View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Ghent University Academic Bibliography

# Allocating Resources for Customizable Multi-Tenant Applications in Clouds Using Dynamic Feature Placement

Hendrik Moens, Bart Dhoedt and Filip De Turck

*Ghent University – iMinds, Department of Information Technology*
*Gaston Crommenlaan 8/201, B-9050 Gent, Belgium*

**Abstract**

Multi-tenancy, where multiple end users make use of the same application instance, is often used in clouds to reduce hosting costs. A disadvantage of multi-tenancy is however that it makes it difficult to create customizable applications, as all end users use the same application instance. In this article, we describe an approach for the development and management of highly customizable multi-tenant cloud applications. We apply software product line engineering techniques to cloud applications, and use an approach where applications are composed of multiple interacting components, referred to as application features. Using this approach, multiple features can be shared between different applications. Allocating resources for these feature-based applications is complex, as relations between components must be taken into account, and is referred to as the feature placement problem.

In this article, we describe dynamic feature placement algorithms that minimize migrations between subsequent invocations, and evaluate them in dynamic scenarios where applications are added and removed throughout the evaluation scenario. We find that the developed algorithm achieves a low cost, while resulting in few resource migrations. In our evaluations, we observe that adding migration-awareness to the management algorithms reduces the number of instance migrations by more than 77% and reduces the load moved between instances by more than 96% when compared to a static management approach. Despite this reduction in number of migrations, a cost that is on average less than 3% more than the optimal cost is achieved.

*Keywords:* Cloud Resource Management, Software Product Line Engineering, Dynamic Application Placement, Feature Placement

## 1. Introduction

In recent years there has been a growing interest in using cloud computing as a means of offloading applications and reducing costs. An efficient way in which costs of cloud deployments may be reduced is through multi-tenancy. In a

traditional model, every client is provided with a separate application instance. In multi-tenant environments, however, a single instance can be used by multiple clients. Every client of the application is referred to as a *tenant* and is considered to be an organization with its own end users. The major advantage of this approach is that it makes it possible to use less application instances to provision the service to each of these tenants, reducing the cost of offering the service. Additionally, this approach makes it easier to scale applications, as sudden increases in numbers of users result in smaller increases of the number of required instances. Spikes in the numbers of end users of one tenant can also be compensated by decreasing numbers of end users of other tenants.

Building customizable multi-tenant applications is however difficult, and it is often hard to make changes that are not just cosmetic configuration changes. Therefore, multi-tenant applications are often offered as a take it or leave it package, with only limited customizability. This approach works well for many application types, especially when tenant needs are very similar, but there are use cases where a very high degree of customizability is required. This is the case in various domains, such as for example document processing, medical communications and medical information management. These application cases are all characterized by the fact that the offered platform is used by a relatively small number of large tenants that each have a large number of end users. Each of these tenants may request its own customizations to the application platform, and as many tenants are large, it is difficult to deny these requests. Currently such customizations are often developed on an ad-hoc basis. This however poses difficulties concerning the management of these customizations and as separate tenants have custom tailored codebases, it becomes impossible to share resources between end users. This problem becomes even more complex when clients are also split up into multiple departments that each require specific customizations and when the application platform is offered to other clients using resellers. An illustration of the various tenant types is shown in Figure 1.

Using feature modeling [1], this issue can be addressed. Feature modeling is an approach where the variability of an application is modeled using a *feature model*. The customizability of the application is represented by a collection of features, a representation of specific functionality that may or may not be added to the application, and their relations. Features can be implemented using aspect oriented programming [2], as configuration changes, or as custom code modules. While feature modeling is an interesting approach for managing the codebase of customizable applications, this still results in customized application binaries, making it impossible to use multi-tenancy in the resulting applications. We previously proposed an approach where applications are separated into multiple interacting components, effectively making sure every feature is implemented in its own service [3]. The entire application is then composed from the various components, thus forming a service oriented architecture. As every code module is itself multi-tenant, the advantages of multi-tenancy can be attained.

Splitting applications into multiple components however impacts the performance of the applications, complicating cloud management. Additionally, the

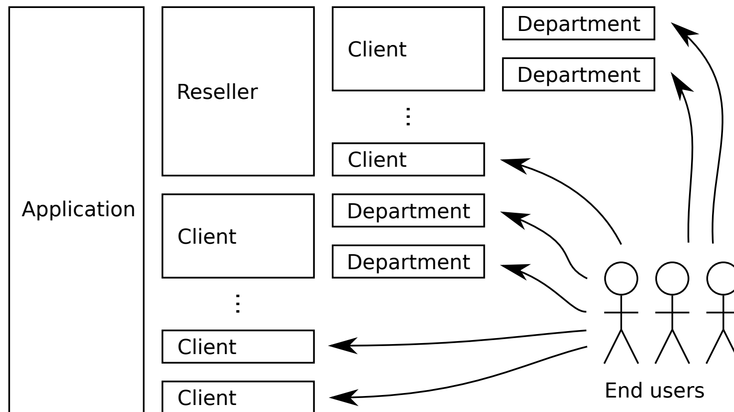



Figure 1: An illustration of a scenario where the application is offered to end users by a hierarchy of the three types of tenants: resellers, clients, and client departments. Resellers can also sell the application to other resellers, and departments may also be further divided into smaller departments. At every level, different application customizations may be required.

chosen features should be taken into account by the management system. It may e.g. be cheaper to use an existing high-performance instance for a tenant that does not pay for such an instance rather than to allocate a low-performance instance specifically for this tenant. We previously addressed resource allocation taking this information into account, referred to as feature placement, in [4] and [5], but the approach however resulted in a static resource allocation, that has to be recomputed periodically. In doing so, the number of migrations is not taken into account, which adversely impacts the performance of the system when services are migrated. Additionally, adding applications is relatively expensive and slow as they can only be added whenever the algorithm is invoked rather than immediately when they are added.

In this article, we focus on dynamic feature placement algorithms that relocate and reconfigure features when changes occur. In computing these changes, the previous state of the system is taken into account, minimizing the number of application changes and instance migrations. We present both ILP-based algorithms and a heuristic algorithm, the Dynamic Feature Placement Algorithm (DFPA). Figure 2 shows the algorithm inputs and how it functions within a cloud management system.

The remainder of this article is structured as follows. In the next section we discuss related work. Afterwards, in Section 3 we describe how the system in which the feature placement algorithm is executed is structured, and how feature modeling is used within the approach. A formal problem representation is presented in Section 4. In Section 5, we present the DFPA. The evaluation setup is presented in Section 6, and the algorithms are then evaluated in Section 7. Finally, we state our conclusions in Section 8.

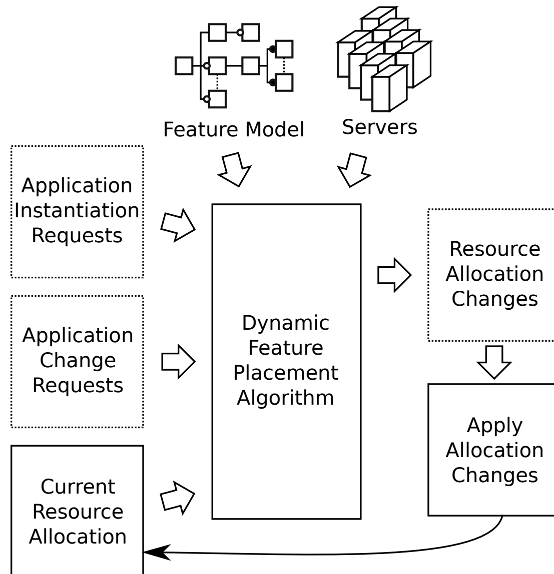



Figure 2: The dynamic feature placement, its inputs and its function within a management system.

## 2. Related Work

To manage variability when building applications, Software Product Line Engineering (SPLE) [6] techniques are used. Instead of managing multiple codebases for different application variants, a single codebase is used, and different variants are generated using SPLE tools. In traditional SPLE applications, the application configuration is however generally decided at compile-time, making it ill-suited for cloud environments. Dynamic SPLE [7] can be used to configure and reconfigure software variants at runtime, making it more suited for cloud environments. This makes it possible to characterize runtime variability and reconfigure applications at runtime. SPLE has been used in cloud environments [8, 9, 10], but the approaches tend to focus mostly on development, deployment and configuration. We however focus specifically on runtime resource allocation for customizable SPLE applications by adding awareness of application variability to the cloud management algorithms. Similarly, other work [11, 12, 13, 14] focuses on the design-time variability of the applications rather than on their runtime management, the latter being the focus of this article.

In this article, we focus on cloud resource allocation [15] and design dynamic management algorithms that are aware of application customizability. In particular, we focus on extending the generic application placement problem [16] and cloud application placement problem [17, 18, 19] to incorporate both multi-tenancy and software variability. The approach we use for multi-tenancy uses component-based applications composed using a Service-Oriented

Architecture (SOA), making the relations between components another important consideration.

Our approach is similar to application component placement approaches [20, 21, 22, 23], where applications consisting of multiple components, represented as a set of Virtual Machines (VMs), are placed within a datacenter taking the relation between components into account. These approaches typically focus on colocation, anti-colocation and other placement constraints used to impact application security, performance, and reliability. These approaches however do not take multi-tenancy on a VM-level into account, meaning the approaches do not support sharing components between different multi-component applications. Our approach further differs by the inclusion of SPLE principles within the management system, making it possible to take application variability into account during resource allocation.

[24] both focuses on VM placement taking energy efficiency into account. Our approach also incorporates server use costs, but differs in that we focus on managing multi-tenant applications where multiple applications can make use of a single instance. Additionally, our algorithm also adds support for software variability within the management algorithm itself. Energy efficiency and server usage costs are incorporated in an application placement system in [25]. The authors however focus on the placement at a VM level, while our approach focuses on managing multi-tenant applications where multiple applications can make use of a single instance, meaning more fine-grained control is needed. Furthermore, our algorithm also adds explicit support for software variability. This enables the management system to dynamically fill in undecided variability, referred to as open variation points [10], at runtime.

Application placement algorithms typically focus on server CPU and memory resources [19, 26, 27, 28] or bandwidth limitations [29]. In this article we make use of a generalized approach where arbitrary resources of different types can be used. Such an approach was previously used in [30], but our approach goes further as it allows the definition of multiple resources rather than just supporting two arbitrary resource types, enabling the management of high-variability applications with heterogeneous resource demands.

This article extends our previous work related to feature placement [4, 5], which focused on the static feature placement problem, and describes and evaluations new dynamic feature placement algorithms that can be used in a context where applications are added and removed through time. The modeling approach we use is further based on our work on feature model conversion [3], which focuses more on how the code modules themselves are defined and how customizable applications within our approach can be designed rather than on how these modules are managed at runtime, which is the focus of this article.

An overview of how the DFPA, which is introduced in this article, compares to the most relevant previous work is shown in Table 1. In the table, we focus specifically on approaches using and supporting multi-component cloud applications. We compare multiple properties for the various publications. These properties are the following:

Table 1: An overview of the relation between the DFPA introduced in this article and previous work.

| | [10], [14] | [17] | [20] | [21] | [22] | [23] | [4], [5] | DFPA |
|---|---|---|---|---|---|---|---|---|
| Multi-component applications | + | + | + | + | + | + | + | + |
| Application variability | + | - | - | - | - | - | + | + |
| Resource management | -[1] | + | + | + | + | + | + | + |
| Dynamic cloud management | N/A | -[2] | + | + | - | - | - | + |
| Service Management | N/A | - | - | - | - | - | + | + |
| Service and VM migrations | N/A | - | - | - | - | - | - | + |
| Generalized resources | N/A | - | + | + | + | + | + | + |
| Server use minimization | N/A | - | - | - | - | - | + | + |

1. Multi-component applications: Whether the approaches support the management of applications consisting out of multiple components.
2. Application variability: Whether application variability and customizability is considered in the work.
3. Resource management: Whether the work addresses the runtime management and resource allocation of these applications.
4. Dynamic cloud management: Whether the management of applications is dynamic (i.e. resource demand can vary over time and application components can be migrated).
5. Service management: Whether the management focuses on services instead of VMs. This makes it possible to consider how application resources are allocated within VMs in addition to how the VMs themselves are allocated.
6. Service and VM migrations: Whether both service and VM migration is supported (i.e. VMs can be moved between nodes and application load can be shifted between VMs without moving the VMs themselves).
7. Generalized resources: Whether the approach supports generalized arbitrary resources, and not just CPU and memory capacity.
8. Server use minimization: Whether the approach takes server utilization into account, enabling energy-efficient resource management.

## 3. Feature Placement

SPLE [6] techniques can be used to model an application as a collection of features and relations between features. Both the features themselves, which encapsulate specific functionality that may or may not be included in an application, and their relations are important. It may for example be the case that the inclusion of a feature implies that other features must be included or conversely that the inclusion of a feature prevents another feature from being

[1]Focuses on modeling of applications, not on managing them at runtime.
[2]On-line algorithm, but does not migrate instances over time.

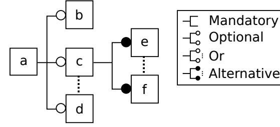


Figure 3: An illustrative example of a small feature model using the Pure::Variants notation.

selected. To make it easier to reason using these relations, feature models are often defined hierarchically. The relation between child nodes can be chosen as one out of four types:

- **Mandatory(a, b)**: If a feature **a** is included, the feature **b** must be included as well.
- **Optional(a, b)**: If a feature **a** is included, the feature **b** may be included. Conversely, the feature **b** must not be included if **a** is not included.
- **Alternative(a, S)**: If a feature **a** is included exactly one of the features contained in the set **S** must be included. If **a** is not included, none of the features in **S** may be included.
- **Or(a, S)**: If a feature **a** is included, at least one of the features contained in the set **S** must be included. If **a** is not included, none of the features in **S** may be included.

In our feature placement approach, applications are constructed using a service oriented architecture and are composed out of various feature instances. Only features that refer to actual code modules are used, while other features such as smaller configuration changes are handled at runtime. The code module of a feature can then be instantiated as a VM in a cloud. We assume that feature instances are multi-tenant, meaning they can be used to serve multiple applications for different clients. For more information as to how features are represented and how feature models containing non-code changes can be mapped to feature models containing feature models we refer to [3]. To support this transformation, and add versatility to the feature model representation, two additional non-hierarchical relation types are used:

- **Excludes(a, b)**: If a feature **a** is included, the feature **b** must not be included and vice versa.
- **Requires(a, b)**: A feature **a** may only be included if the feature **b** is included as well.

Figure 3 shows an illustrative example a simple feature model consisting of six features. The relations between the various features are expressed using the Pure::Variants notation [31] which we also used in our previous work [3, 5]. This model corresponds to three relations: **Optional(a, b)**, **Or(a, {c, d})**, and **Alternative(c, {e, f})**. Based on this model, a specific configuration of features can be selected to be used in an application. A feature can either be

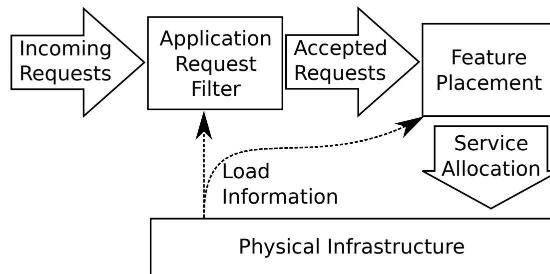



Figure 4: An overview of the relevant components within the cloud management system and the communication flow between the components.

selected, excluded or undecided. A selected feature must always be included for the feature placement of the application to be successful. An excluded feature may not be included in the application under any circumstance. Finally, undecided features may either be included or excluded at runtime, based on what results in the lowest placement cost. It may, e.g. be cheaper to add encryption for a client, even if he has not selected the feature, if only instances with encryption exist than to create a new instance specifically for this client. These undecided features are referred to as *open variation points* [10]. If, for the model in Figure 3, $\{a, c, d\}$ are selected and $\{b\}$ is excluded, $e$ and $f$ remain as open variation points. Only when the application is deployed will it be decided whether $e$ or $f$ are included.

An overview of the workflow when new requests are added to the system is shown in Figure 4. As application placement requests enter the system, the viability of adding them is first evaluated by an application request filter. This filter can make decisions based on multiple factors:

- The amount of system resources required for the request can be determined and compared to the remaining resources available within the datacenter. If there are not sufficient remaining resources, the application request should be rejected.
- Another factor that is important when application allocations are initiated is the consideration of network capacity. This is especially important if some features are allocated in different networks and part of the communication must pass over the Internet. Filters such as this can be created by determining the impact on the underlying network of instantiating these services, which we previously discussed in [32].

This filter is invoked whenever a new application configuration is instantiated, which is when a seller enters the feature configuration for a client within the system. Accepted requests are sent to the feature placement system, which runs algorithms responsible for determining where the services out of which the application consists should be allocated on the physical infrastructure. These allocation changes are then enforced by the feature placement system resulting in a change of the physical allocation. System load information is then

sent back to the application request filter and feature placement components, which can be used to update allocations and to allow or reject future application instantiation requests.

## 4. Feature Placement Model

We previously presented a formal problem formulation for the static feature placement problem in [5]. In this section we briefly describe the problem model, focusing on the variables that are needed to add migration-awareness the model. For a more detailed discussion of the model parameters and formulation we refer to [5]. We will end this section by discussing how migration-awareness can be added to the model to make it useful for dynamic application placement scenarios.

Figure 5 shows the inputs and outputs of feature placement. As an input, the model defines servers, with associated CPU, memory and usage cost information. Additionally, a feature model is defined containing all of the application features and their relations. Individual features, comparable to VMs in traditional application placement scenarios, can be instantiated on a server. Thus, every instance has specific memory requirements. Additionally, we limit the amount of processing that a single instance can do by limiting the amount of CPU resources that can be processed by a single instance. These instance limits are used as it is unrealistic for a VM with limited memory to be able to be able to process an infinite amount of requests (represented by their CPU use). Finally, applications are composed out of multiple features. They are represented by a set of selected features, and a set of excluded features. Some features may be

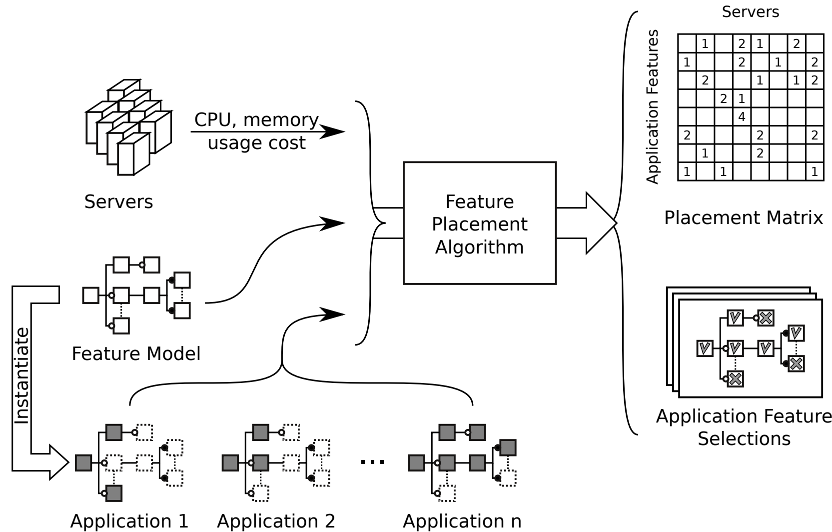



Figure 5: The input and output of the feature placement.

neither selected nor excluded for an application, leaving these features as open variation points. During the execution of the feature placement algorithm, these open variation points are filled in by either selecting or excluding them based on the feature model. The above results in three sets of input variables: servers, features and applications. These are respectively contained in the sets $S$, $F$ and $A$.

The model has two outputs: a placement matrix that defines where features are instantiated and for which applications they are used, and the application feature selection which indicates which features are used for which applications, filling in open variation points. The placement matrix itself consists of two separate parts. First, it describes on which servers features are instantiated. As the amount of CPU resources that can be used by a single feature instance is limited, it is possible for there to be multiple instances of a feature on a server. Secondly, the placement matrix also describes which applications make use of which feature instances and how much CPU capacity of a feature instance is used for every application. The above results in three output variables:

- The instance count, $IC_{s,f}$, represents the number of instances of a feature $f \in F$ that are instantiated on a server $s \in S$ in the solution.
- The placement matrix, $M^{CPU}_{s,f,a}$, shows how much CPU resources are allocated on a given server $s \in S$ for a feature $f \in F$ of an application $a \in A$.
- The binary variables $\Phi_{f,a}$ are used to represent the feature selections: $\Phi_{f,a} = 1$ if the feature $f \in F$ is included for application $a \in A$. Conversely, feature is excluded if $\Phi_{f,a} = 0$.

The objective of the feature placement optimization is to minimize the cost of the placement. This cost consists of two separate components: the cost of using servers and the cost of failing to place applications or their components. The cost of using a server represents an operational cost that can either be an energy cost or the cost of renting a server. The failure cost represents a Service Level Agreement (SLA) cost of failing to place an application correctly. Two separate failure costs are taken into account: an application failure cost is incurred whenever any feature of an application fails to be placed correctly, while the feature failure cost is incurred when a single feature of an application is not placed correctly. These separate costs make it possible to define an additional cost for failing to provide specific essential features of the applications. These costs may be represented as a monetary cost, but also as a virtual cost determined by the management system that varies during the execution of the management system. E.g. failing to place an application may in reality result in a cost of 0 if the service interruption is short but should still have an assigned cost within the management system to prevent it from happening at all.

In the previous discussion, we focused specifically on memory and CPU resources for simplicity. In practice, it is however possible for there to be additional relevant resource types, such as disk space and bandwidth. The model supports this by making an abstraction of the concept of resource, and making a distinction between two resource types:

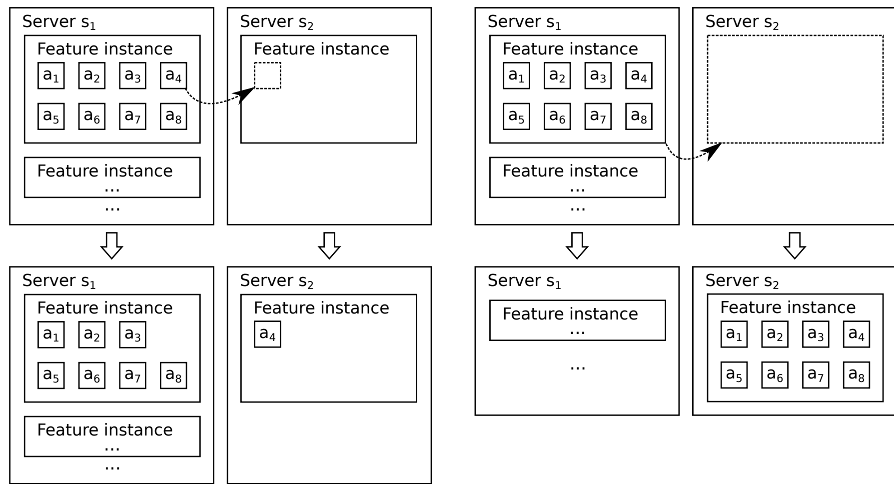
- *Strict resource* types are resources that are required to create an instance of the feature. The typical example of this resource type when studying resource allocation problems is application memory. Another possible example is disk space. To create a VM, this memory and disk demand must be satisfied; otherwise it is impossible to create an instance of the feature. Every server has a specified amount of available strict resources, and every feature instance has a specific strict resource demand.

- *Non-strict resource* types behave differently. The goal of the optimization is to ensure that all of the requested resources of this type are allocated for applications. To achieve this, instances are created and that are responsible for providing (part of) the resource demand of an application. The common example here is CPU capacity: a certain amount of CPU capacity must be allocated to an application to ensure all requests for the application can be handled, but not all of the calls must always be handled on the same instance, especially for larger tenants for whom the application must be distributed over multiple nodes anyway because of their scale. While every server has a specified available non-strict resource demand, feature instances do not have a non-strict resource demand. Instead, applications have a specific non-strict resource demand that must be handled by one or more feature instances. As stated previously, it is possible for feature instances to be limited in the amount of non-strict resources that they can process, which is represented using instance non-strict resource limits.

All of the resources are contained within the set $\Gamma$, strict resources are contained in the set $\Gamma_s$, while non-strict resources are contained in the set $\Gamma_{\overline{s}}$. Generalizing the output variables, we obtain a placement matrix $M^{\gamma}_{s,f,a}$, where $\gamma$ is any of the non-strict variables in $\Gamma_{\overline{s}}$.

### *4.1. Dynamic feature placement: resource migrations*

The static problem model as summarized above can be used to minimize the cost of a placement, in terms of both failed placements and server use, but it does not take the current application allocation into account. By adding the current resource allocation as an input, the number of resource migrations that are required to apply the placement can be determined. Based on how the model is defined, there are two types of resource migrations that should be considered. On one hand, the number of instance count changes can be determined. This represents the number of virtual machines that must be started and stopped. We refer to this migration type as an *instance count change*. Due to the multi-tenant nature of the instances, a second measure for migrations can be considered as well: the amount of resources used for applications that are allocated using other instances. This type of migration, which we refer to as *resource shift*, can also cause network traffic as application data present in one instance may have to be transferred to another instance. An illustration of instance count change and resource shift is shown in Figure 6.




(a) An example of a resource shift migration. No new instances are created, but resources that were allocated for an application $a_4$ within a feature instance on server $s_1$ are moved to another feature instance on server $s_2$.

(b) An example of an instance count change migration. An instance is removed on server $s_1$ and a new instance is created on server $s_2$. (Note that in this example the resources allocated for the applications $a_1$ to $a_8$ are shifted as well.)

Figure 6: An illustration of the difference between resource shift migrations and instance count change migrations.

To model the migrations caused by a placement, the current allocation must be added as an input to the model. This current allocation can be represented using two additional inputs:

1. The previous instance count $IC'_{s,f}$, which indicates the number of instances of a feature $f$ that are currently allocated on a server $s$. This input is required to determine the change in instance count on every instance, which can in turn be used to measure the number of instance count change migrations.
2. The previous resource allocation, represented as $M'^{\gamma}_{s,f,a}$ must be included as an input variable as well. By determining changes between $M'$ and the currently computing allocation $M$, the number of resource shift migrations can be modeled.

The number of instance count increases $IC_+$ can be determined as shown in Equation (1). The increase can be determined by, for every server and feature type, calculating the difference in the number of feature instances between both allocations. We are only interested in feature instance count increases, as only creating a new feature instance incurs a cost, in the form of network load and delays. Therefore decreasing values (when $IC_{s,f} - IC'_{s,f} < 0$) are ignored and replaced by 0 within the sum.

$$IC_+ = \sum_{s \in S} \sum_{f \in F} \max(IC_{s,f} - IC'_{s,f}, 0) \tag{1}$$

To characterize resource shift migrations, we make use of a similar formulation. The difference in resource use between the current placement matrix $M$ and the previous placement matrix $M'$ are determined. Like in the definition of $IC_+$ negative values are removed, and only positive changes are counted. Equation (2) shows how this resource shift can be computed for non-strict resource types. As there may be migrations of resources of different types, this results in a value for every resource type. By normalizing the $M^{\gamma}_+$ values based on the total resource load these different resource types can be combined into a single measure $M_+$. This is shown in Equation (3). In this equation, $T^{\gamma}$ is the total load for non strict resource $\gamma$ in the previous iteration, which is used for the normalization.

$$M^{\gamma}_+ = \sum_{s \in S} \sum_{f \in F} \sum_{a \in A} \max(M^{\gamma}_{s,f,a} - M'^{\gamma}_{s,f,a}, 0) \tag{2}$$

$$M_+ = \frac{1}{|\Gamma_{\overline{s}}|} \times \sum_{\gamma \in \Gamma_{\overline{s}}} \left( \frac{1}{1 + T^{\gamma}} \times M^{\gamma}_+ \right) \tag{3}$$

These model extensions add $IC_+$, which is a measure of instance count change migrations, and $M_+$, which is a measure of resource shift migrations.

*4.2. Iterative migration minimizing model*

The migration minimizing model makes use of an iterative approach for minimizing the cost, instance count increase $IC_+$ and resource shift $M_+$. Within

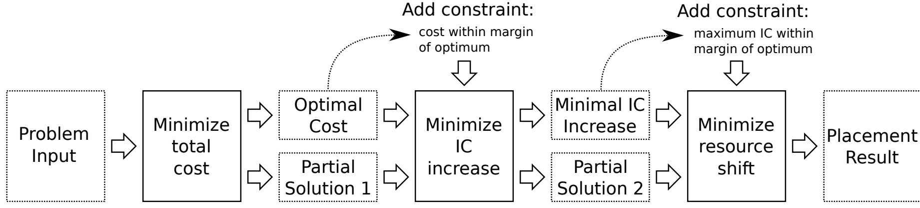



Figure 7: The different optimization steps of the migration minimizing model.

the optimization, we define use of nearness parameter $\alpha \geq 1$, and ensure the cost of the solution will always be within a factor $\alpha$ of its optimal value. The migration minimizing model is illustrated in Figure 7 and consists of three steps:

1. First, the base model is optimized, minimizing the total cost. This results in an optimal cost $C^*$.
2. Subsequently, the instance count increase $IC_+$ is minimized. During this minimization, an additional constraint on the cost is added: $C \leq \alpha \times C^*$. This results in an optimal instance count increase value $IC_+^*$.
3. Finally, the resource shift migrations $M_+$ are minimized. During this optimization, two constraints are added, limiting both the cost and instance count increase migrations: $C \leq \alpha \times C^*$ and $IC_+ \leq \alpha \times IC_+^*$. This last optimization returns a placement result where the total cost $C$ the number of instance count migrations $IC_+$ and the amount of resource shift migrations $M_+$ are all taken into account.

## 5. The Dynamic Feature Placement Algorithm (DFPA)

We designed the Dynamic Feature Placement Algorithm (DFPA) heuristic to solve the problem discussed in the previous section. To create a dynamic algorithm that minimizes the number of migrations between iterations, we design an algorithm that, using the current placement state and a specific change, results in a new application placement. These changes can either be the addition of an application, the removal of an application, or a change to the resource requirement of an application. The latter can be achieved by sequentially removing and re-adding the application with different resource requirements, which is why we focus specifically on application start and stop events. The management algorithm dynamically generates a new solution whenever applications are added or removed, and bases itself on the previous allocation to achieve good placement results with limited resource migrations.

The algorithm is invoked whenever an application is instantiated or halted. A high level overview of the algorithm steps is shown in Figure 8. The algorithm maintains a solution to the placement problem in-memory, which it updates during each of the steps. The initial state of the algorithm is the current placement, before the changes are taken into account. The main steps of the algorithm are as follows:

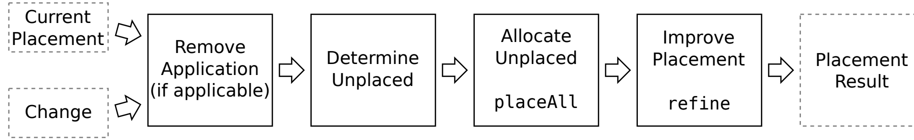



Figure 8: A high level overview of the Dynamic Feature Placement Algorithm (DFPA) steps. The functions `placeAll` and `refine` are discussed in Sections 5.1 and 5.2 respectively.

1. Two inputs are used by the DFPA: the current placement and a change to the current state. The current placement is represented using a placement matrix $M$ and is used as the initial algorithm state. The provided change is either a start or a stop event for an application.
2. If the change is the removal of an application, the application is removed from the current solution, possibly reducing the number of feature instances for some of the application features if this causes feature instances to become unused.
3. Subsequently, the applications that must be placed are determined. This may be an application that must be instantiated now if the change is the addition of an application, but this list may also be larger if an application was not successfully placed in a previous algorithm iteration.
4. In a next step, the applications are placed iteratively. This is done using a `placeAll` function that is described in detail in Section 5.1.
5. Then, an improve operation is used to refine the placement and improve its quality. This is done by selectively removing and re-adding applications, reducing the number of servers used and making it possible to place some applications that would otherwise have failed. This solution refinement process, executed using a `refine` function, is discussed in Section 5.2.
6. Finally, the placement result that has been determined during the previous steps is returned as the placement result.

### *5.1. Placing applications*

The `placeAll` function, illustrated in Figure 9 and shown in Algorithm 1 is responsible for allocating resources for a collection of applications. This function iterates over all of the applications, and allocates resources one by one. The order by which applications are selected is defined by the total cost of failing to place the application, and can be computed by adding the cost of failing the application to the cost of failing of its individual features. Applications with a higher cost are placed first, which is done using a `place` function which will be explained later on. If it is impossible to place the application due to insufficient resources, a reduced version of the application is created: for this version of the application the cost of failing the application itself is 0 (as at this point it has already failed) and only the features that incur a separate failure cost are included. Alternatively, if the application failure cost is already 0, the reduced application is created by removing the feature with the lowest cost of failure. The reduced application is then re-added to the collection of applications that

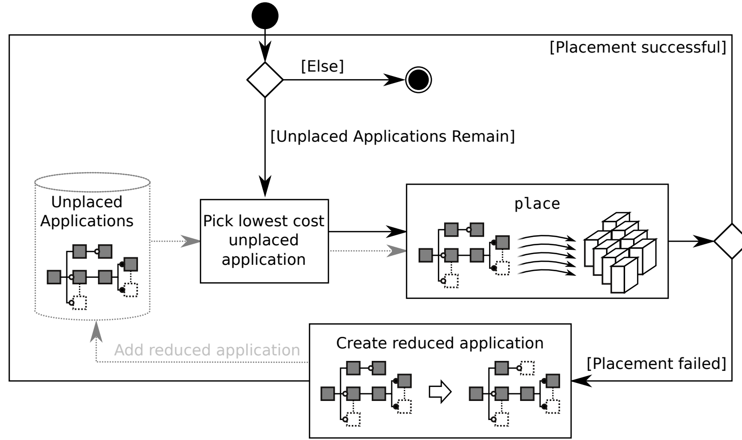



Figure 9: A high level overview of the `placeAll` function which is responsible for allocating all unplaced applications. To achieve this, the function iterates over all unplaced applications and allocates resources for them using the `place` function. If this succeeds, the next unplaced application is allocated. Otherwise, a reduced version of the application is created and added to the collection of unplaced applications. This application will then be placed in a later iteration of the algorithm.

**Data**: The current allocation matrix $M$
**Data**: A collection of applications *toPlace* for which resources must be allocated
**while** *toPlace not empty* **do**
  $a \leftarrow$ application with highest *cost*;
  $toPlace \leftarrow toPlace \setminus \{a\}$;
  $M' \leftarrow$ `place`$(a, M)$;
  **if** *placement of a unsuccessful* **then**
    $a' \leftarrow$ create a reduced version of application $a$;
    **if** $a'$ *still contains features with failure cost* **then**
      $toPlace \leftarrow toPlace \cup \{a'\}$
    **end**
  **end**
**end**
**return** $M$;

**Algorithm 1:** The `placeAll` function.

must be placed, ensuring critical application components will be allocated in a later iteration.

Using this approach, applications are either placed in their entirety, or if this is impossible, an effort is made to place a reduced version of the application. This ensures that important features, for which the cost of failure is high, will still be included even if not all other features can be made available. This approach for allocating collections of applications is based on the application-based feature placement algorithm which we previously presented in [5].

#### *5.1.1. The* `place` *function*

The `place` function is illustrated in Figure 10 and shown in detail in Algorithm 2. First, the function determines the different possible feature configurations where the open variation points are filled in. Then, the algorithm evaluates the possible configurations, using a `chooseBestFeatureSelection` function to select the best configuration. While doing so, it takes into account the current allocation, minimizing the number of new instances needed and maximizing the use of currently existing instances. Once the features that are chosen have been determined, the resources that must be allocated can be determined. This resource demand is then allocated on the existing feature instances using a `placeResidualCapacity` function. If, after allocating resources on existing instances, not all of the resource demand is handled, new feature instances are created using a `doCreateInstance` function. These last two steps are repeated until all demand has been allocated.

In the first step of the `place` function, it determines all of the different possible alternative feature allocations. This is done by at first selecting all of the features that are logically implied by the current feature selection (e.g. by adding parent features of selected features to the collection of parent features) and pruning optional features that are not implied as being included by adding them to the set of excluded features. If at this point there are still features that

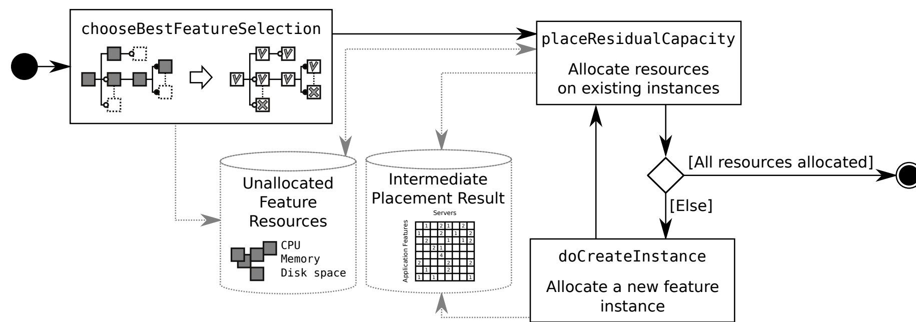



Figure 10: A high level overview of the `place` function which is responsible for allocating a single application. The function first determines a feature selection for the application, resulting in a collection of application features for which resources must be allocated. Resources are then allocated using existing feature instances. When there is no capacity left on existing feature instances, new instances are created.

**Data**: The current allocation matrix $M$
**Data**: An application $a$ for which resources must be allocated
Determine alternative feasible feature configurations $\phi(a)$;
$F \leftarrow$ `chooseBestFeatureSelection`$(M, \phi(a))$;
$d \leftarrow$ resources required for features $F$;
**while** *d not empty* **do**
    $(M, d) \leftarrow$ `placeResidualCapacity`$(M, d)$;
    **if** *d not empty* **then**
        $(M, d) \leftarrow$ `doCreateInstance`$(M, d)$;
    **end**
**end**
**return** $M$;

**Algorithm 2:** The `place` function.

are neither included nor excluded, new configurations are generated where, in every new configuration, one of the undecided features is added to the selected set. This process is repeated until only feature selections where every feature is either selected or excluded are left[3]. This results in a collection $\phi(a)$ containing the possible feature configurations of $a$.

*5.1.2. Choosing the best feature selection*

Next, the `chooseBestFeatureSelection` function is used to determine the best selection of features given the selected application features and the current allocation. In this function, the alternative allocations in $\phi(a)$ are compared using two criteria: the strict resource increase ($SRI$), and the total resource requirement ($TR$). The $SRI$ represents the amount *additional* strict resources that are needed to instantiate this application, thus not counting already existing application instances. The $TR$ measure computes the total resource demand of a configuration, showing its general resource requirement.

A low $SRI$ implies that the cost of allocating the specific feature selection is low, taking into account the current placement, as few additional instances are needed. If the feature selection can be instantiated entirely on existing instances, the $SRI$ value will be zero. A low $TR$ on the other hand implies that the cost of the selection is low in general. When instantiating new applications, we prioritize a low $SRI$ value, and $TR$ values are only minimized when $SRI$ values of two configurations are equal.

The $SRI$ of a feature selection represents the amount of strict resources that must be added to instantiate a specific feature selection taking into account the current placement state. This value is determined in three steps:

1. First, determine how many of the non-strict resources required for this

[3]Note that this process is only dependent on the application and feature model. Thus, these alternatives can be computed once when the application is added and do not have to be computed every iteration.

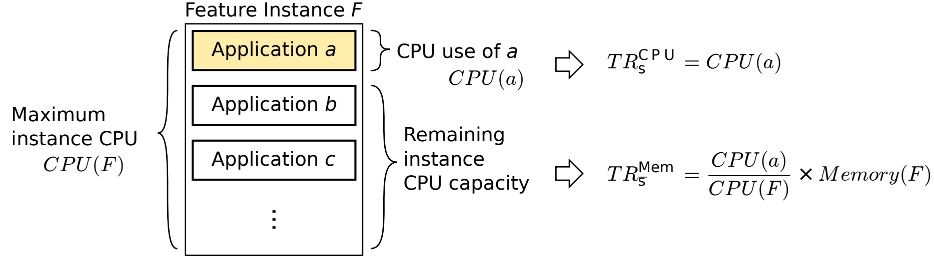



Figure 11: An illustration of how $TR$ values can be determined for an application $a$ using CPU and memory resources. $TR_s^{CPU}$ can be calculated directly based on the application demand. The memory requirement, $TR_{\overline{s}}^{Mem}$ is calculated indirectly as the feature instance's memory demand is shared between the different applications using the instance. The ratio of CPU resources used compared to the total available resources is used to determine the share of the total memory that is dedicated to the application.

allocation can be allocated on existing instances.

2. Based on this, determine the number of new feature instances that can be allocated to place the application.
3. Finally, the amount of strict resources needed to create these instances can be calculated. The resulting value is the $SRI$ for the analyzed application feature configuration.

Equation (4) shows how the $TR$ value for an application is computed. This value is composed out both non-strict resource demands $TR_{\overline{s}}$ and strict resource demands $TR_s$. As strict resource demands are more constraining than non-strict demands, the impact of $TR_{\overline{s}}$ is divided by 2, making it impact the total $TR$ value less than the $TR_s$ value. Non-strict resource demand is calculated by directly measuring the resource demand of an application feature configuration. Strict resource demand is calculated by determining the number of instances required for the configuration. Both $TR_s$ and $TR_{\overline{s}}$ values are normalized by dividing them by the maximum value for any application.

$$TR = \sum_{\gamma \in \Gamma_s} \frac{TR_s(\gamma)}{max_\gamma} + 0.5 \times \sum_{\gamma \in \Gamma_{\overline{s}}} \frac{TR_{\overline{s}}(\gamma)}{max_\gamma} \tag{4}$$

In Figure 11 an illustrative example is shown of how the $TR_s$ and $TR_{\overline{s}}$ values can be computed for CPU and memory resources. More formally, $TR_{\overline{s}}$, as defined in Equation (5), can be computed in a straightforward way: the total resource demand is the sum of the demand for the individual selected features (represented by the variable $required(f, \gamma)$). The $TR_s$ value of a feature selection is computed, as shown in Equation (6), by taking into account both the amount of resources needed for its instances and the share of the instance that is used for this specific configuration. This share, for a given feature $f$, is computed by determining the total amount of non-strict resources that are required for the feature, and comparing this value to the amount of these resources that can

be provided by a single feature instance. This is expressed in Equation (7).

$$TR_{\overline{s}}(\gamma) = \sum_{f \in selected} required(f, \gamma) \tag{5}$$

$$TR_{s}(\gamma) = \sum_{f \in selected} share(f) \times IR_{f}^{\gamma} \tag{6}$$

$$share(f) = \max_{\gamma \in \Gamma_{\overline{s}}} \frac{required(f, \gamma)}{L_{f}^{\gamma}} \tag{7}$$

As mentioned previously, the feature configuration with the lowest $SRI$ value is chosen. If, for two configurations the $SRI$ metrics are equal, the alternative with the lowest $TR$ value is preferred.

*5.1.3. Allocating resources*

Once a feature configuration has been selected, the `placeResidualCapacity` function, shown in Algorithm 3, is used to allocate as much of the demand as possible on existing feature instances. This is done by iterating over every feature that must still be allocated and every feature instance with remaining capacity that is currently allocated.

```
Data: The current allocation matrix M
Data: The demand d that is not yet allocated
Data: The application a for which the demand must be allocated
for every feature f in d do
    for every feature instance of i in M do
        s ← the server on which i is running;
        fCap ← remaining resource capacity of feature f;
        sCap ← remaining resource capacity of server s;
        for r ∈ Γ_s̄ do
            toAssign ← min(fCap(r), sCap(r), d(f, r));
            allocate(a, f, s, r, toAllocate);
            Update the remaining unallocated demand d;
        end
    end
end
return (M, d);
```

**Algorithm 3:** The `placeResidualCapacity` function allocates the remaining demand $d$ on currently existing feature instances.

If not all of the demand is provided after allocating resources on existing instances, additional feature instances must be instantiated. To achieve this, the `doCreateInstance` function, shown in Algorithm 4, is used. This function first determines the features with unassigned capacity, for which new feature instances must be instantiated. The algorithm then iterates over these features, allocating resources for them sequentially. For every feature $f$ that must be

**Data**: The current allocation matrix $M$
**Data**: The demand $d$ that is not yet allocated
**Data**: The application $a$ for which the demand must be allocated
$F^{unassigned} \leftarrow$ features with remaining unassigned capacity;
**for** *every feature* $f \in F^{unassigned}$ **do**
    $N \leftarrow$ determine number of features needed;
    $S' \leftarrow \{s|s \in S \wedge s$ has sufficient free resources$\}$;
    **while** $N > 0$ *and* $S' \neq \{\}$ **do**
        $s \leftarrow$ `selectBestServer`$(S', f)$;
        $N^s \leftarrow$ determine maximum possible number of instances of $f$ on $s$;
        `createInstance`$(M, f, s, \min(N, N^s))$;
        $N \leftarrow \min(N, N^s)$;
    **end**
    **if** $N > 0$ **then**
        **Abort:** Insufficient resources, allocation is impossible;
    **end**
**end**
**return** $M$;

**Algorithm 4:** The `doCreateInstance` function creates additional feature instances for the demand $d$ that is not yet allocated and allocates these resources.

instantiated, the number of times the feature must be instantiated, $N$, is determined using the remaining resource demand $d$. The servers within the datacenter that are capable of running the service and have sufficient free resources are then filtered and put in the set $S'$. The best server in $S'$ for instantiating the feature is then selected, and as many instances as needed and possible are instantiated on the server. This process is repeated until either $N$ new instances have been created, or until no servers are left to allocate instances on. If not all required instances can be created, it is impossible to fully allocate the desired applications. If this happens, the current `place` operation is aborted, and a new feature selection is determined as previously discussed in the description of the `placeAll` function.

*5.1.4. Server selection*

The `doCreateInstance` function compares different servers and makes use of the `selectBestServer`$(S', f)$ function to determine the best server in $S'$ to create instances of $f$ on. This function selects the server $s$ with the highest quality $Q(s)$. To ensure that the algorithm avoids using additional servers, the quality of an unused server is defined as 0. For other servers, this quality is composed of three factors: 1) the quality of remaining resources $QR(s)$ characterizes the desirability of using a server based on the remaining resources on the server; 2) the quality of the fit $QF(s)$ determines whether it is desirable to instantiate the feature on the server by determining the amount of resources that would be remaining on the server afterwards; finally 3) a bonus is added through a binary server use variable $SU(s)$ if an instance already physically exists. Equation (8)

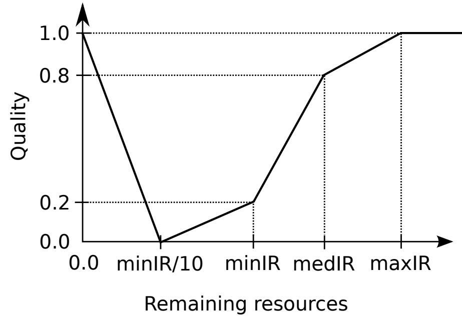



Figure 12: The piecewise linear function used to determine the quality of a fit. $minIR$ is the minimum resource requirement of an instance, $medIR$ is the median resource requirement and $maxIR$ is the maximum resource requirement. If there are more remaining resources than $maxIR$, any other feature can be instantiated on the server, indicating the fit is good. When there are nearly no remaining resources (less than $\frac{minIR}{10}$), the fit is good as well, as in this scenario few resources are wasted.

shows how this server quality can be computed by combining these three factors with different weights.

$$Q(s) = \begin{cases} 0.3 \times QR(s) + 0.5 \times QF(s) + 0.2 \times SU(s) & \text{if } s \text{ used} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

It is preferred to use servers with little remaining resources if possible, as this ensures other servers, with more remaining resources may be used for later, more complicated tasks. This is expressed, for a server, using the quality of remaining resources $QR(s)$ metric. This metric is shown in Equation (9), and represents the desirability of using a server regardless of the instance type that is to be instantiated on it. Within this equation, $remaining(r, s)$ refers to the amount of resources $r$ of $s$ that are currently remaining on the server, while $Ra_s^\gamma$ shows the amount of available resources on the server $s$.

$$QR(s) = 1 - \min_{r \in \Gamma_s} \frac{remaining(r, s)}{Ra_s^\gamma} \tag{9}$$

$QF(s)$ characterizes the quality of the fit of the chosen instance on the server, and represents the quality of remaining resources *after* an instance is allocated. This remaining capacity should either be enough to support new instances or alternatively, it should be very low ensuring few resources are wasted. This value is determined by, for every strict resource type, determining a separate $QF^r(s)$ value which in turn is determined by the amount of resources left after allocating the feature on the server. To these remaining resources, a piecewise linear function, shown in Figure 12, is applied. If after the allocation any other application instance may be instantiated, a high $QF^r(s)$ will be achieved. If after the allocation none of the other application instances may be instantiated, the remaining server capacity will be wasted, so a low $QF^r(s)$ value is achieved, unless the amount of residual resources becomes very low which again indicates

a good fit. The final $QF(s)$ value, defined in Equation (10), is the minimum of all $QF^r(s)$ values.

$$QF(s) = \min_{\gamma \in \Gamma_s} QF^r(s) \tag{10}$$

The $SU(s)$ variable is added to prevent instance migrations. During the execution of the DFPA algorithm multiple feature instances may be removed and added. $SU(s) = 1$ if in the original problem model there was already an instance of the feature on the server, otherwise $SU(s) = 0$. This makes it more likely to select a server on which an instance of the feature already exists, even though this instance may have already been removed previously by the *remove application* step of the algorithm.

*5.2. Refining placements*

After a change to the placement has been made, there are multiple ways in which the placement may be improved. First, it is possible to remove some applications to free resources on a server with low utilization, making it possible to turn it off, and to reallocate the removed applications elsewhere. A second possible improvement can be determined by reconsidering the placement of applications that have a relatively expensive feature configuration. These expensive configurations can occur as the placement algorithm chooses a feature configuration that requires the least new feature instances, even if it requires slightly more resources. After adding and removing other applications, this configuration may however no longer be ideal, so it may be beneficial to re-place this application at a lower cost.

The placement refinement operation starts with an allocation $M$, and generates a new allocation $M'$ by removing and re-adding a collection of applications $\hat{A}$. Subsequently, the quality of $M'$ is compared to the quality of $M$. If the quality of $M'$ is better than that of $M$ (i.e. the placement has a lower cost), this refined placement is returned. Otherwise, the original placement $M$ is returned. The `refine` function is designed to reconsider the placement of a limited collection of applications by ensuring it only moves about as many applications as one server can handle. The number of migrations is further limited by ensuring the refined placement is only enforced if it improves the resulting allocation.

The `refine` function is entirely defined by how the collection of applications $\hat{A}$ is determined. Applications within this set are chosen in two ways: by determining applications that are running on underutilized servers and by determining applications of which the current allocation requires a high amount of resources compared to its minimum resource use. The final application set is determined in three steps:

1. The collection of underutilized servers $\hat{S}$ is determined. This is a subset of all servers in $S$ with a utilization higher than 0% and lower than 70% for all resource types. The collection of applications of which features are allocated on a server $s \in \hat{S}$ is represented by $app^{on}(S)$.

2. The currently allocated applications are evaluated based on the cost feature configuration used to allocate them. For every application $a \in A$, the relative cost of their resource configuration can be computed by comparing their minimum and maximum resource demand. This results in a scalar value where 0 represents an application allocated using minimal resources, and 1 represents the maximum possible resource demand, making the application use more resources than needed. A set of applications $A^{maxCost}$ containing the two highest cost applications is then created.
3. Using the $A^{maxCost}$ and $\hat{S}$ collections the final application set $\hat{A}$ is determined. This is done by, for every server $s \in \hat{S}$, building a collection of applications $app^{on}(s) \cup A^{maxCost}$. Out of all of these collections, the collection containing the fewest applications that results in the highest number of freed servers is chosen as $\hat{A}$.

## 6. Evaluation Setup

### *6.1. Evaluated algorithms*

Based on the formal model discussed in Section 4, two algorithms can be designed using Integer Linear Programming (ILP). These algorithms were implemented using the CPLEX [33] solver, which is capable of optimally solving ILP problem formulations. The following algorithms were implemented:

- The ILP-based Feature Placement Algorithm (FPILP) is based on the formal model presented in the previous section without any considerations for the number of migrations. This algorithm yields an interesting baseline for the lowest possible cost. The results achieved by this algorithm may however not be practical as it does not take migration counts into account.

- The ILP-based Dynamic Feature Placement Algorithm (DFPILP) makes use of the iterative approach discussed in the previous section to determine a solution for the feature placement problem. As discussed, it first minimizes the cost, then the instance count increase, and finally the resource shift migrations.

We compare these ILP-based algorithms to the DFPA presented in the previous section.

### *6.2. Simulation parameters*

For our evaluations we base ourselves on an application use case containing three applications. We evaluate the algorithms for two scenarios with a varying datacenter load. Within the scenario, we use applications defined by the feature model presented in [5], containing the features of three applications: document processing, medical communications and a medical information management application. In total, this model defines 49 different features. Within the evaluations, we make use of a uniform server configuration with a 3GHz CPU and 4GB of memory. Every server is assigned an energy cost of 1. As

previously mentioned in Section 4, this energy cost is not necessarily a direct cost but rather a cost defined by the management system.

Application feature configurations are generated at random by randomly selecting features to include and exclude. For every application, a random cost of failure is chosen within the set $\{2, 4, 8, 16, 32\}$, representing the idea that some applications may have a much higher cost of failure than other. The application demand determined based on the application features and configuration is multiplied with a variable that is randomly chosen using a uniform distribution in the interval $[0.1, 10]$, ensuring there is a variation in application load. Feature failure costs are determined by defining essential features associated with services that must continue functioning, even if the rest of the application fails. If the essential feature is included and correctly provisioned a minimal service is delivered to the end users. If not all of the features can be placed correctly, and placement of an application fails, this minimal service should still be provided. A failure cost out of the set $\{2, 4, 8, 16, 32, 64\}$ is assigned for all essential features. Like for the application fail costs, this value indicates that the costs of feature failure may vary greatly depending on client demands. The maximum feature failure cost is higher than the maximum application failure cost as failing this feature causes a the interruption of critical services, while merely failing the application causes a service degradation.

A schedule is generated to determine when applications are started and stopped. To achieve this, a very large number of applications is generated and shuffled randomly. This ordered list of applications represents the order in which the applications are added. To ensure the datacenter does not become overloaded, as the total resource load of all of the generated applications greatly exceeds the total capacity of the cloud, we define a maximum datacenter resource load. If instantiating an application would result in exceeding the maximum datacenter load, an other application is first chosen to be removed. The resulting schedule will ensure that the datacenter is not overloaded, and will ensure that the application start events would always be accepted by an application request filter as described in Section 3. To choose the application to instantiate, first, the applications are stored as a list $a_i$ where the order of applications is determined by the order in which they were instantiated. The index of the application to be removed is then determined using a Gaussian probability with $\mu = n$ and $\sigma^2 = n/2$, with $n$ the number of applications that are instantiated. This ensures that very long running applications and very new applications have a smaller probability of being removed, which corresponds to client behavior: if an application has been running for a tenant for a very long time, it is a reliable client and there is a lower probability of him leaving, and if a tenant is very new, the application is often used for a period of time while its usefulness is determined and it is thus not stopped very quickly.

The first scenario represents an underloaded datacenter, with a maximum load of 90% for all datacenter resources. This results in an "easy" placement with a low probability of failed placements occurring, which ensures the cost should be almost entirely caused by server use costs. To ensure the ILP-based algorithms can function, the number of server remains limited to 50 In a second

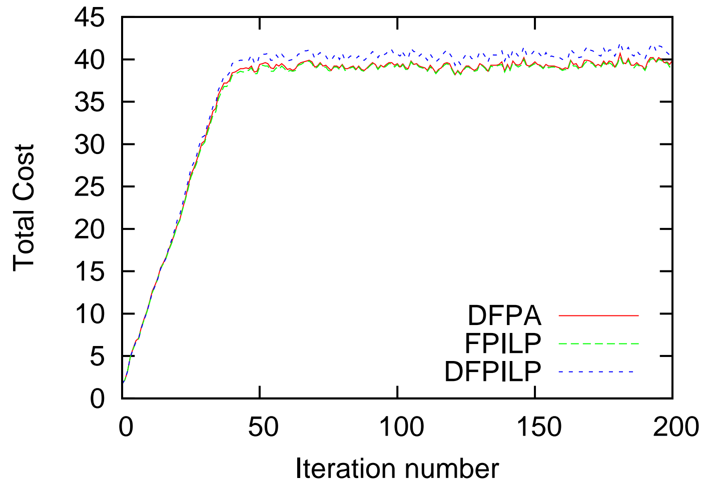



Figure 13: Total placement cost in an underloaded datacenter. Every iteration represents an application start or stop event. The quality achieved by the DFPA heuristic is the same as that achieved by the optimal FPILP algorithm.

scenario, a maximum load of 110% is chosen. In this overloaded datacenter scenario, placement becomes more complex, as more applications will fail to be placed. As this higher maximum load causes the number of applications that are active at any time to increase, the number of servers was reduced to 40 to ensure the ILP-based algorithms could compute solutions acceptably fast. For both scenarios 10 simulations were executed and the results were averaged.

## 7. Evaluation Results

### *7.1. Underloaded datacenter*

Figure 13 shows the total cost throughout the simulation for the three solvers for the underloaded datacenter. In this scenario, there is no significant difference between the performance of the DFPA heuristic and the FPILP algorithm, which always results in the optimal total costs. The DFPILP algorithm, which is based on multiple ILP optimizations performs worst, as it tolerates a slight decrease in solution optimality to reduce the number of migrations. Averaging the cost between iterations 50 and 200 (after the start up period of the evaluation), we observe that the DFPA heuristic on average only results in a cost that is 0.4% higher than the optimal cost.

Comparing the number of instance migrations, shown in Figure 14, we observe that the FPILP which does not take migrations into account results in very high numbers of migrations, making it unusable in dynamic scenarios where applications are started and halted at runtime. Both the DFPA heuristic and the DFILP algorithms result in fewer instance migrations. Due to the higher

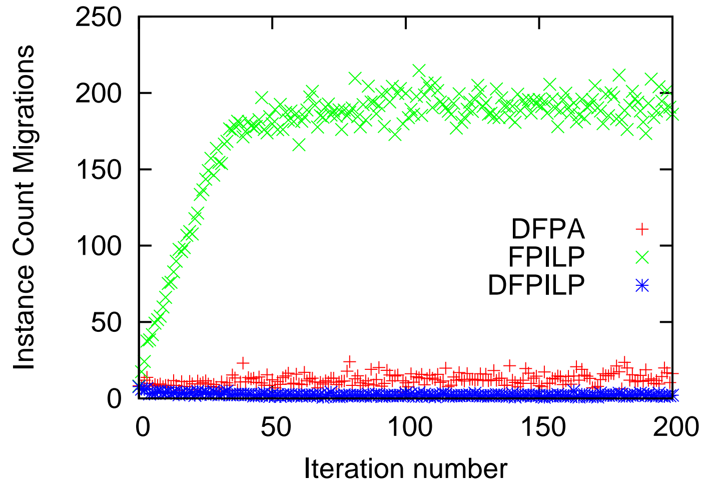



Figure 14: The number of instance migrations for every iteration of the dynamic placement algorithms in an underloaded datacenter. The number of IC migrations of DFPA is much lower than that of the FPILP algorithm which is unaware of migrations and slightly higher than that achieved by the DFPILP algorithm.

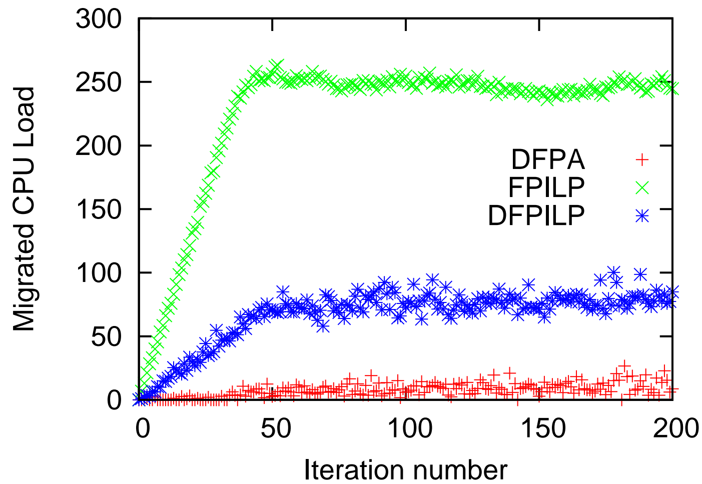



Figure 15: The amount of CPU resources that are moved between instances in subsequent algorithm invocations in an underloaded datacenter. The DFPA places applications while resulting in very few CPU load migrations, even when compared to the DFPILP algorithm.

Table 2: An analysis of the distribution of the total cost of the algorithms. The percentiles were determined using all of the entries between the 50th and 200th iteration.

| Algorithm | 50th pct | 95th pct | 98th pct | 99th pct |
|---|---|---|---|---|
| DFPA | 39 | 48 | 52 | 58 |
| FPILP | 39 | 42 | 42 | 43 |
| DFPILP | 40 | 44 | 44 | 45 |

cost observed previously, the DFPILP algorithm is capable of achieving fewer instance migrations than the DFPA heuristic. When the load shift is compared, shown in Figure 15, the FPILP algorithm again performs worst. Here, the DFPA heuristic outperforms the DFPILP algorithm, migrating noticeably less resources between servers in consecutive algorithm invocations. Compared to the non-dynamic FPILP algorithm, the DFPA algorithm results in a 77.5% reduction in IC migrations and a 96.5% reduction in resource shift migrations.

From this, we conclude that the DFPA performs well in an underloaded datacenter scenario: it achieves a similar cost to the optimal FPILP, but results in much less load shift and instance count migrations. Compared to the iterative ILP-based DFPILP algorithm it results in a lower total cost, and less resource shift at the cost of a slightly higher number of instance migrations.

*7.2. Overloaded datacenter*

The total cost of the algorithms in the overloaded datacenter scenario is shown in Figure 16. Generally, the performance of the DFPA heuristic remains similar to that of both the FPILP and DFPILP algorithms, but there are multiple outliers where performance decreases and a higher cost occurs. Usually, these peaks are temporary and they decrease in the next iterations. Table 2 shows variation of the total cost throughout the evaluation data points (using all entries between iterations 50 and 200, thus taking only the data points into account where the datacenter is fully utilized). Based on this information we observe that in 95% of the algorithm invocations a cost similar to that of the ILP-based algorithms is achieved. In the remaining cases, a performance notably worse than the ILP-based algorithms is observed. On average, the DFPA heuristic on average only results in a cost that is 2.8% higher than the optimal cost. The results for instance count migrations and load shift are comparable to those shown for the underloaded datacenter as can be observed in Figures 17, and 18. Comparing the non-dynamic FPILP algorithm to the DFPA heuristic, we observe a 92.7% reduction in IC migrations and a 96.1% reduction in resource shift migrations.

Based on our evaluation, we observe that the DFPA algorithm achieves good results in an underloaded datacenter scenario, but performs less consistently in an overloaded datacenter scenario. In practice, it is however best to use an access filter that does not permit datacenter overload, as otherwise some application components are bound to fail, making the DFPA an interesting approach as it

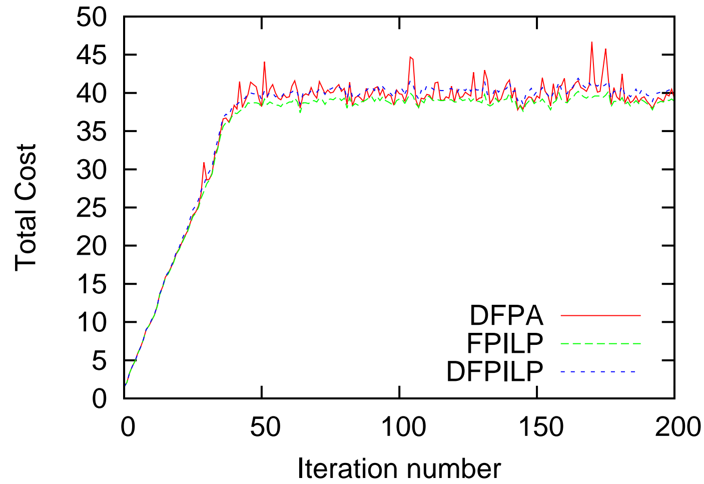



Figure 16: Total placement cost in an overloaded datacenter. The cost of the DFPA is less consistent than that of the ILP-based algorithms: at times the same quality of the FPILP algorithm is achieved, but often spikes in cost occur that are only solved in subsequent iterations.

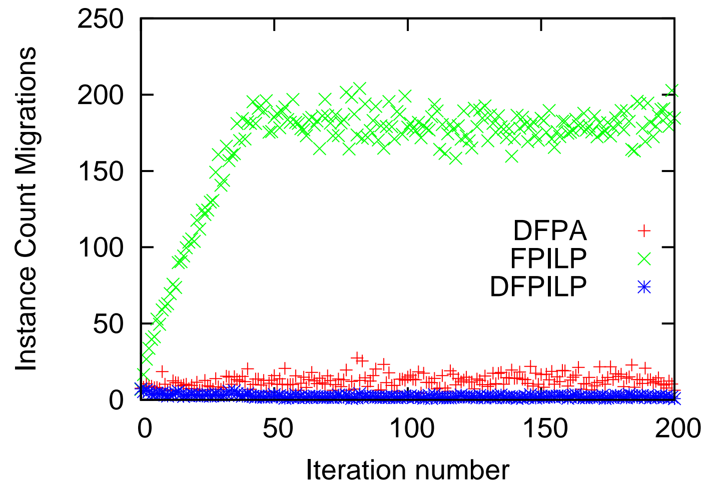



Figure 17: The number of instance migrations for every iteration of the dynamic placement algorithms in an overloaded datacenter. The DFPA results in slightly more IC migrations than the DFPILP algorithm, and significantly less migrations than the FPILP algorithm.

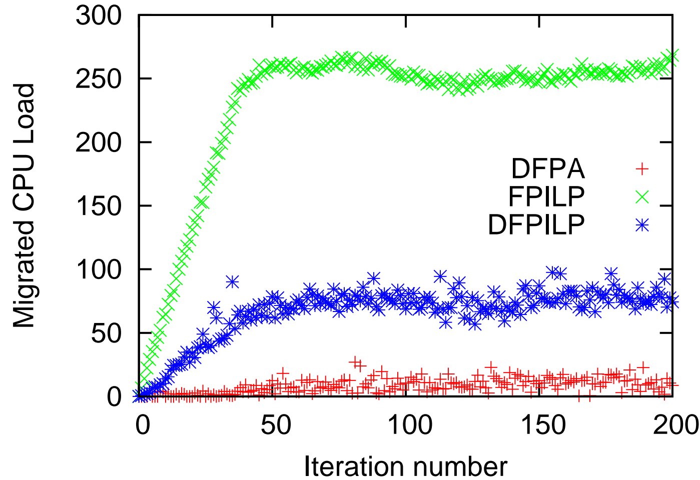



Figure 18: The amount of CPU resources that are moved between instances in subsequent algorithm invocations in an overloaded datacenter. Like in the underloaded datacenter scenario, the DFPA algorithm results in very few CPU load migrations.

does not make use of ILP formulations, making it scale much better than the FPILP and DFPILP algorithms.

## 8. Conclusions

A challenge in contemporary cloud platforms is that it is difficult to create and manage highly customizable applications while still achieving resource sharing through multi-tenancy. In this article we presented the concept of dynamic feature placement, an approach where customizable applications are composed using multiple interacting components and where individual application components can be shared between multiple applications and end users. The presented models and algorithms were designed to take into account dynamic scenarios where applications are started and stopped through time, taking migrations between the various steps into account.

We presented two new algorithms: the DFILP algorithm, an iterative ILP-based algorithm, and the DFPA heuristic. We analyzed the performance of the algorithms comparing them to a static optimal algorithm that is unaware service migrations. In our evaluations, we found that adding migration-awareness to the management algorithms reduces the amount of instance migrations by more than 77% and reduces the load moved between instances by more than 96%. Despite this, the heuristic DFPA algorithm results in a cost that is on average less than 3% more than the optimal cost.

**Acknowledgment**

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT).

[1] K. C. Kang, J. Lee, P. Donohoe, Feature-oriented product line engineering, IEEE Software 19 (4) (2002) 58–65. doi:10.1109/MS.2002.1020288.

[2] R. E. Filman, T. Elrad, S. Clarke, P. M. Aksit, Aspect Oriented Software Development, Addison-Wesley Professional, 2004.

[3] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, F. De Turck, Developing and Managing Customizable Software as a Service Using Feature Model Conversion, in: Proceedings of the 3rd IEEE/IFIP Workshop on Cloud Management (CloudMan 2012), IEEE, 2012, pp. 1295–1302. doi:10.1109/NOMS.2012.6212066.

[4] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, F. D. Turck, Feature Placement Algorithms for High-Variability Applications in Cloud Environments, in: Proceedings of the 13th Network Operations and Management Symposium (NOMS 2012), IEEE, 2012, pp. 17–24. doi:10.1109/NOMS.2012.6211878.

[5] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, F. De Turck, Cost-Effective Feature Placement of Customizable Multi-Tenant Applications in the Cloud, Journal of Network and Systems Managemement 22 (4) (2014) 517–558. doi:10.1007/s10922-013-9265-5.

[6] K. Pohl, G. Böckle, F. Van Der Linden, Software product line engineering: foundations, principles, and techniques, Springer-Verlag New York, Inc., 2005.

[7] M. Hinchey, S. Park, K. Schmid, Building Dynamic Software Product Lines, Computer 45 (10) (2012) 22–26. doi:10.1109/MC.2012.332.

[8] K. Zhang, X. Zhang, W. Sun, H. Liang, Y. Huang, L. Zeng, X. Liu, A Policy-Driven Approach for Software-as-Services Customization, in: Proceedings of the 9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, (CEC/EEE 2007), IEEE, 2007, pp. 123–130. doi:10.1109/CEC-EEE.2007.9.

[9] W. Sun, X. Zhang, C. J. Guo, P. Sun, H. Su, Software as a Service: Configuration and Customization Perspectives, in: IEEE Congress on Services Part II (services-2), IEEE, 2008, pp. 18–24. doi:10.1109/SERVICES-2.2008.29.

[10] R. Mietzner, A. Metzger, F. Leymann, K. Pohl, Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications, in: Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS 2009), IEEE, 2009, pp. 18–25. doi:10.1109/PESOS.2009.5068815.

[11] M. Abu-Matar, H. Gomaa, Feature Based Variability for Service Oriented Architectures, in: Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011), IEEE, 2011, pp. 302–309. doi:10.1109/WICSA.2011.47.

[12] M. Abu-Matar, H. Gomaa, Variability Modeling for Service Oriented Product Line Architectures, in: Proceedings of the 15th International Software Product Line Conference (SPLC 2011), ACM, 2011, pp. 110–119. doi:10.1109/SPLC.2011.26.

[13] S. T. Ruehl, U. Andelfinger, Applying Software Product Lines to create Customizable Software-as-a-Service Applications, in: Proceedings of the 15th International Software Product Line Conference (SPLC 2011), ACM, 2011, pp. 16:1–16:4. doi:10.1145/2019136.2019154.

[14] G. H. Alférez, V. Pelechano, Context-Aware Autonomous Web Services in Software Product Lines, in: Proceedings of the 15th International Software Product Line Conference (SPLC 2011), ACM, 2011, pp. 100–109. doi:10.1109/SPLC.2011.21.

[15] B. Jennings, R. Stadler, Resource Management in Clouds: Survey and Research Challenges, Journal of Network and Systems Management (2014) 1–53doi:10.1007/s10922-014-9307-7.

[16] J. Rolia, A. Andrzejak, M. Arlitt, Automating Enterprise Application Placement in Resource Utilities, in: M. Brunner, A. Keller (Eds.), Self-Managing Distributed Systems, Vol. 2867 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 118–129. doi:10.1007/978-3-540-39671-0_11.

[17] B. Urgaonkar, A. L. Rosenberg, P. Shenoy, Application Placement on a Cluster of Servers, International Journal of Foundations of Computer Science 18 (05) (2007) 1023–1041. doi:10.1142/S012905410700511X.

[18] C. Adam, R. Stadler, Service Middleware for Self-Managing Large-Scale Systems, IEEE Transactions on Network and Service Management 4 (3) (2007) 50–64. doi:10.1109/TNSM.2007.021103.

[19] C. Tang, M. Steinder, M. Spreitzer, G. Pacifici, A scalable application placement controller for enterprise data centers, in: Proceedings of the 16th International Conference on World Wide Web (WWW 2007), ACM, 2007, pp. 331–340. doi:10.1145/1242572.1242618.

[20] X. Zhu, C. Santos, D. Beyer, J. Ward, S. Singhal, Automated application component placement in data centers using mathematical programming, International Journal of Network Management 18 (6) (2008) 467–483. doi:10.1002/nem.707.

[21] D. Breitgand, A. Epstein, SLA-aware Placement of Multi-Virtual Machine Elastic Services in Compute Clouds, in: Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), IEEE, 2011, pp. 161–168. doi:10.1109/INM.2011.5990687.

[22] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, E. Snible, Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-Aware Virtual Machine Placement, in: Proceedings of the 2011 IEEE International Conference on Services Computing, IEEE, 2011, pp. 72–79. doi:10.1109/SCC.2011.28.

[23] L. Shi, B. Butler, D. Botvich, B. Jennings, Provisioning of requests for virtual machine sets with placement constraints in IaaS clouds, in: Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), IEEE, 2013, pp. 499–505.

[24] G. Foster, G. Keller, M. Tighe, H. Lutfiyya, M. Bauer, The right tool for the job: Switching data centre management strategies at runtime, in: Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), IEEE, 2013, pp. 151–159.

[25] J. Xu, J. a. B. Fortes, Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments, in: Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing, IEEE, 2010, pp. 179–188. doi:10.1109/GreenCom-CPSCom.2010.137.

[26] D. Carrera, M. Steinder, I. Whalley, J. Torres, E. Ayguadé, Utility-based placement of dynamic web applications with fairness goals, in: Proceedings of the 11th Network Operations and Management Symposium (NOMS 2008), IEEE, 2008, pp. 9–16. doi:10.1109/NOMS.2008.4575111.

[27] F. Wuhib, R. Stadler, M. Spreitzer, Gossip-based Resource Management for Cloud Environments, in: Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), IEEE, 2010, pp. 1–8. doi:10.1109/CNSM.2010.5691347.

[28] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, A. Tantawi, Dynamic placement for clustered web applications, in: Proceedings of the 15th International Conference on World Wide Web (WWW 2006), ACM, 2006, pp. 595–604. doi:10.1145/1135777.1135865.

[29] C. Low, Decentralised Application Placement, Future Generation Computer Systems 21 (2) (2005) 281–290. doi:10.1016/j.future.2003.10.003.

[30] T. Kimbrel, M. Steinder, M. Sviridenko, A. Tantawi, Dynamic Application Placement Under Service and Memory Constraints, in: S. Nikoletseas (Ed.), Experimental and Efficient Algorithms, Vol. 3503 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 391–402. doi:10.1007/11427186_34.

[31] pure-systems GmbH, pure::variants User's Guide, last accessed: December 2012 (2012).
URL http://www.pure-systems.com/Documentation.116.0.html

[32] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, F. De Turck, Network-Aware Impact Determination Algorithms for Service Workflow Deployment in Hybrid Clouds, in: Proceedings of the 8th International Conference on Network and Service Management (CNSM 2012), IEEE, 2012, pp. 28–36.

[33] IBM ILOG CPLEX 12.4 (2013).
URL http://www-01.ibm.com/software/integration/optimization/cplex-optimizer

**Hendrik Moens** received a Masters degree in computer science from Ghent University in 2010. He is a Ph.D. student in the Department of Information Techology of the Ghent University, and a member of the network and service management research group. His research interests include the management of large scale cloud computing environments and autonomic management systems.

**Bart Dhoedt** received a Masters degree in Electro-technical Engineering (1990) from Ghent University. His research, addressing the use of micro-optics to realize parallel free space optical interconnects, resulted in a Ph.D. degree in 1995. After a 2-year post-doc in opto-electronics, he became Professor at the Department of Information Technology. His research interests include software engineering, distributed systems, mobile and ubiquitous computing, smart clients, middleware, cloud computing and autonomic systems.

**Filip De Turck** leads the network and service management research group at the Department of Information Technology of the Ghent University, Belgium and iMinds. He received his Ph.D. degree from the Ghent University in 2002. He is a full-time professor since October 2006 in the area of telecommunication and software engineering. His main research interests include scalable software architectures for telecommunication network and service management, performance evaluation and design of Cloud management systems.