Autonomic Provisioning and Application Mapping on Spot Cloud Resources

Daniel J. Dubois, Giuliano Casale
Imperial College London, Department of Computing
London, United Kingdom
{*daniel.dubois,g.casale*}@*imperial.ac.uk*

***Abstract*—The spot instance model is a virtual machine pricing scheme in which unused resources of cloud providers are offered to the highest bidder. This leads to the formation of a spot price, whose fluctuations can determine customers to be overbid by other users and lose the virtual machine they rented. In this paper we propose a heuristic to automate the decision on: (*i*) which and how many resources to rent in order to run a cloud application, (*ii*) how to map the application components to the rented resources, and (*iii*) what spot price bids to use in order to minimize the total bid price while maintaining an acceptable level of performance. To drive the decision making, our algorithm combines a multi-class queueing network model of the application with a Markov model that describes the stochastic evolution of the spot price and its influence on virtual machine reliability. We show, using a model developed for a real enterprise application and historical traces of the Amazon EC2 spot instance prices, that our heuristic finds low cost solutions that indeed guarantee the required levels of performance. The performance of our heuristic method is compared to that of nonlinear programming and shown to markedly accelerate the finding of low-cost optimal solutions.**

***Keywords*-cloud provisioning; fluid-approximated queueing networks; spot cloud; resource allocation; random environment; application mapping; bidding strategy.**

I. Introduction

Cloud computing is a popular paradigm for offering compute capacity as a service. In particular, the cloud gives flexibility to decide and modify the speed, the number, and the lease time of virtual machines (VMs). There are several pricing strategies for renting VMs, among which are often mentioned two categories: *on-demand pricing* and *spot pricing*. On-demand pricing guarantees that a resource is available for a fixed price, which is proportional to the time the resource is rented. In spot pricing, instead, resources are offered at a variable price, called the spot price, which changes with the market condition. Spot pricing requires users to bid a maximum price they are willing to pay for. If the bid price is greater than the current spot price, the virtual machine will be charged at the spot price. However, if the spot price exceeds the bid price, the VM can be suddenly reclaimed by the provider and lost. The advantage of spot instances is that their price tends to be lower than the on-demand price. However, their maximum price may temporarily exceed the on-demand price when the cloud provider has shortage of resources. This makes the decision of choosing a bid price both difficult and important. While a number of works have considered this problem in recent years [28], [30], [9], [16], the problem of deciding bid prices in light of performance requirements or constraints on the application architecture is more complex and still poorly understood.

In this paper, we want to help users to take maximum advantage from spot instances by supporting the following decisions:

- What type of virtual resources should be rented for a given application?
- How to efficiently map the components of an application (e.g., web server VMs, a database VMs) to the rented resources?
- What is the lowest bid price that still allows to fulfill quality of service requirements?

Specifically, we focus on applications developed according to the model-driven engineering approach, in which a performance model of the application can be automatically generated through model-to-model transformations. For example, queueing networks can be automatically generated from UML or Palladio Component Model diagrams [21], [3]. The problem of executing the decisions, such as concretely migrating the virtual resources is out of the scope of this paper, which focuses on the decision problem.

The main technical innovations of this paper are as follows:

- a polynomial-time heuristic to jointly solve the bidding and allocation problem, which are in general NP-hard;
- what is, to our knowledge, the first application in the area of bidding of extended queueing network models that include a model of the operational environment. The latter, which is referred to as *random environment* model [6], captures the stochastic nature of the operational environment, in which VMs can be suddenly lost as a result of spot price fluctuations.
- the use of advanced fluid analysis techniques to accurately approximate response time percentiles, which are commonly used to constraint performance in service-level agreements, but which are usually hard to compute in queueing networks. Compared to more complex approximations for accurate percentile assessment, such as Laplace transforms, this method is fast enough for run-time application.

Our heuristic can quickly find a local optimal solution. We validate accuracy of this solution by considering the queueing network model of a real enterprise resource planning (ERP) application and real historical data of Amazon EC2 spot prices. We compare our results with an approach that uses a nonlinear optimization algorithm and show that our heuristics provides better results in less time.

The rest of this paper is organized as follows. Section II gives a motivating example. Section III discusses the problem statement and defines the reference model. Section IV presents our heuristic, that is later evaluated in Sections V and VI. Section VII surveys related work. Lastly, Section VIII concludes the paper and outlines possible extensions.

II. Motivating Example

Let us consider a real multi-tier cloud application, such as the SAP ERP [23]. This application is composed of two components: an application server and a database server. The application must also satisfy some quality requirements in terms of response time in fulfilling requests. The problem we want to solve is to find the cheapest way to run this application on a spot cloud system while maintaining the quality requirements. To help making this decision we assume to have the following information: (*i*) a performance model of the application, which can be represented as a queueing network as shown in [23]; (*ii*) current and historical pricing of the resources that can be rented by the cloud provider; (*iii*) a quality requirement in terms of constraints on the response time.

For example, assume that, after analyzing the performance model of the application and the expected load, we need VMs with different computational requirements (expressed as Amazon Elastic Computing Units, ECUs) for the application server and the database server. Then, we have a very large decision space on how to deploy them in a cloud infrastructure if we have multiple types of resources characterized by different prices and speeds, such as in Amazon EC2. Figure 1 shows four examples of deployment characterized by an increasing level of deployment complexity. In the first deployment, we make the most intuitive decision, that is to choose the two cheapest resources that can fit the two VMs of the application. In the second deployment we can take advantage of cheap large resources by deploying multiple VMs inside a single large cloud resource. In the third deployment we can take advantage of cheap small resources by replicating application VMs into multiple cloud resources with the help of a load balancer. Finally, in the last deployment we can choose the cheapest VM of any size by combining the two previous deployment approaches, thus obtaining the highest degree of flexibility and cost-saving potential.

In our approach we consider the most complex case and also consider that the deployment decision is not only affected by the size of the cloud resources, as in the deployment example above, but it should take into account also additional real-world characteristics that may affect the overall system performance:

- number of CPUs, since having multiple CPUs does not always correspond to a proportional increase in the system throughput;
- load balancing, since balancing the load among multiple VMs does not always correspond to a proportional increase in the system throughput with respect to using a single resource of the same type;
- availability, since a spot instance has a possibility to be lost and become unavailable for some time.

With respect to existing solutions such as [28], [9], [16], we want to increase the level of accuracy by using *fluid-approximated models* based on differential equations to evaluate the system response time. These systems have been shown in [21] to be able to scale well with respect to the system size and to provide information about the distribution of the response times of the overall system in addition to the average. Moreover, the fluid-approximated models can be easily used with tools like LINE [20], [22] to perform *random environment analysis*. Random environments are stochastic models used to describe events occurring in the environment a system operates in [6]. In our particular situation we model the random environment around spot price fluctuations, so to take into account their effect when computing the mean response time and the response time distribution.

III. System Model and Problem Statement

A. System Model

We begin by considering a model for the system under consideration. The system model we propose is composed of the following two parts: *application* and *resources*. Our goal is to determine the *rental* and *allocation* policies, which consist in the amount of computational resources to be rented from a cloud provider, the mapping of the various application components to these resources, and finally the bid price for each resource.

Application: We model the application as a closed queueing network *QN* of M software servers (representing the application components), a delay node (representing user think time), K classes of requests, and a set of constraints on the response time that we defined as Service Level Objective (SLO). A detailed list of application parameters is shown in Figure 2a.

Resources: We consider an environment that has $R+1$ available resource types. Type 0 is a special virtual type used to represent unallocated resources that have zero price and zero rate. Each resource is characterized by a certain rate (processing speed) and a certain number of processors. Moreover, by using historical traces we can also associate

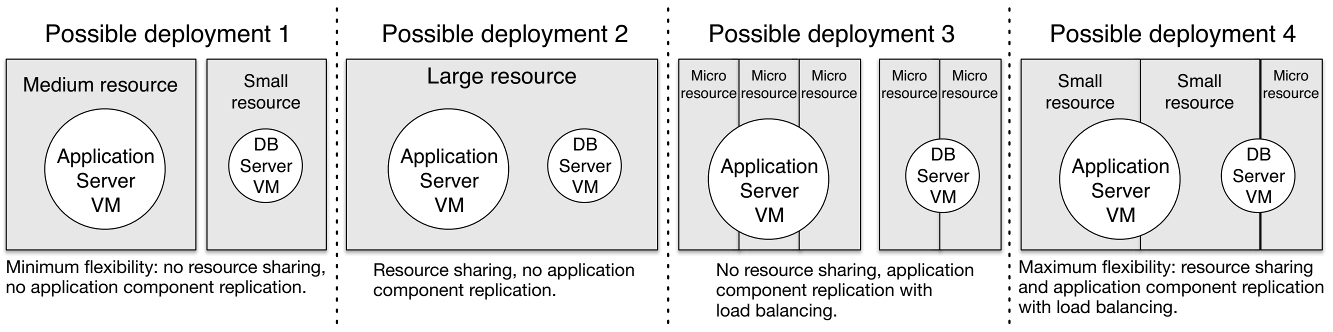


Figure 1. Different strategies for deploying the application components of the SAP ERP application to cloud resources. Application components are represented as circles with size proportional to their ECU requirements. Cloud resources are represented as rectangles with size proportional to their ECU availability. From left to right we show deployment strategies with an increasing level of flexibility and therefore increasing cost-saving potential. The last deployment is the most complex to decide, since it allows multiple application components to be deployed on multiple resources of different types, but it also offers the possibility to use a mix of the cheapest cloud resources.

$p_{m1,m2,k}$	Probability for a request of class k to visit node $m2$ after completing service at node $m1$.
$\mu_{m,k}$	Class service rate. Number of class-k requests completed at software server m in a time unit.
σ_k	Delay node service rate. Number of class-k requests completed at the delay node in a time unit.
N_k	Total number of users of class k in the system. Each user represents a request. This parameter specifies the system workload.
$maxMRT_k$	Maximum mean response time for class-k requests.
$maxRTP_{k,u}$	Maximum response time for the class-k requests in the u-th percentile.

(a) Application parameters.

$Y \in \mathbb{N}$	Total number of resources that can be rented.
T	rental time period.
A	minimum percentage of time in which resources are expected to be available.
$\lambda(r)$	Nominal service rate of resources of type r. The value of $\lambda(r)$ is calculated as the sum of the nominal service rate of each processor of the resource, and is a measure of the total computational capacity (e.g., it may be proportional to Amazon's ECU [1]).
$q(r)$	Number of processors (CPUs) of resources of type r.
$o(r)$	Minimum bid price for renting a single resource of type r for a fixed amount of time T, such that the availability of the resource is at least A. This value can be obtained from historical traces of resource type r and calculated as the minimum bid price that results in an average outbid percentage of at least A for such price.
$c(r)$	Expected price for renting a single resource of type r for a fixed amount of time T when bidding $o(r)$.

(b) Resource parameters.

Figure 2. System parameters.

to each resource a minimum bid price for obtaining an acceptable level of availability, and the actual price that we expect when bidding such bid price. More details on the application parameters are described in Figure 2b.

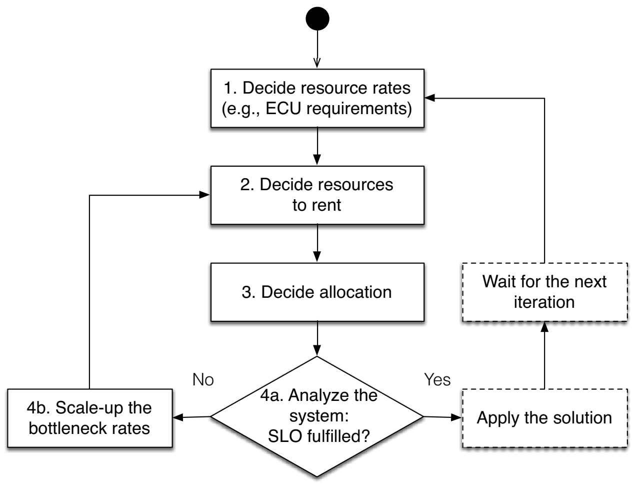
B. Decision Variables and Problem Statement

The system monitors periodically the environment, then it tries to self-adapt the number of cloud resources rented, and the deployment of the software servers on them to optimize the prices and still meet the service requirements. To avoid performance degradation at run-time due to migration and reallocation of software servers, we assume that old resources are deallocated only when the new ones are fully initialized and ready to accept jobs. Based on the considerations above we define our decision variables as follows:

- $t = [t_y]$, $1 \le y \le Y$. *Resource assignment vector*: this is a vector that assigns a resource y to a resource type $t_y \in \mathbb{N}_{[0,R]}$. From this parameter we also define:
  - $\hat{\lambda}_y := \lambda(t_y)$: rate of the resource y.
  - $\hat{c}_y = c(t_y)$: price of the resource y.
- $D = [d_{m,y}]$, $1 \le m \le M$, $1 \le y \le Y$. *Allocation matrix*: where $d_{m,y} \in \mathbb{R}_{[0,+\infty)}$ assigns part of the rate λ of each resource y to each software server m.

The goal is to decide a resource assignment vector t and an allocation matrix D that minimize the sum of the prices of all rented resources. A formalization of the optimization problem is the following:

$$\begin{aligned} \min \quad & \sum_{y=1,\dots,Y} \hat{c}_y \\ \text{s.t.} \quad & \sum_{m=1,\dots,M} d_{m,y} \le \hat{\lambda}_y, \forall y \\ & \mathit{MRT}_k(D) \le \mathit{maxMRT}_k, \forall k \\ & \mathit{RTP}_{u,k}(D) \le \mathit{maxRTP}_{u,k}, \forall u, \forall k \end{aligned}$$



1. Decide resource rates (e.g., ECU requirements)
2. Decide resources to rent
3. Decide allocation
4a. Analyze the system: SLO fulfilled?
No
Yes
4b. Scale-up the bottleneck rates
Apply the solution
Wait for the next iteration

Figure 3. State diagram showing all the steps of our decision-making approach. The approach can be seen as an autonomic feedback loop since it adapts the system at periodic interval by using the most updates prediction data available for the resource prices and the application load.

The first constraint states that it is not possible to allocate to a resource a rate that is larger than the rate of its resource type. The other constraints state that the calculated mean response time and the response time percentiles should be lower than their respective maximums, where $\mathit{MRT}_k(D)$ and $\mathit{RTP}_{u,k}(D)$ are nonlinear functions to calculate the mean response time and the response time percentiles. These functions have all the decision variables and the system parameters described in Figure 2 as input, which are omitted to simplify the notation.

IV. Approach

A. General Idea

The general idea of our approach is to decompose the main problem into simpler subproblems that are solved in an iterative way. Each subproblem obtains its input from the solution of the previous subproblem, as shown in Figure 3: the numbered blocks in the figure represent the subproblems we solve. Our approach is then repeated at regular intervals as a method of pro-active self-adaptation, or in response to unexpected situations that cause a run-time SLO violation as a method of reactive self-adaptation. A general idea of each subproblem we address is described as follows, while details are given in the next subsections.

1. Choosing the minimum computational requirements for each application component. In this step we decide the minimum computational requirements in terms of resource rates (e.g., Amazon's elastic computing units, or simply ECUs) that are needed by each application component to satisfy the quality requirement. At this stage we do not consider the available resources, but we just determine the ECU requirements of the application.

2. Choosing the resources to rent. In this step we calculate the bidding price that guarantees a minimum availability level for the resources and, based on it, we decide which resources to rent. The sum of the ECUs of the rented resources should be large enough to fulfill the ECU requirements of the application decided in the previous step.

3. Choosing the allocation of the application components to the resources. In this step we decide how to allocate the different application components into the rented resources to minimize the negative effects of allocation (e.g., the reduction in performance due to load balancing, as it happens in the third deployment example in Figure 1).

4. Analyzing the overall system and possible scaling-up of bottlenecks. The performance of the overall deployed system is analyzed again taking into account the overhead added by the presence of multiple CPUs and load-balancing. This is also the step in which we consider the effects of the random environment in terms of possibility of losing spot instances in case of overbid of the chosen bid price. If this analysis shows that the chosen resources and allocation do not fulfill the quality requirements anymore, the application ECU requirements of the bottleneck software servers are increased to compensate, and new resources/allocations are decided.

B. Finding the optimal rate for each software server

In this step we want to find a first approximation of the solution of the global problem by assuming that each software server m is deployed on a dedicated hypothetical resource that provides the minimum rate $\hat{\mu}_m$ to process requests such that the SLO constraints are satisfied. In this step we do not consider the characteristics of the real resources (e.g., number of processors, prices, and the random environments information) since a decision on which one to rent will be done in the next steps. The goal of this optimization problem is to decide the minimal rates $\hat{\mu}_m$ that fulfill the constraints on the mean response time and on the response time distribution.

$$\begin{aligned}
\min \quad & \sum_{m=1,\dots,M} \hat{\mu}_m \\
\text{s.t.} \quad & \mathit{MRT}_{k,}(\hat{\mu}) \le \mathit{maxMRT}_k, \forall k \\
& \mathit{RTP}_{u,k}(\hat{\mu}) \le \mathit{maxRTP}_{u,k}, \forall u, \forall k
\end{aligned}$$

To solve this subproblem we use a greedy algorithm that scales down the rates of all the resources as much as it can until one or more bottleneck resources are found for the class of jobs that is closest to the boundary of the constraints. At this point, the rates of the bottleneck resources are fixed, and the algorithm continues to scale down the remaining rates, until all of them have been fixed in the same way.

The pseudocode listing of the algorithm is shown in Figure 4. The function receives as input an initial set of

```
function FINDOPTIMALRATES($\hat{\mu}_{init}, S$)
    $\hat{\mu}, \hat{\mu}_{min} \leftarrow 0$
    $\hat{\mu}_{max} \leftarrow \hat{\mu}_{init}$
    $r = 1, \ldots, S.M$                          ▷ Set of undecided rates
    while $r \neq \emptyset$ do
        $\hat{\mu}(r) \leftarrow (\hat{\mu}_{min}(r) + \hat{\mu}_{max}(r))/2$
        if SLOsatisfied($\hat{\mu}, S$) then
            $\hat{\mu}_{max} \leftarrow \hat{\mu}$
        else
            $vc = findViolatedClasses(\hat{\mu}, S)$
            $bn \leftarrow$ findBottleneckForClasses($r, \hat{\mu}, S, vc$)
            $\hat{\mu}_{min}(bn) \leftarrow \hat{\mu}(bn)$
        end if
        if $\max_r(\hat{\mu}_{max}(r) - \hat{\mu}_{min}(r)) < \epsilon$ then
            $vc \leftarrow$ findClassClosestToSLO($\hat{\mu}, S$)
            $bn \leftarrow$ findBottleneckForClasses($r, \hat{\mu}, S, vc$)
            $r \leftarrow r - bn$
        end if
    end while
    $\hat{\mu} \leftarrow maxRates$
    return $\hat{\mu}$
end function
```

Figure 4. Algorithm for finding the minimum rates $\hat{\mu}$ for each software server.

SLOsatisfied	Checks whether the SLO constraints on the response time are satisfied or not.
findViolatedClasses	Finds the set of classes that violate the SLO constraints.
findBottleneckForClasses	Finds the software servers that are bottleneck for the specified classes.
findClassClosestToSLO	Finds the class that is closest to violating one or more SLO constraints on the response time.

Figure 5. Auxiliary functions that are based on the results of a queuing network evaluation.

arbitrarily large feasible rates $\hat{\mu}_{init}$, and the system model S that contains all the parameters of the application and the resources described in Section III. It returns the optimal rates for each software server as vector $\hat{\mu}$. The variable r is initialized as the set of all available resources that can be scaled. Then, all resources are scaled down using a bisection method until the constraints are violated: minimum rates are increased when the constraints are satisfied and the maximum rates are decreased when the constraints are violated. When the minimum and maximum rates are close enough, the current bottleneck resources are removed from r and the process continues until r is empty. At this point the rate calculated so far is returned as our optimal $\hat{\mu}$. The auxiliary functions used in the algorithm (briefly described in Figure 5) are directly derived from the evaluation of the queueing network and simple operational analysis laws.

C. Finding the real resources to rent

In the previous step we calculated the computational needs in terms of rates of the virtual resources. In this step we want to decide which real resources to rent to provide such computational needs at minimal expense. To make this decision we consider for each real resource y a mean price equal to $\hat{c}_y$, that can be obtained from historical traces given a certain level of desired availability. The goal is to minimize the sum of these costs while ensuring that the rates of all rented resources are large enough to allocate the rates found as the solution of the previous problem.

$$\begin{aligned} \min \quad & \sum_{y=1,\ldots,Y} \hat{c}_y \\ \text{s.t.} \quad & \sum_{y \in 1,\ldots,Y} \hat{\lambda}_y \geq \sum_{m \in 1,\ldots,M} \hat{\mu}_m \end{aligned}$$

This subproblem is a classical integer linear-programming problem (ILP) since the decision variables are integers, and the constraints and the objective functions are linear. This is a well-known NP-hard problem in which we can find an approximate solution using any ILP solver. We implemented a function *findResourcesToRent* to interface with the MATLAB *intlinprog* solver, which accepts the rates of the software servers $\hat{\mu}$ and the system parameters *S* as inputs, and returns the resource assignment vector t.

D. Finding the allocation of the rate for each software server to the real resources

In this step we want to find a good allocation of the rates found so far for each software server to the rented resources. We can combine the allocation of multiple software servers to a single resource and the replication of a single resource to multiple software server, as in the last example of deployment of Figure 1. The allocation decision should minimize the overhead due to load balancing by minimizing the number of associations ($a_{m,y}$) between software servers and resources while still ensuring: (*i*) that each software server obtains at least its minimum rate $\hat{\mu}_m$, (*ii*) that each rented resource y is not providing more than its maximum rate $\hat{\lambda}_y$.

$$\begin{aligned} \min \quad & \sum_{m=1,\ldots,M} \sum_{y=1,\ldots,Y} a_{m,y} \\ \text{s.t.} \quad & a_{m,y} = \begin{cases} 1 & \text{if } d_{m,y} \neq 0 \\ 0 & \text{if } d_{m,y} = 0 \end{cases}, \forall m, \forall y \\ & \sum_{y \in Y} d_{m,y} \geq \hat{\mu}_m, \forall m \\ & \sum_{m \in M} d_{m,y} \leq \lambda_{t_y}, \forall y \end{aligned}$$

To solve this problem we propose an algorithm that finds an approximate allocation by allocating the rates of the

```
function FINDRATEALLOCATION($\hat{\mu}, \hat{\lambda}$)
    $d \leftarrow 0$
    while $\max \hat{\mu} > 0$ do
        $m_{max} \leftarrow argmax_m = \hat{\mu}(m)$
        $y_{max} \leftarrow argmax_y = \hat{\lambda}(y)$
        $transfer \leftarrow \min(\hat{\mu}(m_{max}), \hat{\lambda}(m_{max}))$
        $d(m_{max}, y_{max}) \leftarrow d(m_{max}, y_{max}) + transfer$
        $\hat{\mu}(m_{max}) \leftarrow \hat{\mu}(m_{max}) - transfer$
        $\hat{\lambda}(y_{max}) \leftarrow \hat{\lambda}(y_{max}) - transfer$
    end while
    return $d$
end function
```

Figure 6. Algorithm for finding the allocation of the rates of the software servers to the real resources.

software servers having the largest non-allocated rate to the real resources having the largest available capacity in an iterative process until the rates of all software servers have been allocated.

A listing of this algorithm is shown in Figure 6 as the *findRateAllocation* function. This function takes as input the rates $\hat{\mu}$ we have previously calculated using the *find-OptimalRates* function, and the rented resource rates $\hat{\lambda}_y$, which can be derived from the vector of types t_y calculated using the *findResourcesToRent* function with the relation $\hat{\lambda} = \lambda(t_y)$. In each iteration of the algorithm we find the software server with the highest rate m_{max} and the rented resource with the highest rate y_{max}. Then, we allocate the maximum rate between the rate of m_{max} and the rate of y_{max} by increasing the corresponding value in the allocation matrix $d_{m_{max},y_{max}}$. To avoid reallocating previously allocated rates, we decrement both the rate of m_{max} and the rate of y_{max} by the allocated value. The process is repeated until all the software servers have zero rate.

E. System analysis and scaling-up of the bottleneck server

In this step we check if the SLO constraints still hold when considering the system allocated using the resource assignment vector t and the allocation matrix D found in the previous steps. In our implementation we use the LINE tool [20] to evaluate the mean response time and the response time percentiles, which considers also real resource parameters such as the number of processors, the load balancing, and the random environment model that describes the possibility for a resource to be lost when it is overbid.

If, after calculating the response times, the SLO constraints still hold, we can stop here and return the decision variables t and D calculated so far. These will be used to reconfigure the system and apply the resource rental and allocation decisions.

If the SLO constraints do not hold anymore, it means that the real resource parameters of the proposed allocation had

```
function FINDBOTTLENECKM($\hat{\mu}, S, \alpha$)
    $bestSLOcompliance \leftarrow -\infty$
    $m_* = \emptyset$
    for $m \in 1, \ldots, S.M$ do
        $\hat{\mu}_{tmp} \leftarrow \hat{\mu}$
        $\hat{\mu}_{tmp}(m) \leftarrow \hat{\mu}_{tmp}(m) \times \alpha$
        $t \leftarrow findResourcesToRent(\hat{\mu}_{tmp}, S)$
        $\hat{\lambda} = S.\lambda(t)$
        $d = findRateAllocation(\hat{\mu}_{tmp}, \hat{\lambda})$
        $SLOcomp \leftarrow calcSLOcompliance(\hat{\mu}_{tmp}, d, S)$
        if $SLOcomp > bestSLOcomp$ then
            $bestSLOcompliance \leftarrow SLOcomp$
            $m_* = \{m\}$
        else if $SLOcomp = bestSLOcomp$ then
            $m_* = m_* \cup \{m\}$
        end if
    end for
    return $m_*$
end function
```

Figure 7. Algorithm for finding the bottleneck software servers.

a negative effect on the performance. This can be corrected by identifying one bottleneck server m_* and increasing its rate by a scaling factor α, which is calculated proportional to the amount of *constraint violation*. The bottleneck software server is identified as one of the servers that, when scaled-up by α, have the best effect in reducing the constraint violation of the SLO. To calculate the SLO constraint violation we use the following method. Given a set of i constraints rewritten in the form $V < 0$, where $V = [v_i]$, we define the SLO constraint violation as the maximum value in V. A positive constraint violation means that at least one SLO constraint has been violated.

Finally, to actually determine bottleneck software servers m_* we propose the *findBottleneckM* function, which is shown in Figure 7. This function iterates all the software servers, trying to scale each one up by α and saving the information of the software servers m_* that result in the best reduction of constraints violation. The algorithm then simply recalculates the new resource allocations that would be needed when scaling-up the rate of each software server. Once the bottleneck software servers have been found, we just scale their rate up by α and go back to recalculate the real resources to rent.

F. Convergence of the approach

In this concluding section we give some final remarks on the convergence of each step of our approach.

The problem of finding the optimal rate (step 1) has a guaranteed convergence since it uses the bisection method for fixing the rate of the M resources associated to the software server. The maximum number of queueing network evaluations needed is $O(M \times log_2(max(\hat{\mu}_{init})))$, where M

is the number of software servers and $\hat{\mu}_{init}$ is the vector containing the initial random feasible rates that are given as input to the *findOptimalRates* function.

The problem of finding the real resources to rent (step 2) is NP-hard and solved using an approximated ILP solver. The convergence and the complexity of this step therefore depends on the ILP solver used and its parameters. In this step no queueing network evaluations are performed.

The problem of finding the allocation (step 3) has a guaranteed converge since at each iteration some rate is transferred from the software server with the maximum unallocated rate to the rented resource with maximum rate availability. The maximum number of rate transfers happens when all the M software servers are transferred to all the Y rented resources, therefore the number of iterations of this step is $O(M \times Y)$. Similarly to step 2, this step does not perform any queueing network evaluation during its iterations.

Finally, in the last step it is possible that the final solution computed is not feasible (i.e., it violates the constraints). In this case we need to search for bottleneck servers and scale them up by a factor α. The algorithm to find the bottlenecks tries to scale-up all the software servers one by one, thus resulting in $O(M)$ queueing network evaluations for each search. Each search guarantees that the bottleneck resources speed is increased, thus progressively reducing the violation of the constraints until an optimal solution is found. In some limit situations it is possible that an increase in the rate of a bottleneck resource does not reduce the violation of the constraints, which would prevent the convergence of our approach. These limit cases happen when the contribution to the response time added by the load balancing, the multiple number of processors, and the random environment is too large to be compensated by an increase in rate. Examples of these limit situations are cases with very low resource rates or in which bid prices are characterized by a very low level of availability. In our experiments based on real data we did not experience any of such limit cases, which leads us to think they are contrived examples.

V. Evaluation Setting

The purpose of our evaluation is to give an overview of the behavior of our approach when applied to queueing network models based on real data. In particular, we use public application data measurements from a real SAP ERP study from [23]. For the resources model we use historical traces of spot prices of Amazon EC2 that can be downloaded from [10] and cover a 14-month period up to January 2011. Finally, we instantiate our problem using the generic non-linear solver provided by MATLAB to compare it with our approach. The remainder of this section discusses more in detail the hardware, software, and application models we used to perform our experiments. The results will be presented in Section VI.

A. Hardware and Software

We performed our experiments using a 2.5 GHz Intel Core i7 quad-core processor with 16 GB of RAM running OS X 10.10.3 and MATLAB R2015a. We also used LINE [20] to predict the response times of our queueing network, and we implemented all the functions described in Section IV as MATLAB functions. To allow the evaluation of the effect of allocating the VM of a software server to multiple resources (i.e., replicating it), we have implemented a function to split the nodes of the queueing network according to their application to real resources (*allocateQN*, *splitStation*); moreover, we have implemented an alternative solution to the problem using MATLAB *fmincom* nonlinear solver configured with an interior point algorithm, which we refer as the *exact approach*. This alternative solution considers exactly the same model we solve with our heuristic, but without any particular optimization that can guide the algorithm toward the proper solution. We have chosen this generic solution due to the limited availability of existing approaches that adopt our model formulation.

For the sake of simplicity we omit accurate descriptions of these functions, but they can be downloaded, with all the other MATLAB code we have implemented, from [10]. The provided code can be used to repeat our experiments or to interface it with the run-time monitoring and adaptation module of a cloud system to perform follow-up research on the full autonomic adaptation loop.

B. Application Model

We use an application model based on previous measurements of an industrial enterprise resource planning (ERP) application, SAP ERP. The data of this model and its queueing network representation have been derived from [23]. The application model is represented as a queueing network with exponentially distributed service times, $M = 2$ software servers (representing respectively the CPU of the application server and the CPU of the database server), a delay node representing the user think time, and $K = 3$ classes of requests, which are:

- *dialog step* requests: process and update data on the client-side through the graphical user interface;
- *update* requests: higher priority asynchronous update requests that may be triggered by a dialog step request;
- *update2* requests: lower priority asynchronous update requests that may be triggered by a dialog step request.

The SAP ERP application included additional types of requests, but in the study we are using as reference they were ignored because of their negligible effects on the response times. From the paper we used the information of the service demands, number of users, and number of transactions at each software server and for each class. From this information, we were able to determine a value for the class services rates ($\mu_{m,k}$) and the routing probabilities ($p_{i,j,k}$). A

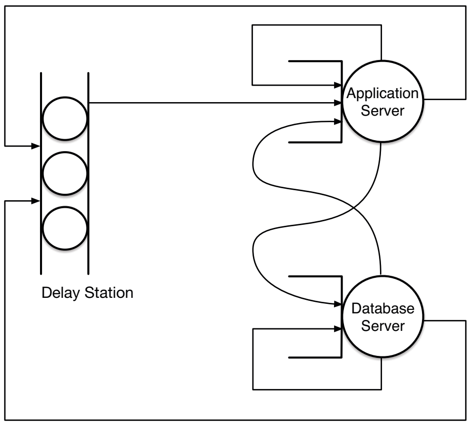


Figure 8. Queueing network representation of the SAP ERP application. The delay station models the user think time, and it is represented as a station with infinite servers. The application server and the database server are represented as regular queues.

Table I
SAP ERP Parameters

Server/class	Service demand [ms]	Service rate $\mu_{m,k}$ [req/ms]
AS dialog step	119.82	0.008346
AS update1	47.92	0.02087
AS update2	32.98	0.03032
DB dialog step	4.541	0.2202
DB update1	1.205	0.8299
DB update2	0.3043	3.286

graphical representation of the queueing network is depicted in Figure 8. In [23] the authors just give an estimation of the overall service demands on the database server, without distinguish the classes of requests. To overcome this problem we assume that the database CPU demand is distributed across the different classes proportionally to the number of users for such classes. The data we have obtained for the class service rate can be seen in Table I, which is calculated as the inverse of the service demand. Additional application parameters are the following:

- N_{dia} (number of users that issue dialog step requests) is arbitrary, but N_{upd} (number of users that issue update requests) and N_{upd2} (number of users that issue update2 requests) are assumed dependent on it, as explained in Section 3.4.1 of [23]. Therefore we consider $N_{upd} = 0.2652 \times N_{dia}$ and $N_{upd2} = 0.06657 \times N_{dia}$.
- σ_k is 0.0001 for all service classes, since we assume an average think time of 10 seconds for each class of users in the system.

C. Resource Model

To determine the resource model we use Amazon EC2 historical spot price traces for each type of resources that are available as text files in [10]. Each line of each trace contains the timestamp and the updated market price for such resource. An example of 100 hours trace is shown in Figure 9.

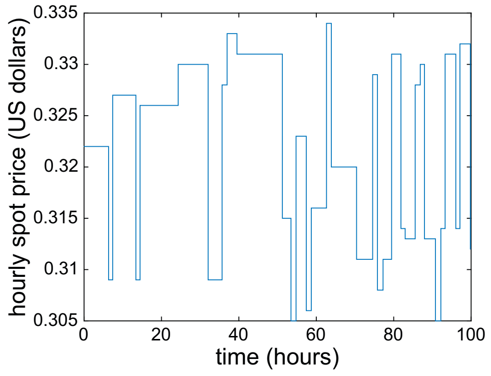


Figure 9. Example of Amazon EC2 spot instance trace for a Windows m2.xlarge VM of the EU-west-1 zone. The trace shows the spot price fluctuations for 100 hours starting on November 30, 2009.

We used these trace files to determine the resource parameters introduced in Section III. In particular, the minimum bid such that the expected life of the resource is as follows:

$$MeanLifeTime = \frac{MeanRecoveryTime \times A}{1 - A} \quad (1)$$

Where A is the desired availability of the resource, and *MeanRecoveryTime* is the average time needed to restart an outbid resource, that is the actual time the resource is unable to process requests after being lost. We run our analysis on some $m1$ and $m2$ Windows instances of the *west-eu* Amazon EC2 zone. We consider $MeanRecoveryTime = 810s$, since it is the typical time to restart a Windows instance, as reported in previous work [18]. After running the analysis, we obtained the parameters shown in Table II, which are the ones we will use in our evaluation. The first column represents the type of the resource, the second the rate of the resource, expressed as Elastic Computing Units (ECU), the third the number of CPUs, which may impact the overall performance. The other columns show the maximum bids and the actual prices for different levels of availability expressed in US dollars. We represent the random environment of the system as the Continuous-Time Markov Chain in Figure 10, which expresses the two possible states of each resource: available (when it is able to process requests) and unavailable (when it is not able to process requests because it is being recovered). Our code for generating our resource model is contained in the class *Survival*, available in [10].

Since the application model has the rates expressed as requests/sec using a reference system that is not expressed in ECU, we have found the conversion rate 1 ECU = 65.1 requests/sec by choosing a rate to the SAP ERP application such that the response time with 1 ECU is equal to the response time measured in [23].

Table II
SPOT PRICES OF AMAZON EC2

Resource	Rate ECU	CPUs	Max bid to have A=90%	Actual prices when A=90%	Max bid to have A=95%	Actual prices when A=95%	Max bid to have A=99.9%	Actual prices when A=99.9%
r	λ_r	q_r	$o_r \mid A = 90\%$	$c_r \mid A = 90\%$	$o_r \mid A = 95\%$	$c_r \mid A = 95\%$	$o_r \mid A = 99.9\%$	$c_r \mid A = 99.9\%$
m1.small	1	1	0.067	0.0653	0.068	0.0659	0.07	0.067
m1.large	4	2	0.266	0.260	0.271	0.2622	0.28	0.2672
m1.xlarge	8	4	0.534	0.5204	0.538	0.5222	0.559	0.5333
m2.xlarge	6.5	2	0.325	0.3139	0.329	0.3164	0.336	0.3203
m2.2xlarge	13	4	0.735	0.7148	0.737	0.7157	0.769	0.7337
m2.4xlarge	26	8	1.468	1.4321	1.47	1.4342	1.54	1.468

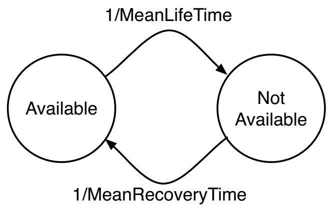


Figure 10. CTMC representing the random environment. In our problem the random environment models the possibility for a resource to be unavailable because of an outbid. The rate for becoming unavailable is the inverse of the resource *MeanLifeTime*, while the rate for becoming available again is the inverse of *MeanRecoveryTime*.

VI. EXPERIMENTS AND RESULTS

We evaluate the real SAP ERP application described in the previous section under different scenarios characterized by a variable number of users to analyze the scalability; with different SLOs, to analyze the behavior in more challenging situations; and finally with a different level of availability, to analyze the effects of the random environment. In each scenario we measure the expected hourly price of the resources, the time needed to compute the solution on our system, and the number of queueing network evaluations.

A. Varying users

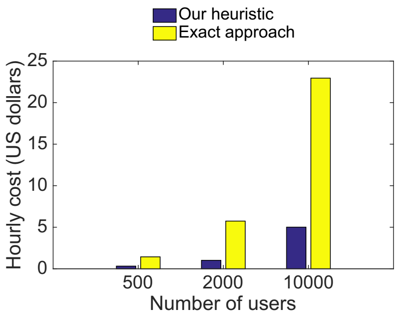
In this experiment we vary the number of dialog users in the system from 500 to 10.000. We fix a SLO that consists of a maximum average response time of 100ms and a maximum 80th percentile of the response time distribution equal to 200ms. Finally, we consider a random environment with availability equal to 95%. By looking at Figure 11 we can see that both our approach and the exact one tend to have a price that grows proportionally with the number of users. The total number of queueing network evaluations tends to be similar for different number of users: in the case of our heuristic we have the convergence at around 25 evaluations, while in the exact approach we often reach the cap of 100 evaluations that has been set to keep the comparison fair. Interestingly, we can see that the execution time is not proportional to the number of evaluations. The reason for this is that the actual time for one queueing network evaluation is proportional to each assignment of software server to a cloud resource. Our heuristic intentionally reduces the level of fragmentation of the assignments to the resources to reduce the overhead due to load balancing, while the exact approach explores too many alternatives that include situations with a high level of fragmentation (i.e., software servers are assigned to a high number of rented resources).

B. Varying SLO

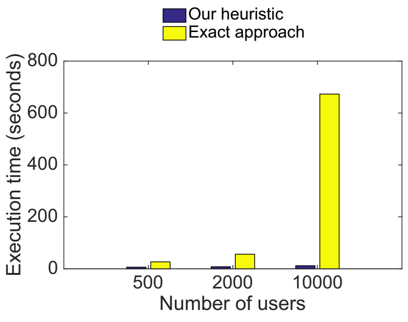
In this set of experiments we show what is the effect of different SLOs constraints on the total hourly price of the system. We consider a maximum response time *maxMRT* that varies from 70ms to 300ms, a value of $maxRTP_{80} = 2 \times maxMRT$, 2000 dialog users, and an availability equal to 95%. The results in Figure 12 show that there is an increase in price when the SLOs are more challenging for both algorithms; however, our approach is still better than the exact one for every different SLO we have considered. From this experiment we can also notice that when the situation becomes more difficult (stricter SLO), we have a significant increase in the number of evaluations and in the time to find the solution. The explanation is that when the SLO is too strict our heuristic requires an increasing number of scaling-up steps.

C. Varying Availability

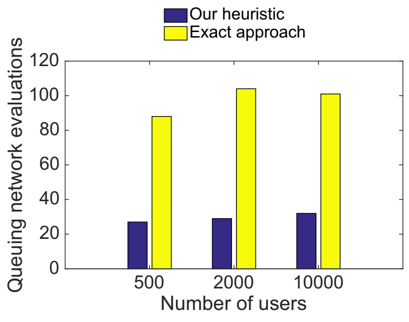
In this other set of experiments we show the effects of different availabilities for the resources, which directly affect the bid price and the probability for a resource to be lost. We consider the cases varying from 90% to 99.9%, while keeping 2000 dialog users, $maxMRT = 100ms$, and $maxRTP_{80} = 200ms$. In Figure 13 we observe that, as we reach the maximum availability the price increases since the system is bidding higher. As we reach the minimum availability we also observe an increase in price since more resources are needed to compensate the failing ones. The best situation is a conservative tradeoff such as 95%, which takes advantage of relatively low spot prices, with a level of availability that is low enough that it does not affect our SLO significantly to require replication. Also in this case our heuristic has a faster converge in terms of time and number of queueing network evaluations than the competing approach for the same reasons we have seen in the other experiments.



(a) Hourly price.

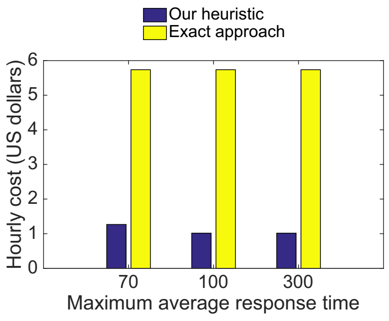


(b) Execution time.

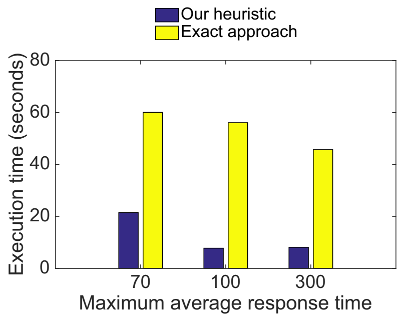


(c) QN evaluations.

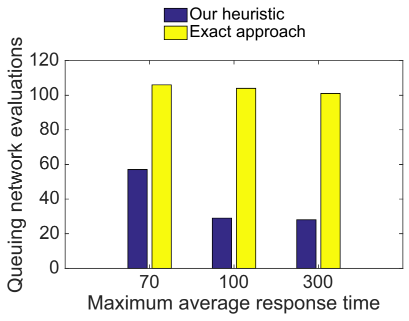
Figure 11. Experiment results when varying the number of users.



(a) Hourly price.



(b) Execution time.



(c) QN evaluations.

Figure 12. Experiment results when varying SLO. The SLO is a maximum limit on the mean value of the response time calculated for a rental time period T.

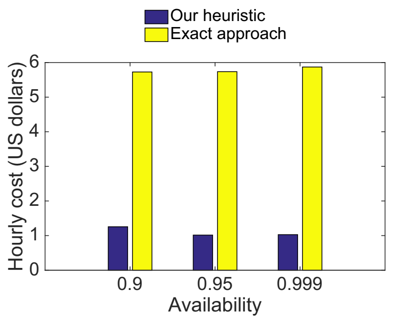
D. Discussion

In this analysis we have seen that our approach is able to outperform an exact algorithm that is based on the MATLAB fmincon interior-point solver. The reason for this result is that our heuristic is able to choose the resources with the best price/ECU ratio and to allocate the application components in such a way that they are not fragmented among cloud resources unless the number of resources is smaller than the number of components. If the number of resources is small, such in the case of 500 users, there is minimal difference between our approach and the exact one. As the number of users increases, or the SLO becomes more restrictive, or the availability decreases, we need more cloud resources to fulfill the SLO. When the number of resources becomes larger than the number of application VMs, the exact approach is not able to choose the correct size of the resources since it tries to resize the partitions of multiple resources, leading to oscillations and slow convergence. The high number of partitions also results in a higher time to evaluate the fluid-approximated queueing network, which ultimately results in large total execution times. Unfortunately, due to the limitations of fmincon we could not express a fitness function that was good enough for the exact approach to converge in every situation. However, in situations in which we observed convergence, the computation of the result was always significantly slower.

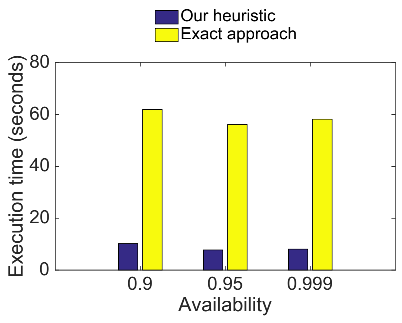
VII. Related Work

In the previous sections we have seen that the main idea of this paper is to combine cost-aware cloud resources provisioning and application mapping into a synergistic autonomic solution that takes into account performance requirements and environmental random factors such as the prices fluctuation of cloud resources and user load.

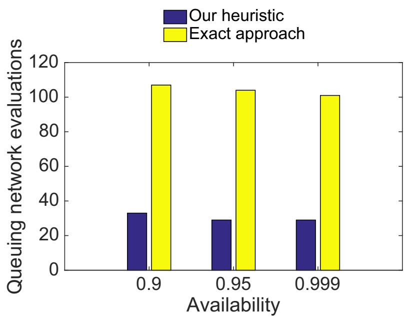
The problem of exploiting different cloud pricing methods such as spot instances has been studied in literature since Amazon introduced the service in 2009. Some works (e.g., [12], [30], [13], [29], [27]) focus on understanding the price dynamics to generate price predictions that can be used to make cost-effective provisioning decision. Other works (e.g., [28], [2], [31]) focus on providing bidding strategies that are specific for spot instances. Our work does not claim to replace these existing method for forecasting resource prices and decide what to bid, but to complement them. In fact in our evaluation we simply assume that bid prices observed so far tend to repeat in the future; however the prediction method is not part of our approach, but simply



(a) Hourly price.



(b) Execution time.



(c) QN evaluations.

Figure 13. Experiment results when varying the availability.

a parameter. That is, in situations when our assumption on the future bid prices is not true, we can seamlessly benefit from the alternative methods cited above without the need to change our approach.

Some different works such as [26], [8], [14] give tools that encourage the use of spot resources by increasing their reliability in case of outbid using recovery techniques based on checkpointing or replication. In our work we are aware of the system reliability thanks to the use of random environment for representing the possibility to lose spot resources; however, the possibility to use reliability-increasing techniques is also orthogonal to our approach and a combined one may result in additional savings in the total price for renting resources.

Finally, research on service placement and load allocation has been specialized to take into account spot pricing models and the possibility to lose resources unexpectedly [4], [32], [15], [33], [17], [16], [9], [5], [11]. With respect to these works we also solve the allocation problem in such a way to minimize the costs while maintaining the desired service level. Our new contribution is that we adopt fluid-approximated performance models [21], which can calculate response time distributions quickly enough to be used at run-time. We also use a random environment model [6] to represent the effects of external events to the system, which for now is limited to price fluctuations, but that can be easily extended to other events expressible as stochastic models. Finally, in our model we also consider the effects of having multiple CPUs in cloud resources (as it is the case for Amazon EC2) and the overhead due to load balancing in case of placement decisions that require resource replication.

VIII. Conclusions

In this paper we have presented a cost-aware approach to support run-time decisions for provisioning cloud resources and allocating application components among them. The benefit of our approach is that it is able to approximate a very complex problem using simple greedy algorithms that are lightweight enough to be used at run-time to support proactive and reactive system adaptation. Moreover, we have shown that we are able to predict and make decisions also when we have a representation of random environmental parameters such as the possibility for spot resources to be lost. The decisions produced by our approach are designed to be used to trigger allocation, deallocation, migration, and replication actions on one or more cloud infrastructures. In our model we assumed that these actions do not affect performance since we consider to keep the system running while they occur; however, this might not be true in every system.

Some future work we have in mind is to introduce in our models and heuristics the possibility to take into account possible overhead in terms of time, performance, and cost that can arise when actually performing adaptation actions on a real system. We also intend to investigate how the approach behaves in presence of different cloud platforms (e.g., federated clouds [19]), services, and alternative ways of expressing the SLOs. Finally, another possible follow-up work is to extend our approach to decentralized cloud systems to improve the scalability and resistance to dynamism, which may contribute to support new emerging cloud paradigms such as volunteer clouds [25] and edge clouds [7], [24].

Acknowledgment

Daniel J. Dubois has been partially funded by the People Programme (Marie Curie Actions) SPANDO FP7 project (FP7-629982). Giuliano Casale has been partially funded by the EPSRC project OptiMAM (EP/M009211/1).

The data generated in this paper are available at: http://dx.doi.org/10.5281/zenodo.20281.

References

[1] Amazon. Ec2 faqs: http://aws.amazon.com/ec2/faqs/.

[2] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under sla constraints. In *MASCOTS '10*, pages 257–266, Aug 2010.

[3] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[4] U. Bellur, A. Malani, and N. Narendra. Cost optimization in multi-site multi-cloud environments with multiple pricing schemes. In *IEEE CLOUD '14*, pages 689–696, 2014.

[5] N. M. Calcavecchia, B. A. Caprarescu, E. Di Nitto, D. J. Dubois, and D. Petcu. Depas: a decentralized probabilistic algorithm for auto-scaling. *Computing*, 94(8-10):701–730, 2012.

[6] G. Casale, M. Tribastone, and P. G. Harrison. Blending randomness in closed queueing network models. *Performance Evaluation*, 82(0):15 – 38, 2014.

[7] A. Chandra, J. Weissman, and B. Heintz. Decentralized edge clouds. *IEEE Internet Computing*, 17(5):70–73, 2013.

[8] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-L. Wang, and F. Cappello. Optimization of cloud task processing with checkpoint-restart mechanism. In *SC' 13*, pages 1–12, 2013.

[9] V. Di Valerio, V. Cardellini, and F. Lo Presti. Optimal pricing and service provisioning strategies in cloud systems: A stackelberg game approach. In *IEEE CLOUD '13*, pages 115–122, 2013.

[10] D. J. Dubois. Heuristic implementation download page: http://www.doc.ic.ac.uk/~djdubois/cac15.

[11] D. J. Dubois, G. Valetto, and E. Di Nitto. Mycocloud: Elasticity through self-organized service placement in decentralized clouds. In *IEEE CLOUD '15*, 2015.

[12] B. Javadi, R. K. Thulasiram, and R. Buyya. Characterizing spot price dynamics in public cloud environments. *Future Generation Computer Systems*, 29(4):988 – 999, 2013. Special Section: Utility and Cloud Computing.

[13] S. Karunakaran, V. Krishnaswamy, and R. Sundarraj. Decisions, models and opportunities in cloud computing economics: A review of research on pricing and markets. In J. G. Davis, H. Demirkan, and H. R. Motahari-Nezhad, editors, *Service Research and Innovation*, volume 177 of *Lecture Notes in Business Information Processing*, pages 85–99. 2014.

[14] S. Khatua and N. Mukherjee. Application-centric resource provisioning for amazon ec2 spot instances. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *LNCS*, pages 267–278. 2013.

[15] J. Li, S. Su, X. Cheng, M. Song, L. Ma, and J. Wang. Cost-efficient coordinated scheduling for leasing cloud resources on hybrid workloads. *Parallel Computing*, 44(0):1 – 17, 2015.

[16] W. Li, P. Svärd, J. Tordsson, and E. Elmroth. Cost-optimal cloud service placement under dynamic pricing schemes. In *IEEE/ACM UCC '13*, pages 187–194, 2013.

[17] J. Lucas Simarro, R. Moreno-Vozmediano, R. Montero, and I. Llorente. Dynamic placement of virtual machines for cost optimization in multi-cloud environments. In *HPCS '11*, pages 1–7, 2011.

[18] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *IEEE CLOUD '12*, pages 423–430, 2012.

[19] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier. A federated multi-cloud paas infrastructure. In *IEEE CLOUD '12*, pages 392–399, 2012.

[20] J. Perez and G. Casale. Line solver http://line-solver.sourceforge.net.

[21] J. Perez and G. Casale. Assessing sla compliance from palladio component models. In *MICAS '13*, pages 409–416, 2013.

[22] J. Perez and G. Casale. Line: A scalable tool for evaluating software applications in unreliable environments. under review., 2015.

[23] J. Rolia, G. Casale, D. Krishnamurthy, S. Dawson, and S. Kraft. Predictive modelling of sap erp applications: Challenges and solutions. In *VALUETOOLS '09*, pages 9:1–9:9, 2009.

[24] M. Ryden, K. Oh, A. Chandra, and J. Weissman. Nebula: Distributed edge cloud for data intensive computing. In *IEEE IC2E '14*, pages 57–66, March 2014.

[25] S. Sebastio, M. Amoretti, and A. Lluch Lafuente. A computational field framework for collaborative task execution in volunteer clouds. In *ACM SEAMS 2014*, pages 105–114, 2014.

[26] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *ACM EuroSys '15*, pages 16:1–16:15, 2015.

[27] V. K. Singh and K. Dutta. Dynamic price prediction for amazon spot instances. In *IEEE HICSS '15*, pages 1513–1520, 2015.

[28] S. Tang, J. Yuan, C. Wang, and X.-Y. Li. A framework for amazon ec2 bidding strategy under sla constraints. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):2–11, Jan 2014.

[29] R. M. Wallace, V. Turchenko, M. Sheikhalishahi, I. Turchenko, Shults, et al. Applications of neural-based spot market prediction for cloud computing. In *IEEE IDAACS '13*, volume 2, pages 710–716, 2013.

[30] P. Wang, Y. Qi, D. Hui, L. Rao, and X. Liu. Present or future: Optimal pricing for spot instances. In *IEEE ICDCS '13*, pages 410–419, 2013.

[31] H. Xu and B. Li. Dynamic cloud pricing for revenue maximization. *IEEE Transactions on Cloud Computing*, 1(2):158–171, 2013.

[32] M. Yao, P. Zhang, Y. Li, J. Hu, C. Lin, and X. Y. Li. Cutting your cloud computing cost for deadline-constrained batch jobs. In *IEEE ICWS '14*, pages 337–344, 2014.

[33] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *IEEE UCC '11*, pages 178–185, 2011.