus, the bids are increased using a proportional-share policy, where the resource "weight", $w[r]$ is the difference between the actual and maximum allocation of the VM (line 27) for the resource $r \in CPU, memory$. If there is a resource with a maximum allocation, the bid distribution becomes straightforward: the algorithm already decreased the bid for this resource, thus the other resource receives the all remaining budget (line 22). [4]

[3]Note that in this case the application controller might just as well suspend the application. This decision can be taken in a loop external to the algorithm.

[4]In this case the application controller could decide to suspend the application.

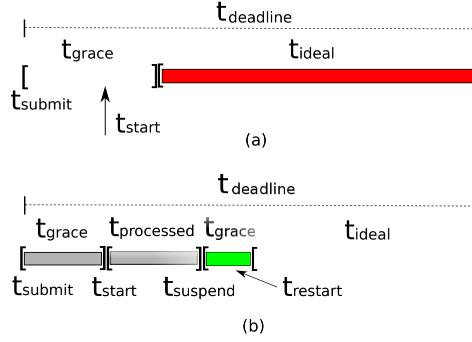


Figure 5: The time in which the application can start or resume.

3.1.1. Example: Deadline-driven Execution of a Static MPI Application

To give an example of how the vertical scaling policy can be used by an application controller, let us consider a tenant that wants to execute a static MPI application with *nvms* VMs and a budget *b*, renewed at a time interval t_{renew} with *r* credits. The SLO requested by the tenant is to finish the application execution before a deadline. The reasoning behind this policy, is that as long as its deadline can be met, the application can reduce its allocation to minimize its execution cost.

Application Model. We consider the following application model. The application is composed of a fixed number of tasks. Each task requires one cpu core and a specified amount of memory. We don't model the communication between tasks. To finish their execution the applications need to perform a certain computation amount (e.g., 1000 iterations). There is a large number of iterative applications (e.g., Zephyr [24]) that follow this model. These applications have a relatively stable execution time per iteration. The execution time of an iteration can be tuned by modifying the resource allocation that each task receives. For example, if each task receives one full core, one iteration can take 1 second. Nevertheless, if the resource allocation drops at half, the same iteration can take 2 seconds.

Terminology. Before explaining how an application controller uses the previously defined policy we give some definitions of the terms used. When submitting her application, the tenant needs to provide the ideal execution time of the application, t_{ideal}, the ideal execution time per iteration step, $t_{refideal}$ and the total number of iterations the application needs to perform, n_{steps}.

The terms that define the application performance are:

$n_{current}$ - the number of iterations the application has performed;

t_{step} - the execution time per iteration step;

t_{ref} - the reference time per iteration step: $t_{ref} = \frac{t_{deadline} - now}{n_{steps} - n_{current}}$;

The terms used in computing the risk of not meeting the deadline are illustrated in Figure 5:

t_{submit} - the time at which the application is submitted;

t_{start} - the time at which the application is started;

$t_{deadline}$ - the remaining time until the deadline set by the tenant; this value is initially set to $deadline - t_{submit}$; during the application execution is computed as $deadline - now$, where now represents the current time. if the execution time per iteration step is higher than t_{ref} then the application risks to miss its deadline.

t_{grace} - the time interval in which the application can be started or restarted without any risk of not meeting the deadline; When starting the application t_{grace} is computed as $t_{deadline} - t_{ideal}$, as seen in Figure 5(a). After the application started executing, t_{grace} is computed as $t_{deadline} - t_{refideal} \cdot (n_{steps} - n_{current})$ as noted in Figure 5(b). As t_{grace} is used now for resuming the application, we are interested in knowing in how much of the remaining time to deadline the remaining computation can be finished.

Algorithm. Now, let us describe the application controller behavior. The application controller starts the application when the resource price is low enough to afford a given allocation for each application VM. During the application execution, to keep the application execution time below a given deadline, the controller adapts the VM bid with the *vertical scaling policy*. The bid is adapted at each application monitoring period, based on the value of t_{step}. If the t_{step} is faster than the reference $0.75 \cdot t_{ref}$, the bid is decreased. Otherwise, if t_{step} is larger than the reference $0.95 \cdot t_{ref}$ the bid is increased. We use these thresholds to avoid too many bid oscillations, and thus, resource re-allocations.

The maximum limit at which the bid can be increased is given by bid_{max}, by distributing all the application's budget over the remaining time to deadline. We apply this distribution to avoid cases in which the application will run out of credits in the middle of its execution. As we assumed that the application receives a budget renewed with r credits every t_{renew}, the total budget the application receives until its deadline is: $(b + r \cdot (t_{deadline}/t_{renew})$.

As part of this adaptation policy, the application controller checks 3 conditions: (i) if it needs to stop the application execution; (ii) if it needs to suspend the application; (iii) if it can restart the application execution when the application is suspended. These conditions are checked as workload variation on the infrastructure leads to high prices. In this situations the application cannot meet its deadline and it is useless to spend more budget for its execution. The first condition becomes true when the application misses its deadline. The second condition happens due to high price periods. In a high price period the application controller cannot afford to keep the reference execution time per step, t_{ref}, and thus it suspends the virtual platform. Finally, the third condition is true when the price drops at an affordable value. If the application cannot resume in t_{grace} time then the controller stops its execution.

3.1.2. Adaptation of a Virtual Platform at Resource Demand Fluctuations

. To show how an application controller can adapt to changes in application resource allocation using the vertical scaling policy due to price variation, we ran a micro-benchmark using an MPI application, Zephyr [24]. Zephyr is a fluid dynamics simulation which runs for a tenant-defined number of iterations. Each iteration performs some

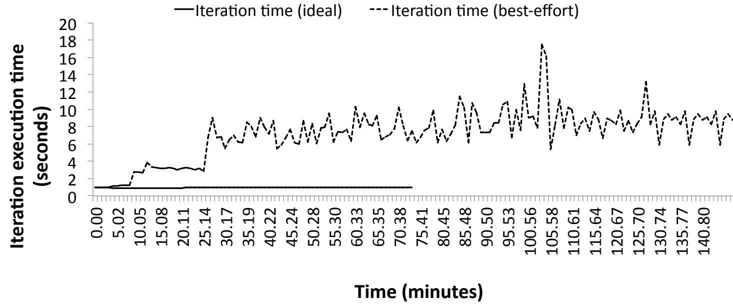


Figure 6: Application execution time variation with a best-effort controller.

computation, and if the application is started with multiple processes, also data is exchanged among them. Zephyr receives as input a configuration file and it simulates a volume filled by fluid for a specified simulated time, Zephyr periodically writes in a log file the following information: the CPU time for each iteration and the current number of iterations. For an SLO-driven Zephyr application, the application controller reads periodically the CPU time for each iteration and it compares it with a reference CPU time computed such that the application finishes its iterations at the tenant's given deadline.

We started an SLO-driven application and four best-effort applications (Zephyr applications running with a controller that does not change the bid during the application runtime) on a node, each in a VM with 4 cores. The SLO-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 bid of 100 CPU credits and 900 memory credits which remains unchanged during their execution. For this setup we used a node from our cloud. The SLO-driven application controller was started at the beginning of the experiment, while the other four application controllers were started during its execution. The last best-effort application was submitted after 20 minutes from the experiment start. The SLO-driven application has an ideal execution time of 77.5 minutes.

For clarity, we first run the Zephyr application with a best-effort controller instead of a SLO-driven one. Figure 6 shows the progress the application makes over time, i.e., its iteration execution time: (i) when it runs alone on the node (the ideal iteration time); (ii) and after the other applications are submitted (the best-effort iteration time). The performance difference in this case is highly noticeable: after all applications started executing, the iteration execution time increased almost 10 times. This degradation is not only due to its reduced resource allocation but also due to the other applications.

Then we ran the application with three different deadlines: 12000 seconds, 9000 seconds and 6000 seconds. We repeated each experiment three times and computed the average of the obtained values.

Figure 7 shows the bid of the SLO-driven controller for CPU resource for the different application deadlines. The bid stabilizes after all the applications were submitted. The application with the smallest deadline demands a maximum allocation and thus, the submitted bid is also much higher than in the other two cases.

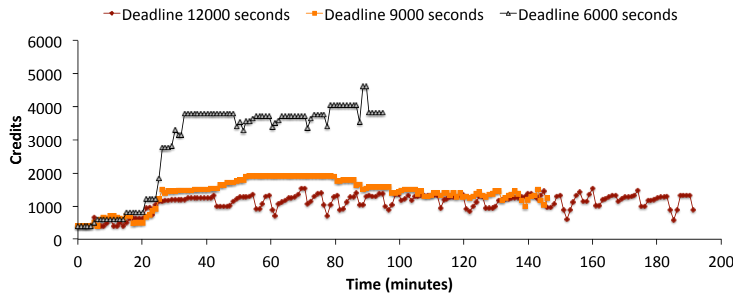
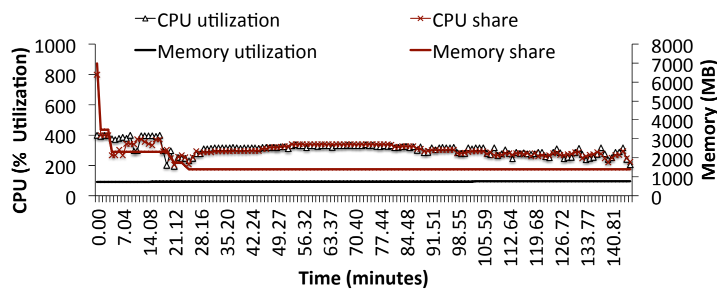
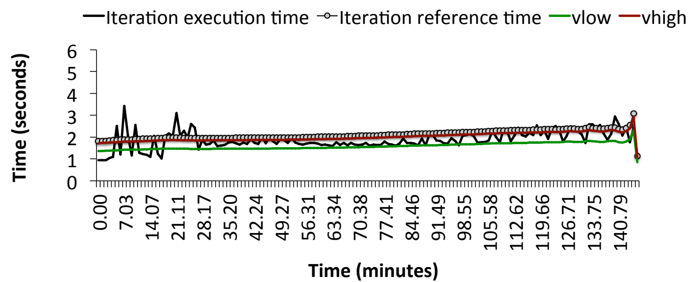


Figure 7: CPU bid variation for three different deadlines.



(a) Resource share vs utilization



(b) Iteration execution time variation

Figure 8: Application resource and performance variation when running with a deadline of 9000 seconds: (a) shows the variation in resource allocation (also called share) and utilization and (b) shows how the controller keeps the perfomance, defined as iteration execution time, between the defined limits.

Figure 8(a) shows the variation in the resource allocation (also called share) and respectively utilization for the application when it is run with a deadline of 9000 seconds. As the behavior is similar for the other two tested deadlines, we omit depicting them. Remember that the VM resource share is the proportional-share the VM gets according to its bid. The resource utilization is how much the application inside the VM consumes. The left axis shows the CPU resource, i.e., percentage of total CPU time, while the right axis shows the memory resource, i.e., in MBs. The best-effort application arrivals can be noticed by looking at the changes in the SLO-driven VM resource share. In this case, the memory share reflects the best these arrivals. When the application started alone on the node the VM's share is the entire node capacity. After each application arrival, this share decreases until it equalizes the VM utilization. After all

the best-effort applications started running, the SLO-driven controller keeps a reduced CPU share as the application can meet its deadline.

Figure 8(b) shows the variation in the SLO-driven application iteration execution time. To understand the behavior of the vertical scaling algorithm, we also give the lower and upper scaling thresholds, v_{low} and v_{high}. We notice that after all the best-effort applications started running, the application controller manages fairly well to keep the iteration execution time between these two thresholds. However, there are cases in which the iteration execution time oscillates. We think that one cause for this variation is the sharing of physical cores between more VM processes. The SLO-driven application receives more CPU than the best-effort applications, and thus less of the VMs in which these best-effort applications run get scheduled on the same physical cores as its own.

3.1.3. Adaptation of a Virtual Platform at SLO Modification

. Merkat's application controller can also react to a tenant imposed condition. To show how it does so, we ran a SLO-driven Zephyr application and changed the tenant-specified deadline during the application execution. 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 SLO-driven application has a budget of 60000 credits, from which it can spend as much as it wants, while the other best-effort applications use a fixed bid of 100 CPU credits and 900 memory credits per VM.

The SLO-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. The application controller's bid adaptation and the allocation changes can be noticed in Figure 9. Figure 10 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 10 from the change in the reference time), 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.

3.2. A Horizontal Scaling Policy

Some tenants, instead of scaling vertically the resource allocation of a VM, may prefer to set a fixed allocation for it and to scale the number of VMs instead. For example, an elastic framework (e.g., Condor, Hadoop) might make better use of VMs with fixed CPU allocations (e.g., 1 core per VM). Such a framework might expand its resource demand when the infrastructure is free and shrink it when the infrastructure is contended. In this case it would be more convenient to give a hint to the tenant on how many VMs she can provision in the next time period of the

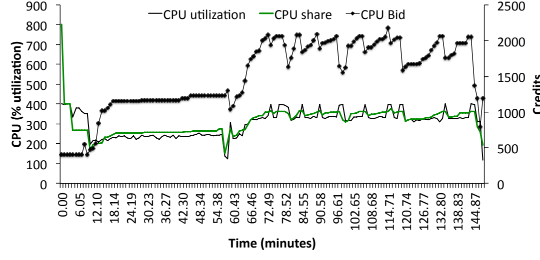


Figure 9: Application's CPU utilization and bid variation due to adaptation to the new deadline.

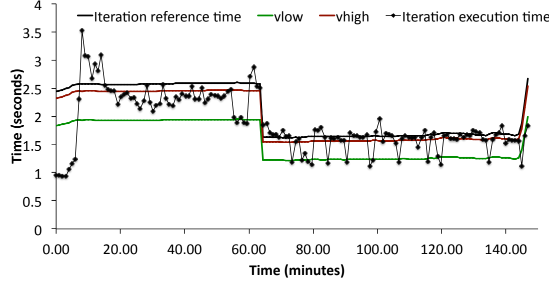


Figure 10: Application's iteration execution time variation due to adaptation to the new deadline. The reference iteration execution time and the two thresholds used in Algorithm 2 are also shown.

application execution. Algorithm 3 describes the method to obtain this hint. Because the proportional-share allocation policy does not provide allocation guarantees, to keep an allocation for each VM closer to the maximum allocation the resource bid needs to be adjusted to resource price fluctuations. This bid value depends on the tenant's available budget and the prices for two resources (CPU and memory).

Algorithm 3 VM upper bound computation

1: **GetUpperBound**(P, $nvms_{max}$, bid_{max}, $alloc_{max}$)

2: $resources \leftarrow \{cpu, memory\}$

3: find maximum value of N $\in [1, nvms_{max}]$ for which $\sum_{r \in resources} \cdot \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}} < bid_{max}$

4: $nvms \leftarrow N$

5: $bid[r] \leftarrow \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}}$, $r \in resources$

6: return (nvms, bid)

The algorithm receives the resource prices, as a vector called $P[r]$, where $r \in \{cpu, memory\}$ the requested number of VMs by the tenant, $nvms_{max}$, and the tenant budget, bid_{max}. The output of the algorithm is a number of VMs, such that each VM might receive an allocation close to a maximum for all its resources at the current price. The value of the bid for each resource can be easily computed from Equation 1. To find the upper bound on the number of VMs, the algorithm performs a binary search between 1 and $nvms_{max}$ by checking at each iteration if the sum of bids the controller needs to submit is less than its remaining budget.

Algorithm 4 Horizontal scaling adaptation policy

1: **HorizontalAdaptation(**$nvms_{old}, bid_{max}, alloc_{max}, v, v_{low}, v_{high}$**)**
2: **if** $v > v_{high}$ **then**
3: $\quad nvms \leftarrow nvms + 1$
4: **if** ($v < v_{low}$ **then**
5: $\quad$ pick vm to release
6: $\quad nvms \leftarrow nvms - 1$
7: $(N, bid) \leftarrow$ **GetUpperBound(**$nvms$**,** bid_{max}**,** $alloc_{max}$**)**
8: **if** $N < nvms$ **then**
9: $\quad$ release (nvms - N) VMs
10: **else**
11: $\quad$ request new $nvms - nvms_{old}$ VMs
12: return $nvms$

The tenant can compute the number of VMs required to meet its application performance goal using the output of the Algorithm 3. To illustrate the use of this policy, Algorithm 4 describes a straight-forward method that scales horizontally the application. 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 tenant's budget constraints, the maximum number of VMs for the next time period is computed using Algorithm 3.

3.2.1. Example: Elastic Execution of a Task Processing Framework

To illustrate the use of a horizontal policy, we consider a tenant that wants to execute bag-of-tasks applications on the infrastructure. The SLO for this application type is to minimize the workload processing time.

Application Model. To manage the bag-of-tasks execution the tenant uses a task processing framework (e.g., Condor) for which it assigns budget at a rate b. The framework has its own scheduler that manages the framework's resources. This scheduler receives application submission requests, keeps them in a queue and dispatches them to nodes when resources become available. At any time, the tenant can retrieve information about the number of running and queued tasks.

Algorithm. To minimize the workload completion time for this framework, a horizontal policy can be designed with the use of Algorithm 4 to provision/release VMs. The policy can provision extra VMs as long as there are tasks in the framework's queue and the framework's budget is enough. If the tenant cannot afford the current number of VMs or no tasks are left in queue, VMs are released.

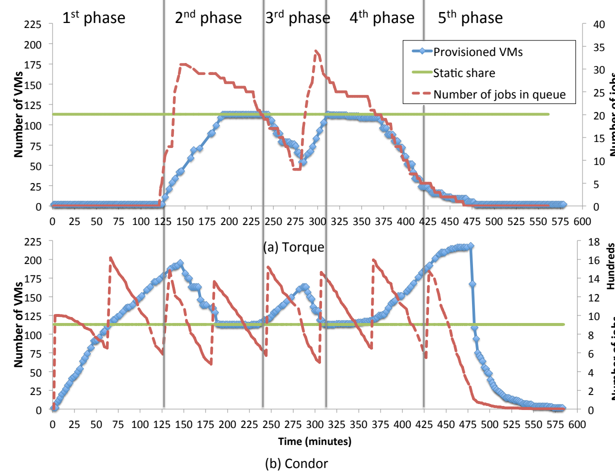


Figure 11: The variation of provisioned VMs versus queued jobs for each framework.

3.2.2. Adaptation of Two Different Virtual Platforms

. Let us illustrate the use of such a policy in a scenario in which a cluster needs to be shared between two commonly used frameworks: Condor [25] and Torque [26]. Tenants 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 that uses Algorithm 4 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. 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 11 shows the number of running and queued jobs in the Torque/Condor's queues and the number of VMs

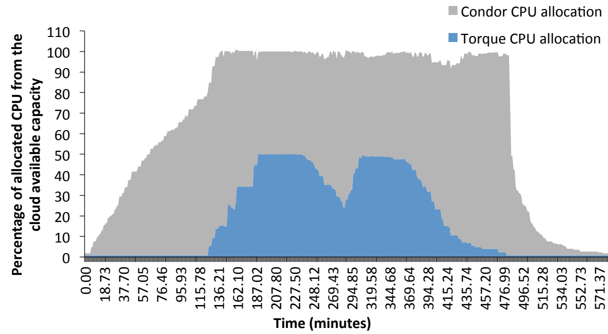


Figure 12: The variation of CPU allocation for both frameworks, as a percentage of the total available CPU capacity of the cloud. We omitted the variation of memory allocation as it is similar.

provisioned over time. If each framework is assigned an equal share of the infrastructure, it would obtain a maximum of 112 VMs (green line from Figure 11). We divided our experiment in five phases. In the first, third and fifth phase 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 12 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 the horizontal scaling policy allows applications to expand and shrink their resource demand according to the resource availability of the infrastructure, thus improving the resource utilization.

3.3. Discussion

We have seen that the virtual platforms react well to both changes in system workload and tenant requirements. Given that the system is not highly dynamic, and the application controller has time to adapt, the application can run with a smaller resource allocation and optimize its budget. The application controller optimizes the application execution cost while allowing other applications with less budget to use resources. However, our previously-presented policies do not cover all the possible cases. For example, the tenant's deadline might be too strict for the current price. In this case, the application controller might need to estimate the future infrastructure load and send feedback to the tenant regarding the minimum deadline it can meet. Starting from the simple mechanisms presented here, to improve the SLO support more complex policies can be developed, based on application profiling and price prediction.

Finally, in both our policies, the controllers use VMs with a predefined maximum size. This size can be set up by the tenant, if for example she knows that her application will never use more than that, or it can be configured by the cloud infrastructure, e.g., as the capacity of the node. Nevertheless, the VM resource allocation changes dynamically,

as the vertical scaling policy can steer it during the application runtime.

4. Implementation and Performance Evaluation

In this Section, we analyze the performance of Merkat in terms of total tenant satisfaction when applications adapt their resource demands to track a given SLO, and we show its flexibility in supporting different tenant types. Tenant satisfaction is an important metric regarding the performance of a resource management system. This satisfaction depends whether that the tenant's SLO is violated or not and on how much the tenant actually valued the execution of her application.

We have implemented a prototype of Merkat and validated it through simulation and on the Grid'5000 testbed [18]. We used simulation to test Merkat's algorithms with a large workload. Running a large workload gives a better insight in the total satisfaction that the system can provide. Due to time and resource limitations, running a large workload on Merkat in a real environment, which lasts for days, or possibly weeks, and with a large number of nodes, would have been unpractical. We discuss next the implementation of Merkat. Then we describe the results we obtained from measuring the total tenant satisfaction provided by Merkat in simulation and on a real-world testbed.

4.1. Performance Results from Simulation with Large Traces

We measured the performance of our system in terms of total tenant 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. Other deadline-based algorithms are available in the state of the art, but a large majority is specific to one application type, e.g., bag of tasks, workflows, environment, e.g., public clouds, or are offline solutions. Moreover, it is difficult to choose among these different algorithms the most representative one. Thus, we chose the EDF policy as it is well-known in the state of the art and is often used as a comparison baseline. It is important to note that cloud managers cannot practically apply EDF or similar algorithms without limiting their support to a predefined set of application goals (e.g., meeting deadlines). Nevertheless, we wanted to compare our system with a centralized system that targets a fixed type SLO.

4.1.1. Simulation Setup

We implemented the algorithms of Merkat in CloudSim [17], 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. To model the applications, we consider the batch application presented in Section 3.1. We simulate

these applications as sets of tasks, with each task requiring one CPU core and a specified amount of memory. The application applies the vertical scaling policy and interacts with the datacenter to change the bids for its VMs. As CloudSim does not model the cost of VM operations, we have also implemented a model for several VM-related performance overheads: memory over-commit, VM boot/resume and VM migration [16].

Tenant Model. To measure the total tenant satisfaction, we model the tenant satisfaction as a utility function of the budget assigned by the tenant to its application and the application execution time. There are different functions which can be used to model this satisfaction, and they depend on the behavior of the tenant. In this paper we define several functions, derived from discussions with scientists at Electricité de France:

- **Full deadline tenants** A common case is of a tenant who wants the application results by a specific deadline. If the application doesn't finish its execution at the deadline, the tenant is unsatisfied.
- **Partial deadline tenants** Some tenants might value partial application results at their given deadline; for example, for a tenant who implemented a scientific method and needs to run 1000 iterations of her simulation to test it, finishing 900 iterations is also sufficient to show the good method behavior.
- **Full performance** Finally, other tenants want the results as soon as possible, but they are also ready to accept a bounded delay. The upper bound of the delay is defined by the application deadline. For example, a developer wants to test a newly developed algorithm. She wants the results as fast as possible, but if the system is not capable to provide them, she might be willing to wait until the morning.

Before discussing the signification of utility functions, we define the following terms: t_{exec} is the application execution time; $t_{deadline}$ is the time from the submission to deadline; t_{ideal} is the ideal execution time, i.e., when the application runs on a dedicated infrastructure; *work_done* represents the number of iterations the application managed to execute until it was stopped; *work_total* represents the total number of iterations; B is the applications budget per budget renewal period and per task. B is assigned by the tenant and reflects the applications importance.

Table 1 summarizes the used utility functions. The full deadline tenant values the application execution at her full budget if the application finishes before deadline. Otherwise, we express her dissatisfaction as a "penalty", which represents the negative value of her budget. The partial deadline tenant is satisfied with the amount of work done until the deadline. Thus the value of her satisfaction is proportional to this amount. The full performance tenant becomes dissatisfied proportionally to her application execution slowdown. We bound the value of her dissatisfaction at the negative value of her budget.

Application Policy. For each tenant type we derive an application-specific execution policy from the vertical scaling policy, presented in Section 2.3, as follows:

- **Full deadline:** The policy is derived from the Algorithm 2. Applications start when the price is low enough to ensure a good allocation. During their execution they adapt their bids and use suspend/resume mechanisms to

tenant Type	Utility Function
full-deadline	B, if $t_{exec} \leq t_{deadline}$, -B otherwise
partial-deadline	B, if $t_{exec} \leq t_{deadline}$, $B \cdot \frac{work_done}{work_total}$ otherwise
full-performance	B, if $t_{exec} = t_{ideal}, max(-B, B \cdot \frac{t_{deadline}+t_{ideal}-2 \cdot t_{exec}}{t_{deadline}-t_{ideal}})$ otherwise

Table 1: Utility functions.

keep a low price in low utilization periods and to use as much resource as their SLO allows in high utilization periods. If, during its execution, the application sees it cannot meet the deadline it stops.

- **Partial deadline:** This policy is similar to the previous policy, but there are two differences: (i) the application suspends when a minimum allocation cannot be ensured (e.g., 30% cpu time or 30% physical allocated memory); (ii) as any work done at the deadline is useful, the application always runs until its deadline
- **Full performance:** This policy is designed for full-performance tenants. The policy is similar to the **Full deadline** policy. Nevertheless, during its execution, the application, instead of tracking a performance reference metric, it tracks a reference allocation defined as the maximum used by the application VM. When the application cannot have a minimum allocation at the current price, the application suspends.

4.1.2. Workload

To evaluate the system performance we use the HPC2N workload trace from Parallel Workloads Archive [27], containing 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. As the memory information was not specified for all the applications missing memory requirements were completed by assigning 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 tenant wants to pay depends on the application's deadline: a tenant 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.

4.1.3. Results

To see how the performance is influenced by the contention level, we measure the obtained satisfaction for each of the 10 obtained workload traces and for full deadline tenants. We repeated each experiment four times and the results

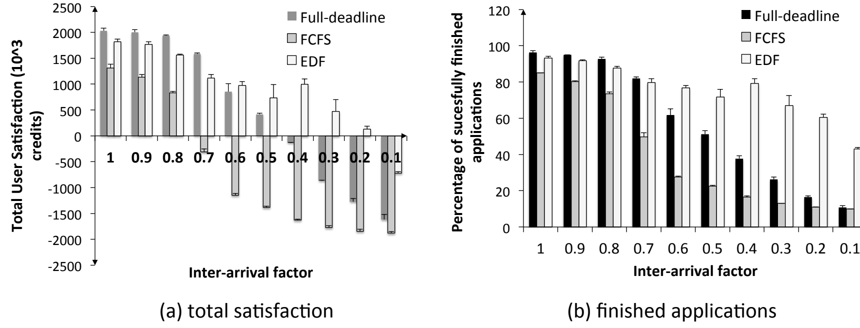


Figure 13: Proportional share market performance in different contention scenarios in terms of: total tenant satisfaction (a) and percentage of successfully finished applications (b). The contention increases from left to right.

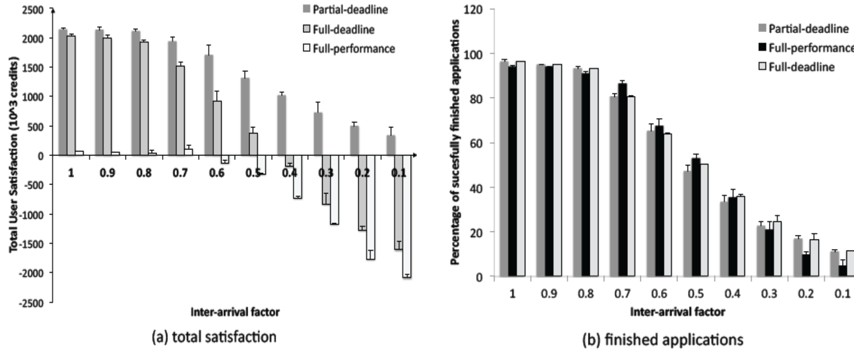


Figure 14: Proportional share market performance for different tenant models in terms of: total tenant satisfaction (a) and percentage of successfully finished applications (b). The contention increases from left to right.

are presented as the mean plus standard deviation.

Figure 13 (a) describes the results of our comparison. For clarity, Figure 13 (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 tenant 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, more applications arrive at the same time and, EDF is capable to take better scheduling decisions: thus more applications with smaller deadlines, get to run on time. Because the deadline urgency is reflected in the application's value, EDF also leads to higher tenant satisfaction. 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" payed by the nature of our system, which allows applications to behave selfishly.

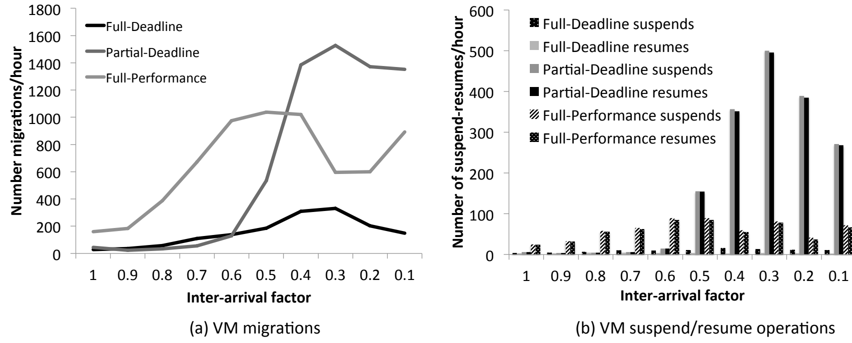


Figure 15: Number of VM operations for each tenant utility policy and for different contention levels. The contention increases from left to right.

We also measured the total tenant satisfaction when tenants have different models for the satisfaction they get from their application execution. Figure 14 describes the proportional-share market performance for each of the different previously discussed tenant models, in terms of total tenant satisfaction and, for clarity, number of successfully finished applications (i.e., their deadlines were met). Figure 14(b) describes the total satisfaction that our system provides to tenants when applications use the each of the three different policies. The best satisfaction is provided by the partial deadline policy, as tenants still gain a positive value from getting the results at their deadline, despite of not having all of them. The worse satisfaction is provided by the full performance policy, as tenants are also more demanding: they get dissatisfied really fast and they perceive a negative value if their jobs don't finish at their deadline. Figure 14(a) shows the percentage of finished applications for each policy. It is interesting to see that in all cases, the percentage remains almost the same. Nevertheless, fewer applications finish in the case of partial deadline tenants, as applications are not stopped before their deadline.

As the application adaptation algorithms lead to VM operations, we measure their cost. We keep the same settings from the previous experiments and we record the number of VM operations during the experiment run. Figure 15 describes the average number of VM operations per hour performed by the VM Scheduler for different tenant utility functions. We notice that the Full-Deadline policy is the most effective as it has the least number of VM operations. This is due to the fact that with this policy the applications are less aggressive in acquiring resources. We notice that in the most highly loaded periods the number of migrations decreases. This is explained by the fact that when there is a high load, many applications don't resume and even don't start their execution as their deadline cannot be met. The Partial-Deadline policy has the most migrations, number which increases with the contention, as even in highly contended periods applications continue running with smaller resource amounts. Moreover, applications start executing whenever they can get a small resource amount. This leads to more bid adaptations, and thus resource allocation changes and more migrations. The number of suspend/resume operations follows the same trend. The reason behind this is that applications resume at a small allocation, as the policy considers that any progress the application makes is useful. The Full-Performance policy follows a similar trend as the Partial-Deadline policy but the

number of migrations becomes higher even when there is not much contention. This is explained by the aggressiveness of the application adaptation policy that always increases the bid to get as much resource as possible.

The results of these experiments show that Merkat can accommodate different tenant types on the same infrastructure while providing good satisfaction.

4.2. Performance Results from Testbed Experiments

To measure the tenant 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 [28]. Maui is often used by HPC organizations to manage their clusters. To run applications, Maui uses algorithms based on FCFS, with backfilling.

4.2.1. Merkat Prototype and Testbed

Merkat is implemented in Python and depends on the Twisted [29], paramiko [30] and ZeroMq [31] 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 [20] 4.6. 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.

We deployed both Merkat and Maui on a cloud of 10 compute nodes on the edel cluster of the Grenoble site from Grid'5000. When using Maui, we deploy the VMs and add them to Maui as physical nodes at the beginning of the experiment. Each node has 2 Intel Xeon E5420 QC processors (4 cores) and 24GB of memory. The VM Scheduler of Merkat 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. The VM images are stored on a NFS server. To speed-up the VM deployment we use copy-on-write VM images. Merkat's VM Scheduler uses a scheduling period of 60 seconds and the application controllers read the application's performance metrics at every 80 seconds. We wanted a scheduling period that is small enough to allow applications to adapt fast their bids, and large enough to allow the VM migrations to finish. 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 [32] as a hypervisor on each node.

4.2.2. Workload

We run 160 Zephyr applications over a time interval of 7 hours, with 1 to 8 processes, each process in one VM and we consider full deadline tenants. We used a Lublin [33] 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.

4.2.3. Results

Platform	% Met Deadlines	Satisfaction	App Suspend/Resume	Migrations/hour	VM Suspend/Resume
Merkat	82.5%	764330	51/51	310	140/140
Maui	58.1%	-178581	-	-	-/-
Merkat/CloudSim	94%	950554	40/40	62	103/103

Table 2: Metrics reflecting the behavior of Merkat compared to Maui and to its simulation implementation in CloudSim.

Table 2 summarizes the results of our experiment. We repeated the experiment 4 times and we averaged the following metrics: (i) the number of successfully finished applications; (ii) the total satisfaction provided by Merkat and Maui; (iii) total number of applications suspended/resumed (iv) number of VM migrations; (v) number of VM suspend/resume operations. We notice that during the experiment 51 applications were suspended and succesfully resumed, with an average number of 140 VM suspend/resume operations. In the same time we recorded an average of 310 VM migrations/hour. When Merkat is used to manage the cloud, the percentage of applications which finish until their deadline increases with 24.1% than when using Maui. Merkat also leads to more tenants being satisfied from running their applications on the cloud than traditional systems like Maui (in Merkat the total tenant satisfaction is 764330 credits, compared to a negative satisfaction of -178581 credits from Maui). This result provides evidence of the effectiveness of Merkat in managing a real cloud.

To analyze the differences between the simulation and the real testbed setup, we also compared the results obtained from the Merkat prototype with the simulation run on the same workload trace (we performed 4 runs and averaged the results). The difference in the number of applications which meet their deadlines in Merkat and CloudSim is 11%, with more applications meeting their SLOs in CloudSim, thus leading to higher user satisfaction. The behavior of Merkat also leads to a higher number of suspended/resumed applications (11 more applications in Merkat), as well as number of VM suspend/resume operations (37 more VM suspend/resume operations in Merkat) together with a significant number of migrations (310 compared to 62).

We believe this overall difference in the number of operations comes from the lack of accurate application modeling in CloudSim. In our experiment with Merkat we use Zephyr as an example application, which uses MPI for process communication. CloudSim relies on much simpler application model, in which the application performance is not influenced by network sharing and performance interference is not reliable modeled. Thus, when running applications on Merkat, Merkat's algorithms have to cope with higher variability in the application performance metrics, which leads to higher variability in bids and resource allocations and thus a higher number of migrations. This number can be reduced by tolerating higher allocation errors in the VM placement algorithm.

5. 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 tenants remains a difficult task. When designing Merkat we encountered two issues: (i) deciding the total credit amount that circulates in the system; (ii) deciding the time period over which the tenant budgets are renewed.

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 tenant satisfaction. If the credit amount is small and new tenants arrive, current tenants might be funded at a rate too small to allow them to run highly urgent applications. In Merkat setting this amount can be done by the infrastructure administrator based on the number of tenants in the system, the price history and the capacity of the infrastructure. If the administrator observes that the price increased considerably in the last period, she can reduce the total credit amount. If new tenants 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 tenants regarding their resource usage versus assigned budget.

The time period over which the tenant budgets are renewed influences the tenant behavior. If the time period is small, tenants will not have incentives to save budget, and thus, to judiciously use resources. If the time period is too large, tenants 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 tenants.

System Stability. Poor adaptation policies, currency management policies or a too small scheduling interval might lead to system thrashing, i.e., the system would spend too much time adapting, wasting resources and leading to poor application performance. 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. 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 in placing communicating application processes as close as possible to each other. Currently, Merkat does not consider the application's or the cluster's topology. However, it can be further extended to allow tenants to express and pay for running their application processes as close as possible to 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 tenants with a production system, Merkat services require self-healing capabilities to make the platform highly available. State-of-the art solutions [34] can be used to achieve these goals.

6. 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.

6.1. Resource Management Systems for Clusters

The emergence of new workload types, like data analytics, raised multiple issues regarding how resources of a cluster should be managed. Currently, clusters need to be shared between multiple tenants, each requiring different frameworks, and having various application performance constraints, e.g., high availability, throughput, latency. Supporting this variety of requirements can lead to a reduced resource utilization of the cluster and inefficient energy consumption. In this context, there are substantial efforts in the state of the art towards designing resource management systems to cope with these issues.

A first class of resource management systems focuses on the flexibility of sharing the cluster between applications requiring different frameworks, with the goal of maximizing the cluster utilization. Mesos [1] 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 [2] 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, and selects from it what resources to use. In Yarn [3] each application has its own application manager that requests resources from a global scheduler, which allocates resources by considering specific application constraints. Borg [35] is a resource manager developed to support different data analytics workloads running on Google's infrastructure. Borg considers a workload composed

of long running critical services and short running best effort jobs. In Borg the infrastructure, composed of multiple clusters, is divided in cells, a cell having the dimension of an entire cluster. Each cell is managed by a Borgmaster which schedules tasks on the available nodes. Tasks are placed on the node with the best score, where the score is computed by considering tenant preferences, high availability, data locality as well as number of possibly preempted tasks and resource fragmentation. Although these systems have flexible architectures, which can support various application types, they do not consider the individual application performance or tenant SLOs. Moreover, when handling contention, these systems rely on priorities or tenant quotas. Apollo [9] is the system used to run data analytics workloads on Microsoft's infrastructure. Apollo manages jobs composed of complex DAGs. The scheduling of each job's tasks is performed independently, using a model similar to the cell state from Omega, in which each job can have an overview of the cluster state. In Apollo, each node also makes available information about future wait time inferred from its task queue, which is then used by the job to place its tasks on nodes. The jobs are classified in latency-sensitive and best-effort, with the tasks of best-effort jobs being scheduled opportunistically, e.g., when resources are idle, on nodes.

Other resource management systems focus on minimizing the energy consumption of the infrastructure [36, 37, 38, 39]. The common approach is to consolidate applications on as few nodes as possible and shut down the idle resources. In this paper we focus on maximizing the resource utilization of the infrastructure while making users to take educated decisions regarding their used resources (through the use of currency).

Finally, other cluster resource management systems focus on providing SLO support. These systems use controllers which allocate resources to applications based on obtained, offline and/or online, performance models. Applications are usually allocated enough resources to meet their SLOs; thus more applications run on the infrastructure and the resource utilization is improved. Earlier systems focused on web applications [40, 41, 10]. 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 [42], or proactively [13]. CloudScale [13] adapts the resource demand of VMs proactively and reactively. The proactive adaptation is done through padding the estimated resource demand of the application running in the VM. The reactive adaptation is done when the resource demand was under-estimated and the resource allocation of the VM needs to be increased. CloudScale leverages its long-term resource demand prediction algorithms to decide when and what VMs to migrate. [11] proposes an architecture based on ”escalation levels”, which divide the actions involved in the resource management process. At the lowest escalation level, the resource configuration of the VM is changed. If this action is not possible, application components are migrated to other available VMs. Then, new VMs are deployed, new nodes are turned on, if nodes are available they are turned on or VMs from public clouds are rent. Although the idea is interesting, in the end, the authors address only the first escalation level by proposing rules to scale the resource allocation of the VM based on utilization thresholds. Policies to mediate resource conflicts between multiple applications are missing (e.g., before renting VMs from public clouds, the allocation of other existing VMs could be shrunk). iSSe [12] is a project which scales the application resource allocation in two ways: (i) first it scales up and down the resource demand

of each VM; (ii) and if the first scaling is not sufficient, it scales the number of VMs. Here the idea is to perform fine-grained resource allocation by re-distributing resources among the VMs belonging to the same application before taking resources from other applications.

Self-tuning virtual machines [43] run deadline-driven scientific applications in a fully decentralised system where applications are altruistic and adapt with no knowledge one from another. Applications run in single VMs on a node. A feedback controller tunes the CPU allocation of the VM such that the application running inside makes enough progress in its computation to meet its deadline. In this way, more applications can run on the infrastructure. A global admission control is used to avoid overloading the system.

More recent resource managers also focus on data analytics applications. Quasar [7] is a resource manager which uses classification techniques to determine the amount of resources required by each application to meet a specified performance and also the performance interference between applications. Heracles [8] targets latency-sensitive applications and dynamically allocates resources like CPU, memory and network to them while also co-locating best-effort applications to maximize the cluster utilization. Heracles uses online monitoring and offline profiling information to take allocation decisions. Bubble-Flux [15, 44] is a resource manager which uses on-line profiling to measure the performance interference between latency-sensitive and batch applications. In all of these systems there are only two types of workloads: best-effort and SLO-driven, where the SLO is a target latency.

Opposed to these systems, Merkat uses a market to distribute resources, leading to a more fine-grain differentiation between application priorities. Moreover, Merkat is designed to be generic enough to support more than one SLO and application type. Nevertheless, Merkat can leverage in its application controllers many of the algorithms proposed in such papers, regarding application performance profiling and avoidance of performance interference between different application types.

6.2. PaaS Systems

Many PaaS systems, commercial [45, 46, 47, 48, 49], research [50, 51, 52], and open source [53, 54, 55, 56] provide runtime support for applications hiding from tenants the complexities of managing resources. These systems, however, provide typically closed environments, forcing tenants 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 tenants the possibility to use it. In contrast to these systems, Merkat allows tenants to run new application types while distributing the resources among them based on their value.

Similar to Merkat, Meryn [57] 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.

6.3. 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 [58]. The reason why markets became so popular in the distributed systems community is the notion of *cost*, which makes tenants more aware of how many resources to acquire for their own use. Based on the pricing model, there are two commonly used market models: commodity markets and auctions.

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 tatonnement algorithms to adjust the resource price to reduce the excess demand close to zero [59]. These algorithms either use estimations of the excess demand [60], or they rely on the participants to send their demand as a function of price [61]. For scalability reasons, resources are allocated in a coarse-grain way, i.e., in terms of CPU slots or number of nodes. One particular case is Libra [62], which allocates CPU proportionally to the application's deadline on each node while a global pricing mechanism, based on the infrastructure utilization, is used to balance supply with demand. The use of the proportional-share policy maximizes the infrastructure utilization, as opposed to a case in which applications are allocated exclusive-access to nodes. Admission control ensures that the applications accepted in the system do not lead to deadline misses for the other applications. However, the same admission control mechanism might prevent urgent applications for running while less urgent applications occupy all the resources. Aneka [63] 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 tenants are willing to pay for resources. Auctions clear the market faster than the price computation algorithms from the commodity markets, allowing tenants 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 [64, 65] or to run bag-of-tasks applications on grids [66]. In this case, tenants bid for an application execution and the scheduler decides which application gets to run through the auction. Nevertheless, these proposed systems address the case of applications with known execution times and static resource requirements. In Popcorn [67] and Spawn [68] applications composed of many tasks can shrink when the resource price is high and expand when the resource price is low. Systems like Tycoon [22] or REXEC [69] implement a proportional-share policy per node to allocate fractional amounts of CPU. Ginseng [70] is a cloud platform that uses an auction to allocate memory to VMs on a node. 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 tenants with a complete system. A dynamic priority scheduler is proposed for Hadoop [71], to allocate map/reduce slots to applications using proportional-share. tenants are assigned a budget and they spend it to run applications on the cluster, by specifying a spending rate, i.e. the tenant's willingness to pay for a slot and for a time period. The budget and the spending rate allow the tenant to control how many slots get assigned

to her applications over time, and thus, to control the application 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 tenant SLOs.

7. 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 tenant 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 tenants 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 tenant's SLO under current price and tenant budget constraints. Merkat supports different applications and tenant 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 tenant satisfaction; (iii) the resource control decentralization has a reasonable performance impact compared to centralized resource management systems.

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] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, 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, USENIX Association, Berkeley, CA, USA, 2011, pp. 22–22.
URL http://dl.acm.org/citation.cfm?id=1972457.1972488

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

[3] 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, E. Baldeschwieler, Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the2013 ACM Symposium on Cloud Computing, SOCC'13, 2013.

[4] OpenStack, http://www.openstack.org/.

[5] A. Gulati, G. Shanmuganathan, A. Holler, I. Ahmad, Cloud-scale resource management: challenges and techniques, in: Proceedings of the 3rd USENIX conference on Hot topics in cloud computing, USENIX Association, 2011, pp. 3–3.

[6] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, A. Rowstron, Scale-up vs scale-out for hadoop: Time to rethink?, in: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13, ACM, New York, NY, USA, 2013, pp. 20:1–20:13.

[7] C. Delimitrou, 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, ACM, New York, NY, USA, 2014.

[8] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, C. Kozyrakis, Heracles: improving resource efficiency at scale, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ACM, 2015, pp. 450–462.

[9] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, L. Zhou, Apollo: scalable and coordinated scheduling for cloud-scale computing, in: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014, pp. 285–300.

[10] Z. Wang, Y. Chen, D. Gmach, S. Singhal, B. Watson, W. Rivera, X. Zhu, C. Hyser, Appraise: application-level performance management in virtualized server environments, IEEE Transactions on Network and Service Management 6 (4) (2009) 240 –254.

[11] M. Maurer, I. Brandic, R. Sakellariou, Enacting slas in clouds using rules, in: Euro-Par 2011 Parallel Processing, Springer, 2011, pp. 455–466.

[12] R. Han, L. Guo, M. M. Ghanem, 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, IEEE, 2012, pp. 644–651.

[13] Z. Shen, S. Subbiah, X. Gu, J. Wilkes, Cloudscale: elastic resource scaling for multi-tenant cloud systems, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11, ACM, New York, NY, USA, 2011, pp. 5:1–5:14. doi:10.1145/2038916.2038921.
URL `http://doi.acm.org/10.1145/2038916.2038921`

[14] H. Fernandez, C. Stratan, G. Pierre, Robust performance control for web applications in the cloud, in: Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER), 2014.

[15] H. Yang, A. Breslow, J. Mars, L. Tang, Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers, ACM SIGARCH Computer Architecture News 41 (3) (2013) 607–618.

[16] S. V. Costache, N. Parlavantzas, C. Morin, 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.

[17] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, 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) (2011) 23–50.

[18] 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, I. Touche, Grid'5000: A large scale and highly reconfigurable experimental grid testbed, International Journal of High Performance Computing Applications.

[19] S. V. Costache, N. Parlavantzas, C. Morin, S. Kortas, Merkat: A market-based slo-driven cloud platform, in: Proceedings of IEEE International Conference on Cloud Computing Technology and Science, CloudCom'13, 2013.

[20] B. Sotomayor, R. Montero, I. Llorente, I. Foster, An Open Source Solution for Virtual Infrastructure Management in Private and Hybrid Clouds, IEEE Internet Computing 13 (5) (2009) 14–22.

[21] C. A. Waldspurger, W. E. Weihl, Lottery scheduling: Flexible proportional-share resource management, in: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, USENIX Association, 1994, p. 1.

[22] K. Lai, L. Rasmusson, E. Adar, L. Zhang, B. Huberman, Tycoon: An implementation of a distributed, market-based resource allocation system, Multiagent and Grid Systems 1 (3) (2005) 169–182.

[23] F. Glover, M. Laguna, et al., Tabu search, Vol. 22, Springer, 1997.

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

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

[26] G. Staples, Torque resource manager, in: Proceedings of ACM/IEEE conference on Supercomputing, 2006.

[27] P. W. Archive, http://www.cs.huji.ac.il/labs/parallel/workload/.

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

[29] Twisted, twistedmatrix.com/.

[30] paramiko, www.lag.net/paramiko/.

[31] ZeroMq, http://www.zeromq.org/.

[32] A. Kivity, Y. Kamay, D. Laor, kvm: the linux virtual machine monitor, in: Proceedings of the Linux Symposium, 2007.

[33] U. Lublin, D. G. Feitelson, The workload on parallel supercomputers: modeling the characteristics of rigid jobs, Journal of Parallel and Distributed Computing 63 (11) (2003) 1105–1122.

[34] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, Zookeeper: wait-free coordination for internet-scale systems, in: Proceedings of the 2010 USENIX conference on USENIX annual technical conference, Vol. 8, 2010, pp. 11–11.

[35] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, in: Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France, 2015.

[36] N. Tziritas, C.-Z. Xu, T. Loukopoulos, S. U. Khan, Z. Yu, Application-aware workload consolidation to minimize both energy consumption and network load in cloud environments, in: 2013 42nd International Conference on Parallel Processing (ICPP), IEEE, 2013, pp. 449–457.

[37] A. Hameed, A. Khoshkbarforoushha, R. Ranjan, P. P. Jayaraman, J. Kolodziej, P. Balaji, S. Zeadally, Q. M. Malluhi, N. Tziritas, A. Vishnu, et al., A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems, Computing (2014) 1–24.

[38] A. A. Chandio, K. Bilal, N. Tziritas, Z. Yu, Q. Jiang, S. U. Khan, C.-Z. Xu, A comparative study on resource allocation and energy efficient job scheduling strategies in large-scale parallel computing systems, Cluster Computing 17 (4) (2014) 1349–1367.

[39] K. Ye, Z. Wu, C. Wang, B. B. Zhou, W. Si, X. Jiang, A. Y. Zomaya, Profiling-based workload consolidation and migration in virtualized data centers, IEEE Transactions on Parallel and Distributed Systems 26 (3) (2015) 878–890.

[40] J. Xu, M. Zhao, J. Fortes, R. Carpenter, M. Yousif, Autonomic resource management in virtualized data centers using fuzzy logic-based approaches, Cluster Computing 11 (3) (2008) 213–227.

[41] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. Mckee, C. Hyser, D. Gmach, R. Gardner, T. Christian, L. Cherkasova, 1000 islands: an integrated approach to resource management for virtualized data centers, Cluster Computing 12 (1) (2009) 45–57.

[42] T. Wood, P. Shenoy, A. Venkataramani, M. Yousif, Sandpiper: Black-box and gray-box resource management for virtual machines, Computer Networks 53 (17) (2009) 2923–2938.

[43] S.-M. Park, M. Humphrey, Self-tuning virtual machines for predictable escience, Cluster Computing and the Grid, IEEE International Symposium on 0 (2009) 356–363. doi:http://doi.ieeecomputersociety.org/10.1109/CCGRID.2009.84.

[44] J. Mars, L. Tang, R. Hundt, K. Skadron, M. L. Soffa, Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, ACM, New York, NY, USA, 2011, pp. 248–259.

[45] AmazonEBS, http://aws.amazon.com/.

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

[47] CloudBees, http://www.cloudbees.com/.

[48] E. Yard, https://www.engineyard.com/.

[49] Heroku, http://www.heroku.com/.

[50] G. Pierre, C. Stratan, Conpaas: a platform for hosting elastic cloud applications, IEEE Internet Computing.

[51] M. Boniface, B. Nasser, J. Papay, S. C. Phillips, A. Servin, X. Yang, Z. Zlatev, S. V. Gogouvitis, G. Katsaros, K. Konstanteli, G. Kousiouris, A. Menychtas, D. Kyriazis, Platform-as-a-service architecture for real-time quality of service management in clouds, in: Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services, ICIW '10, IEEE Computer Society, Washington, DC,

USA, 2010, pp. 155–160.

[52] S. Kächele, F. J. Hauck, Component-based scalability for cloud applications, in: Proceedings of the 3rd International Workshop on Cloud Data and Platforms, CloudDP '13, ACM, New York, NY, USA, 2013, pp. 19–24. doi:10.1145/2460756.2460760.
URL http://doi.acm.org/10.1145/2460756.2460760

[53] Tsuru, http://www.tsuru.io/.

[54] Openshift, http://www.openshift.com/.

[55] AppScale, http://www.appscale.com/.

[56] CloudFoundry, http://www.cloudfoundry.com/.

[57] D. Dib, N. Parlavantzas, 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.
URL http://hal.inria.fr/hal-00940299

[58] C. S. Yeo, R. Buyya, A taxonomy of market-based resource management systems for utility-driven cluster computing, Software: Practice and Experience 36 (13) (2006) 1381–1419.

[59] J. Q. Cheng, M. P. Wellman, The walras algorithm: A convergent distributed implementation of general equilibrium outcomes, Computational Economics 12 (1) (1998) 1–24.

[60] R. Wolski, J. S. Plank, T. Bryan, J. Brevik, G-commerce: Market formulations controlling resource allocation on the computational grid, in: Proceedings of hte 15th International Parallel and Distributed Processing Symposium., IEEE, 2001, pp. 8–pp.

[61] J. Norris, K. Coleman, A. Fox, G. Candea, Oncall: Defeating spikes with a free-market application cluster, in: Proceedings of the 2004 International Conference on Autonomic Computing, ICAC'04, IEEE, 2004, pp. 198–205.

[62] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, R. Buyya, Libra: a computational economy-based job scheduling system for clusters, Software Practice and Experience 34 (2004) 573–590.

[63] C. S. Yeo, S. Venugopal, X. Chu, R. Buyya, Autonomic metered pricing for a utility computing service, Future Generation Computer Systems 26 (8) (2010) 1368–1380.

[64] I. Stoica, H. Abdel-Wahab, A. Pothen, A microeconomic scheduler for parallel computers, in: Job Scheduling Strategies for Parallel Processing, Springer, 1995, pp. 200–218.

[65] C. S. Yeo, R. Buyya, Pricing for utility-driven resource management and allocation in clusters, International Journal of High Performance Computing Applications 21 (4) (2007) 405–418.

[66] D. Abramson, R. Buyya, J. Giddy, A computational economy for grid computing and its implementation in the nimrod-g resource broker, Future Generation Computer Systems 18 (8) (2002) 1061–1074.

[67] O. Regev, N. Nisan, The popcorn market. online markets for computational resources, Decision Support Systems 28 (1) (2000) 177–189.

[68] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, W. S. Stornetta, Spawn: A distributed computational economy, IEEE Transactions on Software Engineering 18 (2) (1992) 103–117.

[69] B. N. Chun, 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, Springer-Verlag, London, UK, UK, 2000, pp. 1–14.
URL http://dl.acm.org/citation.cfm?id=646094.758614

[70] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, A. Mu'alem, Ginseng: Market-driven memory allocation, in: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, ACM, New York, NY, USA, 2014, pp. 41–52.

[71] T. Sandholm, K. Lai, Dynamic proportional share scheduling in hadoop, in: 15th Workshop on Job Scheduling Strategies for Parallel Processing, 2010.