IaaS Service Selection Revisited

Kyriakos Kritikos[1(✉)] and Geir Horn[2]

[1] ICS-FORTH, Crete, Greece
kritikos@ics.forth.gr
[2] University of Oslo, Oslo, Norway
geir.horn@mn.uio.no

Abstract. Cloud computing is a paradigm that has revolutionized the way service-based applications are developed and provisioned due to the main benefits that it introduces, including more flexible pricing and resource management. The most widely used kind of cloud service is the Infrastructure-as-a-Service (IaaS) one. In this service kind, an infrastructure in the form of a VM is offered over which users can create the suitable environment for provisioning their application components. By following the micro-service paradigm, not just one but multiple cloud services are required to provision an application. This leads to requiring to solve an optimisation problem for selecting the right IaaS services according to the user requirements. The current techniques employed to solve this problem are either exhaustive, so not scalable, or adopt heuristics, sacrificing optimality with a reduced solving time. In this respect, this paper proposes a novel technique which involves the modelling of an optimisation problem in a different form than the most common one. In particular, this form enables the use of exhaustive techniques, like constraint programming (CP), such that both an optimal solution is delivered in a much more scalable manner. The main benefits of this technique are highlighted through conducting an experimental evaluation against a classical CP-based exhaustive approach.

1 Introduction

Cloud computing is a new computing paradigm that has revolutionized the way applications can be built and provisioned. Its high adoption is due to the main benefits that it delivers, which include flexible pricing and resource management as well as reduction of costs due to the outsourcing of infrastructure management.

This computing paradigm includes the potential delivery and exploitation of different service models, which include Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS), with a gradual release of management control from the requester to the provider. The most widely used and researched model is the IaaS one. In this model, an infrastructure in the form of a Virtual Machine (VM) is offered to requesters to enable them to create an execution environment for their application components. Apart from this infrastructure, suitable tools are also supplied to requesters to enable them to better exploit this cloud service kind, including suitable restful management APIs.

K. Kritikos et al. (Eds.): ESOCC 2018, LNCS 11116, pp. 170–184, 2018.
https://doi.org/10.1007/978-3-319-99819-0_13

Once cloud computing has been set and evolved, it has also led to the rise of a new application design and provisioning model based on micro-services. This model caters for a better separation and reuse of business functionalities while enables a more flexible adaptation of the micro-service application. In this model, an application is functionally split into a set of services, which are deployed individually in containers in different VMs. As such, to better manage such an application's provisioning, there is a need to cover the selection and adaptation of the underlying IaaS services. This means that an initial set of IaaS services needs to be selected according to the application requirements, while the application can be reconfigured at runtime by either migrating micro-services from one IaaS service to another or creating new instances of the micro-services, e.g., to handle the additional, unexpected workload that might arrive.

Focusing on IaaS selection, various approaches have been proposed [1] which differ along the optimisation solving techniques that they adopt and the optimisation objective kinds that they can handle. This differentiation also impacts the capabilities of each approach with respect to the well-known solving time-to-optimality trade-off. For example, exhaustive approaches, like those adopting techniques like Constraint Programming (CP), are more suitable for delivering optimal results but in the expense of increased solving time. On the other hand, heuristic approaches more rapidly deliver near-optimal results.

In any case, current approaches follow a classical way to model the optimisation problem, where variables are mainly used to denote decisions that need to be taken with respect to which service component should be mapped to which IaaS offerings from those that satisfy its local (e.g., hardware or location) requirements. However, this classical way cannot scale in sight of the plethora of IaaS services available on the market. Just focusing on one big cloud provider like Amazon, one has to select among tens of thousands of IaaS offerings. Even if multiple local constraints are being supplied per service component, this can reduce the offering number to hundreds just for one cloud provider. However, by considering the combinational nature of the optimisation problem to solve, this can lead to a huge solution space that cannot be handled by any exhaustive approach, while the results provided by any heuristic approach would be just non-optimal, as a very small part of the solution space can be examined.

In our opinion, this inherent difficulty in the IaaS service selection problem has not been well and appropriately addressed in the literature. In this respect, this paper goes one step forward by proposing a novel technique which can enable the use of an exhaustive approach to a modified modelling of the optimisation problem in such a way that the main benefits of optimality are supplied in a much more scalable manner. In particular, in this modelling, the decision space is regulated by variables which map the service components to respective VM attributes. As the number of VM attributes is usually quite limited while the size of their value domain is small, this leads to the production of an optimisation problem which is less complex and leads to a quite reduced solution space.

While this modelling is more suitable, it comes also with a certain flaw. In particular, a solution mapping to a value for all these variables might not be

associated with a real IaaS offering. This then required our approach to adopt a smart method to alleviate this. This method involves three main parts: (a) the derivation of the dependencies between the offering cost with respect to the values of the other VM variables in the form of a linear function; (b) the supply of if-then-else conditional statements which enable to reflect other dependencies between the different VM variables so as to further filter irrelevant combinations; (c) the post-processing of the produced solution to map it to a real one which has the least distance to the current, possibly virtual one.

The main benefits of our work are highlighted via an experimental evaluation assessing how well the IaaS selection problem is solved in the context of the Amazon cloud provider. The evaluation results show that our approach scales much better than a classical, exhaustive one and can deliver results of almost the same quality. Once the scalability limits are reached for the classical approach, the proposed one is also able to deliver results of even better quality.

The rest of the paper is structured as follows. The next section reviews the related work and provides important background knowledge. Section 3 analyses the proposed IaaS selection technique. Section 4 reports and discusses the main evaluation results. Finally, the last section concludes the paper and draws directions for further research.

2 Background

2.1 Related Work Analysis

Many approaches [2–4] have been proposed for VM consolidation in data centres. Such approaches tend to solve a similar problem, where instead of mapping application components to VMs, they associate VMs to respective hosts. As such, such approaches could be utilised only in the context of resource management internally within a cloud provider's data centre. However, their techniques are similar or equivalent to those used for solving the IaaS selection problem.

The IaaS selection approaches [1] either focus on local IaaS selection restricted to the context of one application component or on global IaaS selection where the selection concerns all components of a respective application. As this paper focuses over the second and more advanced form of the IaaS selection problem, the analysis is restrained over this form.

The global IaaS selection approaches differ [1] with respect to the solving technique adopted as well as in the number and kind of objectives optimised. Classically, depending on the solving technique used, there can be a trade-off between solving time and optimality. Exhaustive techniques like CP [5] or Linear Programming (LP) [6] attempt to explore the whole solution space so as to derive the most optimal solution. However, this exploration is costly with respect to solving time. In this respect, heuristic approaches have been adopted [7], such as nature-inspired ones, which tend to produce rapidly a sub-optimal solution.

Cost [8] is the dominant optimisation objective that is usually optimised [1]. However, there exist other approaches that also attempt to optimise resource-specific metrics [9], like the number of cores (mapping to the computational

power to be devoted to a micro-service component). Others focus on reducing network latency [10] to guarantee a more suitable service execution time. There is also an approach which attempts to cover multiple levels [11] by being able to map resource-specific metrics to service-specific ones. Such an approach is then able to optimise metrics which reside at the service/application level.

To reduce the solving time by still adopting an exhaustive technique, some approaches attempt to learn from the application execution history. Such a learning enables then to fix some parts of the problem and thus accelerate its solving. Learning-based approaches adopt different ways to conduct this learning. In [12], a combined stochastic programming and learning approach is proposed which attempts to remember bad solutions and to discard them when re-solving the same optimisation problem. On the other hand, the approach in [13] employs a rule-based method to derive the best deployments for both the current application and its components from the application execution history.

The latter kind of approaches is complementary to our work. Such approaches could be employed for further reducing the solving time. However, the main advancement of the state-of-the-art lies on the capability to not require any prior knowledge about the application execution but rely on smart techniques that better and more rapidly explore the solution space by still employing an exhaustive technique to guarantee optimality. In this respect, a better trade-off between optimality and solving time is reached with respect to the state-of-the-art which is the main subject of research here. Further, our work is more scalable with respect to the others due to its capability to rely on a constant solution space when then number of VM offerings is increased.

2.2 IaaS Allocation Problem

The classical IaaS allocation problem attempts to optimise one or more objectives at the IaaS resource level. Let $x_{i,j} \in \{0,1\}$ be a binary decision variable indicating that application component type $i \in \mathbb{I}$ can be hosted on IaaS offering $j \in \mathbb{J}$. It is noted that there are $|\mathbb{I}| \cdot |\mathbb{J}|$ binary decision variables regarding the assignment of all application component types. Furthermore, there are $|\mathbb{I}|$ decision variables $n_i \in \mathbb{n}_i \subset \mathbb{N}_0$ representing the number of instances of application component type i.

The allocation problem has given Q "quality" dimensions for which the goodness of an allocation is measured; e.g., cost- or performance-related dimensions. Let

$$v_q\left(\mathbf{X}, \mathbf{n} \,|\, \boldsymbol{\theta}\right) : \{0,1\}^{|\mathbb{I}| \cdot |\mathbb{J}|} \times |\mathbb{I}| \;\mapsto \mathbb{D}_q \tag{1}$$

be the value function in dimension $q \in \{1, \ldots, Q\}$ given the matrix of the binary allocations $\mathbf{X} = [x_{i,j}]$ and the vector of instance counts $\mathbf{n} = [n_1, \ldots, |\mathbb{I}|]^T$. The vector $\boldsymbol{\theta}$ represents the context parameters for the allocation. The context parameters can be related to cost, performance or any other value that can be considered constant for the allocation problem. As an example, consider the situation where a quality dimension d represents the overall cost of an allocation and θ_j is the cost of IaaS offer j. Then, the value function takes the form $v_d\left(\mathbf{X}, \mathbf{n} \,|\, \boldsymbol{\theta}\right) = \sum_i \sum_j n_i \cdot x_{i,j} \cdot \theta_j$.

For each quality dimension value there is a utility function indicating how good this value is on a normalised scale, i.e. $u_q\left(v_q\left(\mathbf{X}, \mathbf{n} \,|\, \boldsymbol{\theta}\right)\right) : \mathbb{D}_q \mapsto [0,1]$. The utility function is defined as the normalised value with respect to the extreme values of the domain.

$$u_q\left(v_q\left(\mathbf{X}, \mathbf{n} \,|\, \boldsymbol{\theta}\right)\right) = \frac{\sup \mathbb{D}_q - v_q\left(\mathbf{X}, \mathbf{n} \,|\, \boldsymbol{\theta}\right)}{\sup \mathbb{D}_q - \inf \mathbb{D}_q} \tag{2}$$

Two kind of constraints are involved in the problem modelling. The first kind involves component specific constraints that restrict the domain of respective decision variables. For each service component only one IaaS offering must be selected, which implies the following set of constraints

$$\sum_j x_{i,j} = 1 \quad \text{for all } i \tag{3}$$

The second kind of constraints attempts to reflect user requirements posed at the global level. For instance, if we consider the resource level, then we could have constraints for, e.g., VM offering characteristics like the cost and the number of cores. In general, the constraints of this kind can take the following form

$$g\left(\mathbf{X}, \mathbf{n} \,|\, \boldsymbol{\theta}\right) \leq a \tag{4}$$

Additional constraints might also be posed to express further user requirements, like component co-location constraints. The interested reader can find more details about such constraints in [11].

Given that the utility is normalised in all dimensions, each of them is a simple unit less number in the interval $[0,1]$, and the overall allocation utility can be computed as an affine combination of the utility dimensions, also known as the Simple Additive Weighting (SAW) [14] technique. The weights $w_q \in [0,1]$ can be usually calculated by following the Analytic Hierarchy Process (AHP) [15]. The overall utility to be maximised is then given as

$$U\left(\mathbf{X}, \mathbf{n} \,|\, \boldsymbol{\theta}\right) = \sum_{q=1}^{Q} w_q \cdot u_q\left(v_q\left(\mathbf{X}, \mathbf{n} \,|\, \boldsymbol{\theta}\right)\right) \tag{5}$$

subject to the constraints (3)–(4).

The main issue with the above problem formulation lies on the huge solution space as can be seen from the Cartesian product in (1). By considering just one cloud provider (Amazon) and that common hardware constraints (over core number, memory, and storage size) are imposed at the local level which lead to around 400 Amazon cloud offerings matching each application component, this means that for an application with just 3 components, the number of combinations could be at least 3^{400}. Thus, such a solution space is already quite large. So, imagine what would be the case for applications with a greater size. The use of an exhaustive solver would be out of the question, while heuristic techniques would just supply non-optimal solutions as it will be impossible for them to

check a great part of the solution space. This actually requires the proposal of a technique that more smartly explores or even filters the solution space. Such a technique is actually proposed in this paper and will be analysed in the following section.

3 Technique

In order to find a better trade-off between solving time and optimality, our technique attempts to modify the way the IaaS selection problem is modelled. The main rationale is that by changing the solution space and making it much smaller, we could still have the ability to exploit an exhaustive technique.

Indeed, this was the main idea that has been followed. Instead of mapping each service component to all the IaaS offerings that match it, we now associate it with the respective features of an IaaS offering, like the number of cores, the main memory size and so on. This new mapping has the advantage that the number of IaaS offering features is small and the value domain for that features is also small. Further, the problem now becomes independent on the number of IaaS offerings and thus more scalable.

However, this mapping comes with the penalty that the solution that is produced, mapping each service component to a value from the domain of each IaaS offering feature, might be virtual. This is actually quite probable as the offering space of any single provider is smaller than the solution space formulated by the cartesian product of the value domains of its IaaS offering features. In order to cope with this major issue, we have employed two main measures.

First, on the modelling side, we have introduced smart constraints that enable to further reduce the solution space, as it might be initially big, as well as guide the solution process towards picking more suitable combinations of values for the IaaS features.

Second, once a solution has been produced, we employ a post-processing logic aiming at making all IaaS offerings that have been mapped to the application components real. Such logic will be shown to employ a distance measure in order to guide the exploration for the finding of the most suitable, real IaaS offerings.

Both measures are now analysed in the following two sub-sections while the last one attempts to provide the complete modelling of the optimisation problem.

3.1 Smart Constraints

To reduce the solution space of a problem, one kind of measure would be to introduce special constraints which attempt to formulate dependencies between the main problem variables. Such constraints through the respective constraint propagation mechanism enable to restrict the solution space in a great extent.

As indicated in Sect. 2.1, one of the major factors always attempted to be optimised is cost. As such, we got the idea that we could introduce a respective constraint in the optimisation problem which correlates application component cost with the rest of the IaaS feature-based parameters. Such a constraint could be easily formulated if the exact cost model of an IaaS provider

was known. However, even if such a cost model was available, it could be quite complex and might require formulating a great number of logical constraints of the form: **if** $(f_2 == v_{f_2} \wedge f_3 == v_{f_3} \ldots \wedge f_m == v_{f_m})$ **then** $(f_1 = 0.1)$, where f_k represents IaaS feature k and f_1 is the feature representing the cost. Unfortunately, logical constraints are difficult to handle in any kind of mathematical programming paradigm. They also create major scalability issues when their number is large.

In this respect, another idea came to our mind. Instead of attempting to formulate all possible logical constraint combinations, we could introduce just a single function enabling to model the needed correlation. This then led us to resort to linear regression techniques which have exactly this goal: to map one parameter or variable to a set of other variables. Thus, in the end, we could express *cost* as a function of the IaaS features for each cloud provider. This could then take the following constraint form: $f_1 = R_p(f_2, f_3, \ldots f_m)$, where $R_p(\cdot)$ is the regression function for IaaS provider p.

We could employ non-linear regression techniques instead but this did not seem to be actually needed as we were able to produce a relative accurate linear cost function for two of the most major IaaS providers, i.e., Amazon and Google.

However, the derived function does not exactly and completely solve the current issue. It provides a mapping that enables us to become independent of cost and be able to derive it through the rest of the variables. However, as IaaS offering cost maps to a quite large value domain, this action enabled us to significantly reduce the initial solution space.

To still follow the idea of formulating dependencies, the next clever development that has been performed was to introduce a restricted form of logical constraints for a widely used feature with a quite small domain. Such logical constraints will not thus be great in number and could be still easily handled by an exhaustive technique like Constraint Programming (CP).

This led us to focus on the *number of cores* feature which happens to have the smallest value domain among the most widely used IaaS features while also plays an important role in influencing IaaS offering cost. As such, we just processed the whole IaaS offering space of each cloud provider and attempted to create mappings from each value of the *number of cores* feature to the respective minimum and maximum value that has been anticipated for the rest of the features, including *cost*. This led to the definition of the following form of constraints:

$$\begin{aligned}&\textbf{if } (f_2 == v_{f_2} \wedge p == 1) \textbf{ then}\\ &\quad (\min v_{f_1} \leq f_1 \leq \max v_{f_1}) \wedge (\min v_{f_3} \leq f_3 \leq \max v_{f_3})\\ &\quad \ldots \wedge (\min v_{f_m} \leq f_m \leq \max v_{f_m})\end{aligned}$$

where f_2 is the *number of cores* feature and p is a variable that denotes a certain IaaS provider.

By combining the above two constraint forms, the solution space is reduced as cost feature is automatically calculated by a function while the different values of the core number feature guide the solution process and enable us to pick more

correct values for the remaining IaaS features. This leads to a smarter solution space exploration that can rapidly diverge to the optimal solution.

3.2 Solution Post-processing

The produced solution may not be a valid one. The combination of IaaS feature values in the context of a certain IaaS provider does not guarantee that exactly a real IaaS offering can be designated. The introduction of smart constraints remedies slightly this but there is still a need for correcting this derived solution.

Such a correction or alignment is performed by examining the IaaS offering space of all providers to find a real offering which is as much as possible close to the derived virtual one. This involves first finding only the most relevant offerings from all providers via a normal matchmaking step, which can be performed ultra rapidly by employing unary matchmakers like the one in [16], and then performing the local search over them to find the most appropriate real IaaS offering.

The distance between the virtual and a real IaaS offering is calculated according to the following definition: $D\,(\text{real}, \text{virtual}) = \Delta_{\text{utility}}\,(\text{real}, \text{virtual}) + \Delta_{\text{position}}\,(\text{real}, \text{virtual})$, where Δ is a difference function. The first factor attempts to penalise the real IaaS offering based on the actual parameters that participate in the optimisation objectives of the IaaS selection problem. While the second factor attempts to penalise the real IaaS offering based on the distance of the position of the respective IaaS offering feature value within the (ordered) value domain of that feature.

By considering that the respective optimisation objective is only cost, the first term of the distance function could take the following form:

$$\Delta_{\text{utility}}\,(\text{real}, \text{virtual}) = \frac{cost_{\text{real}}}{cost_{\text{virtual}}} \cdot 100000$$

where $cost_{\text{real}}$ and $cost_{\text{virtual}}$ represent the cost of the two offerings. On the other hand, the second term of the distance function can be expressed as:

$$\Delta_{\text{position}}\,(\text{real}, \text{virtual}) = \sum_{b} \left| I_b\,(f_{b,\text{virtual}}) - I_b\,(f_{b,\text{real}}) \right| \cdot 1000$$

where $I_b(\cdot)$ represents the index function of the feature numbered as b which returns the position in the feature's (ordered) value domain for a specific value of that feature. The feature value is represented by $f_{b,\text{virtual}}$ and $f_{b,\text{real}}$ in the case of the virtual and real offering, respectively.

As it can be seen, the distance formula attempts to penalise more when we move far away from the expected utility of the solution and less when we pick more distant values for each feature with respect to its ordered value domain. This leads to imposing two levels of penalisation. As it will be shown in the evaluation section, this distance measure was enough for finding the right real solution out of a virtual one.

3.3 Optimisation Problem Formulation

The general process for solving the IaaS selection problem according to our approach follows three main steps: (a) problem formulation; (b) problem solving; (c) solution alignment, where the last step applies the respective distance-based search (see Sect. 3.2) for each application component with respect to the virtual IaaS offering derived for it.

In this subsection, we focus on the first process step by attempting to modify the formulation of the classical IaaS selection problem (see Sect. 2.2). Please note, though, that the same principles are followed which regard the use of the AHP and SAW techniques as well as linear utility functions.

The classical IaaS selection problem is, first of all, relaxed by replacing the binary decision variables with variables based on the smart constraints. The main decision variables of the relaxed optimisation problem comprise:

(a) component-to-feature variables of the form $x_{i,b}$ indicating that a certain value from the domain of feature b has been selected with respect to application component i. Thus, in contrast to the original problem, the domain of such variables now is a certain value set and not the boolean domain with just two possible values; and
(b) instance number variables for components as in the case of the original/class problem formulation; and
(c) variable p which denotes the IaaS provider.

This means that we have the introduction of one new decision variable, the modification of the first variable kind and the maintenance of the second with respect to the original problem.

The original constraints of the problem, (3)–(4) remain the same, and we are still maximising the overall utility (5). However, we do have a differentiation on the concrete level with respect to (4). In particular, the value of the different parameters at the global level can be actually easily computed from the sum of these parameters at the local level for each application component multiplied by the number of instances of that component (as we are considering mainly resource characteristics). For instance, the overall cost could be computed by the following formula: $v_1 = \sum_i x_{i,1} \cdot n_i$ if we consider that v_1 is the value of *cost* parameter which is indexed as 1 while $x_{i,1}$ maps to the local cost of the virtual IaaS offering for component i.

However, we do have now the introduction of new constraints, the smart ones, as indicated in Sect. 3.1.

$$\textbf{if } (p == P) \textbf{ then } x_{i,1} = R_p\left(x_{i,2}, x_{i,3}, \ldots x_{i,|\mathbb{J}_P|}\right) \quad \text{for all } i, P \tag{6}$$

$$\textbf{if } (x_{i,2} == v_{2,l} \wedge p == P) \textbf{ then} \tag{7}$$

$$\min v_{x_{i,1}} \leq x_{i,1} \leq \max v_{x_{i,1}} \wedge \min v_{x_{i,3}} \leq x_{i,3} \leq \max v_{x_{i,3}}$$
$$\ldots \wedge \min v_{x_{i,|\mathbb{J}_P|}} \leq x_{i,|\mathbb{J}_P|} \leq \max v_{x_{i,|\mathbb{J}_P|}} \quad \text{for all } l, P$$

Constraint (6) indicates that if a certain IaaS provider P is selected, the cost for each application component should be computed by applying the regression

function for that provider over the remaining VM features. While Constraint (7) introduces the smart constraints reflecting the dependencies between the *number of cores* and the rest of IaaS offerings for IaaS provider P.

Discussion and Implementation Details. As it can be seen from the above formulation, the optimisation problem does include a greater number of constraints which, however, enable to better explore as well as filter the solution space.

Such a problem is not linear so it cannot be solved by employing different exhaustive technique kinds. On the contrary, CP seems to be the most suitable candidate as it can handle both non-linear and logic-based constraints, while it is also able to cater for the introduction of both integer- and float-based variables.

Based on this analysis, our implementation has relied on using the MiniZinc language for specifying the constraint optimisation problem as well as different kinds of CP solvers which can be deemed as best for the new IaaS selection problem formulation depending on the number of optimisation objectives involved. The use of MiniZinc enabled us to easily evaluate a great set of CP solvers and find those that have the best possible performance.

4 Evaluation

The goal of the experimental evaluation was to assess whether the performance and optimality of our proposed approach does advance the state-of-the-art. To conduct such evaluation we have relied on a certain experimental framework able to control the way the optimisation problem is formulated according to certain configuration parameters. The experiments were performed in a laptop with the following characteristics: (CPU: Inter Core i5-2430M with 2 cores and 2.4 GHz frequency, RAM: 6 GB, Disk: 500 GB SSD).

4.1 Experiment Configuration

Three main evaluation parameters were considered:

- cost as a parameter for evaluating the optimality of the examined approaches for only single objective optimisation problems;
- the solution utility as the parameter for evaluating the optimality of the examined approaches for multi-objective optimisation problems
- the solving time, i.e., the time required for solving a certain model of an optimisation problem, including any kind of solution post-processing time.

Each experiment was conducted in a series of steps by step-wisely varying one configuration parameter while leaving the rest stable. Each experiment step was computed (30) times and average values from the raw data were calculated for each approach considered and each from the above evaluation parameters.

The examined approaches were the following: (a) a classical problem formulation approach denoted as *OLD*; (b) a new problem formulation approach based

on our work without the solution post-processing denoted as *NEW*; (c) the same approach as the previous one along with the solution post-processing denoted as *NEW_FIXED*. Each approach was implemented in Java and relied on the use of the best possible solver according to the actual problem at hand (variation point is the number of objectives as indicated in the previous section). To not make each solver run forever, a certain time limit was introduced (100 s) for the solving process in order to also reduce the execution time of the experiments.

The experimental framework involves using different configuration parameters to control the way the optimisation problem is generated: (a) the number of application components; (b) the number of cloud providers; (c) the number and kind of IaaS features; (d) the number of optimisation objectives. To keep the problem complexity low so as to also evaluate in an error-free manner the *OLD* approach, the cloud provider number was kept to the minimum (1, the Amazon provider) while the kind of IaaS features considered were the most common (core number, memory & disk size). Thus, to conduct the experiments, we varied mainly the component and optimisation objective number. We should also note that we have taken as a base all the actual real IaaS offerings available at the time of the experiments for Amazon AWS.

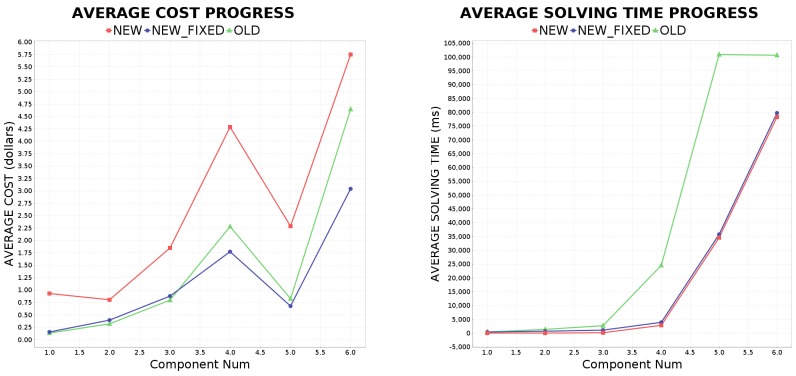
As there is no actual benchmark for evaluating IaaS selection approaches, we have relied on randomly creating IaaS service requests for each application component in each experiment step execution. Each such request attempts to randomly select a specific value from the value domain of each IaaS feature considered (out of the 3 ones in the current experiment configuration). This looks like a more correct way to produce the respective requests as we can consider that there is already widespread knowledge about which are the most suitable values for each IaaS feature across the whole developer community.

4.2 Experiment Analysis

Two main experiments have been conducted, which are now analysed below, having as their main variation point the number of optimisation objectives.

First Experiment. In this experiment, we consider only *cost* as the main optimisation objective and attempt to vary the number of application components from 1 to 6. The respective experiment results are depicted in Fig. 1.

The solving time results are quite expected. The two variants of the proposed approach seem to scale much better than the classical approach. Further, the classical approach already reaches the time limit when the component number equals to 5. The performance of the two proposed approach variants is similar. This means that the post-processing step does not occupy a great proportion of the overall approach execution time. In fact, the respective search time is mainly proportional to the number of matched IaaS offerings and application components. So, as the application component number linearly increases and the match number remains more or less stable, the post-processing time also increases linearly. So, the exponential behaviour in the two variants' performance is mainly due to the exponential increase of the solution space.



(a) Cost results (b) Solving time results

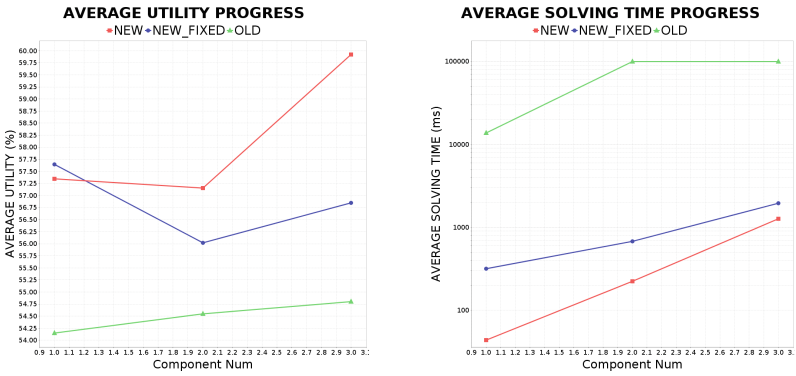
Fig. 1. 1st experiment results

Concerning cost, i.e., the current optimality parameter, we can see that the non-aligned approach variant does not perform so well with respect to the rest of the approaches. This is mainly related to the precision of the linear regression function. As this precision is imperfect, we expect that the difference between the utility derived by this approach variant and the utility of the other approaches will be increased when the application component number increases. This is the actual case in the experiment results. With the sole exception that the utility difference between *NEW* and *OLD* gets reduced at some time point, mainly due to the deterioration of the utility on the side of the *OLD* approach.

Such a deterioration is mainly due to the fact that the *OLD* approach is starting to have a hard time in better exploring the solution space. Such that when the time limit is eventually reached, the quality of the solution deteriorates significantly. This gives the opportunity for the overall proposed approach, the *NEW_FIXED* to surpass the *OLD* one when the component number becomes 4.

Second Experiment. In the second experiment, the same control parameter is varied (from 1 to 3) but the number of optimisation objectives is now 2. These objectives include *cost* and *total number of cores*. The combination of these objectives make sense as there is usually a trade-off between computation power and cost. The respective results from this experiment are shown in Fig. 2.

As it can be seen, the *OLD* approach already reaches its time limit when the component number is two. This signifies that the increase in the number of objectives makes the exploration of the solution space more time consuming such that the exponential increase in that space's size makes the respective solver to more rapidly exceed the time limit posed. On the other hand, the two variants of the proposed approach are much more scalable while their solving time is always below 2 seconds. The time difference is again quite small between these variants, mainly due to the post-processing time penalty. This time penalty seems to be increased quite slightly with the increase in the component number.



(a) Overall Utility results (b) Solving time results

Fig. 2. 2nd experiment results

The overall utility results seem a little bit surprising. As we can see, the best approach is now *NEW* followed by *NEW_FIXED* and *OLD*. This looks more correct as *NEW* has more freedom to find the virtual solution exhibiting the best possible trade-off while *NEW_FIXED* is restrained over the capabilities of the current offerings locally matching each application component. Such capabilities might thus be less performant than those of the virtual solution found. The bad utility result of *OLD* is mainly due to its hard time to explore the solution space. Which is rather immediate than in the case of single-objective problems. While not shown here, due to page restriction reasons, the only case where *OLD* is better than the rest of the approaches is with respect to the overall cost (i.e., a part of the objective set) and only when the component number equals to 1.

4.3 Discussion

As it can be observed from the experiment results, our novel approach is much scalable and performant than the classical IaaS selection approach. Further, it is able to find a better solution in most of the cases, due to the solution space restrictions that the classical approach is facing. Only when the solution space is quite small, the classical approach could be considered as slightly better in optimality but such a case is not so frequent in reality. This validates the superiority of our approach which opens up new opportunities for solving IaaS and service selection problems in general in a much more optimal and rapid way.

5 Conclusions

This paper has presented a novel approach which exhibits a better trade-off between optimality and solving time for the IaaS selection problem. In particular, this approach models differently this optimisation problem and enables as such

the scalable use of state-of-the-art exhaustive solvers for optimally solving it. The approach involves changing the decision variables as well as introducing smart constraints in the model of the optimisation problem. This enables to reduce the solution space significantly as well as have a better way to explore it. Due to a side-effect of the modified problem modelling, the proposed approach involves a solution post-processing step which attempts to guarantee that the components of the application at hand are mapped to real IaaS offerings.

The following future work directions will be pursued. First, we plan to support more cloud providers apart from Amazon and Google in our implementation as well as more thoroughly evaluate our approach in the increased solution space that will be formulated. Second, we plan to expand the modelling of the optimisation problem to cover multiple levels of abstraction. Third, we will explore whether a learning-based method could be additionally employed to further reduce the solving time of our approach. Finally, we will investigate whether additional smart constraints can be incorporated into the optimisation problem model such that the solution post-processing can be avoided.

Acknowledgements. The research leading to these results has received funding from European Union's Horizon 2020 programme under grant agreement No 731664 (concerning the Melodic EU project).

References

1. Sun, L., Dong, H., Hussain, F.K., Hussain, O.K., Chang, E.: Cloud service selection: state-of-the-art and future research directions. J. Netw. Comput. Appl. **45**, 134–150 (2014)
2. Dong, J., Jin, X., Wang, H., Li, Y., Zhang, P., Cheng, S.: Energy-saving virtual machine placement in cloud data centers. In: CCGrid, pp. 618–624. IEEE/ACM (2013)
3. Casalicchio, E., Menascé, D.A., Aldhalaan, A.: Autonomic resource provisioning in cloud systems with availability goals. In: CAC, Miami, Florida, USA, vol. 1(1–1), p. 10. ACM (2013)
4. Jayasinghe, D., Pu, C., Eilam, T., Steinder, M., Whally, I., Snible, E.: Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement. In: SCC, Washington, DC, USA, pp. 72–79. IEEE Computer Society (2011)
5. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier Science Inc., New York (2006)
6. Van Hentenryck, P., Saraswat, V.: Strategic directions in constraint programming. ACM Comput. Surv. **28**(4), 701–726 (1996)
7. Dastjerdi, A.V., Buyya, R.: Compatibility-aware cloud service composition under fuzzy preferences of users. IEEE Trans. Cloud Comput. **2**(1), 1–13 (2014)
8. Chaisiri, S., Lee, B.S., Niyato, D.: Optimization of resource provisioning cost in cloud computing. IEEE Trans. Serv. Comput. **5**(2), 164–177 (2012)
9. Soltani, S., Elgazzar, K., Martin, P.: QuARAM service recommender: a platform for IaaS service selection. In: UCC, Shanghai, China, pp. 422–425. ACM (2016)
10. Klein, A., Ishikawa, F., Honiden, S.: Towards network-aware service composition in the cloud. In: WWW (2012)

11. Kritikos, K., Plexousakis, D.: Multi-cloud application design through cloud service composition. In: Cloud, pp. 686–693. IEEE Computer Society, June 2015
12. Horn, G.: A vision for a stochastic reasoner for autonomic cloud deployment. In: Second Nordic Symposium on Cloud Computing & Internet Technologies (NordiCloud 2013), pp. 46–53. ACM, September 2013
13. Kritikos, K., Magoutis, K., Plexousakis, D.: Towards knowledge-based assisted IaaS selection. In: CloudCom, pp. 431–439. IEEE Computer Society, December 2016
14. Hwang, C., Yoon, K.: Multiple Criteria Decision Making. Lecture Notes in Economics and Mathematical Systems. Springer, Heidelberg (1981). https://doi.org/10.1007/978-3-642-48318-9
15. Saati, T.: The Analytic Hierarchy Process. McGraw-Hill, New York (1980)
16. Kritikos, K., Plexousakis, D.: Novel optimal and scalable nonfunctional service matchmaking techniques. IEEE Trans. Serv. Comput. **7**(4), 614–627 (2014)