Exploiting Network–Topology Awareness for *VM* Placement in *IaaS* Clouds

Stefanos Georgiou, **Konstantinos Tsakalozos**, and **Alex Delis**
University of Athens, GR15748, Athens, Greece
{*st.georgiou, k.tsakalozos, ad*}*@di.uoa.gr*

***Abstract*—In contemporary *IaaS* configurations, resources are distributed to users primarily through the assignment of virtual machines (*VMs*) to physical nodes (*PMs*). This resource allocation is typically done in a way that does not consider user preferences and is unaware of the underlying network layout. The latter is of key significance as cost of the cloud's internal network does not grow linearly to the size of the physical infrastructure. In this paper, we focus on *IaaS* clouds built on the highly fault-tolerant and scalable *PortLand* networks. We examine how the performance of the could can benefit from *VM* placement algorithms that exploit user-provided hints regarding the features of sought *VM* interconnections within a virtual infrastructure. We propose and evaluate two such *VM* placement algorithms: the first seeks to rapidly place the required *VMs* as closely as possible on the *PortLand* network starting with the most demanding virtual link and by following a greedy approach. The second approach identifies promising neighborhoods of *PMs* for deploying the virtual infrastructure sought. Both methods try to reduce the network utilization of the physical layer while taking advantage of the *PortLand* layout. Moreover, we seek to minimize the time expended for the placement decision regardless of the size of the infrastructure. Our experimentation shows that our methods outperform the traditional methods (first-fit) in respect to network usage. Our greedy approach reduces the network traffic routed through the top–level core-switches in the *PortLand* topology by up to 75%. The second approach attains an additional 20% improvement.**

Keywords*—topology-aware; hint-aware; network optimization; virtual machine placement

I. Introduction

Effective resource management in *IaaS* clouds is essential for the productive use of the underlying physical infrastructure. The provision of virtual machines (*VM*s) featuring their own IP, CPU(s), main-memory, disk space and network bandwidth is the outcome of the "*VM* placement" phase; during this period, *VM*s are assigned to physical machines (*PM*s) so that user–requests for service are facilitated. In this paper, we tackle the *VM* placement problem in a physical infrastructure whose network fabric is organized using the *PortLand* approach [1]. It is well established that network operating costs alone in data centers may account for up to 20% of the overall power bill [2]. Moreover, conventional tree-like network architectures deployed in modern data centers often encounter oversubscription and network resource contention especially at their core top-levels [3], [4]. This leads to bottlenecks and corresponding delays in rendered services. As an alternative network fabric, *PortLand* can play a significant role in the effective management of cloud computational resources while making a better job in managing the available network bandwidth. In addition, *PortLand*-based clouds may help restrain the rising cost of the traditional tree-like network structures as *PM*s are added into the infrastructure. We also expect that the inherently enhanced management of the *PortLand*-networks will ultimately assist the scalability of the cloud.

Planning tools for *VM* consolidation almost exclusively focus on server resources and frequently disregard the impact *virtual infrastructures* (*VI*s) have on the cloud internal network [5]. Efforts that optimize the networking utilization [6] do overlook the diversity of the requested *VM*s as well as the ways with which these *VM*s are linked together. In this paper, we propose two *VM*-to-*PM* placement algorithms that exploit user provided hints regarding the resource demands of entire *VI*s. As multiple *VM*s are joined together in a operational workflow (i.e., *VI*) to accomplish a task, the above hints are about:

- bandwidth needs for specific pairs of *VM*s in the *VI*,
- anti-collocation constraints for pairs of *VM*s.

Our two proposed algorithms realize a *Virtual Infrastructure Opportunistic fit (VIO)* as well as a *VIcinity-BasEd Search (VIBES)*. *VIO* attempts to place *VM*s as close as possible to each other on the physical network. The algorithm commences by placing the two *VM*s that demand the highest bandwidth for the *Virtual Link (VL)* connecting them. *VIBES* on the other hand, seeks to identify an appropriate *PortLand* neighborhood to accommodate the entire *VL* and then applies *VIO* in this vicinity. While selecting the "areas" of the underlying network fabric which are to be used for placement, the suggested methods take advantage of the *PortLand* layout and properties to reduce network operational costs. Moreover, they help maintain healthy oversubscription usage ratios. Our key contribution is that the placement of *VM*s on *PM*s connected through a *PortLand*-fabric assisted by user hints for their deployment yields noteworthy network resource utilization enhancements.

We have developed and experimented with a simulated environment to compare our proposed placement methods against the *First-Fit Decreasing (FFD)* [7] approach often used as a resource allocation option. *FFD* is expected to offer viable results for *VM* placement in practice [8]. Our approach consistently outperforms *FFD* and shows up to 60% lower network utilization over the physical substrate. When experimenting with high bandwidth *VI*s, our *VIO* placement algorithm displays 75% less usage of core/top-level links, while our *VIBES* technique further reduces the stress of core switches up to 95% when compared to *FFD*. Finally, we examine the performance impact of our approach as we scale both the cloud infrastructure and the *VI*s. We show that planning times of our work remain below 200ms even in infrastructures of more than $8,000$ *PM*s and *VI*s of more than 100 *VM*s.

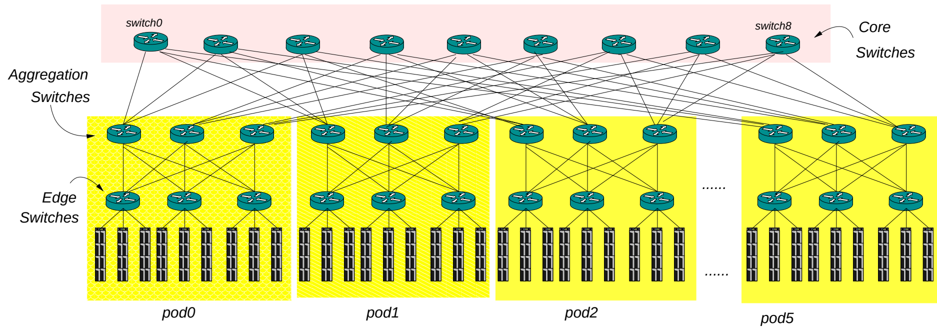


Fig. 1. Sample *PortLand* topology with number of ports per switch k=6

II. Network–Optimized *VM* Scheduling Problem

Users request services in the form of *VM*s from virtualization-based data-centers. By and large, users remain "agnostic" of the policies that govern the assignment of the requested *VM*(s) on *PM*(s) in the underlying *IaaS*-infrastructure. It is also often the case that users ask for entire *virtual infrastructures (VIs)* in the form of a group of *VM*s linked in specific ways through *VL*s featuring diverse characteristics intended to serve specific application needs. In many environments, the deployment of such a comprehensive *VI* is carried in an isolated manner as the middleware considers the placement of each *VM* individually and without giving the due consideration on how *VM*s are linked over the network fabric.

The consumption of the bandwidth among *VM*s in a data-center is an issue that has to receive attention as indiscriminate placement of service-nodes will certainly lead to significant operational bottlenecks. Recent studies [6], [9], [10] on traffic patterns generated by *VM*s in intra-communication for cloud-infrastructures reveal:

1) *Skewed distribution of network traffic*: while 80% of communication remains at low bandwidth levels, 4% of the traffic increases by a tenfold. This implies that the cloud administration cannot make consistently accurate estimations on intra-*VI* traffic.
2) *Traffic rate remains relatively stable*: despite the above highly skewed distribution of traffic, traffic rates present minor fluctuations. This suggests that users likely possess knowledge of the minimally fluctuating traffic that will unfold among the *VM*s of their virtual infrastructures even before the placement occurs.

In addition to the above, a user can always offer useful information on the network requirements of her *VM*s in the requested *VI*. In this context, a cloud-provider can produce *VM* placement solutions that reduce the power costs and yield better *QoS* characteristics through lower oversubscription of network resources [11]. In this paper, we propose a *VI* placement approach in which the provider may elect to put into good use anti-collocation constraints for *VM*s and exploit fundamental characteristics of *PortLand* network fabric. *PortLand* is a layer-2 network that demonstrates promising scaling and fault-tolerance. As Fig. 1 shows, *PortLand* entails the classic notion of "*neighborhood*" –encountered in fat-tree networks– in the form of *pods*. The factor k, which expresses the number of ports on all *Core*, *Aggregation* and *Edge* switches, characterizes each *PortLand* deployment. Each *PortLand* infrastructure contains k pods and supports a total of $\frac{k^3}{4}$ *PM*s. This "effortless" configuration renders *PortLand* an excellent candidate for real-life data-centers. It is this aforementioned combination of user hints and the characteristics of *PortLand* that allow for the rapid creation of network-optimized placement plans.

III. Related Work

The scheduling of *VMs* in parallel with focus on networking has recently attracted much attention. In [6], an approximation algorithm is proposed to solve the *VM* placement problem with minimized network traffic. However, this work only considers the network aspect of the physical infrastructure, treating every *VM* as equally demanding in terms of server resources. Elastic tree [11] employs a network-wide power manager which dynamically turns network elements (switches and links) on or off, and routes the active flows appropriately; this work is complimentary to ours. In [12], elastic trees are extended and a simplified approach of *VM* to *PM* mapping is proposed; yet, diversified resource requirements of *VMs* are not taken into account. Similarly, [13], [14] promotes traffic awareness through monitoring the network and implements live–migration techniques to offer network optimization. Here, migrations stress the network and may also affect live *TCP* connections. In our work, user-offered hints provide resource utilization information prior to the *VM* deployment and thus, we do not consider after placement migration.

In [5], the *VM* placement is treated as a knapsack problem trying to satisfy the maximum possible placement demands. [15] optimizes a placement approach for minimum power usage in its *PM*s; the approach though does not consider management of network resources as *VM*s are treated as individual entities and optimizations apply only on server–resources such as CPU, RAM and disk. Similar to our approach, in [16], [17] neighborhood allocations are also considered. However, here the authors do not offer extensive flexibility in terms of communication and resource requirements, as well as the flexibility of the placement itself. In [18], we utilize a variety of hints and constraints including *VM* collocation suggestions and favoring specific *VMs*. However, we do not exploit hints regarding *VI* bandwidth needs that could result in improved network utilization. Compared to prior approaches [5], [6], [12], we consider *VM*s as heterogeneous, communicating groups of entities operating atop the *PortLand*; this allows for promising outcomes regarding the networking infrastructure.

IV. Overview

Figure 2 depicts the key elements of our approach. *PM*s along with *PortLand* make up the physical infrastructure that receives oversight from a middleware; the latter consists among others, of the following two elements: *a)* a *deployer* component often 3rd-party provided whose responsibility is to deploy compiled plans for a virtual infrastructure (*VI*) under formation, and *b)* a *planner* whose job is to execute a placement algorithm that would determine how *VI*s are to be assigned by the *deployer*. The *planner* takes as input user suggestions regarding desired features in their requested *VI*s and exploits information about the current resource allocations across the underlying machinery.

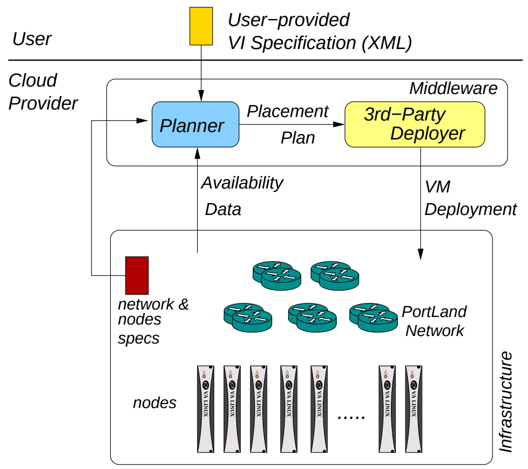


Fig. 2. Overview of our approach (including the *planner*.

A user formulates her request for a *VI* by launching an *XML* document that outlines not only conventional requirements (cores, memory, etc.) but also, the way each of the participating *VM*s is connected with its counterparts. More specifically, we expect that for each *VL*, the user would indicate an estimate of the average network bandwidth to be "consumed" by its two connected *VM*s. Listing 1 presents such a *VL* deployment request; it solicits two *VM*s whose *VL* is set at 20Mbps. Here, there is no stated requirement regarding the anti-collocation of the two *VM*s on a single *PM* as the pertinent clause has been set to *off*.

Listing 1. Designating a *VI* of 2 *VM*s

```
<TaskInfo>
 <vmList>
  <VirtualMachine>
   <cores>6</cores><ram>12</ram>
   <disk>1000</disk>
  </VirtualMachine>
  <VirtualMachine>
   <cores>2</cores><ram>21</ram>
   <disk>2000</disk>
  </VirtualMachine>
 </vmList>
 <vlList>
  <VirtualLink>
   <vm1 reference="VirtualMachine[1]"/>
   <vm2 reference="VirtualMachine[2]"/>
   <bandwidth>20</bandwidth>
   <antiCollocation>off</antiCollocation>
  </VirtualLink>
 </vlList>
</TaskInfo>
```

Alternatively, we could explicitly promote operational independence and fault-tolerance by setting this anti-collocation constraint to *on* and so, explicitly asking the *planner* to come up with a respective solution.

The *planner* works off an initial listing of the available physical infrastructure offered by the cloud–provider. Such a description of physical resources appears in Listing 2 and it is "fed" to the *planner* as Figure 2 shows. In this listing, the network is set to be of type *PortLand* and the cloud machinery includes 1024 *PM*s each having 32-cores, 64 GBytes of main-memory, 6 TBytes of disk and a network connection at 100 Mbps. At any time in its operation, the planner maintains structures clearly indicating the availability of all "hard"-resources made available initially to it. While in operation, the planner keeps this information up to date as Figure 2 depicts; the physical infrastructure makes the level of its resource availability known as soon as *VM*s depart.

Listing 2. Physical Infrastructure Initially Available

```
<infrastructureinfo>
 <nettop>PortLand</nettop>
 <nodeinfo>
  <entry>
   <card>1024</card>
    <nodetemplate>
     <cores>32</cores><ram>64</ram>
     <disk>6000</disk><nlink>100</link>
    </nodetemplate>
   </entry>
  </nodeinfo>
</infrastructureinfo>
```

Below, we introduce two policies that the planner may use in order to appropriately place *VM*s: *Virtual Infrastructure Opportunistic fit (VIO)* and *VIcinity-BasEd Search (VIBES)*. Both methods attempt to reduce the network utilization of physical links and present low decision time overhead regardless of the infrastructure size.

V. Virtual Infrastructure Opportunistic-fit (VIO)

Drawing inspiration from the *nn-embed* greedy algorithm proposed in [19], we have implemented an Opportunistic algorithm that places a *VI* while considering the provided list of the *VL*s. Each *VL* references two *VM*s as well as the required network bandwidth between them. Information regarding *VM*s requirements includes CPU, RAM and disk storage. The availability of resources is kept in a graph that contains a node for every *PM* and an edge for every physical link (*PL*s) in the infrastructure.

Our algorithm needs to assign both *VM*s of a *VL* to *PM*s with enough available resources. At the same time a path with sufficient bandwidth connecting the two *VM*s must exist. In absence of an anti-collocation constraint, the best placement for the *VM*s of a *VL* is to be co-located on the same *PM*. In this way we eliminate the need to consume the network bandwidth of the physical infrastructure. If we are unable to fit both *VM*s on the same host, we place them on the nearest possible *PM*s.

In case anti-collocation constraints and/or resource shortage do not allow a *VM* to be placed on any *PM* or a *path* cannot be created between the *PM*s of a *VL* our algorithm employs backtracking that reverts to a proper amount of previously *VM*-to-*PM* assignments before resuming its search. Each *VL* is associated with a revert counter that limits the backtracking attempts.

Alg. 1 outlines our *VIO* placement method and Table I summarizes notations used. Placement begins by first sorting the list of *VL*s in descending bandwidth order. The sorted list along with all the *PM*s is provided as input to the recursive Alg.1. There are two cases in handling a *VL* (**switch** statement): either a *VM* has to be assigned to a *PM* or both referenced *VM*s are already assigned and we just need to verify the validity of the path connecting them.

For the assignment of the first *VL* of the *VI* performed in the *place* method call, we consider the *PM* with the highest weighted sum of all available resources (CPU, RAM, disk

Algorithm 1 PlaceVLs

Input: `vList`: The sorted list of *VL*s we attempt to place
`i`: The index in the `vList` of the *VL* we attempt to place
`pmList`: The list of physical machines we are considering for placement
Output: true if the *VL* placement leads to a successful *VI* placement, otherwise false

```
if no more VLs to place then
  return true
end if
vl := vList[i]
switch vl.placedVMs do
  case at most one VM placed
    while place(vl, pmList) do
      if PlaceVLs(vList, i+1, pmList) then
        return true
      else if !revert(vl) then
        return false
      end if
    end while
    setProblematicVL(vl)
    return false
  case both VMs placed
    if !verify_and_find_path(vl) then
      setProblematicVL(vl)
      return false
    else
      if PlaceVLs(vList, i+1, pmList) then
        return true
      else
        return revert(vl)
      end if
    end if
  end case
end switch
```

storage); we term this *PM* the *most available* one. With this approach we promote a load balanced environment as *VI*s are deployed around different *PM*s. If the *most available PM* has insufficient remaining resources to host the *VM* pair of the first *VL*, or an anti-collocation request is present between the respective *VM*s, we place the most resource demanding *VM* of the *VL* on the *most available PM* and look for a nearby *PM* to accommodate the second *VM*. Should we fail to assign both *VM*s of the first *VL*, we reject the *VI* placement request. Should we succeed, we bootstrap a list of used *PM*s (*placed_PM_list*) by storing the chosen *PM(s)* in it. The *placed_PM_list* plays a pivotal role in placing the rest of the *VL*s.

If we are to handle a *VL* with none of its *VM*s already placed we attempt to place the two *VM*s of the *VL* on one of the already used *PM*s stored in *placed_PM_list*. If that fails we look for the closest possible *PM*s of the last *PM* on that list. We always attempt to place the *VM*s of each *VL* as close as possible to previously assigned *VM*s since they are likely to communicate with each other. In case the *VL* at hand contains an anti-collocation request, we follow a different strategy. First, we attempt to assign the most resource demanding *VM* on one of the *PM*s in *placed_PM_list*. If that fails, we choose the last *PM* in *placed_PM_list* and attempt to find one of its neighbors capable of hosting the pertinent *VM*. Second, we search for a nearby *PM* capable of hosting the *VL*'s second *VM*. For a *VL* with any of its *VM*s already assigned to a *PM*, we either attempt to place the *VM* on the same *PM* (no anti-collocation constraint present) or we find an appropriate nearby *PM*.

In light of a *VL* with both of its *VM*s assigned (meaning that the respective *VM*s are also part of other *VL*s), no *VM* assignment is necessary. Yet, we must always verify the satisfaction of any anti-collocation constraints as well as the network requirements set by the user. If the two *VM*s are placed on separate *PM*s, a *path* with enough bandwidth must be found between the two *PM*s. In order for a *path* to be admissible for a specific *VL*, every physical link (*PL*) of the *path* needs to have at least as much available bandwidth as the *VL* requires.

The placement of the entire *VI* is considered successful as soon as all *VL*s of the *VI* are assigned. In this case Alg.1 returns true.

TABLE I. SUMMARY OF ABBREVIATIONS USED

Used notations	
PM	Physical Machine
VI	Virtual Infrastructure
VM	Virtual Machine
VL	Virtual Link between a pair of *VM*s
PL	Physical Link in the network graph
path	A list of *PL*s that connects two *PM*s
path length	The number of *PL*s (or hops) in a *path*
cc_i	Core capacity of PM_i
uc_i	Cores used in PM_i
cr_i	Memory capacity of PM_i
ur_i	Memory used in PM_i
cs_i	Storage capacity of PM_i
us_i	Storage used in PM_i
cb_i	Bandwidth capacity of PL_i
ub_i	Bandwidth used in PL_i
n_{pms}	Number of *PM*s in the infrastructure
n_{vms}	Number of *VM*s in the *VI*

Two operations in our *Opportunistic-fit VIO* approach that play a crucial role are the finding of nearby *PM*s and backtracking from a state where we are not able to satisfy all requirements. In what follows we describe in detail these two operations.

Finding nearby *PM*s: finding a *PM*'s neighbors involves calculating all network *paths* towards all other *PM*s given a fixed amount of network hops (*path length*) between the original *PM* and its neighbor. For each of the *paths* and its Physical Links (*PL*s) list, we use the link's bandwidth ub_i to calculate a weight:

$$W_{path} = \sum_{i \in list} ub_i$$

Out of all *path*s between a neighbor and the original *PM*, we choose the one with the lowest W_{path}. We always verify that a *path* is valid for the existing topology since in *PortLand* a switch must never forward a packet out along an upward-facing port if the ingress port for that packet is also an upward-facing port. Having compiled a list of the best paths (one *path* for each neighbor of a *PM*), we sort this list in ascending W_{path} order. When trying to place a *VM* on a neighboring *PM*, we iterate over this the *path* list. Placement of the *VM* can fail either due to lack of resources of the neighboring *PM* or due to inadmissibility (not enough bandwidth) of the *path*. In either case, we continue with the next path of our sorted list. *VL*s consume resources of the *PL*s, thus a *path* is admissible only if every *PL* in the *path* has at least as much available bandwidth as the *VL* requires. In case we are not able to find a *PM* that can both host the *VM* and provide a *path* that would satisfy the requirements set by the assignment of *VM*s on its ends, we extend the *path length* by increasing the number of hops allowed among *PM*s. We term the amount of hops "Radius". In *PortLand*, the first radius level allows 2 hops, while each subsequent level allows 2 additional hops. We end the process of trying to find a neighbor as soon as the maximum allowed radius of 6 hops is reached.

Backtracking and reverting *VL*s: During our *Opportunistic-fit*, we might reach a "dead-end" state where we cannot place the *VL* at hand. This *VL* placement inability is due to any of the following three reasons: *a)* there is no *PM* with enough resources to host a *VM* of the *VL*, *b)* we cannot allocate a path with enough bandwidth between the *VM*s of the *VL*, *c)* an anti-collocation constraint is violated. Upon reaching this state, we mark the *VL* at hand as *problematicVL* (*setProblematicVL* method call) and proceed with what we term *backtracking*. During the backtracking process we revert the placement of the *VL*s. The *VL*'s reverting order is determined by the recursive nature of Alg.1; that is *VL*s are reverted in the opposite order they have been placed.

Reverting firstly involves freeing the network bandwidth in case the respective *VM*s are placed on separate *PM*s. Freeing the network resources involves deallocating the bandwidth reserved by the *VL* on every *PL* in the *path* between the *PM*s where the *VM*s were assigned. Secondly, when we have reverted all *VL*s a *VM* is part of, we also deassign the *VM* from its *PM* host, deallocating the resources reserved for it (cores, RAM and disk usage).

We stop backtracking (reverting the *VL* placements) as soon as any *VM* of the problematic *VL* is reverted back to unassigned. If both *VM*s of a problematic *VL* are placed (backtracking caused by the failure to verify the *VL* placement), we need to revert to the last partial placement where any of the involved *VM*s is not yet placed. We note that a *VM* may be part of multiple *VL*s, thus reverting that *VM* means reaching a partial placement state where all the *VL*s referencing the *VM* are also reverted. If *problematicVL* had none of its *VM*s placed when it reached the "dead-end", we backtrack only one step, reverting a single *VL* and continuing the search.

As soon as backtracking stops, we continue our search for a placement. However, in order not to reach the same placement "dead-end" we need to mark the unsuccessful placement. We do so by not allowing the *VM* of the problematic *VL* to be hosted on the same *PM* as before.

To confine the extent of our reverting process and terminate the search for a placement accordingly, we utilize a *revert counter* on each *VL*. In case the maximum amount of reverts has been reached for a *VL*, we terminate the placement and reject the *VI*. This counter limits our backtracking attempts and allows our algorithm to maintain reasonable decision times.

Algorithm 2 Revert

Input: `vl`: the virtual link we are reverting
Output: true if we are stopping the backtrack process and resuming our search, otherwise false

```
deallocateResources(vl)
revertingVL := getProblematicVL()
if (vl shares a VM with revertingVL AND the shared vm is now unassigned) OR
revertingVL.placedVMs==0 then
    unsetProblematicVL();
    return true
end if
return false
```

VI. VICINITY-BASED SEARCH (VIBES)

In *VIBES* we realize a two-phase approach according to which we first find a fitting neighborhood of *PM*s for our *VI* and then we utilize our *VIO* algorithm to place the *VI*. In essence we create a subgraph of our physical infrastructure that guarantees an overall resource availability throughout its *PM*s. This subgraph is used as input to *VIO* in producing the final *VM* placement. The reduced search space enables us to shorten execution time and enhance the placement quality.

In the first phase of our approach we formulate a group of *PM*s in close vicinity that can possibly host our *VI*. We call this group of *PM*s a neighborhood. The neighborhood formation is achieved with a call to the *getNeighborhood* method in Alg.3. *PortLand* proves to be ideal in forming neighborhoods as it provides already clustered *PM*s in its *pods*. We start the neighborhood formation by finding the edge switch with the biggest sum of available host and network resources considering the *PM*s it connects. Should we require a larger neighborhood, we locate the pod with the most available resources. The neighborhood expands even further by progressively merging the next most available pod to the set of the already selected pods. The distance, in terms of *path length*, is not considered when selecting the pod to be merged to the neighborhood. The reason for this is that in *PortLand* the distance between pods is fixed. The only metric we have to define in order to identify which pod to merge next is its resource availability. We denote resource availability of a resource r as: ar_i=cr_i-ur_i, where cr_i indicates the capacity of the resource for the included *PM*s, and ur_i reflects how much of the resource is currently being used. We rank all neighborhoods using the following formula:

$$Score_{neigh} = w_c \sum_{PM_i \in neigh} (w_{cpu} ac_i + w_{ram} am_i + w_{disk} as_i)$$
$$+ w_n \sum_{PL_i \in neigh} ab_i$$

Each of the three resources (*cores*, *RAM* and *disk*) must have their individual availability calculated. Weights w_c and w_n represent the overall significance of host and network resource significance. Similarly, weights w_{cpu}, w_{ram} and w_{disk} correspond to CPU, RAM and disk importance.

Algorithm 3 VicinitySearch

Input: `infra`: Our infrastructure with information on *PM*s and the topology
`vinfra`: The *VI* to be placed, with its given *VM*s and *VL*s
`size_limit`: The maximum allowed *PM*s a neighborhood is allowed to contain
Output: true if application was fully placed, otherwise false

```
min_PMs:= amount of PMs a switch can accommodate
vList := sort(vinfra.virtualLinkList)
while true do
    neigh := getNeighborhood(infra, min_PMs, size_limit)
    if size_limit was reached then
        break
    else if Opportunistic(vList, 0, neigh) then
        return true
    else if neigh includes the PMs of only a single switch then
        min_PMs:= amount of PMs in a pod {request a pod}
    else
        min_PMs:= neigh.size + amount of PMs in a pod {request a merge of pods}
    end if
end while
return false
```

In the *getNeighborhood* method of Alg.3 we also need to verify the admissibility of a neighborhood. To do so, we sum-up each *PM* related resource of the neighborhood and require that they sum to at least the sum of each respective resource requested for the *VI*. This means that the neighborhood should include for example as many free cores as the sum of cores the *VM*s of the *VI* require. If this requirement is not met, we search for a larger neighborhood. At this point, we do not compare the neighborhood's internal network bandwidth availability against the total bandwidth requirements of the *VI* since we are likely to be hosting multiple *VM*s on each of our neighborhood's *PM*s, eliminating some of the communication costs.

Should the neighborhood have less *PM* related resources than requested, it is expanded with the addition of pods. The search for a large enough neighborhood continues until we are either presented with enough available resources, or our search window is growing larger than a set amount (*size_limit*). Beyond that point, we assume that including more *PM*s in our search will most likely only expand the decision time without reaching a successful placement as we already include the most resource-free *PM*s in our neighborhood. The administration of the cloud is free to tune this parameter so as not to expend valuable time and resources in trying to achieve a placement. When we reach this *size_limit*, the algorithm rejects the *VI*.

In the second phase of our algorithm, we attempt to place our *VI* in the *PM*s of the neighborhood returned from the first phase. For the placement decision we employ our *Opportunistic-fit* algorithm presented in Section V, providing it with the list of *PM*s of the selected neighborhood. However, the placement can possibly fail. This can happen either due to inadequate bandwidth in the neighborhood, inability to satisfy anti-collocation constraints, or imbalanced resource availability. For example we could have two *PM*s in the neighborhood with 2 and 4 available cores respectively, but our *VI* included two *VM*s with 3 cores required each, therefore we cannot attain a successful placement. If such a placement failure occurs, we return to phase one and repeat the process, forcing *getNeighborhood* to return a bigger neighborhood. To this end we update and utilize the minimum amount of *PM*s the returned neighborhood should have (*min_PMs*).

VII. Evaluation

The evaluation of our work is based on simulation of physical infrastructures. The algorithms as well as the simulated infrastructure are implemented in *Java* using the *JgraphT*–graph library that provides multiple pathfinding and traversal algorithms. Our tests were run on a Intel Q6600 processor.

The infrastructure consists of *PM*s with 32 cores, 64GBytes RAM and storage of 6TBybes, linked through 1Gbps Ethernet connections; this configuration was based on *IBM x3850 X5* Servers. Our *PortLand* switches are connected through 1Gbps links as well. Unless stated otherwise, the simulation is run on a 1,024 *PM* infrastructure. Regarding the weights used in our evaluation, we set the computational and network weights of our vicinity search algorithm to w_c=0.65 and w_n=0.35 as these values produced placements of good quality. For CPU, RAM and disk weights we use w_{cpu}=w_{ram}=w_{disk}=0.33. We also set the maximum allowed vicinity size to be 5-times the amount of the given *VI*'s *VM*s, namely $size\_limit$=$5 * n_{vms}$. Lastly, we set the pre-determined amount of maximum allowed reverts for each *VL* to 8.

Our objectives in this evaluation are to: *1)* compare the performance of our algorithms against other typically used placement approaches, *2)* determine the behavior and placement time in light of diverse physical infrastructures, *VI* sizes and workloads and *3)* examine core and aggregation network switch utilization in each algorithm for communication-heavy *VI*s. We want to stress the importance of user-provided bandwidth needs, therefore we compared our *VIO* and *VIBES* algorithms with the *FFD* baseline approach which is network-agnostic.

The *VI*s used in our experiments display three different data flow topologies termed *Pipeline*, *Data Aggregation* and *Epigenomics*. All three topologies are inspired by workflow structures analyzed in [20] and are shown in Fig.3. Our evaluation scenario involves placing the following default *VI*s:

1) 20% Pipeline *VI*s (Fig.3(a)) with 5 *VM*s
2) 30% Data Aggregation *VI*s (Fig.3(b)) with 14 *VM*s
3) 50% Epigenomics *VI*s (Fig.3(c)) with 20 *VM*s

As we request *VI*s for deployment, we also randomly select already deployed *VI*s for removal. The removed *VI*s are 30% of the total *VI*s placed. In what follows, we examine the network usage with respect to switch utilization per infrastructure tier. We also consider the impact of scaling three different factors of the problem, namely the infrastructure size, *VI* size, and the bandwidth levels of the tested *VI*s.

• **Overall network utilization and average** *path length***:** In this experiment, we examine the network utilization while we gradually place additional *VI*s. We choose to take snapshots of the network usage within our infrastructure at different *PM* load levels, as we keep adding *VI*s. We term load of a *PM* the weighted sum of all its resources utilization. We begin at 10% load and we gradually reach up to 90%.

In Fig.4 we show the performance of the 3 placement algorithms. Our proposed methods outperform *FFD* by up to 3 times. The *VI* rejection rate established by the three algorithms remains almost identical for *FFD* and *VIO*, while *VIBES* displays up to 2% higher rejection rate. The tendency of *VIBES* to reject more *VI*s is explained by the limit on the maximum allowed vicinity size. Inspecting Fig.5 gives us a view of how the average *path length* fluctuates among the three algorithms, with *VIBES* exhibiting up to 40% less hops than *FFD*, and 20% less hops against *VIO*. This experiment points into the fact that *VIBES* places nodes on groups of *PM*s more effectively than its two counterparts.

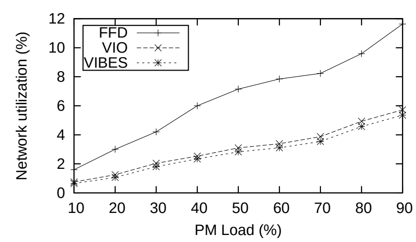


Fig. 4. Network Utilization per load level

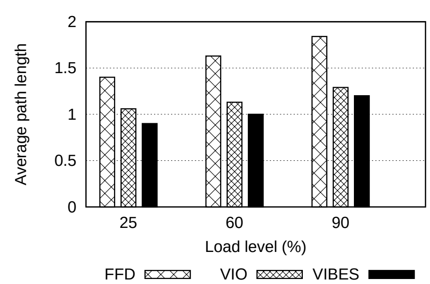
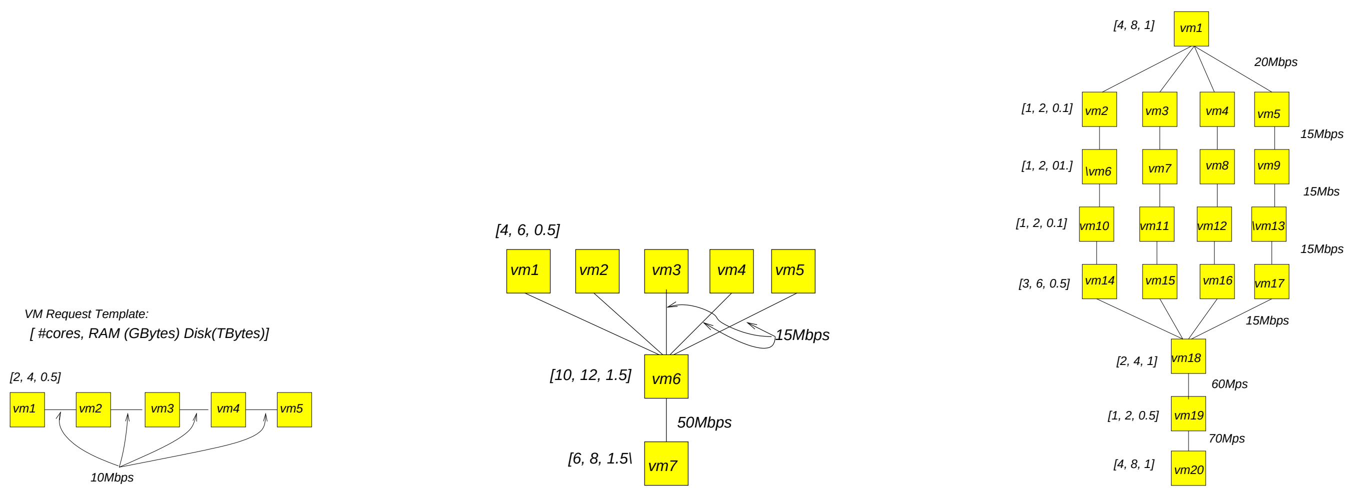


Fig. 5. Average *path length* on different load levels

• **Scaling the bandwidth requirements of the** ***VI*s:** Here, we stress the network by increasing the average bandwidth required by the *VI*s. We do so by increasing the bandwidth needs by a factor of 3. We gradually increase the amount of *VI*s placed until we reach 90% resource utilization of the *PM*s. Fig.6 shows the network utilization of core, aggregation, and edge switches for the 3 placement policies.

FFD displays high core and aggregation *physical link (PL)* usage, since it is treating each *VM* as an isolated entity instead of being part of a *VI*. Compared to *FFD*, *VIO* and *VIBES* present 75% and 95% respectively less core switch *PL* usage.



(a) Pipeline *VI*. (b) Data Aggregation *VI*. (c) Epigenomics *VI*.

Fig. 3. The *VI* topologies of our evaluation

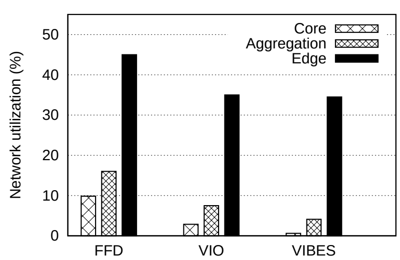


Fig. 6. Network utilization per switch level

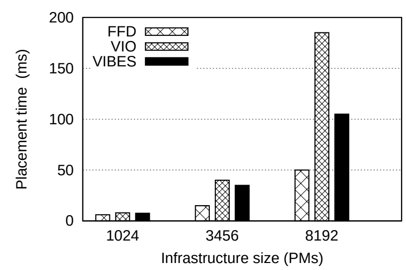


Fig. 7. Average decision times for different data center sizes

In regards to the aggregation *PL*s, *VIO* and *VIBES* reduce the utilization by a factor of 2 and 4 respectively compared to *FFD*. The reason for *VIO* to show higher utilization in core/aggregation switches compared to *VIBES* is the following: assume that we remove a *VI* from a load-balanced infrastructure and then, we ask for a new *VI* that calls for more resources than those released by the just-removed *VI*. *VIO* will likely choose to start the placement on one of the just off-loaded *PM*s even though the other neighbouring *PM*s are heavily loaded. Consequently, some of the *VM*s of the *VI* will not be placed in *PM*s under the same edge switch due to this low *PM* resource availability. In turn, this causes *VIO* to look for neighbours on a larger *path length*, utilizing aggregation and at some cases even core switches. However, *VIBES* treats this situation more effectively. The neighbourhood in which *VIBES* attempts to place the new *VI* is a group of closely linked *PM*s that makes sure we utilize the least amount of core/aggregation *PL*s. Therefore, even at high loads and bandwidth-demanding *VI*s, *VIBES* makes minimal use of core switches. We must also note that during high-bandwidth demand tests *FFD* was constantly bringing multiple edge *PL*s to an over-committed state of up to 150%. At low *PM* loads we observe an average of 3% over-committed edge links, while at a maximum load this number reaches up to 13%.

• **Scaling the physical infrastructure size:** We now look at the decision time of our algorithms as we scale the physical infrastructure (by adding more *PM*s) while maintaining the default amount of *VM*s in the *VI*s (5 *VM*s for Pipeline, 14 *VM*s for Data Aggregation, and 20 *VM*s for Epigenomics). In order to get an accurate measurement of decision times for both the successful and the unsuccessful placement requests, we deploy *VI*s until we reach a 20% rejection rate. The decision time for successfully placed *VI*s are on average 50-80% less than the decision times for rejecting *VI*s. This is because rejecting a *VI* involves exploring a much larger search space that requires much time for reverting *VL*s and reattempting the placement. We evaluate the performance of the three algorithms in physical infrastructures of three sizes, decided by the *PortLand* k factor (the number of ports on each switch): 1,024 *PM*s (k=16), $3,456$ *PM*s (k=24) and $8,192$ *PM*s (k=32). The results are shown in Fig.7. Our algorithms remain comparable to the fast nature of *FFD* with *VIO* and *VIBES* exhibiting decision times of up to 185ms and 95ms respectively, against *FFD*'s 50ms. The extra time required by our algorithms is due to the need to execute complex *path* calculations within our topology graph, as well as the time spent for backtracking. In addition, the vicinity search algorithm evidently succeeds in making use of the reduced search space provided by the neighbourhood subgraphs to lessen decision time by up to 50% compared to *VIO*. This performance lead is extended as we add more *PM*s in the infrastructure.

• **Scaling the virtual infrastructure size:** In this final experiment we measure the impact of the *VI* sizes on the placement decision time. We gradually increase the average number of *VM*s that a *VI* includes, while keeping the infrastructure size to 1024 *PM*s. We select three group ranges for the *VI* sizes:

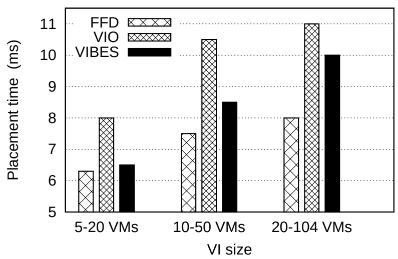


Fig. 8. Average decision times for different *VI* sizes

1) 5-20 *VM*s: 5 *VM*s for Pipeline flows, 14 for Data Aggregation, 20 for Epigenomics
2) 10-50 *VM*s: 10 *VM*s for Pipeline flows, 30 for Data Aggregation, 52 for Epigenomics
3) 20-104 *VM*s: 20 *VM*s for Pipeline flows, 80 for Data Aggregation, 104 for Epigenomics

The Epigenomics workflows have a pre-set topology that we scale by increasing the amount of *VM*s operating in parallel (spreading a job to more *VM*s than the default amount of 4). As previously, we stop our simulation as soon as we reach a rejection rate of 20%. The decision time of the three algorithms is presented in Fig.8. Evidently none of the three algorithms seems to be largely affected by the *VI* size, maintaining a low decision time of under 12ms even for very large *VI*s.

VIII. Conclusions and Future Work

In this paper, we propose two algorithms that perform the placement of *VI*s made of several *VM*s while taking into account user-provided constraints and network usage hints. Our approach keeps the user agnostic of the underlying infrastructure while exploiting hints and the *PortLand*-specific network topology. We show how the information regarding bandwidth for intra-*VI* network combined with knowledge of the cloud's topology can be crucial for the efficient operation of the physical infrastructure. Through our evaluation, we demonstrate the advantages our algorithms have over traditional placement methods. Such methods typically ignore the cloud's structure and often resort to post-placement complex and network-demanding live migrations. Instead, our focus is to address the efficient management of the network resource during the initial placement of the *VM*s. In the future, we intend to improve the *VM* placement of the second phase of our *VIBES* algorithm and further reduce the network utilization. We will test our approach in other network topologies such as *BCube* [21] and *VL2* [22]. Finally, we plan to exploit the reduced network utilization provided by our work to optimize power usage of network switches [11].

Acknowledgment

This work was partially supported by the Sucre EU FP7 ICT project. We thank the reviewers for their comments.

References

[1] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: a Scalable Fault-tolerant Layer 2 Data Center Network Fabric," in *Proc. of the ACM SIGCOMM Conf.*, Barcelona, Spain, August 2009.

[2] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 68–73, December 2008.

[3] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: A Scalable and Fault-tolerant Network Structure for Data Centers," in *Proc. of the ACM SIGCOMM Conf.*, Seattle, WA, October 2008.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable Commodity Data Center Network Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, October 2008.

[5] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A Scalable Application Placement Controller for Enterprise Data Centers," in *Proc. of the 16th Int. Conf. on WWW*, Banff, Canada, May 2007.

[6] X. Meng, V. Pappas, and L. Zhang, "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement," in *Proc. of the 29th INFOCOM Conf.*, San Diego, CA, March 2010.

[7] A. C.-C. Yao, "New Algorithms for Bin Packing," *Journal of the ACM*, vol. 27, no. 2, pp. 207–227, April 1980.

[8] G. Dosa, "The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I)=(11/9)OPT(I)+6/9," in *17th European Symp. on Computer–Aided Process Engineering*, Bucharest, Romania, May 2007.

[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The Nature of Data Center Traffic: Measurements & Analysis," in *Proc. of the 9th ACM Internet Measurement Conference (IMC)*, Chicago, IL, November 2009.

[10] D. Ersoz, M. S. Yousif, and C. R. Das, "Characterizing Network Traffic in a Cluster-based, Multi-tier Data Center," in *Proc. of the 27th IEEE ICDCS Conf.*, Toronto, Canada, June 2007.

[11] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving Energy in Data Center Networks," in *Proc. of the 7th USENIX NSDI Conf.*, Berkeley, CA, 2010.

[12] V. Mann, A. Kumar, P. Dutta, and S. Kalyanaraman, "VMFlow: Leveraging VM Mobility to Reduce Network Power Costs in Data Centers," in *Proc. of the 10th Int'l IFIP TC 6 Conf. on Networking - Volume Part I*, Berlin, Heidelberg, May 2011, Springer-Verlag.

[13] V. Shrivastava, P. Zerfos, Kang-Won Lee, H. Jamjoom, Yew-Huey Liu, and S. Banerjee, "Application-aware Virtual Machine Migration in Data Centers," in *30th IEEE INFOCOM Conf.*, Shanghai, China, April 2011.

[14] S. Hallett, G. Parr, and S. McClean, " Network Aware Cloud Computing for Data and Virtual Machine Placement," in *12th Ann. PostGraduate Symp. on the Convergence of Telecommunications, Networking and Broadcasting*, Liverpool, UK, June 2011.

[15] J. Xu and J.A.B. Fortes, "Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments," in *Proc. of the 2010 IEEE/ACM Int. Conf. on Green Computing and Communications*, Dec. 2010.

[16] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 242–253, August 2011.

[17] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Co-NEXT '10 Proc. of the 6th Int'l Conf.* November 2010, ACM.

[18] K. Tsakalozos, M. Roussopoulos, V. Floros, and A. Delis, "Nefeli: Hint-based execution of workloads in clouds," in *Proc. of the 30th IEEE ICDCS Conf.*, Genoa, Italy, June 2010.

[19] V. M. Lo, S. V. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, B. Nitzberg, J. A. Telle, and X. Zhong, "OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures," *Int. Journal of Parallel Programming*, vol. 20, no. 3, pp. 237–270, June 1991.

[20] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M-H. Su, and K. Vahi, " Characterization of Scientific Workflows," in *3rd Workshop on Workflows in Support of Large-Scale Science*, Austin, TX, Nov. 2008.

[21] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a High Performance, Server-centric Network Architecture for Modular Data Centers," in *Proc. of the ACM SIGCOMM Conf.*, Aug. 2009.

[22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a Scalable and Flexible Data Center Network," in *Proc. of the ACM SIGCOMM Conf.*, Aug. 2009.