VM Placement in non-Homogeneous IaaS-Clouds

Konstantinos Tsakalozos, Mema Roussopoulos, and Alex Delis

University of Athens, Athens, 15748, Greece,
{k.tsakalozos, mema, ad}@di.uoa.gr

Abstract. Infrastructure-as-a-Service (IaaS) cloud providers often combine different hardware components in an attempt to form a single infrastructure. This single infrastructure hides any underlying heterogeneity and complexity of the physical layer. Given a non-homogeneous hardware infrastructure, assigning *VM*s to physical machines (*PM*s) becomes a particularly challenging task. *VM* placement decisions have to take into account the operational conditions of the cloud (e.g., current *PM* load) and load balancing prospects through *VM* migrations. In this work, we propose a service realizing a two-phase *VM*-to-*PM* placement scheme. In the first phase, we identify a promising group of *PM*s, termed *cohort*, among the many choices that might be available; such a cohort hosts the virtual infrastructure of the user request. In the second phase, we determine the final *VM*-to-*PM* mapping considering all low-level constraints arising from the particular user requests and special characteristics of the selected cohort. Our evaluation shows that in large non-homogeneous physical infrastructures, we significantly reduce the *VM* placement plan production time and improve plan quality.

1 Introduction

Infrastructure-as-a-Service (IaaS) cloud providers often face the following challenge: they must offer uniform access (resource provision) over a non-uniform hardware infrastructure. Non-homogeneous infrastructures may be the product of hardware upgrades, where old resources are left operational alongside new ones, or federated environments, where several parties are willing to share hardware resources with diverse characteristics.

Resource management of non-homogeneous hardware resources has been extensively studied [7]. Typically, a resource management system receives, queues, and finally matches user job requirements with the characteristics of the offered hardware. For instance, scheduling jobs in the Grid requires choosing an appropriate Grid site that complies with the user requirements. The advent of the clouds has introduced very strict abstractions over the physical resources. IaaS clouds restrict users from specifying the exact physical resources to be consumed when instantiating virtual machines (*VM*s). Cloud consumers remain agnostic of the underlying physical infrastructure. Only high-level resource requirements such as CPU and RAM are stated in user requests. In return, the cloud offers new options for load balancing. Live *VM* migration allows for relocation of jobs

to offloaded hardware inside the cloud in a manner transparent to the user. Thus, the *VM*-to-physical machines placement policies must be revisited in the context of the cloud to take into account both the new enhancements and the additional constraints that cloud abstractions offer.

In this paper, we focus on the problem of instantiating entire virtual infrastructures in large non-homogeneous IaaS clouds. We introduce a service implementing a two-phase mechanism. In the first phase, we synthesize infrastructures out of existing promising physical machines (*PM*s). These dynamically-formed physical infrastructures, termed *cohorts*, host the user-requested *VM*s. In the second phase, we determine the final *VM*-to-*PM* mapping considering all low-level constraints arising from the particular user requests and special characteristics of the most promising selected *cohorts*. Compared to other constraint-based *VM* scheduling systems [8, 9, 20], the novelty of our approach mainly lies in the first phase. During this phase, besides resource availability, we also take into account properties such as migration capabilities, network bandwidth connectivity, and user-provided deployment hints. This helps prune out many possible *cohorts* within the cloud, and thus reduces the time required to produce a deployment plan in the second phase. We express both the selection of hosting nodes and the production of *VM*-to-*PM* mappings as constraint satisfaction problems and we use cloud-resources to solve these problems. Our evaluation shows that this approach *1)* scales effectively for hundreds of *PM*s, *2)* reduces plan production time by up to a factor of 9, and *3)* improves plan quality by up to a factor of 4, when compared to a single-phase *VM* placement approach.

2 Overview of Our Approach

We assign user-requested *VM*s to cloud-provided *PM*s through a service implementing a two-phase optimization process. During the first phase we select a subset of *PM*s with properties that best serve the *VM* placement. We term these *dynamically formed subsets* of *PM*s *cohorts*. Cohorts may entail *PM*s from a single rack and/or machines spread across the network. In the second phase, we solve a constraint satisfaction problem that yields a near optimal *VM*-to-*PM* mapping. Constraints emanate from user-provided deployment hints and internal cloud specifications such as hardware resource characteristics and administration preferences. The goal in selecting a subset of all available *PM*s, during the first optimization phase, is to reduce the number of constraints and the search space during the second phase.

To serve a user request for a virtual infrastructure, a single cohort has to be selected to host all *VM*s involved. The main idea in cohort selection is that we need to assist future load balancing requests in the context of an IaaS cloud. Since load balancing is better performed among *PM*s supporting live migration, we favor cohort formation among such nodes. In case, the user-requested resource requirements exceed the *VM* hosting capacity of all *PM* pools supporting live migration, we must merge neighboring pools to form larger ones. We synthesize

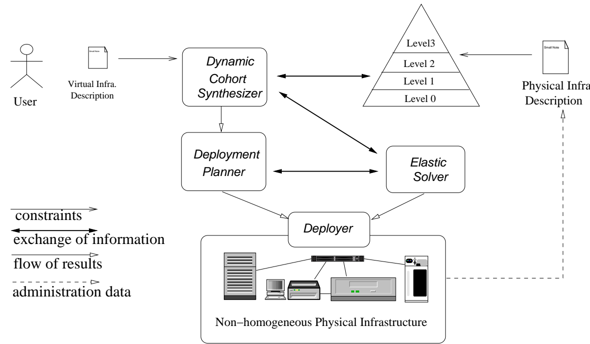


Fig. 1. High level view of our approach.

cohorts based on a *4-level hierarchy*, depicted as a triangle in Figure 1. These four levels are defined as follows:

- At **level 0**, all groups of *PM*s that support live migration make up corresponding cohorts. Load balancing through *VM* migration for these cohorts is transparent. In addition, each *PM* that does not feature live migration makes up a cohort on its own.
- At **level 1**, we may form cohorts from diverse groups of level 0. Cohorts of level 1 may involve *PM*s supporting both live migration as well as migration through suspend/resume.
- **Level 2** features cohorts among which migration is infeasible due to hardware incompatibilities, insufficient network bandwidth, etc.
- **Level 3** is a single cohort consisting of the entire non-homogeneous physical infrastructure and so it signifies the maximum amount of resources available.

When serving a user request, we try to satisfy all virtual infrastructure requirements through a single *cohort*. The search for such a cohort starts from level 0 and may reach up to level 3. The higher the level of the selected cohort is, the more computational resources are needed to solve the *VM* assignment problem in the second phase. As more *PM*s are available in cohorts generated at higher levels, the search space of the *VM*-to-*PM* mapping increases. Cohort selection is formulated as a constraint satisfaction problem discussed in detail in Section 4.

In the second phase, where the final *VM*-to-*PM* assignment is produced, all fine-grained constraints of the selected cohort are taken into account in a constraint satisfaction problem. These fine-grained constraints refer to *a)* all specifics regarding the hosting capacity including features and resource availability in *PM*s, *b)* user-provided deployment hints and *c)* cloud administrative goals.

Figure 1 presents a high-level view of our approach. The user submits a request for a virtual infrastructure to the *Dynamic Cohort Synthesizer*. The user request includes both *VM* specifications and deployment hints. The *PM*s of the selected cohort along with the user request are forwarded to the *Deployment*

Planner that produces the final *VM*-to-*PM* mapping. We term this mapping *deployment plan*. For all user requests encountered all respective deployment plans are delivered to the *Deployer*, that in turn interacts with the cloud's facilities and coordinates each *VM* instantiation.

Both the *Dynamic Cohort Synthesizer* and the *Deployment Planner* have to solve constraint satisfaction problems. They do so using resources of the cloud itself. The *Elastic Solver*, of Figure 1, is a service providing access to a set of *VM*s in the cloud that form a distributed constraint satisfaction solver.

3 User Provided Hints & Constraints

Shown in Figure 1, the deployment of a virtual infrastructure starts with the user submitting a request for *VM*s to our service. The user request is an infrastructure description (XML document [19]) with the following sections:

1. Description of all *VM*s and resource requirements.
2. One or more possible infrastructure deployments, each one accompanied with its own set of deployment hints. The deployment hints are translated into user-provided constraints that drive the cloud's *VM*-to-*PM* assignment. Table 1 shows some commonly used hints.
3. The conditions under which a transition from one infrastructure deployment to another is required. A transition may enable and/or disable deployment hints associated with the respective infrastructure deployments. In turn, this may call for *VM* migrations.

Apart from the constraints derived from hints in the infrastructure description, our approach also leverages constraints describing the internal physical cloud infrastructure. Such constraints refer to both the availability of resources and high-level resource management goals that the cloud administration may require (e.g., *PowerSave* hint of Table 1).

Within the *Dynamic Cohort Synthesizer*, hints are realized as cohort evaluation cost functions. The same hints are realized by the *Deployment Planner* as specific deployment plan evaluation functions driving the *VM*-to-*PM* assignment. In other words, deployment hints are interpreted in different ways depending on the phase of our approach. Some hints are even ignored during the cohort selection phase. *ParVMs* is a typical example of how the same hint is treated in different ways. During cohort selection, we penalize cohorts that provide fewer *PM*s than the number of *VM*s referenced in the *ParVMs* deployment

Table 1. Commonly used deployment hints.

FavorVM	Try to reserve a single *PM* for a specific *VM*.
MinTraf	Minimize traffic by co-deploying a set of *VM*s on the same *PM*.
ParVMs	Spread *VM*s across separate *PM*s.
PowerSave	Reduce the number of *PM*s used for *VM* deployment.
EmptyNode	Offload a specific *PM*.

hint, whereas during the final *VM*-to-*PM* assignment we penalize plans that place the *VM*s referenced in the same *ParVMs* hint on the same *PM*. To reduce the number of constraints considered, the *Dynamic Cohort Synthesizer* ignores certain hints such as the *MinTraf*.

After a deployment plan is applied –through respective *VM* placement operations – the deployment hints used in the production of the *VM*-to-*PM* mapping are not discarded. Some deployment hints have "side-effects" on future deployment plans. A deployment hint is marked to have "side-effects" if it has to be considered during the deployment of virtual infrastructures of future user requests. Deployment hints with "side effects" are known to the cloud administration and are marked as such upon their implementation. A typical example of such a hint is the *FavorVM*, which calls for a *VM* to be placed on an offloaded *PM*. This *PM* should be kept offloaded as long as the *VM* referenced in the deployment hint is online. Consequently, the deployment of future virtual infrastructures should also respect any *FavorVM* hints already in place.

4 Synthesis of Dynamic Infrastructures

Selecting a single subset of all *PM*s (cohort) requires iterating over the levels discussed in Section 2. The *Dynamic Cohort Synthesizer* ranks the cohorts of the same level based on metrics provided by the cloud administration. These metrics may reflect the cohort's load, its reliability (e.g. redundant hardware), or even higher level properties such as its prospect of load exchange with other cohorts.

Algorithm 1 gradually explores all cohort levels in search of a promising "neighborhood". The input of the algorithm consists of a) the user *request*, b) the number of candidate cohorts (k) which should be used as the starting seed for the dynamic formation of the next level cohorts, c) the *load_threshold* (i.e., average CPU utilization) over which a cohort is considered to be overloaded and d) a resource availability factor *(overcommit)* indicating how many times the resources of a cohort should surpass the resources requested. Both input parameters *overcommit* and k allow cloud administrators to tune the quality of the produced deployment plans. The *overcommit* parameter ensures that the *Deployment Planner* will have enough space to search for a *VM*-to-*PM* mapping during the second optimization phase of our approach. The k parameter allows cloud administrators to reduce the amount of lower level cohorts used as a starting point in cohort synthesis. Since each cohort synthesis attempt results in a simulated annealing execution, high k values reduce the danger of getting trapped into a local optimum. This is because each execution of the simulated annealing starts with a different cohort as seed.

Starting from *level 0*, we first rank cohorts given that we need to satisfy the provided user request (*CohortRanking* function call of line 2). We also filter out cohorts that do not satisfy the overload threshold and the resource availability factor (*CohortFiltering* call of line 3). If all cohorts are filtered out, then we must search higher level cohorts using the while loop of lines 4 to 12. In line 6,

Algorithm 1 *Dynamic Cohort Synthesizer*

Input: *request*: user request for a virtual infrastructure

k: The top-k cohorts will be returned

load_threshold: Threshold over which the cohort is considered overloaded

overcommit: How many times the cohort's resources must surpass the requested resources

Output: Set of cohorts we will consider for deployment

```
1: level := 0 ; ranked_cohorts := ∅
2: ranked_cohorts := CohortRanking(level,request);
3: ranked_cohorts := CohortFiltering(ranked_cohorts, load_threshold, overcommit,
       request);
4: while ranked_cohorts = ∅ and level < 4 do
5:    ranked_cohorts := ∅
6:    graded_cohorts := CohortRanking(level,request);
7:    for i := 0 ; i < k; i++ do
8:       good_cohort := MergeCohorts(level, graded_cohorts[i], request,
             load_threshold, overcommit)
9:       ranked_cohorts := ranked_cohorts ∪ {good_cohort}
10:   end for
11:   level := level + 1
12: end while
13: return ranked_cohorts
```

we use the *CohortRanking* function to grade all cohorts of the level indicated by variable *level*. The top-k highest scoring cohorts of the current level are used as a starting point in exploring the next level up. Merging lower level cohorts is performed in the *MergeCohorts* call of line 8. Below, we outline the functionality of the following routines: *CohortRanking*, *CohortFiltering* and *MergeCohorts*.

Cohort Ranking: Each cohort maintains the following key properties: *a)* the number of *PM*s it contains, *b)* resource availability indicators including CPU average load, total/unused RAM, hard disk capacity, redundancy and high availability features. For simplicity, we elaborate the first two indicators in the discussion that follows, *c)* average bandwidth of network connections among *PM*s within the cohort and *d)* a set of cohort-specific characteristics (e.g., CPU architecture).

Static characteristics of cohorts at level 0 are provided by the cloud administrator (with the *"Physical Infrastructure Description"* of Figure 1). In this regard, we expect the administrator to specify the properties of all *PM*s as well as the cohorts supporting live migration. Recall that each live migration group of *PM*s is a level 0 cohort and each *PM* that does not support live migration forms a cohort by itself. Using the *PM* properties we compute a score for every available cohort at level 0 as follows: initially we evaluate the resource availability within the cohort:

$$ResEval_0(Cohort) = \sum_{i \in R} w_i * (Provided_i(Cohort) - Required_i) \quad (1)$$

where R is the set of resources including average CPU, RAM, and network bandwidth utilization rates. $Provided_i$ and $Required_i$ represent the provision and requirement of resource i respectively. The weights w_i are set by the administrator to reflect the importance of each resource and to normalize the intermediate results. These administrator-defined weights allow our approach to be tuned to match specific requirements and administrative preferences. Next, we

evaluate requirements resulting from both user provided hints and administrator's imperatives:

$$ConstrEval(Cohort) = \sum_{j \in Constr.} w_j * Constraint_j(Cohort) \quad (2)$$

where $Constraint_j$ are cost functions expressing user hints and administration preferences as we describe in Section 3. Again, w_j indicates constraint importance. Each constraint function takes as input a cohort and returns the degree of the human-provided preference/hint satisfaction ($Constraint : Cohort \mapsto [0, 1]$). We designate the sum of *ResEval* and *ConstrEval* to be the cohort's overall score:

$$Score(Cohort) = ResEval(Cohort) + ConstrEval(Cohort) \quad (3)$$

The $ResEval(Cohort)$ at levels above 0 are computed recursively as follows:

$$ResEval_n(Cohort) = \frac{\sum_{s \in S} ResEval_{n-1}(s)}{|S|} \quad (4)$$

where n represents the level and S is the set of all lower level cohorts within the $Cohort$.

Cohort Filtering: Algorithm 1 filters out cohorts that are not worthy to be considered in the final *VM* placement. This action takes place in line 3 and within the function *MergeCohorts* of line 8. Filtering uses the *overcommit*, *load_threshold* limits as well as information regarding the specific resources requested by the user. We exclude cohorts that do not comply with the rules of Table 2. *Rule 1*, avoids stressing overloaded cohorts. *Rule 2* ensures that the candidate cohorts have sufficient resources so that in the second phase (final *VM*-to-*PM* mapping) there will be enough options to choose from and produce a high-scoring plan. Finally, *Rule 3* functions in levels 0 and 1 and filters out migration-incompatible combinations of cohorts.

Cohort Merging: Through the merging of cohorts of a certain level we synthesize more comprehensive cohorts at the next level up. This operation is required when the hosting capacity of each and every cohort at the current level does not suffice to address all user resource demands. The cohort to be formed must have both the *VM* hosting capacity and the characteristics to match the user request. Thus, the goal of the merging process is to produce a number of high-scoring cohorts and then choose the "best". The formula used to compute the cohort ranking is also used here to designate this best selection. We have formulated

Table 2. Cohort Filtering Rules.

Rule 1:	A cohort's resource availability is below a certain "threshold" (set as input in Alg. 1)
Rule 2:	A cohort's resource availability is less than "overcommit" times the requested quantity
Rule 3: levels 0 & 1	Mismatch with user-provided specifications, such as differences in CPU architecture

Algorithm 2 Simulated-Annealing

Input: *same_iterations*: Maximum number of iterations yielding no improvement

T: Temperature

GetNeighborOf(): Space exploration function

Score(): Score function

seed: Starting seed of space exploration

Output: A near-optimal solution

```
1:  same := 0
2:  best_solution := current_solution := GetInitialSolution(seed)
3:  while same <same_iterations do
4:    new_solution := GetNeighborOf(current_solution)
5:    D = Score(new_solution) - Score(current_solution)
6:    if ( T > 10^-5 AND e^(D/T) > Random()) OR (T < 10^-5 AND D > 0) then
7:      current_solution := new_solution
8:    end if
9:    if Score(new_solution) > Score(best_solution) then
10:     best_solution := new_solution ; same := 0
11:   end if
12:   same++ ; T := 0.99 * T
13: end while
14: return best_solution
```

the above selection as an optimization problem whose constraints are the user needs and administration preferences. As the number of these constraints and their combinations while merging cohorts may increase exponentially to the size of the physical infrastructure, we resort to finding a near-optimal solution. To this end, we employ a stochastic and easily parallelizable approach –depicted in Algorithm 2– that is based on simulated annealing [11].

The primary objective of Algorithm 2 is to create from a lower-level *seed* cohort, a new formation at the current level. The solution has to be a single cohort that complies with the rules of Table 2. Towards achieving this objective, *GetNeighborOf* forms new potential "coalitions of resources" through navigation among cohorts of the current level. The function generates new cohorts by merging neighboring lower-level cohorts connected through at least one network route. More specifically, the lower level cohorts are randomly selected with only one requirement: each potential merging operation will result in a single cohort with its nodes adequately networked (i.e., preferably nodes that have direct physical links). Every cohort produced by *GetNeighborOf* is assigned a score within Algorithm 2 with the help of Eqn. 3. This computation is carried out efficiently as the resource evaluation is a single average sum and the constraint evaluation uses a reduced, in terms of size, set of constraints.

Algorithm 2 input parameters, T and *same_iterations* are used to designate the termination condition of the simulated-annealing procedure. As the algorithm visits more solutions, its temperature (T) drops. In high temperature states, we are allowed to choose a new solution even if it is worse than the one we currently have at hand. In this respect, Algorithm 2 avoids getting trapped in local optima. The maximum number of allowed consecutive iterations that yield no improvement is defined by the *same_iterations* parameter. As soon as the value of *same_iterations* is reached, we assume that a local near-optimum

solution is found. As we show in Algorithm 1, *MergeCohorts* is called k times. Every time, we use as input a different cohort from the top-k cohorts of the previous level to serve as *seed* in Algorithm 2.

5 Deployment Plan Production

All properties of the selected cohort (i.e., *PM*s, resource availability, network connections and cohort-specific characteristics) along with the user-provided constraints (deployment hints) are used as input to the *Deployment Planner* that ultimately produces the actual mapping of *VM*s-to-*PM*s in a way similar to the one discussed in [19]. The *Deployment Planner* also takes into account previously encountered deployment hints referring to virtual infrastructures already deployed and operational on (some) nodes of the selected cohort. However, since these infrastructures are already deployed, the set of deployment hints can be trimmed down only to those hints marked to have "side effects" as we discuss in Section 3. We employ Algorithm 2 to produce the mapping of *VM*s to *PM*s. Again, we need to designate two aspects: a) how to select a neighbor solution commencing with a *seed*-mapping and b) how to ascertain the value of the produced candidate solution.

In general, the plans neighboring a specific deployment plan p are designated by the following formula:

$$N_p = \{P \in \mathit{AllPlans} \mid \mathit{Prob}(P(v) \neq p(v)) = d, \forall v \in V\}, \tag{5}$$

where V is the set of *VM*s under deployment and d is the value of the probability that a *VM* v is to be deployed on a *PM* other than the one currently set in plan p. High values of d result in producing almost random deployment plans that render Algorithm 2 ineffective. On the other hand, significantly reducing d may trap the search process for a neighbor(s) into local optima. A detailed discussion on this plan generation procedure can be found in [19].

The input scoring function is implemented as the weighted sum of the cost evaluation functions corresponding to the constraints relevant to the user request at hand. For a given deployment plan m the *Score* is:

$$\mathit{Score}(m) = \sum_{\mathit{Const}_i \in C_s} w_i \mathit{Const}_i(m) \tag{6}$$

where C_s is the set of constraints and w_i the respective user/administrator imposed weights indicating the importance of each constraint.

6 Elastic Solver Service

Simulated annealing is inherently parallelizable and can effectively harvest the distributed nature of the same cloud infrastructure we administer. Every time the simulated annealing is invoked, it can use different seeds (e.g. Algorithm 1 line 8). In this manner even if one execution gets trapped into a local optimum, a plausible solution can be ultimately found by another execution.

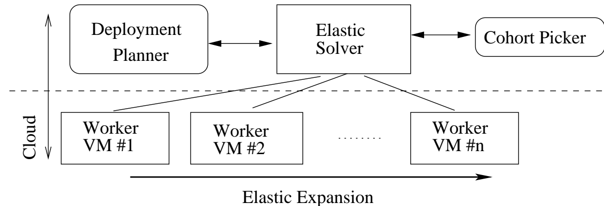


Fig. 2. High level view of the *Elastic Solver*.

The *Elastic Solver* is a service providing virtual infrastructures used to solve the constraint satisfaction problems described in the previous sections. As Figure 2 depicts, the solver follows a master-workers architecture and interacts with both the *Dynamic Cohort Synthesizer* and the *Deployment Planner*. As the virtual infrastructures employed by the *Elastic Solver* are hosted in the cloud, this service also acts as a special cloud client. Worker *VM*s requested by the solver are configured so that as soon as they come online they register with the master component of the *Elastic Solver* and declare their availability. These workers remain idle until a request for solving a cohort-merging or a *VM* placement problem arrives. In [18], we show how we are able to dynamically adjust the exact number of worker nodes so that our "profit" in using this service is maximized. We employ an iterative process that periodically adds or removes worker nodes in an attempt to assess the *VM*'s performance and to properly adjust the number of worker nodes. Larger elastic solver infrastructures result in better deployment solutions and reduced deployment time but at the cost of higher maintenance overhead as they reserve more resources.

7 Evaluation

Our evaluation examines the performance of deployment plan production for a wide range of different infrastructures. A comparison of our constraint-based approach against other *VM* placement algorithms can be found in [19]. We have simulated two network topologies, shown in Figure 3, denoted as LAN and star. In both topologies, there are groups of *PM*s supporting live migration. The two topologies differ in the way these groups are connected amongst themselves. A LAN can be created by lining up switches, each one leading to a single live migration group (bottom left of Figure 3). In the star topology (right half of Figure 3), a central switch connects N other switches and each switch leads to N live migration groups. N is the fan-out of the star topology. In our evaluation, the network's fan-out also indicates the number of *PM*s inside any live migration group in both the star and LAN topologies.

With the exception of the network connections inside a live migration group, all other network links are assigned their bandwidth randomly out of three distinct values: 100, $1,000$ and $10,000$ $Kbps$. We expect connection within a live migration group to be dedicated and of high bandwidth, thus, we assign them the

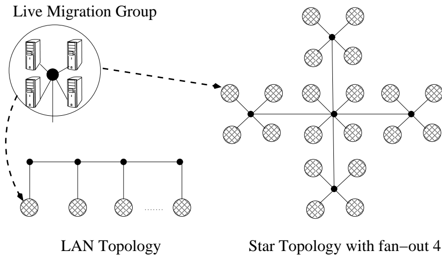


Fig. 3. Simulated network topologies.

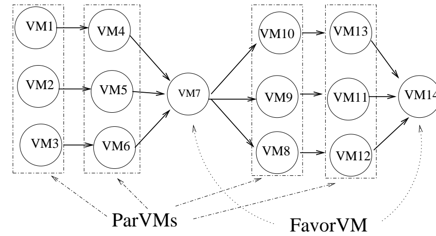


Fig. 4. LIGO inspired virtual infrastr.

maximum bandwidth of $10,000$ $Kbps$. Apart from the connectivity bandwidth, infrastructure non-homogeneity is also introduced through the characteristics of the *PM*s. Each live migration group is made of identical *PM*s belonging to one of three classes. Each of the three equally-sized classes has *PM*s of specific capacity in hosting *VM*s. Here, this capacity depends on the amount of available RAM. Therefore, classes feature *PM*s of 16, 8 and 4 *GB* of RAM respectively. In addition, we assume 20% of all live migration groups to be incompatible with the user request (e.g. due to different CPU architecture). In our simulation, whenever we set an average load percentage, we randomly deviate from it up to 10%. The load is realized as a reduction of the available resources of the infrastructure. Each *PM* features a single CPU from which *VM*s reserve a fraction. The CPU reserve fraction is explicitly stated in the user's virtual infrastructure request.

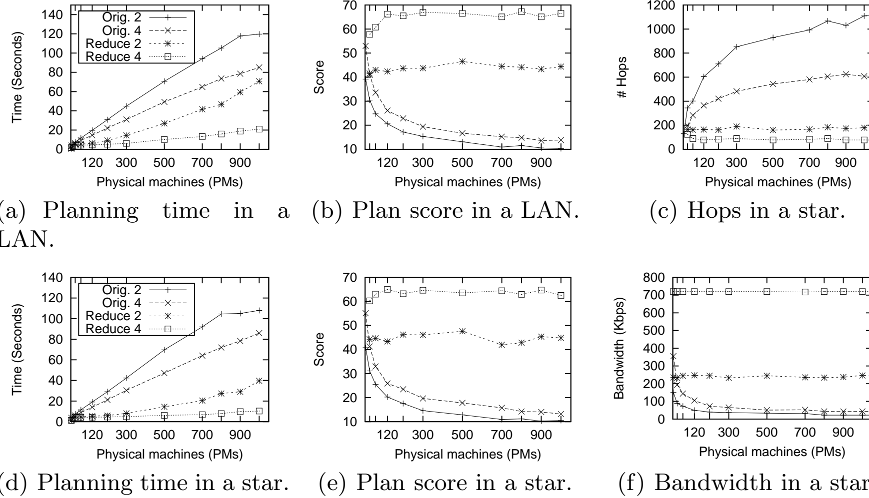
Workload Description: We have run experiments with all the workflows described in [3]. Here, we present the results of a virtual infrastructure request resembling the Laser Interferometer Gravitational Wave Observatory (LIGO) Inspiral Analysis Workflow. Figure 4 presents the *VM*s involved along with the deployment hints provided by the user. The requested virtual infrastructure is made of 14 *VM*s. Each *VM* reserves 1 *GB* of RAM and 10% of the CPU available on the hosting node. Table 3 summarizes all user hints and their weights used in this experiment. Apart from the user hints, we also employ administrative deployment hints (Table 4) so as to promote the deployment of *VM*s in neighboring *PM*s. The input parameters of Algorithm 1 are as follows: *load_threshold* is 10%, *overcommit* is set to 6, we perform 200 *same_iterations* and we set k to 100.

Table 3. User deployment hints.

Hint	Involved VMs	Weight
`ParVMs` (x4)	VMs {1,2,3},{4,5,6},{8,9,10},{11,12,13}	40.0
`FavorVM` (x2)	VM 7 & VM 14	40.0

Table 4. Administrative hints.

Hint	Description	Weight
`Reduce Groups`	Reduce the number of live migration groups	40.0
`Reduce Dist`	Reduce the network hops between migration groups	40.0



(a) Planning time in a LAN. (b) Plan score in a LAN. (c) Hops in a star.

(d) Planning time in a star. (e) Plan score in a star. (f) Bandwidth in a star.

Fig. 5. Evaluating our approach when increasing the infrastructure's size.

Scaling the Infrastructure: Figures 5(a) and 5(d) show the time required for producing a deployment plan as we increase the number of *PM*s from 30 to 1000. In configurations denoted as "Orig." we use only the *Deployment Planner* for plan production whereas in configurations denoted as "Reduce" we also employ the *Dynamic Cohort Synthesizer* to reduce the search space. For each of the two topologies, we offer two variations corresponding to different fan-out values. "Orig. 2" and "Reduce 2" correspond to configurations where the fan-out is 2 whereas for "Orig. 4" and "Reduce 4" the fan-out is 4.

To have a full understanding of the improvement, we must also examine the score of the produced plans, as computed through Equation 1. Figures 5(b) and 5(e) show the plan score for the LAN and star topology respectively. The *Deployment Planner* alone fails to produce high-scoring deployment plans as the size of the infrastructure increases. The reason for this is that the selection of *PM*s tends to disperse the virtual infrastructure across the physical infrastructure. The extra proximity cost functions that promote plans utilizing neighboring *PM*s are not enough to concentrate the user's *VM*s. The option of increasing the weight of the proximity functions proves to be ineffective in large infrastructures. High weight values for proximity administrative hints render the user provided hints insignificant playing a minor role in plan production. Instead, when we reduce the hosting infrastructure, *Deployment Planner* manages to produce high-scoring plans. The cohort selection renders the proximity cost most effective.

Network Efficiency when Scaling the Infrastructure: In non-homogeneous infrastructures, low-bandwidth network connections used by several *VM*s may quickly become a performance bottleneck. To this end, we try to concentrate *VM*s of the same virtual infrastructure in the same neighborhood of *PM*s. We use two metrics to measure network efficiency a) the "packet hop count" and

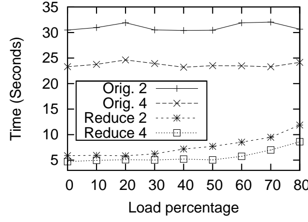


Fig. 6. Plan time under increasing load.

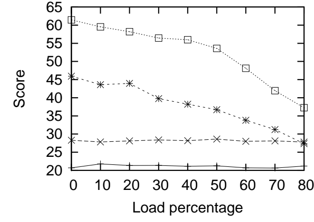


Fig. 7. Plan score under increasing load.

b) the "average minimum path bandwidth". For each pair of *VM*s there is a path of minimum length over the physical network that connects the two *PM*s where *VM*s are deployed. We are interested in two properties of this min-path, a) its length and b) the connection with the minimum bandwidth along this min-path. The length of the min-path is the number of network switches (hops) a packet must pass starting from the source *VM* until it reaches the target *VM*. The "packet hop count" metric is the summation of all min-path lengths of all possible *VM* pairs. We use "packet hop count" to estimate the network latency within the virtual infrastructure. The connection with the minimum bandwidth in a min-path also determines the maximum bandwidth of the communication among the source and target *VM*s -as the min-path is often the only path. The "average minimum path bandwidth" also takes into account all min-paths of all possible *VM* pairs and produces an average of the minimum bandwidth. We use the "average minimum path bandwidth" as an indicator of the virtual network bandwidth within the virtual infrastructure.

Figures 5(c) and 5(f) show the average number of hops and the bandwidth in plans produced in a star topology when we use the *Dynamic Cohort Synthesizer* (Reduce 2 & 4) and when we do not (Orig. 2 & 4). The respective LAN results are similar. When selecting a cohort in our approach, the search space shrinks radically and remains unchanged regardless of the set of *PM*s available in the infrastructure. The confinement of *VM*s to a small sub-infrastructure with dedicated network connections such as those within live migration groups increases the average minimum bandwidth.

Infrastructure Load: Increasing the load of the cloud reduces the capacity of the physical infrastructure to host *VM*s. We fix the number of *PM*s to 100 and gradually increase their load from 0% to 80%. Figures 6 and 7 show that high load has little impact on the performance of the *Deployment Planner* operating alone, without the help of *Dynamic Cohort Synthesizer*. This is because the number of constraints in the respective constraint satisfaction problem remain the same regardless of load. Any constraints depicting the load on a particular *PM* must be taken into consideration by the *Deployment Planner* regardless of whether the *PM*'s load is low or high. In contrast, in our two-phase approach there is a performance degradation as the load increases. Yet, the worst performance we get in an unrealistic scenario, with 80% load, is still higher than the respective "Orig." configuration.

8 Related Work

The assignment of *VMs*-to-*PMs* can be reduced to the job assignment problem should *VMs* correspond to jobs and *PMs* to processing elements. The job assignment problem has been extensively studied [15, 24], yet it is regularly revisited as application areas emerge. The placement policy in [16] exploits the tendency of *VMs* to have certain properties in common. In [23] a two level control management system is used for the placement of *VMs* to *PMs* using combinatorial and and multi-objective optimization to address potentially conflicting placement constraints. [5] reformulates the problem as a multi-unit combinatorial auction. In [17], placement constraints are treated as separate dimensions in a multi-dimensional *Knapsack* problem. User and administrative preferences expressed through constraints are also employed in [8, 9, 20, 19]. Often, as the number of constraints increases more resources are needed to solve the constraint satisfaction problem. Here, we address the issue of *VM* placement scalability through the introduction of dynamically-formed cohorts.

Network connectivity is of critical importance for data centers [2] and has influenced our experimental group formation. [14] outlines an approach that builds networks of *VM* test-beds over physical infrastructures via simulated annealing. Algorithms that improve the embedding of virtual networks to physical layouts are considered in [6]. In [13], the *VM* placement is mainly addressed from the network traffic perspective.

The use of heuristics is a common approach among systems performing load balancing in data centers. *vManage* [12] describes a low overhead solution for managing load in an infrastructure hosting *VMs*. The *VM* placement policies primarily consider properties of both platform (e.g., power management) and the virtualization layer (e.g., SLA violations). Likely instability issues are also addressed. *Sandpiper* [22] detects and monitors performance bottlenecks in a cluster hosting *VMs*. Two approaches are evaluated in the decision making mechanism that produces the *VM* migration actions: the first, termed *black-box*, remains fully OS-and-applications agnostic while the second, termed *gray-box*, exploits statistics originating from both the OS and the application-layer. Compared to both *Sandpiper* and *vManage*, our approach addresses performance bottlenecks by exploiting provided preferences and not by monitoring the operation of *VMs*. Modeling the *VM* load is deemed important in the *black-box* approach used in IaaS clouds. [10, 4, 21] classify *VM* workloads and develop metrics to model the encountered workloads in an effort to reduce *VM* migration costs.

In our approach, we do not try to predict the performance requirements of *VMs* but we trust user-provided hints to avoid performance bottlenecks. Through *PM* grouping, we reduce the search space of the constraint satisfaction problem in the second phase (*VM* to *PM* mapping). In this way, we address both potential scalability issues [8, 9, 20, 19], and balance load. Compared to [14, 6, 13], our approach is more general as it makes use of constraints expressing high-level properties. Finally, compared with other site-based approaches [1], our approach synthesizes physical infrastructures (cohorts) on-the-fly.

9 Conclusions

In this paper, we examine the problem of *VM* placement in non-homogeneous IaaS cloud environments. We propose a service realizing a two-phase approach that manages diversified resources. Compared to a heavyweight monolithic approach, our scheme can scale to several hundreds of physical nodes. In fact, the quality of the deployment plans we produce remains largely unaffected by the size of the physical infrastructure. A key concern has been the confinement of the solicited virtual infrastructure into a dynamically-adjusted set of physical nodes whose size and properties match the user requests. As a result, overall plan quality is improved since latency amongst deployed *VM*s is reduced and the average bandwidth increases dramatically. Our future work includes exploring the use of other constraint satisfaction algorithms and refining the cost and revenue functions of the *Elastic Solver* service.

Acknowledgments. We thank the reviewers for their comments. This work has been partially supported by the *EU D4Science I&II* FP7-projects.

References

1. Opennebula. http://www.opennebula.org (Nov 2010)
2. Al-Fares, M., Loukissas, A., Vahdat, A.: A Scalable, Commodity Data Center Network Architecture. In: Proc. of the ACM SIGCOMM Conference. pp. 63–74. ACM, Seattle, WA (August 2008)
3. Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.H., Vahi, K.: Characterization of Scientific Workflows. In: 3rd Workshop on Workflows in Support of Large-Scale Science. pp. 1–10. Austin, TX (Nov 2008)
4. Bobroff, N., Kochut, A., Beaty, K.: Dynamic Placement of Virtual Machines for Managing SLA Violations. In: Proc of the 10th IFIP/IEEE International Symposium on Integrated Network Management. Munich, Germany (May 2007)
5. Breitgand, D., Epstein, A.: SLA-aware Placement of Multi-Virtual Machine Elastic Services in Compute Clouds. In: IFIP/IEEE International Symposium on Integrated Network Managemen. Dublin, Ireland (May 2011)
6. Chowdhury, N., Rahman, M., Boutaba, R.: Virtual Network Embedding with Coordinated Node and Link Mapping. In: Proc. of IEEE INFOCOM. Rio de Janeiro, Brazil (April 2009)
7. Cierniak, M., Zaki, M.J., Li, W.: Compile-Time Scheduling Algorithms for a Heterogeneous Network of Workstations. The Computer Journal 40(6), 356–372 (1997)
8. Hermenier, F., Lorca, X., Menaud, J., Muller, G., Lawall, J.: Entropy: a Consolidation Manager for Clusters. In: Proc. of the 2009 ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. Washington, DC (March 2009)
9. Hyser, C., McKee, B., Gardner, R., Watson, B.J.: Autonomic Virtual Machine Placement in the Data Center. HP Laboratories HPL-2007-189 (2008)
10. Khanna, G., Beaty, K., Kar, G., Kochut, A.: Application Performance Management in Virtualized Server Environments. In: Proc of the 10th IEEE/IFIP Network Operations and Management Symposium. Vancouver, Canada (April 2006)
11. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by Simulated Annealing. Science 220, 671–680 (May 1983)

12. Kumar, S., Talwar, V., Kumar, V., Ranganathan, P., Schwan, K.: vManage: Loosely Coupled Platform and Virtualization Management in Data Centers. In: Proceedings of the 6th international conference on Autonomic computing. pp. 127–136. ACM, Barcelona, Spain (June 2009)
13. Meng, X., Pappas, V., Zhang, L.: Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In: Proceedings of IEEE INFOCOM. San Diego, CA, USA (March 2010)
14. Ricci, R., Alfeld, C., Lepreau, J.: A Solver for the Network Testbed Mapping Problem. SIGCOMM Computer Communications Review 33(2), 65–81 (April 2003)
15. Rotithor, H.: Taxonomy of dynamic task scheduling schemes in distributed computing systems. Computers and Digital Techniques, IEEE Proceedings (Jan 1994)
16. Sindelar, M., Sitaraman, R.K., Shenoy, P.: Sharing-Aware Algorithms for Virtual Machine Colocation. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures. San Jose, California, USA (June 2011)
17. Singh, A., Korupolu, M., Mohapatra, D.: Server-Storage Virtualization: Integration and Load Balancing in Data Centers. In: Proc. of the 2008 ACM/IEEE conference on Supercomputing (SC'08). pp. 53:1–53:12. Austin, TX (2008)
18. Tsakalozos, K., Kllapi, H., Sitaridi, E., Roussopoulos, M., Paparas, D., Delis, A.: Flexible Use of Cloud Resources through Profit Maximization and Price Discrimination. In: Proc. of the 27th IEEE Int. Conf. on Data Engineering (ICDE'11). Hannover, Germany (Apr 2011)
19. Tsakalozos, K., Roussopoulos, M., Floros, V., Delis, A.: Nefeli: Hint-based Execution of Workloads in Clouds . In: Proc. of the 30th IEEE Int. Conf. on Distributed Computing Sytems (ICDCS'10). Genoa, Italy (June 2010)
20. Wang, X., Lan, D., Wang, G., Fang, X., Ye, M., Chen, Y., Q.B. Wang, Q.: Appliance-Based Autonomic Provisioning Framework for Virtualized Outsourcing Data Center. In: Proc. of the 4th Int. Conf. on Autonomic Computing. p. 29. Washington, DC (2007)
21. Weng, C., Li, M., Wang, Z., Lu, X.: Automatic Performance Tuning for the Virtualized Cluster System. In: Proc. of the 29th IEEE International Conference on Distributed Computing Systems. Montreal, Canada (June 2009)
22. Wood, T., Shenoy, P., Venkataramani, A., Yousif, M.: Black-box and Gray-box Strategies for Virtual Machine Migration. In: Proc of the 4th USENIX Symposium on Networked Systems Design and Implementation. Cambridge, MA (2007)
23. Xu, J., Fortes, J.A.B.: Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments. In: Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing. Hangshou, PR of China (Dec 2010)
24. Yeo, C.S., Buyya, R.: A taxonomy of market-based resource management systems for utility-driven cluster computing. Softw. Pract. Exper. 36(13) (November 2006)