On the Use of a Proportional-Share Market for Application SLO Support in Clouds

Stefania Costache[1,2,⋆], Nikos Parlavantzas[2,3], Christine Morin[2], and Samuel Kortas[1]

[1] EDF
[2] INRIA Centre Rennes - Bretagne Atlantique
[3] INSA Rennes

Abstract. Virtualization provides increased control and flexibility on how resources are allocated to applications. However, common resource provisioning mechanisms do not fully use these advantages; either they provide limited support for applications demanding quality of service, or the resource allocation complexity is high. To address these issues we developed Themis, a market-based application management platform. By limiting the coupling between the applications and resource management, Themis can support diverse types of applications and performance goals while ensuring maximized resource usage. In this paper we present the performance of Themis when users execute batch applications with different Service Level Objectives such as deadlines.

1 Introduction

Cloud computing is attractive to execute increasingly dynamic and complex applications on a High Performance Computing infrastructure. An organization can efficiently share its physical resources between different application types (e.g., MapReduce, MPI, or other legacy applications) by allowing each application to run in its own virtual cluster (a set of virtual machines configured with the software packages needed by the user) with limited interference from the infrastructure's administrator. Recent Platform-as-a-Service solutions, both commercial [1, 8] and research [16] hide the complexity of deploying and configuring these virtual clusters, providing users with support to develop and run their applications with no concerns regarding infrastructure's resource management complexities.

However, a remaining challenge is the design of resource management policies to share resources fairly between applications, in terms of their priority and user-specified Service Level Objectives (SLOs). The common cloud resource-provisioning model is "on-demand" virtual machine (VM) provisioning. This model relies on a First-Come-First-Served policy to schedule virtual machines. This would not be a problem if the infrastructure capacity were enough for all

⋆ The author is currently a PhD student at INRIA and EDF R &D. This work is supported by ANRT through the CIFRE sponsorship No. 0332/2010.

user requests in the highest demand periods. Nevertheless, this is rarely the case, as expanding the resource pool is expensive. Thus, it is preferable to solve the contention periods when they appear, around deadlines (e.g., conference or project deliverable), rather than acquiring more resources.

To address this challenge we proposed Themis [10], a platform for application and resource management. To allocate VMs to applications, Themis uses a proportional-share market on top of a virtualized infrastructure. VMs are bought from the market at a (user or application) specified cost (i.e. bid) while the CPU time and memory amount allocated to each of them varies in time according to the total resource demand and the costs of other VMs. To keep the correct CPU and memory amounts allocated to each VM, Themis migrates them between physical nodes. We evaluated the resource allocation algorithms of Themis when applications compete for CPU time [10] and implemented it in a real prototype.

In this paper we analyze the performance of the implemented proportional-share market when applications buy amounts of *different* resources (i.e., CPU and memory) to meet *different* SLOs. Simulations with a real workload trace show that even with simple adaptation policies and using only current knowledge, the system behaves well in terms of application performance and number of VM operations (e.g., migrations, suspend/resume).

This paper is organized as follows. Section 2 presents the context of our work and introduces Themis. Section 3 details the proposed resource management policies. Section 4 describes the evaluation of the proposed resource management policies and Section 5 concludes the paper.

2 Context

This Section describes the context of our work. We first introduce the application model considered in this paper. Then we give an overview of Themis.

2.1 Application Model

Although Themis supports a wide variety of applications, in this paper we focus on batch applications 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, applications need to perform a certain computation amount (e.g. 1000 iterations). There is a large number of iterative applications that follow this model, for example scientific simulators (e.g., Code_Saturne [9]). These applications have a relatively stable iteration execution time. The iteration execution time can be tuned by modifying the resource allocation received by each task. For example, if each task receives one full core, one iteration can take 1 second. If the resource allocation drops at half, the same iteration can take 2 seconds.

2.2 Themis Overview

We previously developed Themis, a market-based platform for application and resource management on clouds [10]. A variety of solutions were proposed to use

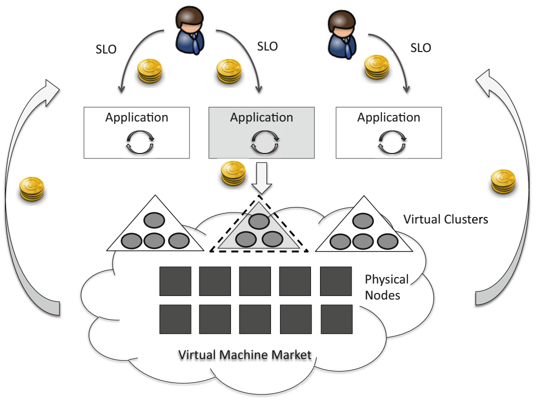


Fig. 1. Themis eco-system

a market to schedule jobs on clusters [11, 21, 20, 7, 22, 6, 3, 19], to schedule jobs on grids [4] or to run parallel applications [17]. Market-based systems force users to assign throughful priorities to their applications as they have to pay for their execution. If the system is too loaded, a user that doesn't need to run her application immediately will postpone its execution until a less loaded period, as the execution cost is lower. Works that apply market mechanisms focus on providing better user satisfaction than traditional resource management systems, usually batch schedulers. However, these systems neither consider application adaptation, nor user SLOs.

Figure 1 gives an overview of Themis. Users receive budgets of credits from a central banking service and use them to run their applications on the cloud. In Themis, applications are funded at a rate established by users, representing the maximum execution cost supported by the user, i.e., budget.

Themis regulates the resource allocation between applications by using a proportional-share market [13, 7, 14, 18]. Applications provision VMs from this market by specifying bids for them. Users are charged for the cost of used resources, i.e., the VM bids, at each scheduling period. In Themis resources (i.e. CPU and memory) are allocated to VMs by using a proportional-share allocation rule. With the proportional-share scheduling mechanisms, each VM i is assigned a bid b_i and receives a share of $b_i/\Sigma b$ of the infrastructure resources. For example, let's take two VMs A and B that want to use the CPU resource of a physical node. The VM A has a bid of 1 and the VM B has a bid of 2. In this case, A receives 33% CPU time and B receives 66%. This mechanism is already supported by current hypervisors and achieves good system utilization through fine-grained resource allocation.

Themis runs each application in a private virtual cluster and allows the application to adapt its resource demand by changing the number of VMs or the resource allocation (i.e., CPU and memory) for each VM by changing the VM bids. Applications individually adapt their resource demand to meet their SLOs (e.g. deadlines) and to react to fluctuations in the resource prices.

VM Allocation on the Proportional-Share Market. We apply the proportional-share market to allocate to each VM a fraction of CPU time and physical memory on a physical node. Themis periodically applies the proportional-share allocation rule system-wide. For simplicity, we consider that the storage capacity is sufficient to accommodate all the VM images.

As the system load or the bids change in time, to ensure an allocation corresponding to the submitted bids, VMs might need to be migrated between nodes. To limit the number of migrations, each VM is allowed to have a certain allocation error. To explain what the allocation error is, we introduce the following terms. We define a_{ri} as the maximum resource allocation a VM receives for a resource r from the capacity of its current node proportional to the bids of all the VMs running on this node. We define a_{rh} as the resource allocation the VM would receive for a resource r if the total infrastructure capacity is considered. Then, if the VM is not migrated, its allocation error is the maximum error over all resources: $max_r(e_r)$, where $e_r = \frac{a_{ri}-a_{rh}}{a_{ri}}$. The goal of the migration algorithm is to ensure that each VM has an allocation error below a given threshold (e.g. 10%). To select the VMs to be migrated at each scheduling interval, Themis uses a tabu-search heuristic [12]. Tabu-search is a local-search method for finding the optimal solutions of a problem. The heuristic runs for a specific number of iterations. At each iteration the heuristic tries to move the VM with the maximum allocation error that is not in the tabu list to the physical node that minimizes it. The heuristic stops if it cannot improve the VM allocation error for the last iterations (i.e. 100 iterations) or if reaching a better solution involves a number of migrations higher than a threshold.

Application Resource Demand Adaptation Policy. On top of the proportional-share market, applications can adapt in two ways: (i) by changing their resource demand (i.e. number of virtual machines) to cope with modifications in their workload (e.g. changes of computation algorithms, additional started modules, etc.); (ii) by changing their bids (which are re-considered at each scheduling period) to cope with fluctuations in price. In this paper we focus on the last case, for which we developed a simple policy. The application uses only information regarding its current CPU and memory allocation and resource prices. Based on this information, the application tries to keep the value for its remaining execution time, or the predicted time for the next iteration, close to a user-defined reference metric. For this we use simple heuristics: the application decreases the bids for its resources when its performance value drops below the given target (i.e. the predicted execution time becomes smaller than the remaining time to deadline) and it increases them otherwise.

In this paper we extend our previous work in several ways. First, we extend the resource allocation algorithms to support for multiple resource allocation. Second, we define a set of adaptation policies for different deadline-based SLOs. These policies use the current application resource allocation and resource price to adapt the application's bids for VMs and obtain the desired resource allocation for meeting the application's SLO (they vertically scale the VMs). Finally, we show how Themis can support these different user SLOs and how its resource allocation algorithms cope with both CPU and memory dynamic allocation.

3 SLO Policies on the Proportional-Share Market

In this section we define a set of SLOs users typically require for their applications and a set of policies to adapt the application resource demand to these SLOs.

User and SLO Modeling. After studying the needs of HPC users from an organization (e.g. at Electricité de France), we determined two classification criterias: the required time of their application results and the required application results.

Based on the required time of their application results, we found that there are two classes of users:

- deadline users: they want the application results by a specific deadline. For example, a user needs to send her manager a simulation result by 7pm.

- performance users: they want the results as soon as possible but they are also ready to accept a bounded delay. This delay is defined by the application deadline too. 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.

Based on the required application results, we found two classes of preferences:

- strict results: to provide useful results the application needs to finish all its computation before its deadline.

- partial results: some users might value *partial* application results at their given deadline; for example, for a user 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.

We combined these classes and obtained four user types: (i) deadline-strict; (ii) deadline-partial; (iii) performance-strict; (iv) performance-partial. We think that these categories can be representative for other organizations as well.

Additional Mechanisms. Besides adapting their bids, to minimize the execution cost applications can apply two policies: (i) delay their execution if the price is too high; (ii) suspend their execution when the price becomes too high, and resume it later when the price drops.

Algorithm 1 describes the conditions the application uses to start, resume or suspend its execution. The suspend policy is run at each scheduling interval by the application. The start/resume policies are run by Themis on behalf of the application, if the application hasn't started or is suspended.

The *StartResume* policy computes the initial payment for *nvms* VMs with T_{alloc} allocated resources by using the current market price. If this payment is greater than the maximum afforded budget, bid_{max}, then the application postpones its execution/resume with a random amount of time bounded by t_{wait}.

Algorithm 1 . Application Start/Resume/Suspend Policies

```
1: **StartResume**($bid_{max}, nvms, T_{alloc}, price_{current}, t_{wait}$)
2: bid = compute initial payment(price_current, $T_{alloc}$, nvms)
3: start = False
4: if $bid[CPU] + bid[memory] < bid_{max}$ then
5:     start = True
6: if start = False then
7:     wait random time period between now and $t_{wait}$
8:
9: **Suspend**($value, T, bid, bid_{max}$)
10: if ( $value < T$) and
    $bid[CPU] + bid[memory] < bid_{max}$ then
11:     if $suspend\_iterations < max\_iterations$ then
12:         $suspend\_iterations = suspend\_iterations + 1$
13:         suspend = False
14:     else
15:         suspend = True
```

The *Suspend* procedure describes the conditions the application uses to suspend its execution. To minimize the execution cost, the application suspends itself when its performance metric, *value*, is bigger than the reference T and the application cannot afford to improve it. To avoid cases in which all applications would suspend at the same time, before suspending its execution, the application waits for a random number of scheduling periods, defined by *max_iterations*.

Application Policies. We derive a set of application specific policies that consider the types of users and goals previously presented. These policies run periodically during the application execution. To ensure the best chance to finish its execution before a deadline, an application starts as soon as the price drops enough so the application can afford a minimum allocation for each VM (e.g. 25% of CPU time). Then, during its execution it applies the different policies according to its SLO. These policies are the following:

- deadline-strict: Applications start when the price is low enough to ensure a good allocation (i.e. 75% CPU time). During their execution they adapt their bids to keep a low price in low utilization periods and to use as much resource as their SLO allows in high utilization periods. If the application cannot pay for the resources needed to meet its SLO it suspends its execution. In this way it leaves resources for other applications and avoids wasting credits for nothing. The application resumes if the price drops enough to allow it to finish its execution within the deadline. If, during its execution, the application sees it cannot miss the deadline it stops.

- deadline-partial: This policy is similar to the previous policy. Nevertheless, there are two differences: (i) the application suspends only 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 does not stop its execution when it sees it cannot meet its deadline anymore.

- performance-strict: The policy is similar to the **deadline-strict** policy and it follows the same algorithm. However, during its execution, the application, instead of tracking a performance reference metric, tries to keep a maximum allocation given its budget. When the application cannot have a minimum allocation at the current price, the application suspends.

- performance-partial: This policy is similar to the previous one but is used by users accepting partial results. However, as for the *deadline-partial* policy, the application does not terminate before the deadline is reached.

4 Evaluation

This section describes the evaluation of our proposed resource management policies. With our evaluation we seek an answer to the following two questions:

- How does the system perform in terms of user satisfaction when their applications behave strategically and adapt their resource demands?
- What is the performance overhead of the application adaptation, considering that application adaptation leads to VM operations?

4.1 Implementation

Themis is implemented in CloudSim [5], a Java event-based cloud simulator. CloudSim can be used to model applications, workload submission scenarios and varios resource management policies. The simulated environment is composed of a datacenter, its VM allocation policy that runs periodically, and multiple applications, created dynamically during the simulation run. Applications are created according to their submission times, taken from a workload trace, and are destroyed when they finish their execution. During its lifetime, each application runs the resource demand adaptation policy and interacts with the datacenter to change the bids for its VMs. We extended our previous implementation [10] by introducing dynamic memory allocation and overheads for VM boot and suspend/resume operations; then we implemented the proposed policies.

4.2 Evaluation Metrics

The performance of a resource management system can be measured in different ways. Traditional metrics include application wait time, resource utilization or number of missed deadlines. Nevertheless, these metrics do not reflect accurately the total user satisfaction, which represents an important metric in showing how well resources are managed. To quantify the total user satisfaction, the aggregate user satisfaction can be used. We model the user satisfaction as a function of the budget assigned by the user to its application and the application execution time, i.e., a utility function. As there are four different user types, we obtain four utility functions.

Nevertheless, 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., if the application runs on a dedicated infrastructure. $work_{done}$ represents the number of iterations the application managed to execute until it was stopped and

Table 1. Utility functions

User	Utility Function
Deadline-strict	B, if $t_{exec} \leq t_{deadline}$, 0 otherwise
Deadline-partial	B if $t_{exec} \leq t_{deadline}$, $B \cdot \frac{work\_done}{work\_total}$ otherwise
Performance-strict	$max(0, B \cdot \frac{(t_{deadline} - t_{exec})}{(t_{deadline} - t_{ideal})})$

$work_{total}$ represents the total number of iterations. B is the application's budget per time scheduling period and per task. B is assigned by the user and reflects the application's importance.

Table 1 summarizes the used utility functions. The deadline-strict user values the application execution at her budget rate if the application finishes before deadline, otherwise she assigns a value of zero. The deadline-partial user is satisfied with the amount of work done until the deadline. Thus the value of the application execution is proportional with this amount. The performance-strict user becomes dissatisfied proportional to her application execution slowdown. We bound the value of her dissatisfaction at zero. The utility function for the performance-partial user is a combination between the deadline-partial and the performance-strict functions.

4.3 VM Operations Modeling

We implemented in CloudSim a model for several VM performance overheads:

- resource allocation. We assume pessimistically that the application's performance degrades proportionally with the allocated memory fraction, when this fraction is less than the demanded memory.

- VM boot/resume. We simulate the VM boot and resume times separately. The VM boot time is modeled by sending the application a message that the VM was created with a delay of 30 seconds. The VM resume time is modeled by assigning to the VM a processing capacity of 0 for 30 seconds.

- VM migration. We compute the migration time as the time to transfer the VM memory state by using the available network bandwidth of the current host. We assume that the bandwidth is shared fairly between all the requests. The available network bandwidth is computed by considering all the suspend and resume operations that occur on the considered host at the current scheduling period. We model the migration performance overhead as 10% of CPU capacity used by the virtual machines in which the application is running. We chose this overhead as previous work found that migration brings 8% performance degradation for HPC applications [15].

4.4 Workload Modeling

To evaluate the system performance we use a real workload trace as it reflects the user behavior in a real system. Such traces are archived and made publicly available [2]. As a workload trace, we chose the HPC2N file as it has detailed information regarding memory requirements of the applications. This file contains information regarding applications submitted to a Linux cluster from Sweden. The cluster has 120 nodes with two 240 AMD Athlon MP2000+ processors each. We assigned to each node 2 GB of memory. For applications with no memory information, we assigned a random memory amount, between 10% and 90% 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 scaled the inter-arrival time with a factor between 0.1 and 1 and we obtained 10 traces with different contention levels. A factor of 0.1 gave a highly contended system while a factor of 1 gave a lightly loaded system.

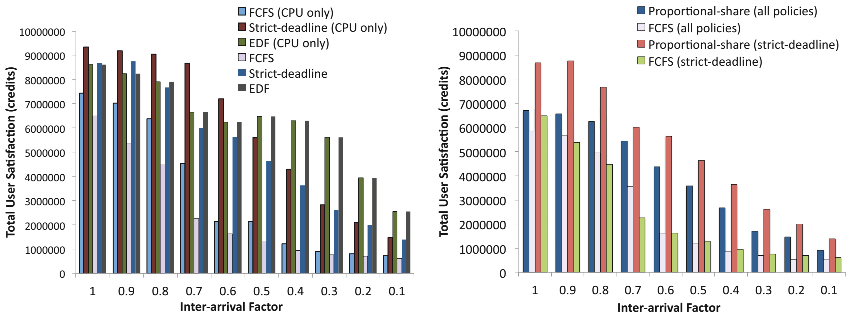
We consider that all applications have a deadline and a re-chargeable budget. As we couldn't find any information regarding application deadlines, we assigned synthetic deadlines to applications. The budgets assigned to applications are inversely proportional to the application's deadline factor.

4.5 Results

Figure 2(a) describes the total satisfaction that the system provides to users when applications use the *strict-deadline* policy compared to well-known algorithms like FCFS and EDF. We selected this policies as we wanted to use the same comparison criteria as in previous work [10] and this policy performed good in terms of user SLO satisfaction. We didn't include the other three policies due to lack of space. To see the behavior of our algorithms when both CPU and memory need to be dynamically allocated, we measure the total user satisfaction when the memory is enough for all the requests and when the memory is constrained at 2GB RAM per physical node. We notice that when both CPU and memory need to be allocated our market out-performs FCFS much more than in the case of CPU only allocation. The proportional-share mechanism allows applications to run with a less than required amount of resources.

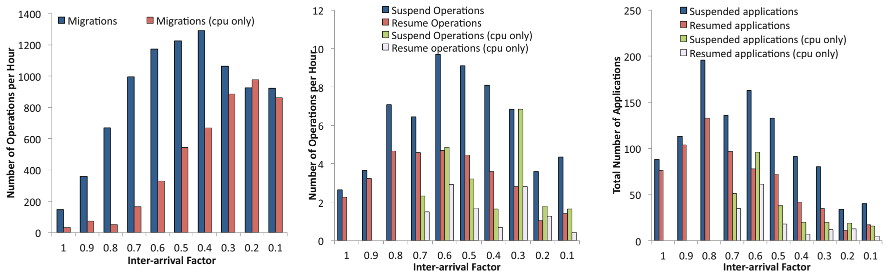
Compared to EDF, we notice a performance degradation that increases with the system load. As the inter-arrival time decreases, EDF is capable to take better scheduling decisions: more applications with smaller deadlines, and in the same time higher budgets, get to run on time. This provides better user satisfaction than our system and FCFS. Then, our system is decentralized: each application acts selfishly and independently to meet its own application SLO while with EDF, the central scheduler sorts applications by their deadline and executes the application with the smallest deadline first.

Figure2(b) describes the total satisfaction that the system provides to users when applications use different policies. Themis allocates both CPU and memory. In this scenario, each application selects a policy randomly from the four ones we provide. We notice a performance degradation compared to the case when all applications use a *strict-deadline* policy. Applications using a policy like *strict-performance* spend all their budget to try to receive a maximum allocation, leading to other applications to miss their own deadlines. In the case of



(a) (b)

Fig. 2. Proportional share market performance in terms of total user satisfaction in two cases: (a) when the strict-deadline policy is considered and (b) when all the policies are considered



(a) (b) (c)

Fig. 3. Performance of VM allocation algorithm in terms of (a) average number of migrations per hour (b) average number of suspend/resume operations per hour and (c) total number of applications suspended and resumed

the *strict-deadline* policy, applications are cost effective, leaving a larger share of resources for other applications.

Figure 3 describes the number of VM operations performed by the VM allocation algorithms and the total number of applications suspended/resumed during the experiment. To perform this experiment we selected the *strict-deadline* policy, but any other policy would have been appropriate too. We make two observations: (i) the number of VM operations decreases when the system is highly loaded; (ii) the number of VM operations when there are multiple allocated resources is significantly higher than in the case of one resource. The first observation is explained by the fact that when there is a high load, more applications don't start or resume their execution. The second observation is intuitive: applications adapt their bids for multiple resources, leading to more errors in VM allocations and thus, more migrations.

To conclude our results, we need to stress that each application that runs on Themis adapts individually to the market condition and its SLO (e.g. deadline). This selfish application behavior leads to a performance degradation (or also known as the Price of Anarchy), compared to when applications collaborate or when they operate under strict, centralised control. To illustrate why this uncoordinated behavior can be bad, let us take the case of suspend/resume at price fluctuations. When multiple applications suspend, the other running applications receive a higher resource allocation and drop their bids. This creates a favorable condition for the suspended applications to resume again. However, when the other applications resume, the price increases leading them to another suspend. The performance degradation is the "price" payed by the nature of our system, that allows applications to behave selfishly.

5 Conclusions

In this paper we analyzed the performance of a proportional-share market mechanism implemented in Themis, an application and resource management platform. In Themis, applications autonomously adapt their resource demand to meet their SLOs, disregarding the other infrastructure occupants.

We extended Themis with multi-resource allocation algorithms and we simulated the application behavior by considering four SLO-driven resource demand adaptation policies. Our simulation results show the efficiency of the proportional-share market. Our policies behave reasonably in terms of application performance and number of VM operations. When the system is lightly loaded our policies lead to a better system performance than well-known scheduling schemes. As each application adapts autonomously to its own SLO, it is intuitive that the system's performance degrades in high load periods.

As future work we plan to implement a resource regulation mechanism in which applications can be more aware of the other infrastructure occupants. For example, when there are not enough free resources to satisfy all arriving applications we can use a double auction in which applications already running on the infrastructure can sell their resources to more urgent applications. We also plan to do more experiments with Themis on a real testbed.

References

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

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

[3] AuYoung, A., Chun, B., Snoeren, A., Vahdat, A.: Resource allocation in federated distributed computing infrastructures. In: Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure, vol. 9 (2004)

[4] Buyya, R., Abramson, D., Giddy, J., Stockinger, H.: Economic models for resource management and scheduling in grid computing. Concurrency and computation: practice and experience 14(13-15), 1507–1542 (2002)

[5] Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software Practice and Experience 41(1), 23–50 (2011)
[6] Chun, B.N., Buonadonna, P., AuYoung, A., Ng, C., Parkes, D.C., Shneidman, J., Snoeren, A.C., Vahdat, A.: Mirage: A microeconomic resource allocation system for sensornet testbeds. In: Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (2005)
[7] Chun, B.N., Culler, D.E.: REXEC: A decentralized, secure remote execution environment for clusters. In: Falsafi, B., Lauria, M. (eds.) CANPC 2000. LNCS, vol. 1797, pp. 1–14. Springer, Heidelberg (2000)
[8] CloudFoundry, `http://www.cloudfoundry.com/`
[9] CodeSaturne. Codesaturne: a finite volume code for the computation of turbulent incompressible flows. International Journal on Finite Volumes (2004)
[10] Costache, S.V., Parlavantzas, N., Morin, C., Kortas, S.: Themis: Economy-based automatic resource scaling for cloud systems. In: Proceedings of IEEE International Conference on High Performance Computing and Communications (2012)
[11] Ferguson, D., Yemini, Y., Nikolaou, C.: Microeconomic algorithms for load balancing in distributed computer systems. In: International Conference on Distributed Computer Systems, vol. 499 (1988)
[12] Glover, F., Laguna, M., et al.: Tabu search, vol. 22. Springer (1997)
[13] Lai, K.: Markets are dead, long live markets. SIGecom Exch. 5, 1–10 (2005)
[14] Lai, K., Rasmusson, L., Adar, E., Zhang, L., Huberman, B.: Tycoon: An implementation of a distributed, market-based resource allocation system. Multiagent and Grid Systems 1(3), 169–182 (2005)
[15] Nagarajan, A.B., Mueller, F., Engelmann, C., Scott, S.L.: Proactive fault tolerance for hpc with xen virtualization. In: Proceedings of the 21st International Conference on Supercomputing (ICS), p. 23 (2007)
[16] Pierre, G., Stratan, C.: Conpaas: a platform for hosting elastic cloud applications. In: IEEE Internet Computing (2012)
[17] Regev, O., Nisan, N.: The popcorn market. online markets for computational resources. Decision Support Systems 28(1), 177–189 (2000)
[18] Sandholm, T., Lai, K.: Dynamic proportional share scheduling in hadoop. In: 15th Workshop on Job Scheduling Strategies for Parallel Processing (2010)
[19] Sherwani, J., Ali, N., Lotia, N., Hayat, Z., Buyya, R.: Libra: a computational economy-based job scheduling system for clusters. Software Practice and Experience 34, 573–590 (2004)
[20] Stoica, I., Abdel-Wahab, H., Pothen, A.: A microeconomic scheduler for parallel computers. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 200–218. Springer, Heidelberg (1995)
[21] Waldspurger, C.A., Hogg, T., Huberman, B.A., Kephart, J.O., Stornetta, W.S.: Spawn: A distributed computational economy. IEEE Transactions on Software Engineering 18(2), 103–117 (1992)
[22] Yeo, C.S., Buyya, R.: Pricing for utility-driven resource management and allocation in clusters. International Journal of High Performance Computing Applications 21(4), 405–418 (2007)