A logic programming approach to VM placement

Remo Andreoli[1][a], Stefano Forti[2][b], Luigi Pannocchi[2][c],
Tommaso Cucinotta[1][d] and Antonio Brogi[2][e]

[1]*Sant'Anna School of Advanced Studies, Pisa, Italy*

[2]*Department of Computer Science, University of Pisa, Pisa, Italy*

Corresponding authors: remo.andreoli@santannapisa.it, stefano.forti@unipi.it

Keywords: VM Placement, Cloud Computing, Resource Management, Declarative Reasoning, Optimization

Abstract: Placing virtual machines so to minimize the number of used physical hosts is an utterly important problem in cloud computing and next-generation virtualized networks. This article proposes a declarative reasoning methodology, and its open-source prototype, including four heuristic strategies to tackle this problem. Our proposal is extensively assessed over real data from an industrial case study and compared to state-of-the-art approaches, both in terms of execution times and solution optimality. As a result, our declarative approach determines placements that are only 6% far from optimal, outperforming a state-of-the-art genetic algorithm in terms of execution times, and a first-fit search for optimality of found placements. Last, its pipelining with a mathematical programming solution improves execution times of the latter by one order of magnitude on average, compared to using a genetic algorithm as a primer.

1 Introduction

Recently, the problem of Virtual Machine (VM) placement gained renewed interest in the field of telecommunications (Attaoui et al., 2023; Cucinotta et al., 2022) – with the advent of Network Function Virtualization (NFV) (Cai et al., 2023), where Virtual Network Functions (VNFs) are deployed in a private cloud infrastructure of a network operator – as well as in cloud-edge settings (Sonkoly et al., 2021) – having to deal with limited resource capacity of edge hosts. These paradigms require flexible management of physical resources, along with the ability to promptly reconfigure VM allocation in response to changes in the network state or VM requirements.

In the following, we take the viewpoint of a Telco provider having to deploy VNFs as a set of VMs within its NFV infrastructure. Affinity constraints may be used when low-latency communications are needed, so to place VMs onto the same host, and anti-affinity ones when service availability is needed, by placing VM replicas onto different hosts. The goal of the Telco provider is to minimize the number of used hosts to reduce operational costs.

Solutions to this type of problem usually employ a mixed-integer linear programming (MILP) approach and state-of-the-art (SOTA) solvers to determine an optimal solution (Filho et al., 2018). However, optimality comes at the price of high execution times and possibly cumbersome encodings of non-numerical constraints (e.g. affinities or anti-affinities), especially in the presence of large infrastructures. More recently, declarative approaches have been proposed to heuristically solve application placement problems in cloud-edge landscapes (Forti et al., 2022). Declarative approaches are more concise than MILP to formulate, easier to extend with new requirements, and faster on average at determining eligible (yet suboptimal) solutions, see e.g. (Massa et al., 2023a).

In this article, we pursue the reconciliation of declarative approaches with MILP focusing on the problem of placing VMs onto hosts, providing the following contributions: i) an open-source declarative Prolog prototype[1], declPacker, implementing four *heuristic strategies* to determine VM placements accounting for hardware, network, and (anti-) affinity requirements, while reducing the number of hosts; ii) the *integration* of our declarative prototype as a way

[a] https://orcid.org/0000-0002-3268-4289
[b] https://orcid.org/0000-0002-4159-8761
[c] https://orcid.org/0000-0002-6250-4939
[d] https://orcid.org/0000-0002-0362-0657
[e] https://orcid.org/0000-0003-2048-2468

[1]Available at: https://retis.santannapisa.it/~tommaso/papers/closer24.php

to determine an upper-bound on the number of hosts into a SOTA, MILP solution which employs a genetic algorithm as its default upper-bounding strategy; and iii) the *assessment* of our proposal over real, increasingly complex, problem instances from a Vodafone industrial use (released in (Cucinotta et al., 2022)) comparing our approach with the SOTA solution w.r.t. execution times and optimality of found placements.

2 Problem statement

Consider a network provider that needs to deploy many VNFs in the form of multiple VMs and minimize the number of hosts. A VNF team is in charge of sizing the virtualized infrastructure requirements to support a maximum traffic volume (e.g., connected users, requests per minute) and to meet the desired end-to-end latency of deployed service chains. The capacity planning problem is determined by the number of interconnected VMs and their characteristics. The virtualized infrastructure needs to be deployed onto physical hosts to be provisioned. Providers usually buy hosts in large batches with identical hardware specifications, i.e., with the same "host blueprint", which fits their needs. With such hardware, the deployment plan specifies how many VMs are needed to deploy each VNF component, with a given VM specification (e.g., CPU cores, RAM, network bandwidth), and what are their associated affinity and anti-affinity constraints, useful to meet performance (i.e., minimizing experienced latencies) and reliability (i.e., through VM replicas across distinct hosts) requirements, respectively.

As an example, consider a lifelike motivating scenario where we have to deploy 6 VMs onto hosts accommodating 44 virtual CPU cores, 420 GB of RAM, and a network throughput of 15000 Mbps each, net of the resources used for management purposes. We consider only CPU, RAM, and network requirements, similarly to (Cucinotta et al., 2022), as we use the same open data-set for the evaluation in Sect. 4, albeit additional resources can be considered as well.

In this example, the deployment plan consists of two VMs for each of the following types: i) *small*, requiring 10 vCPUs, 50 GB of RAM, and a throughput of 2500 Mbps; ii) *medium*, requiring 15 vCPUs, 100 GB of RAM, and a throughput of 5000 Mbps; and iii) *large*, requiring 30 vCPUs, 200 GB of RAM, and a throughput of 10000 Mbps.

The provider preferably requires support for latency-sensitive communications. Hence, it configures an affinity constraint between the *small* VMs (`vm1, vm2`), so that they possibly run on the same host. We consider affinity constraints as *soft*, i.e., optional. To improve the availability of the services offered by the *medium* VMs (`vm3, vm4`), the provider imposes an anti-affinity constraint among them – i.e., that they run on different hosts. Anti-affinity constraints are considered as *hard*, assuming that the host blueprint is large enough to accommodate at least one instance of the *large* VM (`vm5, vm6`).

Affinity and anti-affinity requirements can also involve sets of VMs for akin reasons, in such cases they are called cross-affinity and cross-anti-affinity constraints. Again, cross-affinity constraints are considered soft, while cross-anti-affinity constraints are treated as hard. In our scenario, the infrastructure provider sets a cross-affinity constraint within *small* and *medium* VMs (to reduce latency among those), and a cross-anti-affinity constraint among *medium* and *large* VMs (to enhance service availability).

Overall, we tackle the following problem:

Given a set of VMs and a host blueprint, determine a valid placement of those VMs onto a set of hosts complying with the blueprint, such that:

(*a*) *it meets all VM requirements in terms of CPU, RAM and network throughput,*

(*b*) *it meets all soft (cross-)affinity constraints and hard (cross-)anti-affinity constraints, and*

(*c*) *it minimizes the number of spanned hosts.*

Figure 1 shows an optimal solution to the above problem instance, featuring 4 hosts. Note that `vm1` and `vm2` meet their affinity constraint, and that `vm3` is deployed along with them to (partially) comply with the cross-affinity constraints between *medium* and *small* VMs. The anti-affinity constraint between the *medium* VMs is obtained by placing `vm4` onto a host by itself. Last, the placement complies with the cross-anti-affinity constraint between *medium* and *large* by placing `vm5` and `vm6` onto their hosts.

All these considered, optimally solving the depicted VM placement problem incurs exploring a combinatorial search space, hence worst-case exptime complexity, typical of bin-packing (NP-hard) problems. In the next section, we will present a declarative solution to such a problem and showcase it over the illustrated scenario.

3 Methodology

In this section, we discuss and illustrate the feature of our declarative methodology and associated open-

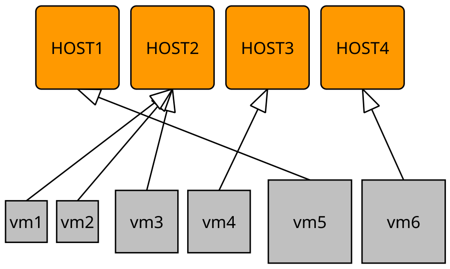


Figure 1: Placement solution for the motivating scenario.

source Prolog[2] prototype declPacker.

Knowledge representation We illustrate a set of simple fact declarations used to model and solve instances of the considered problem. First, the blueprint of the physical hosts onto which VMs are to be placed is specified through a single fact of the form

```
hostBluePrint((CPUCap, MemCap, NetCap)).
```

where `CPUCap`, `MemCap` and `NetCap` denote the CPU, RAM, and network throughput available at each physical server in terms of million instructions per second (MIPS), Megabytes, and Megabit/s, respectively.

Then, we denote each VM to be placed as

```
vm(Id, (CPUReq, MemReq, NetReq)).
```

where `Id` is a unique VM identifier, and `CPUReq`, `MemReq`, and `NetReq` are the CPU, RAM, and network throughput requirements for the VM at hand. We assume VM requirements rely on the same unit of measurement used to specify host capabilities.

Affinity and anti-affinity constraints over sets of VMs are specified as facts of the form

```
affinity(VMList).
antiaffinity(VMList).
```

where `VMList` is the list of identifiers of the VMs involved in the specified affinity or anti-affinity constraint, i.e. that are to be placed onto the same or different hosts, respectively.

Similarly, cross-affinity and cross-anti-affinity constraints are denoted as facts like

```
xAffinity(VMList1, VMList2).
xAntiaffinity(VMList1, VMList2).
```

where `VMList1` and `VMList2` are the lists of identifiers involved in the cross-constraint.

[2]A Prolog program is a finite set of *clauses* of the form: `a :- b1, ... , bn.` stating that a holds when b1 $\wedge \cdots \wedge$ bn holds, where n$\geq$0 and a, b1, ..., bn are atomic literals. Clauses with an empty condition are also called *facts*. Prolog variables begin with upper-case letters, lists are denoted by square brackets, and negation by \+.

Overview Fig. 2 below lists the main predicate `declPacker/3` of our prototype which exploits the knowledge representation discussed above. Given a placement `Strategy` among the available ones (i.e., `optimal` and `heuristic`), `declPacker/3` determines an eligible `Placement` for the VMs declared in the knowledge base onto `NumberOfHosts` hosts with capacity as declared in `hostBlueprint/1`. Particularly, `declPacker/3`: (1) it collects all `VMs` to be placed into a list through predicate `toPlace/1` (lines 3, 5), which simply finds all `vm/2` facts declared in the knowledge base; (2) it splits the cross-constraints and ignore invalid affinities constraints so to comply with the `hostBlueprint/1` through predicates `splitXconstraints` and `cleanupInvalidAffinities` (line 2); and (3) it determines an eligible `Placement` for those `VMs` by applying the specified search `Strategy` and computing the associated `NumberOfHosts` with predicate `declPacker/4` (line 4). Steps (2) and (3) constitute the *preprocessing* and the *placement* step of our solution. As a result of the preprocessing step, the declarative placement strategy of declPacker (described in the following) can work by only exploiting `hostBlueprint/1`, `vm/2`, and simple `affinity/2` and `antiAffinity/2` constraints.

```
1  declPacker(Strategy, Placement, NumberOfHosts) :-
2      toPlace(VMs),
3      splitXconstraints(), cleanupInvalidAffinities(),
4      declPacker(Strategy, VMs, Placement, NumberOfHosts).

5  toPlace([VM|List]) :- findall(V, vm(V,_), [VM|List]).
```

Figure 2: Bird's-eye view of our prototype.

Placement step A VM placement is represented as a list of lists like

```
[ [VM11, VM21, ..., VMP1], ... , [VM1K, VM2K, ..., VMRK] ]
```

where the `J`th list `[VM1J, VM2J, ..., VMSJ]` store the identifiers of the `S` VMs placed onto the host `J`. For instance, the above placement depicts `K` hosts and places `P` VMs onto host `1`, `Q` VMs onto host `2`, and so on, up to `R` VMs onto host `K`.

The placement step (3) is described in Fig. 3. Predicate `place/3` (lines 7–10) inputs from `declPacker/4` a non-empty `[VM|List]` of VMs to be placed and a previously built eligible placement `OldPlacement` (initially empty, line 7). At each recursive step, `place/3` places a new `VM` by relying on predicate `placeVM/4` (line 8) to extend `OldPlacement` into a new eligible `TmpPlacement`, which includes an eligible placement for `VM`. It then recurs (line 9), and stops at an empty list of VMs to be placed (line 10).

Predicate `placeVM/4` (line 11–20) recursively scans a previous eligible placement `[H|Hs]` (i.e., a list of hosts) onto which the current `VM` could be placed. It relies on predicates `fits/2`, `affinityOK/3` and `antiAffinityOK/2` to determine whether `VM` can be placed on a considered host `H`. Particularly, `placeVM/4` distinguishes the following four cases:
(1) `VM` is placed onto host `H` meeting all its CPU, RAM and network requirements, and its affinity and anti-affinity constraints, then `VM` is added to the host `H` (line 12), which in turn is appended to the `NewPlacement`, between already scanned hosts `Pre` and hosts `Hs` not yet considered (line 13). Recursion ends as we have an eligible placement for `VM`.
(2) `VM` is placed onto host `H` meeting all its CPU, RAM, and network requirements and anti-affinity constraints, but ignoring its (soft) affinity constraints (line 15), then `VM` is added to the host `H` as in the previous case (line 16). Recursion ends as we have an eligible placement for `VM`.
(3) `VM` is not placed onto host `H` and recursion goes on to the next candidate host for supporting `VM`, by including `H` in the list `Pre` of visited hosts (line 18).
(4) `VM` is placed onto a new host as the list of candidate hosts is finally empty (line 19–20). This last case reasonably assumes that a host can at least support the largest VM and avoids checking predicate `fits/2`.

As mentioned above, predicates `fits/2`, `affinityOK/3`, and `antiAffinityOK/2` are used to check the eligibility of the new placement. We now briefly comment on their functioning.

Predicate `fits/2` (lines 21–26) checks that `VM` can be placed onto host `H` by meeting its CPU, RAM, and network throughput requirements, also considering other VMs previously allocated at `H`. It does so by first retrieving the requirements (`VMCPU, VMMEM, VMNET`) of `VM` and the corresponding capabilities (`HCPU, HMEM, HNET`) of the host blueprint (line 22). Through predicate `allocatedResources/2`, it retrieves the amount of resources (`AllocatedCPU, AllocatedMEM, AllocatedNET`) allocated to VMs already placed onto `H` (line 23). Last, it checks that adding the requirements of `VM` to the previous allocation does not exceed the capacity of `H` for what concerns CPU, RAM, and throughput (lines 24–26).

Predicate `affinityOk/3` retrieves all affinity constraints involving `VM` and recursively checks that there exists no VM `V` $\neq$ `VM` within an affinity constraint with `VM` that is placed onto a host `H2`, such that `H2` $\neq$ `H`. Predicate `antiAffinityOk/2` analogously retrieves all anti-affinity constraints involving `VM` to check that no other VM `V` in anti-affinity with `VM` is placed onto `H`.

Example. By repeatedly querying predicate

```
place([vm1,vm2,vm3,vm4,vm5,vm6], [], P).
```

over the knowledge base representing the motivating scenario of Sect. 2 returns three distinct eligible placements of up to five hosts. Namely:

```
P = [[vm5], [vm3, vm2, vm1], [vm4], [vm6]];
P = [[vm5], [vm4, vm2, vm1], [vm3], [vm6]];
P = [[vm5], [vm3], [vm2, vm1], [vm4], [vm6]].
```

Note that the first placement corresponds to the one sketched in Fig. 1, the second one is identical to the first up to swapping `vm3` and `vm4`, the last one exploits one extra host by placing `vm3` and `vm4` onto dedicated hosts – ignoring the soft cross-affinity constraint with the *small* VMs `vm1` and `vm2`. □

Placement strategies The solution that we have described applies an uninformed first-fit strategy to determine an eligible placement, as it explores the list of VMs in the order they appear in the knowledge base. By sorting the list of VMs to be placed according to some strategy, it is possible to apply some heuristics to our search for an eligible placement.

declPacker provides other three functioning modes out-of-the-box, namely:

- `optimal`, which retrieves all eligible placements along with their number of hosts, and sorts them by increasing the number of hosts to return an optimal placement relying on the minimum number of hosts, alas incurring in exp-time complexity,
- `heuristic`, which sorts the `VMList` by first ranking them according to a `Rank` and then sorting them according to a specified `Order`, namely `ascending` or `descending`. We currently support two ranking methods: `resourceDemand`, which ranks the VMs and sorts them according to the following function

$$R(VM) = \frac{VMCPU}{HCPU} + \frac{VMMEM}{HMEM} + \frac{VMNET}{HCPU} \quad (1)$$

  where the resource requirements are scaled via min-max normalization; `numberOfConstraints`, which ranks the VMs according to the number of occurrences in (cross-)affinity e (cross-)anti-affinity rules. The search space is explored according to the `SortedVMs` list.

Example. Now, querying predicate `declPacker` in `optimal` mode returns the following placement

```
P = [[vm5],[vm3,vm2,vm1],[vm4],[vm6]].
```

which corresponds to the optimal one of Fig. 1. Similarly, querying the `heuristic` version sorting by descending number of constraints, we obtain

```
declPacker(firstfit, VMList, Placement, NumberOfHosts) :- place(VMList, [], Placement), length(Placement, NumberOfHosts).

place([VM|List], OldPlacement, NewPlacement) :-
    placeVM(VM, OldPlacement, [], TmpPlacement),
    place(List, TmpPlacement, NewPlacement).
place([], Placement, Placement).

placeVM(VM, [H|Hs], Pre, NewPlacement) :-         % VM is placed on existing H with (anti-)affinity constraints
    fits(VM, H), affinityOk(VM, H, Pre), antiAffinityOk(VM, H),
    append(Pre, [[VM|H]], TmpP), append(TmpP, Hs, NewPlacement).
placeVM(VM, [H|Hs], Pre, NewPlacement) :-         % VM is placed on existing H without affinity constraints
    fits(VM, H), antiAffinityOk(VM, H),
    append(Pre, [[VM|H]], TmpP), append(TmpP, Hs, NewPlacement).
placeVM(VM, [H|Hs], Pre, NewPlacement) :-         % VM is not placed on existing H, try next one in list
    placeVM(VM, Hs, [H|Pre], NewPlacement).
placeVM(VM, [], Pre, NewPlacement) :-             % VM is placed on new H (Hp: host blueprint supports at least the largest VM)
    affinityOk(VM, [VM], Pre), append(Pre, [[VM]], NewPlacement).

fits(VM, H) :-
    vm(VM, (VMCPU, VMMem, VMNet)), hostBlueprint((HCPU, HMEM, HNET)),
    allocatedResources(H, (AllocatedCPU, AllocatedMem, AllocatedNet)),
    AllocatedCPU + VMCPU =< HCPU,
    AllocatedMem + VMMem =< HMEM,
    AllocatedNet + VMNet =< HNET.
```

Figure 3: Declarative placement step.

```
P = [[vm2,vm6],[vm1,vm4],[vm3],[vm5]].
```

which meets all hardware requirements, ignores the soft (cross-)affinity requirements over VMs, and meets all (cross-)/anti-affinity requirements. □

4 Experimental assessment

This section presents an experimental assessment of `declPacker` and a comparison of our approach to the SOTA solution in (Cucinotta et al., 2022) in terms of the suggested number of hosts and solve time. Their work introduces three approaches to the optimal VM placement problem: i) a standard MILP-based formulation that expresses the problem stated in Sect. 2 as decision variables and mathematical constraints, ii) a simple First-Fit (FF) heuristic that allocates the VMs one after the other in the first feasible host; and iii) a genetic algorithm (GA) meta-heuristic.

As pointed out by the authors, a traditional MILP-based solvers deal with low-level mathematical calculations with no awareness of the high-level description of the problem under scrutiny. They provide guarantees regarding the optimality of the solution but are significantly slower than other approaches due to the NP-hardness of the considered problem. A solution to avoid unbearably long execution times is to set an upper bound to the number of hosts to the MILP formulation obtained through a (fast-in-practice) heuristic approach, e.g. FF or GA.

In this regard, we replicated the experimental environment[3] presented in (Cucinotta et al., 2022) and expanded it with experimental results using declPacker. The results for each of their solvers were provided to us by the authors themselves. The placement problems consider the same homogeneous physical infrastructure as described in their paper, as well as in our motivating example in Sect. 2. Namely, each host has the following specification: 44 CPUs, 420 GB of RAM and 15000 Mbit/s of network bandwidth.

Every approach is assessed and compared using the open data set published in (Cucinotta et al., 2022), which proposes a set of 152 placement problems tackled by Vodafone in the optimization of its capacity planning decisions. The majority of problems require the placement of fewer than 100 VMs. Only 5 complex problems require more than 1000 VMs.

Assessing declPacker As a preliminary step, we selected the best placement strategy of declPacker to be compared with the SOTA approaches. As mentioned, the `optimal` strategy incurs in very high solve times,

[3]The MILP-based examples have been solved using ILOG CPLEX version 12.9. Our `declPacker` is based on SWI-Prolog version 9.0.4. FF and GA are written in Python 3 with numpy module. All the experiments have been performed on a dedicated server equipped with an Intel(R) Xeon(R) CPU E5-2640 v4 @2.40GHz and 64 GB of RAM.

therefore it is discarded from this preliminary comparison. Fig. 4 shows the outcomes of declPacker with varying `Rank` and `Order` heuristic parameter. Notice that there is no clear positive correlation between the number of hosts and the problem size because the latter is expressed in terms of the number of VMs only. The number of (anti-)affinity constraints is not considered in the sorting, but it is part of the complexity.

There are a total of 4 heuristics: i) "Most demanding VM first", which corresponds to ranking method `resourceDemand` and order `descending`; ii) "Least demanding VM first", rank `resourceDemand` and order `ascending`; iii) "Most constrained VM first", rank `numberOfConstraints` and order `descending`; and finally iv) "Least constrained VM first", rank `numberOfConstraints` and order `ascending`. Solve times are not considered in this step, as they are mostly equivalent and subject to experimental errors. The results show that ascending order (i.e., "Least `Rank`-ed VM first") is not a good evaluation criterion for the open data set under analysis. Secondly, the "most constrained VM first" heuristic turned out to be the best one. Indeed the placement problems in the open data feature VMs with similar hardware requirements, making the `ResourceDemand` ranking method not as good as `numberOfConstraint`. Each of the above strategy, suggests on average 29, 32, 28, and 31 hosts.

Since the FF approach presented by (Cucinotta et al., 2022) belongs to the same family of heuristics, it is compared to declPacker in this preliminary step. The solve time is equivalent in every placement scenario, but FF suggests 3 additional hosts on average and 52 maximum, compared to our best strategy. This is because declPacker allows for more flexibility in terms of ranking and order, whereas FF "blindly" places the VMs without considering the characteristics of the placement problems. Although the differences may seem negligible for all heuristic approaches, additional hosts translate to much longer solve times when aiming for optimality. Given the outcomes of this section, the rest of the experimental evaluation will only consider heuristic declPacker with rank `numberOfConstraints` and order `descending`.

declPacker vs a Genetic Algorithm This section presents a comparison between the best strategy offered by declPacker (for the use case under analysis) and the GA approach described in (Cucinotta et al., 2022). The latter is run with a population of 50 candidates and a max of 25 iterations. Such values have been determined by the authors themselves as a good exploration/exploitation/solving-time trade-off for the experimental assessment.

Fig. 5a depicts the suggested number of hosts and

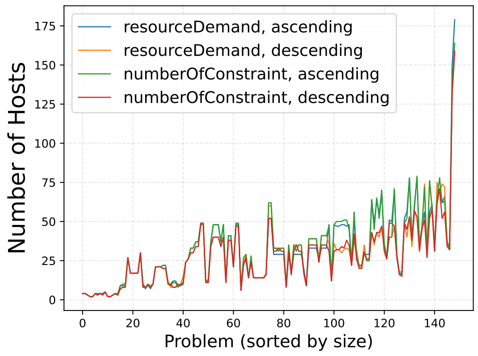
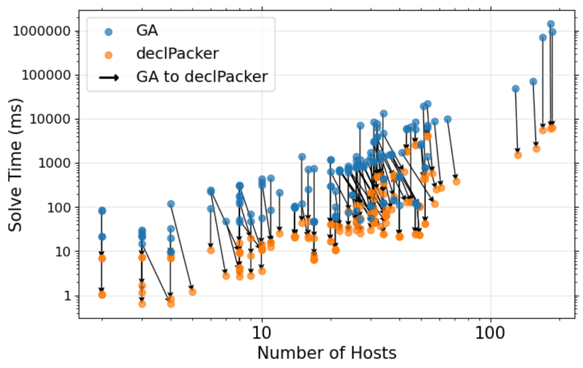


Figure 4: Comparison of declPacker strategies.

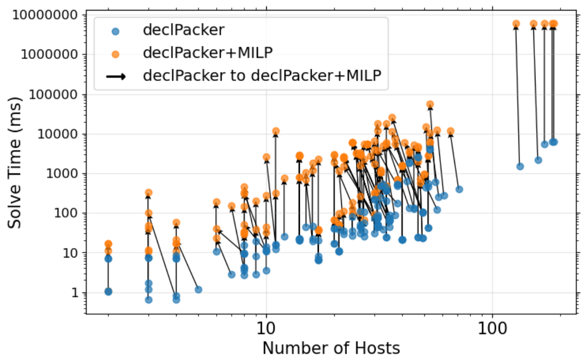
the corresponding solve time for every placement scenario, outlining the 1-to-1 comparison between the two methodologies. Arrows represent the advantage, or disadvantage (depending on the arrow orientation), of declPacker over GA in the two evaluation criteria. For instance, declPacker demonstrates better solve time, outperforming GA 99.95% of times, and this is highlighted by the significant presence of downward arrows in the plot. Our approach performs worse on a handful of placement problems where the solve time is under 1 second, making the time difference negligible. Oblique arrows express a trade-off between declPacker and GA over one of the two evaluation criteria. For instance, declPacker suggests the same number of hosts 60% of times, but a worse upper-bound to the number of hosts 40% of times with respect to the GA. The latter is depicted by the arrows pointing right side. Anyhow, the upper-bound difference is no more than 2 additional hosts on average, with a worst-case of 16 hosts. In comparison, FF suggests 5 additional hosts on average and a worst-case of 53 hosts compared to GA.

Regarding solve times, declPacker saves 22s on average, with a maximum time save of 1400s. Note that this is biased due to the large solve time difference for 5 problems, which are significantly more complex than the rest (outliers on the top-right region of the plots). Another important aspect is the variability in solve time: GA experiences greater instabilities due to the stochastic nature of evolutionary algorithms. For instance, the placement problem with the longest solve time returns a standard deviation of over 200s in 20 reruns for GA, whereas declPacker is much more stable with a standard deviation below 1s.

Achieving an optimal placement We now compare the host upper-bounding capability of our approach with SOTA approaches. The number of hosts as from the previous section has been integrated in the MILP formulation as a new upper-bound to the num-



(a) declPacker vs GA



(b) declPacker vs declPacker+MILP

Figure 5: Problem-to-problem comparison of the suggested upper-bound to the number of hosts and related solve times.

ber of hosts, thus allowing for faster convergence with a reduced search space. The approaches to be compared are declPacker-upper-bounded MILP (or simply declPacker+MILP), FF+MILP, and GA+MILP. We removed from the analysis the 5 complex problems, as they did not return the optimal solution within a time limit of 6000s. In (Cucinotta et al., 2022), a 600s time limit was used for analogous reasons.

In our experiments, declPacker+MILP performs better than the SOTA approaches in combination with MILP. GA+MILP reaches optimality faster than the rest (2s on average), as it is the approach that suggests the best upper-bound for optimality. However, it is burdened by the execution time of the upper-bounding phase which is 22s on average (and highly variable). FF+MILP reaches optimality in 7s on average, and it suggests an upper-bound in less than 1s. declPacker+MILP reaches optimality in 3s on average, while also showing a pre-processing step as fast as FF.

If optimality is not the main goal, using declPacker by itself may be a fair option, as shown in Fig. 5b, showing the near-optimality of declPacker. In particular, the latter saves 11s of solve time on average compared to running MILP (upper-bounded with declPacker), while suggesting 2 additional hosts on average. This is a 6% gap from the optimal solution.

5 Related work

Prior work tackled the problem considered in this article (Masdari and Zangakani, 2020). Alongside the goal of minimizing the number of hosts needed for a deployment, other objectives can be mixed, such as keeping the workload balanced among the machines (Hieu et al., 2014; Gupta et al., 2013). Some authors focused on NFV (Alicherry and Lakshman, 2012; Cucinotta et al., 2022; Ma et al., 2015), considering network-awareness in the problem formulation.

Classical approaches rely on a MILP formulation and leverage available solvers (Cucinotta et al., 2014; Cucinotta et al., 2022; Saber et al., 2015). These works aim at formal optimality guarantees, but suffer of some practical feasibility issues due to their possibly long execution times (Andreoli et al., 2023). Other approaches based on *ad-hoc* heuristics (Oechsner and Ripke, 2015) are often very efficient in solving large problems, but they cannot guarantee optimality. Neural Networks and ML techniques have also been proposed (Long et al., 2020; Khoshkholghi et al., 2019; Cucinotta et al., 2022), which are more adaptable by having training and solving exploration phases, but they are not very fast on retraining.

As mentioned in the introduction, recently, Prolog-based approaches have been proposed to solve akin application placement problems, considering different orthogonal aspects, e.g. data-awareness (Massa et al., 2022), environmental sustainability (Forti and Brogi, 2022), or intent satisfaction (Massa et al., 2023b). Besides solving a different problem, such proposals are usually limited to solving the decision version of those placement problems without target optimisations. To the best of our knowledge, only (Massa et al., 2023a) has previously tried to combine a declarative approach with MILP resolution.

6 Concluding remarks

In this article we proposed declarative heuristic solution to a VM placement problem, made available as an open-source Prolog prototype called declPacker.

Experimental results over real data from an industrial dataset show that declPacker can be used as a stand-alone reasoner for quick decision-making, saving on average 99% of execution times and determining solutions that are only 6% far from optimal w.r.t. MILP-based solutions. For complex scenarios, its combined use with MILP improves their solve time by 10× compared to a genetic algorithm upper-bounding strategy – always allowing the identification of an optimal placement.

Future work includes extending declPacker with further constraints, mixing it with neural network solutions in the spirit of neuro-symbolic approaches, and assessing it in real testbed settings.

ACKNOWLEDGEMENTS

Work partly funded by projects: *Energy-aware management of software applications in Cloud-IoT ecosystems* (RIC2021_PON_A18) funded over ESF REACT-EU resources by the *Italian Ministry of University and Research* through *PON Ricerca e Innovazione 2014–20*; and *hOlistic Sustainable Management of distributed softWARE systems (OSMWARE)*, PRA_2022_64, funded by the University of Pisa.

REFERENCES

Alicherry, M. and Lakshman, T. (2012). Network aware resource allocation in distributed clouds. In *2012 Proceedings IEEE INFOCOM*, pages 963–971.

Andreoli, R., Gustafsson, H., Abeni, L., Mini, R., and Cucinotta, T. (2023). Optimal Deployment of Cloud-native Applications with Fault-Tolerance and Time-Critical End-to-End Constraints. In *Proceedings of the 16th IEEE/ACM International Conference on Cloud Computing*, Taormina (Messina), Italy.

Attaoui, W., Sabir, E., Elbiaze, H., and Guizani, M. (2023). VNF and CNF Placement in 5G: Recent Advances and Future Trends. *IEEE Transactions on Network and Service Management*, pages 1–1.

Cai, X., Deng, H., Deng, L., Elsawaf, A., Gao, S., Nicolas, A. M. D., Nakajima, Y., Pieczerak, J., Triay, J., Wang, X., Xie, B., and Zafar, H. (2023). Evolving NFV towards the next decade, ETSI White Paper No. 54.

Cucinotta, T., Lugones, D., Cherubini, D., and Jul, E. (2014). Data Centre Optimisation Enhanced by Software Defined Networking. In *IEEE 7th International Conference on Cloud Computing*, pages 136–143.

Cucinotta, T., Pannocchi, L., Galli, F., Fichera, S., Lahiri, S., and Artale, A. (2022). Optimum VM placement for NFV infrastructures. In *IEEE International Conference on Cloud Engineering*, pages 205–212. IEEE.

Filho, M. C. S., Monteiro, C. C., Inácio, P. R. M., and Freire, M. M. (2018). Approaches for optimizing virtual machine placement and migration in cloud environments: A survey. *J. Parallel Distributed Comput.*, 111:222–250.

Forti, S., Bisicchia, G., and Brogi, A. (2022). Declarative continuous reasoning in the cloud-iot continuum. *J. Log. Comput.*, 32(2):206–232.

Forti, S. and Brogi, A. (2022). Green application placement in the cloud-iot continuum. In *Practical Aspects of Declarative Languages - 24th International Symposium, PADL 2022*, volume 13165 of *Lecture Notes in Computer Science*, pages 208–217. Springer.

Gupta, A., Kale, L. V., Milojicic, D., Faraboschi, P., and Balle, S. M. (2013). Hpc-aware vm placement in infrastructure clouds. In *Proceedings of the IEEE International Conference on Cloud Engineering*, pages 11–20.

Hieu, N. T., Francesco, M. D., and Jääski, A. Y. (2014). A virtual machine placement algorithm for balanced resource utilization in cloud data centers. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing*, page 474–481. IEEE.

Khoshkholghi, M. A., Taheri, J., Bhamare, D., and Kassler, A. (2019). Optimized service chain placement using genetic algorithm. In *IEEE Conference on Network Softwarization*, pages 472–479.

Long, S., Li, Z., Xing, Y., Tian, S., Li, D., and Yu, R. (2020). A reinforcement learning-based virtual machine placement strategy in cloud data centers. In *IEEE International Conference on High Performance Computing and Communications*, pages 223–230.

Ma, W., Medina, C., and Pan, D. (2015). Traffic-Aware Placement of NFV Middleboxes. In *2015 IEEE Global Communications Conference*, pages 1–6.

Masdari, M. and Zangakani, M. (2020). Green cloud computing using proactive virtual machine placement: Challenges and issues. *J. Grid Comput.*, 18(4).

Massa, J., Forti, S., and Brogi, A. (2022). Data-aware service placement in the cloud-iot continuum. In *Service-Oriented Computing - 16th Symposium and Summer School, SummerSOC 2022, Revised Selected Papers*, volume 1603 of *Communications in Computer and Information Science*, pages 139–158. Springer.

Massa, J., Forti, S., Dazzi, P., and Brogi, A. (2023a). Declarative and linear programming approaches to service placement, reconciled. In *16th IEEE International Conference on Cloud Computing, CLOUD 2023*, pages 1–10. IEEE.

Massa, J., Forti, S., Paganelli, F., Dazzi, P., and Brogi, A. (2023b). Declarative provisioning of virtual network function chains in intent-based networks. In *9th IEEE International Conference on Network Softwarization*.

Oechsner, S. and Ripke, A. (2015). Flexible support of VNF placement functions in OpenStack. In *Proc. 1st IEEE Conference on Network Softwarization*, pages 1–6.

Saber, T., Ventresque, A., Marques-Silva, J., Thorburn, J., and Murphy, L. (2015). Milp for the multi-objective vm reassignment problem. In *IEEE 27th Intl. Conf. on Tools with Artificial Intelligence*, pages 41–48.

Sonkoly, B., Czentye, J., Szalay, M., Németh, B., and Toka, L. (2021). Survey on placement methods in the edge and beyond. *IEEE Commun. Surv. Tutorials*, 23(4):2590–2629.