2nd IEEE International Conference on Cloud Computing Technology and Science

Cost-Optimal Outsourcing of Applications into the Clouds

Immanuel Trummer*, Frank Leymann†, Ralph Mietzner† and Walter Binder‡
*Artificial Intelligence Laboratory, Ecole Polytechnique Fédérale de Lausanne
Email: immanuel.trummer@epfl.ch
†Institute of Architecture of Application Systems, University of Stuttgart
Email: {leymann, mietzner}@iaas.uni-stuttgart.de
‡Faculty of Informatics, University of Lugano
Email: walter.binder@usi.ch

***Abstract*—Commercial services for provisioning software components and virtual infrastructure in the cloud are emerging. For customers, this creates a multitude of possibilities for outsourcing part of the IT-stack to third parties in order to run their applications. These possibilities are associated with different running costs, so cloud customers have to determine the optimal solution. In this paper, we present and experimentally evaluate an algorithm that solves the corresponding optimization problem.**

We assume that applications are described as templates, fixing the deployment structure and constraining the properties of the used soft- and hardware components. Different parts of the application may be outsourced to different providers and several levels of outsourcing can be considered. However, dependencies between different parts of the application have to be respected. Our algorithm decomposes the application graph in a first step in order to discover all suitable cloud provisioning services from a registry. It determines the optimal solution by representing the problem as constraint optimization problem that can be solved by an existing solver implementation.

***Keywords*-Cloud computing; Minimizing application running costs; Selecting optimal providers; Constraint programming**

I. Introduction

Cloud computing [1] can be seen as a modern realization of the utility computing vision that was already formulated by John McCarthy in 1961: "... computing may someday be organized as a public utility ...". Cloud computing allows customers to outsource applications into the *cloud*, using *provisioning services* from one or several third-parties. Depending on the level where the IT-stack is divided, this is called *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)*, or *Software as a Service (SaaS)*. Amazon [2] for example offers virtual servers that can be rented for an hourly charge (IaaS)—eventually with installed middleware or databases for additional license fees. Salesforce [3] hosts business software that customers can subscribe to (SaaS). Choosing different levels of outsourcing and different providers may result in different running costs for the same application. If applications consist of several deployment trees, specific levels of outsourcing and different providers may be chosen for each of them. The number of cloud provisioning offers is currently growing and so is the number of possibilities for running an application in the clouds. It becomes crucial for cloud customers to choose the best one in order to minimize their running costs, especially for long-running applications. The original scientific contributions of the paper are *i)* an algorithm for determining the optimal combination of provisioning services for running an application, and, *ii)* an implementation of the algorithm that we evaluated with several benchmarks.

The remainder of the paper is organized as follows. Sect. II gives an overview of the adopted algorithm and the context within it is executed. In Sect. III, we introduce the general concepts by means of an example which will also be used throughout the remainder of the paper. Sect. IV introduces the formal models for applications, provisioning services, and provisioning plans. Sect. V describes how the algorithm decomposes the application in order to find applicable provisioning services and Sect. VI explains how the algorithm searches for the cost-optimal solution. In Sect. VII, we shortly describe the implementation and show some evaluation results. Sect. VIII compares our approach with related work and Sect. IX concludes the paper.

II. Algorithm Overview

The algorithm that we present in this paper is executed by the *Provisioning Optimization Engine* (see Fig. 1). Its goal is to find an optimal combination of provisioning services for running a given application. The input is a template of this application, a description of the private software and hardware resources available for running parts of the application, and constraints on the provisioning plan prescribing the use of specific services. The provisioning optimization engine consults a public registry in which cloud providers advertise services in a machine-readable form. The output of the provisioning optimization engine is a plan that describes which provisioning services to use in order to minimize the running costs.

The deployment structure of the application is described as a graph; nodes may represent all kinds of soft- and hardware components. The optimization algorithm consists of two steps. In the first phase (see Sect. V) it enumerates all possibilities of partitioning the application and requests corresponding services for each partition from the registry. The second part of the algorithm (see Sect. VI) searches for an optimal way of provisioning the desired application by transforming the provisioning planning problem into a constraint optimization

978-0-7695-4302-4/10 $26.00 © 2010 IEEE
DOI 10.1109/CloudCom.2010.64

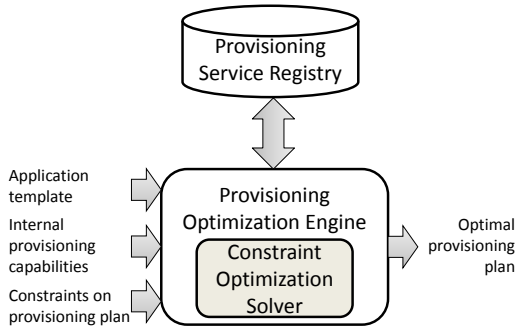

Fig. 1. Optimization architecture

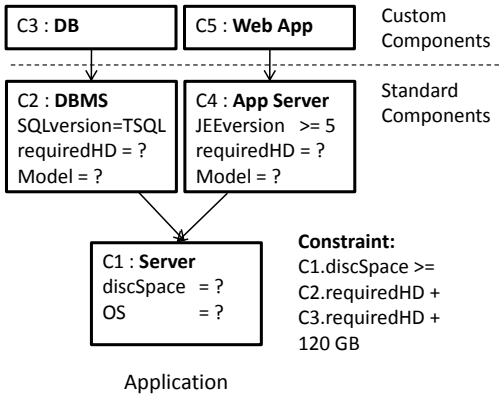

Fig. 2. Example: Application Template

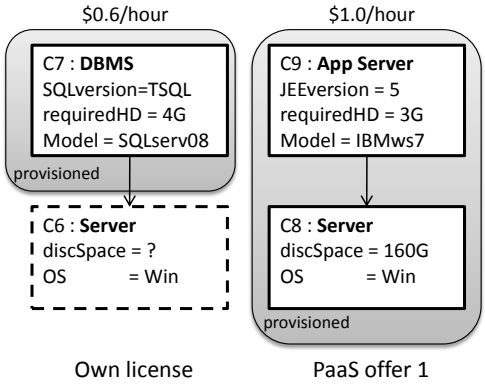

Fig. 3. Example: Available services

problem (COP). Constraint programming is a generic formalism that allows to formulate problems of a broad range of domains (like operations research and artificial intelligence). Transforming the problem allows to use existing constraint solver implementations in order to solve it efficiently.

III. Example Scenario

In this section, we present a motivating scenario. We will refer to it later in order to explain our formal model and the algorithm. Let us assume that we want to determine the best way of running a simple Customer Relationship Management (CRM) application for our enterprise within a cloud environment. Fig. 2 shows a graphical representation of the corresponding application template as deployment graph, similar to UML deployment diagrams. The graphical representation is meant to reflect the formal model that we will present in the next section.

The application consists of five components, numbered C1 to C5 in Fig. 2. The type of the different components is indicated in the top line. Component C1 is of type "Server", a DBMS (C2) as well as an application server (C4) are deployed on it. The application server acts as container for a Web interface (C5), the DBMS as container for the corresponding database (C3). DBMS, application server and server are standard components and we might be able to find corresponding cloud services. The Web interface and the database schema however are specific to the CRM application, hence we have to deploy and manage them ourselves. Each component type is associated with a type-specific set of attributes that describe the component in more detail. Let us assume that the SQL code creating the database (component C3) requires support for T-SQL (a specific SQL extension). Hence we constrain the property “SQLversion" of the DBMS as shown in Fig. 2 while other properties remain unconstrained. In a similar way we constrain the application server to support Java at least in version 5. We can also define constraints on combinations of attributes instead of single attributes. In the example we state that the available disc space of the server machine must be at least equal to the required hard disc for installing and running DBMS and application server plus 120 GB. These 120 GB represent the disc capacity needed for the database and the Web interface. Fig. 3 shows two available provisioning services. They are represented as graphs, similar to the application graph. Note the difference between the components that a service provisions and the components that a service requires in order to deploy on them. In our scenario, Amazon offers to provision a virtual server machine together with IBM WebSphere Application Server running on it against a fee of $1.0 per hour. Additionally, we can buy licenses and administrate Microsoft SQL Server DBMS ourselves. The total licensing and management costs per Microsoft SQL Server instance would be $0.6 per hour. In order to deploy this software, we need a Windows server.

The input to the provisioning optimization engine would be a description of the application as template (graphically represented in Fig. 2), a description of our own capability of providing and running the Microsoft SQL Server software (Fig. 3, left side) and a constraint stating that both the custom components cannot be outsourced. The provisioning engine would then retrieve the Amazon provisioning service (Fig. 3, right side) from a public registry. One provisioning plan would be to use the service from Amazon and deploy our own software license on the virtual Amazon server. The total running costs would be $1.6 per hour. This plan implies a mapping between the components from the services to the components of the template: C6 and C8 to C1, C7 to C2

and C9 to C4. If available, another possibility would be to use a service (not shown in Fig. 2) provisioning a virtual server machine with installed DBMS and application server as one package. It would be up to the optimization engine to compare all possibilities in order to find one with minimal running costs. Note that we assume that the requirements on the infrastructure of one application instance are known in advance. Dynamic scalability could be achieved on a higher level by adapting the number of instances that are created following our optimal provisioning plan.

IV. Formal Model

In this section, we introduce a formal model for describing applications, provisioning services and provisioning plans.

A. Components

Components are used to describe the application that has to be provisioned and to describe the provisioning services that are available for provisioning parts of it. Components are described by a type and some type-specific attributes. We denote by $\mathcal{T}$ the set of all component types (like "DBMS"). $\mathcal{A}_t = \{a_1, a_2, \ldots\}$ designates the set of attributes for some type $t \in \mathcal{T}$, $\mathcal{A} = \bigcup_{t \in \mathcal{T}} \mathcal{A}_t$ the set of all attributes of all types. Each of the attributes has a domain of possible values (e.g. the set {SQL92, SQL99, ...} for attribute "SQLversion"). For attribute a we denote by $dom(a)$ the set of possible values. By $DOM = \bigcup_{a \in \mathcal{A}} dom(a)$, we denote the set of all possible attribute values.

B. Applications

Definition 1 (Application Template): An *application template* is a tuple $ap = (G_{ap}, R_{ap})$ where

- $G_{ap} = (C_{ap}, D_{ap}, T_{ap}, V_{ap})$ is called *application graph.*
- $R_{ap} = \{r_1, \ldots, r_n\}$ is a set of global constraints.

G_{ap} corresponds to a directed graph where the components $c \in C_{ap}$ are the nodes and the deployment relationships between the components $d \in D_{ap}$ the edges. The semantic of a pair $(c_1, c_2) \in D_{ap}$ is that component c_2 is deployed on component c_1. The graph has to be acyclic since no component can be deployed on itself. It may be disconnected since an application can consist of several deployment trees. The function $T_{ap} : C_{ap} \to \mathcal{T}$ assigns components to their types, the function $V_{ap} : C_{ap} \times \mathcal{A} \to \mathcal{P}(DOM)$ assigns attributes to a set of allowed values. Note that V_{ap} is a partial function, for each component it assigns only those attributes to values that belong to the type of this component. The restrictions $r_i \in R$ allow to model dependencies between the attributes of different components (we will abstain from a more formal definition here). An example would be the disc space constraint in Fig. 2. Such constraints could also relate quality of service properties of the application to component attributes and set minimum requirements on them.

C. Provisioning Services

Definition 2 (Provisioning Service): A *provisioning service* $s \in S$ is described by a tuple (G_s, P_s, c_s) where

- $G_s = (C_s, D_s, T_s, V_s)$ is called the *service graph.*
- $P_s \subseteq C_s$ designates the set of *provisionable components.*
- $c_s \in \mathbb{N}$ represent the costs per time unit for using the service.

The definition of the service graph corresponds to the definition of application graphs. The service graph must be connected, hence we refer to it as *service tree*. The nodes of the service graph represent either components that the service can provision (*provisionable components*) or components that the service requires in order to deploy on them (*prerequisite components*). Only the root of the service tree may be a prerequisite component. Again, the service graph is labeled by the functions T_s and V_s that describe type and properties of the components. For provisionable components, each attribute is assigned to exactly one value by V_s. The service may be able to deploy on a range of different components, so V_s may assign several possible values per prerequisite component and attribute. We denote by $S = \{s_1, s_2, \ldots, s_n\}$ the total set of provisioning services, including public services as well as our own provisioning capabilities. Note that we do not impose restrictions on the maximal number of service invocations. This seems a reasonable assumption for public cloud services but represents a simplification for private provisioning capabilities.

D. Provisioning Actions

Definition 3 (Provisioning Action): A *provisioning action* corresponds to the possibility of using one specific service for provisioning one specific part of the application graph. It is described by a tuple $pa = (s, M)$ where

- $s = (G_s, P_s, c_s)$ is a provisioning service.
- $M : C_s \to C_{ap}$ is the *component mapping.*

The component mapping maps each component of the provisioning service to its *match*, an application component. Mapping a prerequisite component of the service tree to a component of the application graph means that the service can deploy on this component. Mapping a provisionable component of the service to an application component means that the service would provision this component. By $M[P_s]$ we denote the application components that can be provisioned by service s, given component mapping M. Provisioning components C1 and C4 of the example application (Fig. 2) by the service from Amazon (Fig. 3) would be a provisioning action.

The mapping function has to satisfy several constraints. *i)* Each application component must be assigned to a compatible component in the service graph. We say that a component $c_1 \in C_{ap}$ is compatible with a component $c_2 \in C_s$, written $(c_1 \sim c_2)$, if they are of the same type $(T_{ap}(c_1) = t = T_s(c_2))$ and if they have at least one common value for each attribute $(\forall a \in \mathcal{A}_t : V_{ap}(c_1) \cap V_s(c_2) \neq \emptyset)$. *ii)* The component mapping function M has to be injective (different service components cannot be mapped to the same application component). *iii)* The mapping has to be consistent with regards to the deployment

structures of service tree and application graph. So if one component is deployed on another in one graph, the match of the first component must also be deployed on the match of the second in the other graph (see (1)). *iv)* M must be a total function. So each component of the provisioning service must have a match among the application components. But not every component of the application must have a match among the service components.

Later we will show an algorithm that is able to create a component mapping step by step for two given trees. So we will work with mappings that do not comply with the last condition (until they are completed) and call them *partial component mappings*. In the following we will assume that the application template, ap, and the set of available services, S, are fixed. We denote by $PA = \{pa_1, pa_2, \ldots, pa_k\}$ the total set of possible provisioning actions then.

$$\forall c_1, c_2 \in C_s : \\ (c_1, c_2) \in D_s \iff (M(c_1), M(c_2)) \in D_{ap} \quad (1)$$

E. Provisioning Plans

Definition 4 (Provisioning Plan): A *provisioning plan* $P \subseteq PA$ is a set of provisioning actions representing a valid possibility for provisioning the given application.

A provisioning plan for the example application was described in Sect. III using the two services shown in Fig. 3. For each application component, a plan must contain exactly one provisioning action that provisions it. On the other hand, a component can be mapped to the prerequisite components of different provisioning actions. All service components that are assigned to one application component must be compatible. Finally, we assume that some of the application components are already bound to specific ways of provisioning (e.g. custom components for which no publicly available services can be discovered, like C3 and C5 in Fig. 2). By $PA_{fix} \subseteq PA$ we denote the set of provisioning actions that have to be included in every provisioning plan (see (2)). For a given application and set of services, we denote by $\mathcal{P}$ the set of all possible plans for provisioning the application. For any $P = \{(s_1, M_1), (s_2, M_2), \ldots\} \in \mathcal{P}$ we denote by $cost(P)$ the total running costs that the plan implies (see (3) - c_s designates the running costs of service s as in Def. 2). We simplify and assume that no additional running costs remain for outsourced components. Note that the same service may be used several times in the same application instance for different parts of the application. An optimal provisioning plan P_{opt} fulfills (4).

$$PA_{fix} \subseteq P \quad (2)$$

$$cost(P) = \sum_{(s,M) \in P} c_s \quad (3)$$

$$\forall P \in \mathcal{P} : cost(P_{opt}) \leq cost(P) \quad (4)$$

V. Determining all Possible Provisioning Actions

In this section, we will explain how all interesting provisioning services can be retrieved for a given application, and how the set of applicable provisioning actions can be derived out of them. Within the listings we will use the following notations.

Algorithm 1 Determine component mappings among services and applications

```
1: function FIND_MAPPINGS(G_s, G_ap, m)
2:     if ∀c_1 ∈ C_s ∃c_2 ∈ C_ap : (c_1, c_2) ∈ m then
3:         return {m}
4:     end if
5:     C_s^av ← {c_1 ∈ C_s | ∄c_2 ∈ C_ap : (c_1, c_2) ∈ m}
6:     C_ap^av ← {c_2 ∈ C_ap | ∄c_1 ∈ C_s : (c_1, c_2) ∈ m}
7:     c_s ←_oneOf {c ∈ C_s^av | D_s^-1(c) ∉ C_s^av}
8:     if c_s = root(G_s) then
9:         C_ap^m ← {c ∈ root(G_ap) ∩ C_ap^av | c ~ c_s}
10:    else
11:        C_ap^m ← {c ∈ D_ap(m(D_s^-1(c_s))) ∩ C_ap^av | c ~ c_s}
12:    end if
13:    M ← ∅
14:    for all c_ap ∈ C_ap^m do
15:        m̃ ← m ∪ {(c_s, c_ap)}
16:        M ← M ∪ find_mappings(G_s, G_ap, m̃)
17:    end for
18:    return M
19: end function
```

We will use the deployment graph relation $D_{ap} \subseteq C_{ap} \times C_{ap}$ as function: $D_{ap}(c)$ denotes the set of all components deployed on $c \in C_{ap}$ and $D_{ap}^{-1}(c)$ the container component of c ($D_{ap}^{-1}(c) = \epsilon \notin C_{ap}$ if c has none). The same notation applies for the service graph. $root(G_{ap})$ or $root(G_s)$ denotes the set of components that are not deployed on other components. Given a subset of application components $\widetilde{C} \subseteq C_{ap}$ we denote by $G_{ap}^{\widetilde{C}}$ the corresponding vertex induced subgraph of the application graph (the same notation applies for the service graph). By C_{ap}^{fix} we refer to the application components for which the provisioning is already fixed by the provisioning actions in PA_{fix} (like C3 and C5 in the example application):

$$C_{ap}^{fix} = \{c \in C_{ap} | \exists (s, M) \in PA_{fix} : c \in M[P_s]\} \quad (5)$$

A. Finding Possible Mappings between Component Trees

Given a service tree and a subtree of the application graph, Alg. 1 determines all valid mappings from the service tree to (part of the) application tree. We will use this function in order to find all possible provisioning actions for one specific service. It has three input parameters: the service tree $G_s = (C_s, D_s, T_s, V_s)$, the subtree of the application graph $G_{ap} = (C_{ap}, D_{ap}, T_{ap}, V_{ap})$, and a partial completed component mapping m among these two graphs ($m = \emptyset$ for non-recursive calls). The output of the function is the set of all possible, complete mappings between the two component trees. Each instance of the function tests at first whether the partial mapping m is already complete and returns the mapping as result in this case (line 2). If the mapping is not complete then there are still service components that are not assigned to an application component (they are *available*). The function arbitrarily selects one service component (c_s) that was not matched, yet, but whose container component (if it has

Algorithm 2 Determine set of possible provisioning actions

```
1: function PROVISIONING_ACTIONS(G_ap, C_ap^fix)
2:     PA ← ∅
3:     for all c ∈ C_ap do
4:         at ← {c}; S ← ∅
5:         while
   ∃(c1, c2) ∈ D_ap : c1 ∈ at ∧ c2 ∉ (C_ap^fix ∪ at)  do
6:             at ← at ∪ {c2}
7:         end while
8:         if c ∉ C_ap^fix ∧ c ∈ root(G_ap) then
9:             S ← S ∪ find_services(G_ap^at, at)
10:        end if
11:        S ← S ∪ find_services(G_ap^at, at − {c})
12:        for all s = (G_s, P_s, c_s) ∈ S do
13:            M ← find_mappings(G_ap^at, G_s, ∅)
14:            for all m ∈ M do
15:                PA ← PA ∪ {(s, m)}
16:            end for
17:        end for
18:    end for
19:    return PA
20: end function
```

any) was already matched (line 7). Hence the service tree is treated bottom-up. A list of candidate application components, C_{ap}^{m}, is calculated to which c_s could be mapped (line 8). The deployment structure of the two graphs restricts the set of candidates: If c_s is the root component of the service tree then the only possible match is the root of the application tree. Otherwise, the candidates are application components deployed on the match of the container component of c_s (see (1)). Additionally, the candidates must not be assigned to any other service component and they must be compatible with c_s. Each candidate is tried out: the pair consisting of c_s and the respective candidate c_{ap} is added to the partial mapping and the function calls itself recursively with the extended mapping as parameter. The result is the union between the return values of all recursive calls.

B. Generating the Set of Applicable Provisioning Actions

Alg. 2 generates the set of applicable provisioning actions using the service registry. Its input parameters are the graph of the application $G_{ap} = (C_{ap}, D_{ap}, T_{ap}, V_{ap})$ and the set of components $C_{ap}^{fix} \subseteq C_{ap}$ with a predetermined way of provisioning. Its output is the set of all applicable provisioning actions. The basic idea of the function is to divide the application graph into trees and request corresponding provisioning services for each of them. The for-loop from line 3 to 18 iterates over all application components. In each iteration a tree is created with the current application component as root. The nodes of this tree are determined by a flooding algorithm that starts from the root component and adds all deployed components for which no provisioning action has been fixed.

The function *find_services* represents the interface to the service registry. Its first parameter is a deployment tree, the second parameter the subset of components within this graph that the requested service has to provision. The registry returns all services whose deployment tree is a subtree of the requested one, meaning that they are able to provision at least a part of the required components. Only if the root of the currently examined deployment tree is also a root within the total application graph, services without prerequisite component are applicable (line 8). Services with prerequisite component are applicable in each case (line 11). Note that we can retrieve all relevant services with a linear number (in the number of application components) of requests to the registry.

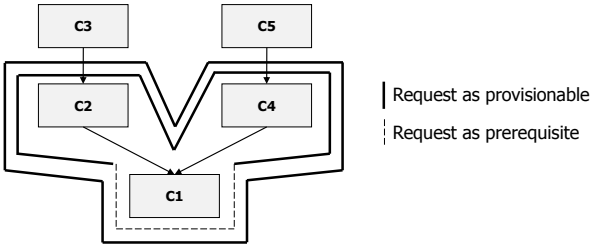

Fig. 4. Partitioning of example application

For each service in the total set of retrieved services—with and without prerequisite component—the function *find_mappings* is used in order to determine possible component mappings (line 13). For each service and each component mapping, a corresponding provisioning action is added to the result set of function *provisioning_actions*. After all application components have been tried as root for the provisioning action, the result set of the function (variable PA) contains all possible provisioning actions for the given application. Fig. 4 shows the two requests that would be sent for our example application (excluding requests for prerequisite components only). One request for services that are able to provision a subtree of tree C1-C2-C4. Another request for services that are able to deploy C2 and/or C4 on top of C1.

VI. Representation as Constraint Optimization Problem

Constraint satisfaction problems [4] consist of a set of variables with an associated value domain each, and of a set of constraints that link the variables. *Constraint optimization problems* (COP) additionally integrate an objective function that has to be maximized or minimized. The following subsections describe how our problem can be represented in this formalism.

A. Variables of the COP

The main idea is to introduce a variable for each application component that represents the provisioning action by which it should be provisioned. This variable is called $\mathbf{prov_c}$ (we refer to it as *provisioning variable*) where $c \in C_{ap}$ represents the component it refers to. Its value domain is the set of provisioning actions that provision this component: $PA_c \subseteq PA$. In a solution to the COP, each provisioning variable is assigned

to a provisioning action such that the global running costs are minimal. If a group of application components is provisioned by the same provisioning action, their provisioning variables are assigned to the same value. By adopting this COP structure, we already make sure that each solution to the COP corresponds to a provisioning plan in which each component is provisioned exactly once. The variable $\mathbf{cost_c}$ designates the costs for provisioning the component $c \in C_{ap}$, we refer to it as *costs variable*. Obviously, it is linked to the chosen way of provisioning for this component. $\mathbf{val_c^a}$ represents the value of attribute $a \in \mathcal{A}_t$ of component $c \in C_{ap}$. The initial value domain are all values that are allowed by the application template $V_{ap}(c, a) \subseteq dom(a)$.

B. Constraints and Goal Function of the COP

In the following we will represent groups of constraints as first-order equations. The COP variables will be marked within the equations in order to separate them visually from the constants. The following constraints need to be added for all application components ($\forall c \in C_{ap}$), all component attributes ($\forall a \in \mathcal{A}_t$), and for all possible provisioning actions ($\forall pa \in PA$ with $pa = (s, M)$ and $s = ((C_s, D_s, T_s, V_s), P_s, c_s)$):

$$\nexists c \in C_{ap} : \mathbf{prov_c} = pa \vee \forall c \in M[C_s] : \mathbf{prov_c} = pa \quad (6)$$

$$\mathbf{prov_c} = pa \Rightarrow \mathbf{cost_c} = \mathrm{distributeCosts}(pa, c) \quad (7)$$

$$\mathbf{prov_c} = pa \Rightarrow \forall c \in M[C_s] : \mathbf{val_c^a} \in V_s(M^{-1}(c), a) \quad (8)$$

$$\bigwedge_{r \in R_{ap}} r \quad (9)$$

$$\text{minimize} \sum_{c \in C_{ap}} \mathbf{cost_c} \quad (10)$$

Constraint (6) is due to the fact that provisioning actions may provision more than one component. So, either all of them are provisioned by this action or none. By constraint (7), we link specific ways of provisioning a component to the corresponding costs. If several components are provisioned by some provisioning action, the whole running costs are associated with the costs variable of one of them. The costs variables of the others are set to zero. We use the auxiliary function $\mathrm{distributeCosts(pa, c)}$ that assigns the whole costs of a provisioning action to one arbitrary component that is provisioned by it. Choosing one specific provisioning action for provisioning some component has an influence on the values of its attributes. This is expressed by constraint (8). Several global constraints may already be associated with the application description. Constraint (9) corresponds to the conjunction of all global constraints. Constraint (10) corresponds to the goal of our COP: we want to minimize the total running costs of the provisioning plan which is the sum of the costs variables over all components.

VII. Evaluation

We implemented a testbed in order to evaluate the algorithm experimentally. Our testbed generates application deployment graphs, global constraints and corresponding provisioning actions randomly. The test cases are then transformed into a COP as described in Sect. VI. We measure the time it takes to transform the test case into a COP and to solve it by a constraint solver. This is the critical part of the algorithm in terms of performance since solving the generated COP is NP-hard (we allow arbitrary application constraints). Our testbed can be configured by a set of parameters such as the number of application components. In the following we will analyze the influence of these parameters on the performance. We executed our tests on a server machine running Linux Debian version 2.6.26. As constraint solver we used the G12 solver [5]—an open-source, non-commercial product—in version 1.1.2. We restricted the virtual memory for the solver process to 2 GB in order to demonstrate that our approach does not result in high memory consumption. It used one AMD Opteron Quad-Core processor with 3 GHz.

TABLE I
Testbed parameters

Category	Parameter	Value range
General	No. component attributes	50
	No. attribute values	100
Application	Tree height	1 - 4
	No. trees	2
Service	No. provisionable components	1 - 5
	Prob. of service with prerequisite	25 %
	Prob. prerequisite matches	80 %
	Costs	1 - 50

In the following we will present several test series. For each test series we vary some of the configuration parameters of our testbed and analyze the impact on the performance. We generate 50 different test cases for each configuration and report the arithmetic mean of the computation times. We set the number of deployment trees that the application consists of to 2 and the maximal height of these trees to 4. We assume that each component is described by 50 attributes with value domains of cardinality 100. The probability to generate a provisioning action without prerequisite component is set to 75%, provisioning actions provision between 1 and 5 components. We assume a probability of 80% that services with prerequisite component can be applied (if an arbitrary provisioning service is chosen for the component they want to deploy on). The cost of a service is chosen arbitrarily between 1 and 50 units. Table I summarizes these parameters. We optimized the representation of the COP in comparison to the version presented in Sect. VI. First, we replaced the Boolean equations (7) and (8) by *table constraints* linking for each component the values of its provisioning variable with costs and attribute values. Table constraints express admissible value combinations for a subset of COP variables as tuple set and are handled more efficiently by the solver than Boolean equations. Second, we applied a pretreatment in order to integrate only those attribute variables into the COP that are addressed in global constraints.

A. Influence of the Number of Provisioning Actions

For the first test series (see Fig. 5) we generated applications without global constraints. We will justify later that having no global constraints at all corresponds to a more difficult

case. For the first test series, we successively increased the number of provisioning actions while keeping the number of application components at 40. Without global constraints, we are still able to calculate an optimal provisioning for applications with 40 components and 120 provisioning actions within less than 9 seconds in average. Note that the number of components corresponds to the number of components for which the provisioning is not fixed, yet. The application may have more components. Also the total number of provisioning services in the registry may be much higher than the number of provisioning actions.

B. Influence of the Number of Application Components

In series 2 (see Fig. 6), we set the number of provisioning actions to 105 and increase the number of application components. While the computation time grows exponentially with the number of provisioning actions, this is obviously not the case for the number of components. This is understandable since—from another point of view—provisioning planning corresponds to selecting an optimal subset out of a set of available provisioning actions. The complexity of this problem depends primarily on the number of provisioning actions.

C. Influence of the Number of Constraints

Adding global constraints to the application helps pruning the search space and accelerates the search. For the last test series (see Fig. 7), we assumed a fixed ratio of 1 to 3 between application components and applicable provisioning actions. We compared two versions of the COP: one with global constraints and one without. We added constraints of the form $a_1 + \ldots + a_n \leq 80 * n$ where the a_i represent attributes of different components and $n \leq 5$. Since attribute values of provisioned components are chosen arbitrarily between 1 and 100, these constraints are satisfied with a probability of 80 %. For the version with global constraints, we assumed that each component is involved in 2 constraints in average (this seems a moderate assumption since components are described by 50 properties). For 50 application components and 150 provisioning services, the version of the problem that integrates constraints can be solved in average about 10 times faster than the other. This justifies our claim from before that having no constraints at all corresponds to a more difficult case.

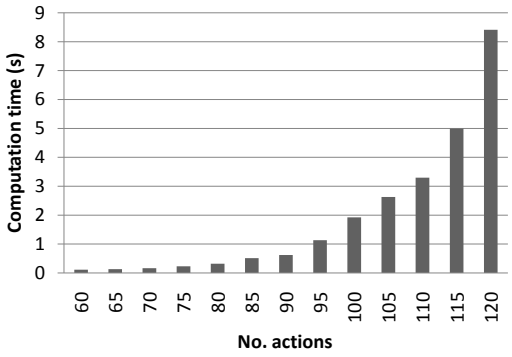

Fig. 5. Benchmark results: test series A

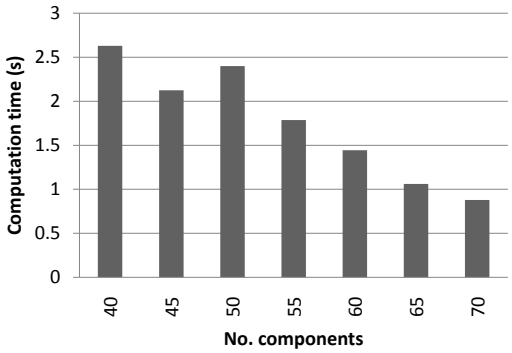

Fig. 6. Benchmark results: test series B

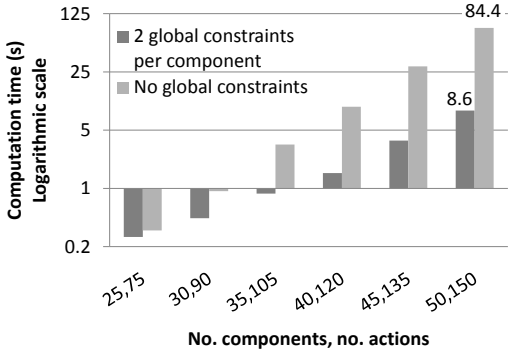

Fig. 7. Benchmark results: test series C

VIII. Comparison With Related Work

A general analysis about the risks and benefits of using public cloud computing offers can be found in [6]. In our case we assume that the typical risks associated with cloud computing—e.g. concerning privacy and service availability—have been found acceptable. The financial benefits of using cloud offers have been studied in several publications. Some general formulas are presented in [6], in [7] the benefits of purchasing versus leasing storage from cloud services are compared. Different execution modes of a scientific workflow are studied in [8], finding that cloud computing can be a cost-effective solution especially for data-intensive applications.

Our scenario and our data model for describing applications and provisioning services is most similar to the Cafe model [9]–[11]. However, Cafe uses an external file, the *variability descriptor*, for representing variability within the application. We use a template based model instead (similar to the one presented in [12]) which facilitates discovery and transformation into a COP. In [11], a simple algorithm for choosing the best cloud providers is presented. However, it performs only local optimization and no experimental evaluation has been published. A more sophisticated algorithm for optimizing the usage of Storage as a Service offers is presented and evaluated in [13]. Similar to our approach, a COP solver is used in

order to determine the optimal storage location for specific data items and the optimal placement for administrators. The optimal outsourcing is calculated with regards to a specific time window. Our work could be extended as well in order to take into account that services may be offered in different temporal granularities. However, the approach is specific to storage outsourcing and does not deal with deployment graphs. Cost-optimal provisioning of Hadoop applications in the cloud was examined in [14]. In [15], cost-optimal scheduling for applications in cloud environments is performed using heuristic algorithms. However, the applications are described as workflows and not as deployment graphs.

The problem of finding the best combination of provisioning services for a given application can be compared to the problem of optimal component placement studied in Grid computing (e.g. [16]). However, applications are usually described as sets of components with associated requirements on the infrastructure in this context. We generalize this model by describing applications as deployment graphs and by considering different levels of outsourcing. If provisioning services are realized as Web services (see [10]) our optimization problem could be seen as a form of quality-aware service composition. In [17] linear integer programming is used for solving this type of problems. While we study cost optimization out of the perspective of the cloud customer, other publications (e.g. [18]–[20]) treat cost-optimization problems for the cloud provider and are hence complementary to our approach.

IX. Conclusion

In this paper we addressed the challenge of minimizing application running costs by outsourcing carefully selected parts into the cloud. This problem is of growing practical importance since the number of IaaS, PaaS, and SaaS offers is increasing. We presented an algorithm that works on generic representations of applications and provisioning services as graphs. In a first step, it decomposes applications according to all possible levels of outsourcing and retrieves corresponding services from a registry. In the second step, it transforms the problem of cost-optimal outsourcing into a constraint optimization problem which is solved by a standard solver implementation. We presented benchmarks derived from solving randomly generated COPs. We calculated a cost-optimal provisioning for applications consisting of up to 50 components, given 150 provisioning actions, within less than 85 seconds in average (even less than 9 seconds for a moderate number of global constraints). Investing significantly more time in order to find the optimal solution to much more complicated problems may still be a good investment - even slightly suboptimal solutions for the running costs may add up to big additional sums over the time, especially if used for running several application instances over a long time. Including dynamic aspects directly into our model would be an interesting point of future research for cases where scaling in the granularity of application instances is not possible or desirable. Extending the model to different billing strategies would be a challenging research topic as well.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "SOSOA: Self-Organizing Service-Oriented Architectures" (SNF Sinergia Project No. CRSI22_127386/1). We also thank Radoslaw Szymanek for his support in optimizing the COP formulation.

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[2] "Amazon." [Online]. Available: http://www.amazon.com/

[3] "Salesforce." [Online]. Available: http://www.salesforce.com/de/

[4] F. Rossi, P. V. Beek, and T. Walsh, *Handbook of constraint programming*, 1st ed. Amsterdam: Elsevier, 2006.

[5] "NICTA - cp platform." [Online]. Available: http://www.nicta.com.au/

[6] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[7] E. Walker, W. Brisken, and J. Romney, "To lease or not to lease from storage clouds," *Computer*, vol. 43, no. 4, pp. 44–50, 2010.

[8] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, pp. 1–12.

[9] T. Unger, R. Mietzner, and F. Leymann, "Customer-defined service level agreements for composite applications," *Enterprise Information Systems*, vol. 3, no. 3, pp. 369–391, 2009.

[10] R. Mietzner and F. Leymann, "Towards provisioning the cloud: On the usage of Multi-Granularity flows and services to realize a unified provisioning infrastructure for SaaS applications," in *2008 IEEE Congress on Services - Part I*, Honolulu, HI, USA, 2008, pp. 3–10.

[11] R. Mietzner, T. Unger, and F. Leymann, "Cafe: A generic configurable customizable composite cloud application framework," in *CoopIS 2009 (OTM 2009)*, vol. 5870, 2009, pp. 357–364.

[12] W. Arnold, T. Eilam, M. Kalantar, A. Konstantinou, and A. Totok, "Pattern based SOA deployment," *Service-Oriented Computing–ICSOC 2007*, pp. 1–12.

[13] S. Uttamchandani, K. Voruganti, R. Routray, L. Yin, A. Singh, and B. Yolken, "BRAHMA: Planning Tool for Providing Storage Management as a Service," in *Proceedings of the IEEE International Conference on Services Computing*, 2007.

[14] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *1st Workshop on Hot Topics in Cloud Computing, HotCloud*, 2009.

[15] S. Pandey, L. Wu, S. Guru, and R. Buyya, "A Particle Swarm Optimization-based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 2010, pp. 400–407.

[16] T. Kichkaylo and V. Karamcheti, "Optimal resource-aware deployment planning for component-based distributed applications," in *13th IEEE International Symposium on High performance Distributed Computing, 2004. Proceedings*, 2004, pp. 150–159.

[17] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-aware middleware for web services composition," *IEEE Transactions on software engineering*, vol. 30, no. 5, pp. 311–327, 2004.

[18] C. Fehling, F. Leymann, and R. Mietzner, "A Framework for Optimized Distribution of Tenants in Cloud Applications," in *Proceedings of the 2010 IEEE International Conference on Cloud Computing (CLOUD 2010)*. IEEE, Juli 2010, pp. 1–8.

[19] J. Li, J. Chinneck, M. Woodside, and M. Litoiu, "Deployment of Services in a Cloud Subject to Memory and License Constraints," in *IEEE International Conference on Cloud Computing, 2009. CLOUD'09*, 2009, pp. 33–40.

[20] X. Li, Y. Li, T. Liu, J. Qiu, and F. Wang, "The Method and Tool of Cost Analysis for Cloud Computing," in *Proceedings of the 2009 IEEE International Conference on Cloud Computing*. IEEE Computer Society, 2009, pp. 93–100.