Cost-Optimal Cloud Service Placement under Dynamic Pricing Schemes

Wubin Li, Petter Svård, Johan Tordsson, and Erik Elmroth
Department of Computing Science, Umeå University
SE-901 87 Umeå, Sweden
Email: {wubin.li, petters, tordsson, elmroth}@cs.umu.se

Abstract—Until now, most research on cloud service placement has focused on static pricing scenarios, where cloud providers offer fixed prices for their resources. However, with the recent trend of dynamic pricing of cloud resources, where the price of a compute resource can vary depending on the free capacity and load of the provider, new placement algorithms are needed. In this paper, we investigate service placement in dynamic pricing scenarios by evaluating a set of placement algorithms, tuned for dynamic pricing. The algorithms range from simple heuristics to combinatorial optimization solutions. The studied algorithms are evaluated by deploying a set of services across multiple providers. Finally, we analyse the strengths and weaknesses of the algorithms considered. The evaluation suggests that exhaustive search based approach is good at finding optimal solutions for service placement under dynamic pricing schemes, but the execution times are usually long. In contrast, greedy approaches perform surprisingly well with fast execution times and acceptable solutions, and thus can be a suitable compromise considering the tradeoffs between quality of solution and execution time.

Keywords—Cloud Computing; Dynamic Pricing; Service Placement; Deployment Optimization

I. INTRODUCTION

Cloud services are typically encapsulated in virtual machines (VMs), and are deployed by instantiating VMs in a virtualized infrastructure. When deploying such services, it is desirable to find an optimal placement, that is an optimal choice of cloud provider(s), considering for example Service Level Agreement (SLA) terms, power consumption and performance. By deploying cloud services across several cloud providers instead of using just one, users can gain benefits like cost reduction, load balancing and better fault tolerance, and also avoid vendor lock-in.

In the expanding cloud computing market, there are many cloud providers with comparable offers, for example GoGrid [1] and Amazon [2] which offer capacity on a hourly, monthly, semi-annual, and annual base. Historically, most providers have used fixed pricing schemes, i.e., the price of a compute unit is constant regardless of the available capacity at the provider. Recently, the concept of dynamic resource pricing is becoming increasingly popular. Such schemes enable cloud providers to attract more customers by offering a lower price if they have excess capacity. Amazon for example has introduced spot instances which enable users to bid for unused Amazon EC2 capacity. Instances are charged the Spot Price, which is set by Amazon and fluctuates periodically depending on the supply of and demand for the spot instance capacity [3]. Such
types of dynamic pricing schemes provide cloud customers with the flexibility of ad-hoc provisioning while receiving significant price savings.

As a consequence of the static pricing schemes used by commercial providers, most cloud service placement research has focused on static pricing scenarios where cloud providers offer fixed pricing schemes for their resources. In this paper however, we propose methods and algorithms to find cost-optimal deployment of services across multiple cloud providers in dynamic pricing scenarios. We study a number of algorithms for placement optimization and evaluate them by running deployments on a cloud platform using our general approach to service deployment, the Service Deployment Optimizer (SDO), presented in a previous contribution [14].

The remainder of the paper is organized as follows. Section II gives a short background on cloud services and deployment. Related work is discussed in Section III. Section IV briefly defines the studied problem and outlines our optimization algorithms. Section V presents the experimental evaluation in an environment with three clouds. Our conclusions are given in Section VI followed by acknowledgments, and a list of references.

II. BACKGROUND

A. Cloud services

In virtualized cloud environments, a cloud service is provisioned as a VM or a collection of VMs. A VM of a certain type is known as a component and a service can consist of multiple components. For example, a typical three-tier web application has a presentation layer component, a business layer component and a database component. Note that there can be several instances of each component. Information about the service composition in terms of components, functional and non-functional requirements and elasticity bounds may be described in a document, the service manifest. The elasticity bounds are upper and lower limits for how many instances of a component that are allowed to be provisioned at a given time and are commonly associated with elasticity rules for when to scale up or down the number of instances of a component. The service manifest may also contain any constraints on the service, e.g., geographical location or requirements for data protection. An extensive implementation of this kind of a service manifest can be found in [19].
Figure 1 illustrates the different phases in the lifecycle of a cloud service. The packaging the service into components and building the service manifest is known as the construction phase of the service lifecycle. In order to get the service running, it needs to be deployed. During the deployment phase, a suitable cloud provider, or a set of providers, is identified. The service components are then contextualized and transferred to the selected provider where they are installed. Once the VMs have been booted and are accessible to outside peers, the deployment phase is complete. The service lifecycle then moves to the operation phase in which the service is managed by the cloud provider to ensure efficient and robust service delivery. Notably, VM recontextualization may be needed in the operation phase to enable adaptation of VM behavior in response to internal changes in the service to which the VM belongs or to external changes affecting the execution environment of the VM [6]. When the service is no longer needed, it can be undeployed. During the undeployment phase, the cloud provider shuts down the running VMs and removes the service assets such as disk images. This paper focuses on the service deployment phase which is discussed in more detail in the upcoming sections.

B. Service Deployment

Providers offering services to customers are known as Service Providers (SPs). Since most SPs often do not control enough hardware resources, they deploy their services to Cloud Providers. This deployment process can be complex. In a previous contribution [14], where we design and implement a general approach to service deployment, the Service Deployment Optimizer, we divide the deployment process into six stages. These stages are cloud provider discovery and filtering, service manifest construction, negotiation and deployment optimization, service contextualization, service data transfer and SLA creation. We then identify requirements for all of these stages as well as for the deployment process as a whole.

Of interest to this contribution is the requirement for the negotiation and deployment optimization stage which states that the deploying party must be able to negotiate with available providers for service hosting offers. The requirement also states that it is not always desirable to deploy the whole service to the same provider but for reasons such as security, performance, and fault tolerance it can be preferable to split the service between several providers. This means that a negotiation can be performed for part of a service, not necessarily the whole manifest. Based on the results of these negotiations and data such as reputation statistics, which could be gathered and evaluated by third-party entities, the deploying party must then decide where to deploy the service. In this contribution, we refer to this process as Service Placement.

C. Service Placement

During the deployment of a service, a decision is taken on which provider, or combination of providers, is to be used to host the service. If several providers are used, the service manifest is split to create a number of sub-services which are then independently mapped to providers. Note that constraints such as affinity and anti-affinity can limit how the service can be decomposed [9].

Taking into account these requirements, the placement algorithm usually strives to optimize a given objective, for example to minimize the cost or the risk by splitting the service and finding the optimal combination of providers. The selection process is commonly performed as a negotiation process, where the SP asks the providers for offers on hosting a service or parts of a service. Based on the results of these negotiations and other possibly available information such as previous experience with the providers (reputation assessments), the SP decides on where to deploy the service.

III. RELATED WORK

Over the last years, there has been a significant research effort in optimizing allocation of VMs in clouds, commonly with cost and performance as optimization objectives [8], [15], [17], [22], [24]. This research field, commonly referred to as cloud placement, scheduling, and/or brokering, started out focusing on static environments. Recently, the field has been extended to also include dynamic scenarios, including changes in cloud provider prices. Examples of the latter include work by Andrzejak et al., who propose a probabilistic model to optimize cost and performance under dynamic pricing schemes. They use an SLA model with tasks bound by deadline and budget, where varying numbers of VMs can be allocated to optimize these goals [5]. An evaluation based on historical spot instance prices from Amazon EC2 combined with publicly available grid workloads demonstrates how users can achieve large cost savings by bidding for high-CPU instances, and also achieve a balance between cost and service level (job deadline).

Similarly but from a cloud provider’s perspective, to optimize the revenue and energy cost while satisfying the demands of customers, Zhang et al. presents a MPC (Model Predictive Control) based resource management mechanism to dynamically adjust the capacity allocated to each VM type [26]. Experimental evaluations show that, compared with static allocation strategies, the proposed approach combining market economy and optimal control theory is very promising.

Service placement in multi-cloud scenarios has also been studied extensively in the past. Our previous contribution on this topic includes a novel cloud brokering approach that optimizes placement of virtual infrastructures across multiple clouds [23], which compared to single cloud deployment improves performance, lower costs, or provide a combination thereof. For scenarios where parameters such as pricing schemes and VM types are continuously changed, we propose a linear integer programming model for dynamic cloud scheduling via migration of virtual machines [16]. The
proposed model can be applied in various scenarios through selections of corresponding objectives and constraints, and offers the flexibility to express different levels of migration overhead when restructuring an existing virtual infrastructure where services are being hosted. Lucas-Simarro et al. go a step further to implement a scheduler capable of taking autonomous placement decisions based on different pricing schemes. In case of dynamic pricing scenarios, the scheduler decisions are based on a prediction model that estimates the price of the VMs in the next period [18].

A field closely related to the optimization of VM placement is study of the actual cloud provider pricing mechanisms. Wee studies the development of spot instance prices on Amazon EC2 [25], the most well-known cloud system with real-time pricing. While Wee observes that spot instances are around 50% cheaper than reserved instances, the observed deviations in spot instance prices over a year are very small, with only a few percent reduction in the cheapest prices. Ben-Yehuda et al. analyze the historical prices of EC2 spot instances and reverse engineer the pricing scheme. They conclude that prices are not market-driven but rather randomly generated within a tight interval [7]. Analogously, a statistical model of spot instance prices in public cloud environments is presented by Bahman et al. in [12], which fits Amazon’s spot instances prices well with a good degree of accuracy. To capture the realistic value of the cloud compute commodities, Bhanu et al. employ financial option theory and treat the cloud resources as real assets. The cloud resources are then priced by solving the finance model [21].

IV. Algorithms

In this paper, we study the cost-optimization problem from the perspective of a service provider, which can be simply formulated as follows: Given $n$ cloud providers provisioning resources with dynamic pricing schemes, our goal is to find the placement solution that minimizes the cost of deploying a service with $q$ components across those cloud providers. Notably, unlike the related works mentioned in the previous section, we assume dynamic pricing of provisioning requests, and thus, we assume that the price of hosting a service or a part of a service is not known to the service provider prior to negotiation with the target provider.

To investigate the effects of our dynamic pricing strategy, a number of placement algorithms for cost-minimization are evaluated in this contribution. We define an optimal algorithm (Permutation) that through exhaustive search finds the best solution. We also define a greedy heuristic and another approximation (First-fit) to the optimal algorithm. For the sake of comparison in the later evaluation, we also introduce two naive algorithms: Round-robin and Random. For the sake of clarity, we omit requirements on affinity or anti-affinity between deployed instances but we remark that such requirements can be included as additional constraints, as presented in [9].

A. Random

The Random algorithm outlined in Algorithm 1 partitions the service into components and deploys each component to a random cloud provider (see lines 3 and 4 in Algorithm 1). After that, negotiations with cloud providers are performed (see line 9) accordingly. Note that in the presented algorithms, \texttt{negotiate}(X, I) is an atomic operation that represents a bargaining action with a service request $X$ against cloud provider $I$. This action returns the cost of hosting $X$ in cloud provider $I$. If cloud provider $I$ does not accept the request $X$ (e.g., due to insufficient capacity), the cost is denoted as $+\infty$, indicating that hosting $X$ in $I$ is infeasible (see lines 10 and 11 in Algorithm 1).

\begin{algorithm}[h]
\caption{Random(Components, Clouds)}
\begin{algorithmic}[1]
    \State Input: Components = \{C\_0, C\_2, ..., C\_q-1\}, Clouds = \{I\_0, I\_2, ..., I\_n-1\}
    \State /* Randomly map service components to clouds. */
    \State mapping $\leftarrow \emptyset$;
    \For {$C \in \text{Components}$}
        \State $I_p$ $\leftarrow$ Randomly select a cloud $I_p \in \text{Clouds}$;
        \State mapping[$C$] $\leftarrow$ $I_p$;
    \EndFor
    \For {$I \in \text{Clouds}$}
        \State $X \leftarrow$ \{C $\in$ Components $|$ mapping[C] = I\};
        \If {$X \neq \emptyset$}
            \State cost $\leftarrow$ \texttt{negotiate}(X, I);
            \If {cost = +\infty}
                \State return N/A;
            \EndIf
        \EndIf
    \EndFor
    \State \textbf{return} mapping;
\end{algorithmic}
\end{algorithm}

B. Round-robin

The Round-robin algorithm maps the service components to cloud providers in a circular fashion. While this algorithm does not in anyway strive to find an optimal solution, it is simple and fast. The algorithm is outlined in Algorithm 2. For a component $C$ in the service, if there is no available cloud provider that can host it within $n$ rounds of negotiation, the algorithm fails (see lines 18 and 19).

C. Greedy

The Greedy algorithm strives to find the best match for each component, without considering how this affects the other components. For many problem classes, this type of algorithm tends to give good results while being simple and easy to implement. The Greedy algorithm used in this evaluation is outlined in Algorithm 3. For each component $C$ in the service, the lowest cost provider is selected to host it (see lines 6 – 13).
D. Permutation

The optimal Permutation algorithm, outlined in Algorithm 5 evaluates all possible permutations of components on all providers to find the global maximum. It first generates all possible partitions of a set of components. In combinatorics, the number of possible partitions of a set of size \( n \) is referred to as the Bell number, and denoted by \( B_n \). This can be calculated as \( B_n = \sum_{i=0}^{n} \frac{n!}{i!} \), where \( \{ \frac{n!}{i!} \} \) is the Stirling number of the second kind which is the number of ways to partition a set of \( n \) objects into \( i \) non-empty subsets [11]. For optimization purpose, given the number of cloud providers \( k \), we only need to generate \( B_{n,k} \) partitions for a service with \( n \) components using Algorithm 5.

All generated partitions are then evaluated via a recursive exhaustive search algorithm presented in Algorithm 4. To narrow the search space as much as possible, a branch-and-cut strategy is adopted. For each component set \( \Omega \) in a partition, two different branches are created to find a better solution. Each non-occupied cloud provider \( I \) is evaluated (see lines 11 – 26). The first branch just skips placing \( \Omega \) in \( I \), and continues with the next non-occupied cloud provider, keeping the cumulative cost unchanged. The other branch evaluates whether the sum of the cumulative cost and the cost of placing \( \Omega \) in \( I \) is higher than the optimum already obtained. If so, the current branch is stopped; otherwise, the algorithm continues with mapping \( \Omega \) to cloud \( I \) and adding the corresponding cost to the cumulative cost.

E. First-fit

The First-fit algorithm is a simplification of the Permutation algorithm. As soon as a feasible solution is found, the algorithm exits. This means that there is no guarantee a global optimum is found. This algorithm is outlined in Algorithm 6 (same as the optimal Permutation algorithm) but uses a parametric setting \( firstfit = true \) to halt upon finding the first feasible solution.

V. EVALUATION

Multiple external factors affect the results of the service deployment algorithms, including provider pricing schemes and workloads, SLA-tiered pricing, and specifications of the service(s) to deploy. We try to make reasonable assumptions about these factors in our evaluation and to avoid bias, the results are interpreted at a higher level, focusing on the overall trends rather than on exact numbers. The purpose of the evaluation is to highlight the conceptual differences between the proposed algorithms in as realistic environments as possible. The evaluation setup is discussed in detail below.

A. Evaluation setup

1) Testbed configuration: The tests are run on 3.30 GHz Intel Core i5-2500 machines with 8 GB of RAM and Gigabit Ethernet. The operating system is Linux stable 3.6. We host
Algorithm 4: TraversePartition(tmapping, partition, Clouds, indicator, cost, firstfit)

/* partition = \{Ω₀, Ω₁,..., Ωₘ₋₁\} where Components = \bigcup \{Ωᵢ \mid i \in [0,m-1]\}
   \forall i,j \in [0,m-1], i \neq j : Ωᵢ \cap Ωⱼ = ∅ */
1. if firstfit = true & & optimum \neq +∞ then
2. return;
3. end
4. if indicator ≥ m then
5. if tmapping.size() = m then
6. optimum ← cost;
7. mapping ← tmapping;
8. end
9. end
10. else
11. for I ∈ Clouds do
12. r ← cost;
13. /* check if I is not occupied. */
14. if \forall Ω ∈ partition & & tmapping[Ω] \neq I then
15. c ← negotiates(Ω[indicator], I);
16. if cost + c < optimum then
17. tmapping[Ω[indicator]] ← I;
18. cost ← cost + c;
19. TraversePartition(tmapping, partition, Clouds, indicator + 1, cost);
20. end
21. end
22. else
23. continue;
24. end
25. cost ← r;
26. tmapping[Ω[indicator]] ← ∅;
27. end

Algorithm 5: PFFAlg(Partitions, Clouds, firstfit)

Input: Partitions = \{G₀, G₁,..., Gₚ₋₁\}, Clouds = \{I₀, I₁,..., Iₙ₋₁\}
/* Traverse each partition G ∈ Partitions, and store the optimal mappings between Partitions and Clouds */
1. mapping ← ∅;
2. optimum ← +∞;
3. for G ∈ Partitions do
4. /* tm is a temporary mapping. */
5. tm ← ∅;
6. TraversePartition( tm, G, Clouds, 0, 0, firstfit );
7. if optimum ≠ +∞ then
8. return mapping;
9. end
10. return N/A;

TABLE I

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>small</th>
<th>medium</th>
<th>large</th>
<th>xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU(#cores)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

TABLE II

<table>
<thead>
<tr>
<th>Tiers</th>
<th>no. of instances</th>
<th>instance types</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE</td>
<td>1 ~ 4</td>
<td>S, M, L</td>
</tr>
<tr>
<td>LO</td>
<td>1 ~ 4</td>
<td>M, L, XL</td>
</tr>
<tr>
<td>DB</td>
<td>1 ~ 2</td>
<td>L, XL</td>
</tr>
</tbody>
</table>

three cloud providers on this testbed. These providers are configured with the Optimis cloud toolkit [10], which is a set of independent components. The Optimis toolkit can be adopted, either in full or in part, by cloud providers that provide infrastructure resources and by service providers that use these capacities to deliver services. In the evaluation, we use three components from the Optimis toolkit. The first is SLA Management, a service and client based on the WS-Agreement protocol [4] which is used for negotiating and creating SLAs between cloud providers and service providers [13]. The second is Admission Control, which is responsible for accepting or rejecting services for deployment in a provider. For clarity, the Admission Control algorithm used is a simple threshold-based function that accepts service requests if there is enough capacity. The third is the (previously discussed) SDO, which implements the service deployment and placement processes. For the purpose of this evaluation, the SDO is modified by implementing the placement algorithms discussed in Section IV.

2) Services: We use a service consisting of a typical three-tier Web application comprised of a front-end (FE), a logic (LO) and a database (DB) tier in the evaluation. To evaluate the algorithms against different services with diverse configurations, we vary the number of instances and the instance types for each tier, as shown in Table II. In addition, we use four different sizes of VMs, defined in terms of number of CPU cores used, as presented in Table I. We also use three different service availability SLAs, namely Bronze, Silver and Gold, which are summarized in Table III.

By varying these parameters, we get a considerable amount of different services. For example, the number of instances in the FE tier ranges from 1 to 4, and for each instance, its type is defined to be small or medium. Thus, there are 14 \((\sum_{i=1}^{4} C_i^{4})\) different configurations for the FE tier. Finally, incorporating the SLA types listed in Table III, we end up with 2940 \((14 \times 14 \times 5 \times 3)\) different services to evaluate our placement algorithms with.

3) Cloud providers: Three cloud providers with different average background load and free capacity are used in our experiments. In this contribution, we assume that the total capacity of each provider is uniform, i.e., 96 unit-capacity VMs. To be able to define a dynamic provider pricing scheme based on real-world conditions, the background load of the cloud providers must be realistic. To model the background
loads for these cloud providers, we use the Google cluster data trace [20], which consists of 30 days of usage data for a 12k-machine cell in May 2011.

The workload consist of traces for jobs, tasks and machines and contains over 60 GB of data. To get the total CPU load of the cluster as a relative value, we aggregate the CPU usage of all running tasks and divide this absolute value with the total capacity of the cluster. We use data from a 24 hour period and as seen in Figure 2. The load, represented by the blue line, varies between 15% – 60% during this interval. In order to highlight the impact of the dynamic provider pricing schemes, we scale the CPU load by a factor of 1.6 so that the load instead varies between 25% – 96%. This scaled load is represented by the black line in Figure 2.

![Fig. 2. CPU loads of the cluster.](image)

In each of the 2940 tests, one point in Figure 2 is randomly chosen as the background loads for each of the three cloud providers. These background loads are then combined with the service size and required SLA level by the provider when pricing a service request, as discussed below.

4) Dynamic pricing strategy: In order to evaluate the five placement algorithms, the providers must support dynamic pricing. Since little work has been done in researching how such schemes should work aside from modeling how prices are set for Amazon spot instances through reverse engineering [7], we therefore chose to implement a simplistic dynamic pricing scheme for the purpose of the evaluation.

Our dynamic pricing scheme is a straightforward function where the price set for a service offer is defined as a function of three factors: the background workload of the cloud provider, the size of the service request, and the required SLA level (availability). A step-function is defined to incorporate the required SLA level by multiplying the unit price depending on the currently available capacity. This means that the price for deploying a service with a high availability requirement is inversely proportional to the free capacity of the provider, and that highly loaded cloud providers charge more for high availability. The pricing function is illustrated in Table III.

The pricing scheme for the cloud providers are given by:

$$f(X, I) = g(X, I) \times \frac{X}{C(I)}$$

where $X$ denotes the size of the service request, $C(I)$ represents the available capacity of cloud provider $I$, and $g$ is the step-function defined in Table III.

With these configurations, we evaluate the behavior of the studied algorithms in terms of execution time, rounds of negotiations, and quality of solutions.

B. Evaluation results

Figure 3 and Figure 4 present the execution time (in seconds) for each algorithm with respect to the number of components in a service request. The value is given both as the average value and the standard deviation. As illustrated, the execution time for the Permutation algorithm increases dramatically as the size of the service becomes larger. This is due to the fact that Permutation algorithm is in essence a brute-force approach, which iterates through the search space that contains all possible partitions of components set. The number of partitions $B_{n,k}'$ rises rapidly as the number components $n$ increases, e.g., when $n = 10$, and the number of cloud providers $k = 3$, the number of components are $B_{10,3}' = 9842$.

![Fig. 3. Execution time for algorithms.](image)

In contrast, the Random, Greedy, Round-robin, and First-fit algorithms require much shorter execution time. The execution time of Greedy and Round-robin increases linearly with different slopes for increasing number of components. Considering the fact that the execution time for an algorithm is proportional to the number of negotiations, we do not present the numbers of negotiations here. In our evaluation, a round of negotiation lasted for around 0.6 seconds.
To evaluate the ability of algorithms to find optimal solutions, we reduce the total capacity of the providers by half, i.e., 48 single core VMs, and repeat the same experiments. As presented in Table IV, some of the algorithms now fail to find an optimal solution due to this capacity constraint of the providers.

After investigating the ability of the algorithms to find a solution at all, we next study the quality of the found solutions. We quantify the distance between suboptimal solutions and optimal solutions by defining the cost overhead, \( \alpha \), as

\[
\alpha = \frac{\text{cost} - \text{optimum}}{\text{optimum}},
\]

where \( \text{cost} \) is the cost of deploying the service request gained by the algorithm, and \( \text{optimum} \) is the cost of the optimal solution for the service. The cost for cases where no solution is found is denoted by \( +\infty \). A lower \( \alpha \) indicates a better solution, as it is closer to the optimal solution.

Notably, the Permutation algorithm always finds the optimum solution if one exists. Therefore, for the Permutation algorithm, \( \alpha = 0 \) is always true. For the other algorithms, since \( \text{cost} \geq \text{optimum} \), we have \( \alpha \geq 0 \).

We divide \( \alpha \) values into 5 intervals, i.e., \([0\%−25\%], (25\%−50\%], (50\%−75\%], (75\%−100\%], \) and \((100\%−+\infty)\). We then calculate the percentage of solutions whose \( \alpha \)-values fall into each interval for each algorithm and present the results in Figure 5. The Permutation algorithm is not included since its solutions always have an \( \alpha \)-value of 0.

Finally, to compare the performance of the algorithms with each other, we employ a similar metric as Equation (1):

\[
\beta = \frac{\text{cost}_1 - \text{cost}_2}{\text{cost}_2},
\]

where \( \text{cost}_1 \) and \( \text{cost}_2 \) are two solutions gained by two compared algorithms respectively.

We aggregate the results and present the average of \( \beta \) values in Table V. A value in the table represents an average \( \beta \), i.e., the comparison result for the algorithms in the corresponding column and row, respectively.

From Figure 5 and Table V, we conclude that the Greedy algorithm performs very well. For 90\% of all 2940 services, the deployment cost using Greedy is within 25\% of the optimal cost. The corresponding number for first-fit is around 50\%. Conversely, Round-robin and random perform much worse, with deployment costs twice that of the optimal solution in almost half of the cases.

To summarize the evaluation, Permutation is the best algorithm for finding optimal solutions. As it evaluates the entire search space, it either finds optimal solutions, or confirms that no solution is available. The downside of the Permutation algorithm is that the number of negotiation rounds grows rapidly and thus, execution time quickly grows infeasible. The First-fit algorithm on the other hand terminates when the first feasible solution is found, if one exists. This means that the quality of the solution always is the same as the first solution obtained by the Permutation algorithm. Consequently, the results shown in Table IV demonstrates that most of the solutions generated by the First-fit algorithm are suboptimal. Similar results are observed on Round-robin and Random and we also remark that while the Random algorithm by chance
might generate the optimal solution, it also has the largest percentage of no solution found cases. Interestingly, the very fast Greedy algorithm finds optimal solutions in more than half of all cases, and for 90% of the rest of cases, the quality of solution is within 25% from optimal. Greedy thus seems to be a very good trade-off between the quality of the solution and execution time.

As discussed previously, the exact numbers in the evaluation depend on the provider pricing schemes, background workload, size and composition of services, etc. However, we observe that the Greedy algorithm seems to perform very well.

VI. CONCLUDING REMARKS

In this contribution, we study a series of algorithms for cost-optimal cloud service deployment under dynamic pricing schemes. We perform an experimental evaluation using simulated deployments on cloud providers with dynamic pricing schemes. We then compare the algorithms with respect to execution time, ratio of successfully solved deployment cases, and the quality of the solution. Our experiments suggest that the greedy algorithm is a promising approach as it is very fast and also finds good solutions in most cases.

We believe that results of this research could be helpful in the design of scheduling algorithms and mechanisms in cloud environments with dynamic pricing schemes.

VII. ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive and valuable feedback, improving the quality of this work. We also thank Ahmed Ali-Eldin for sharing his knowledge and materials on workload analysis. Financial support has in part been provided by the European Community’s Seventh Framework Programme ([FP7/2010-2013]) under grant agreements no. 257115 (OPTIMIS), the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control, and the Swedish Government’s strategic effort ESSENCE.

REFERENCES