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 cloud 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 (VMs) featuring their own IP, CPU(s), main-memory, disk space and network bandwidth is the outcome of the “VM placement” phase; during this period, VMs are assigned to physical machines (PMs) 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 over-subscription 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 PMs 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 (Vls) have on the cloud internal network [5]. Efforts that optimize the networking utilization [6] do overlook the diversity of the requested VMs as well as the ways with which these VMs 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 Vls. As multiple VMs are joined together in an operational workflow (i.e., VI) to accomplish a task, the above hints are about:

- bandwidth needs for specific pairs of VMs in the VI,
- anti-collocation constraints for pairs of VMs.

Our two proposed algorithms realize a Virtual Infrastructure Opportunistic fit (VIO) as well as a ViciDay-BasEd Search (VIBES). VIO attempts to place VMs as close as possible to each other on the physical network. The algorithm commences by placing the two VMs 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 VMs on PMs 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 Vls, 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 Vls. We show that planning times of our work remain below 200ms even in infrastructures of more than 8, 000 PMs and Vls of more than 100 VMs.
II. NETWORK–OPTIMIZED VM SCHEDULING PROBLEM

Users request services in the form of VMs 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 VMs linked in specific ways through VLs 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 VMs are linked over the network fabric.

The consumption of the bandwidth among VMs in a data-center is an issue that has to receive attention as indiscriminate placement of service-nodes will certainly lead to significant bandwidth needs that could result in improved network utilization. Compared to prior approaches [5], [6], [12], we consider VMs as heterogeneous, communicating groups of entities operating atop the PortLand; this allows for promising outcomes regarding the networking infrastructure.

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 PMs; the approach though does not consider management of network resources as VMs 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 VMs 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. PMs 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 VIs are to be assigned by the deployer. The planner takes as input user suggestions regarding desired features in their requested VIs and exploits information about the current resource allocations across the underlying machinery.

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 VMs 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 VMs. Listing 1 presents such a VL deployment request; it solicits two VMs whose VL is set at 20Mbps. Here, there is no stated requirement regarding the anti-collocation of the two VMs on a single PM as the pertinent clause has been set to off.

Listing 1. Designating a VI of 2 VMs

```
<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>
  <vList>
    <VirtualLink>
      <vm1 reference="VirtualMachine[1]"/>
      <vm2 reference="VirtualMachine[2]"/>
      <antiCollocation off/>
    </VirtualLink>
  </vList>
</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 PMs 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 VMs depart.

Below, we introduce two policies that the planner may use in order to appropriately place VMs: 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 VIs. Each VL references two VMs as well as the required network bandwidth between them. Information regarding VMs 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 (PLs) in the infrastructure.

Our algorithm needs to assign both VMs of a VL to PMs with enough available resources. At the same time a path with sufficient bandwidth connecting the two VMs must exist. In absence of an anti-collocation constraint, the best placement for the VMs 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 VMs on the same host, we place them on the nearest possible PMs.

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 PMs 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 VIs in descending bandwidth order. The sorted list along with all the PMs 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 VMs 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 VLs we attempt to place
i: The index in 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

1: if no more VLs to place then
2: return true
3: end if
4: vl = vList[i]
5: switch vl.placedVLs do
6: case at most one VM placed
7: while placeVL(vList, i+1, pmList) do
8: if PlaceVL(vList, i+1, pmList) then
9: return true
10: else if !verify(vl) then
11: return false
12: end if
13: end while
14: setProblematicVL(vl)
15: return false
16: end case
17: if verify_and_find_path(vl) then
18: setProblematicVL(vl)
19: return false
20: else
21: if PlaceVL(vList, i+1, pmList) then
22: return true
23: else
24: return revert(vl)
25: end if
26: end if
27: end case
28: end switch

Two operations in our Opportunistic-fit VI approach that play a crucial role are the finding of nearby PMs 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 PMs: finding a PM’s neighbors involves calculating all network paths towards all other PMs 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 (PLs) list, we use the link’s bandwidth \( b_i \) to calculate a weight:

\[
W_{\text{path}} = \sum_{v_i \in \text{path}} b_i
\]

Out of all paths between a neighbor and the original PM, we choose the one with the lowest \( W_{\text{path}} \). 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 \( W_{\text{path}} \) 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. VLs consume resources of the PLs, thus a path is admissible only if every PL in the path has at least as much available bandwidth as the VL requires.

Backtracking and reverting VLs: 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 have at least as much available bandwidth as the VL requires.

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

<table>
<thead>
<tr>
<th>TABLE I. SUMMARY OF ABBREVIATIONS USED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used notations</td>
</tr>
<tr>
<td>PM</td>
</tr>
<tr>
<td>VI</td>
</tr>
<tr>
<td>VM</td>
</tr>
<tr>
<td>VL</td>
</tr>
<tr>
<td>PL</td>
</tr>
<tr>
<td>path</td>
</tr>
<tr>
<td>path length</td>
</tr>
<tr>
<td>cc</td>
</tr>
<tr>
<td>uc</td>
</tr>
<tr>
<td>cr</td>
</tr>
<tr>
<td>ur</td>
</tr>
<tr>
<td>cs</td>
</tr>
<tr>
<td>us</td>
</tr>
<tr>
<td>cb</td>
</tr>
<tr>
<td>ub</td>
</tr>
<tr>
<td>n_pms</td>
</tr>
<tr>
<td>n_vms</td>
</tr>
</tbody>
</table>

Summary: The index in the vList (or hops) in a path |

Storage capacity of PM_i |

Storage used in PM_i |

Bandwidth capacity of PL_i |

Bandwidth used in PL_i |

Number of PMs in the infrastructure |

Number of VMs in the VI |
allocate a path with enough bandwidth between the VMs of the VL, c) an anti-collocation constraint is violated. Upon reaching this state, we mark the VL at hand as problemVL (setProblematicVL method call) and proceed with what we term backtracking. During the backtracking process we revert the placement of the VLs. The VL’s reverting order is determined by the recursive nature of Alg.1; that is VLs are reverted in the opposite order they have been placed.

Reverting firstly involves freeing the network bandwidth in case the respective VMs are placed on separate PMs. Freeing the network resources involves deallocating the bandwidth reserved by the VL on every PL in the path between the PMs where the VMs were assigned. Secondly, when we have reverted all VLs 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 VMs 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 VMs is not yet placed. We note that a VM may be part of multiple VLs, thus reverting that VM means reaching a partial placement state where all the VLs referencing the VM are also reverted. If problematicVL had none of its VMs 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 revert 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: VI, the virtual link we are reverting
Output: true if we are stopping the backtrack process and resuming our search, otherwise false
1: deallocateResources(vI)
2: revertingVL := getProblematicVL()
3: if (vI) shares a VM with revertingVL AND the shared vm is now unassigned) OR revertingVL.placedVMs==0 then
4: unsetProblematicVL();
5: return true
6: end if
7: 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 PMs 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 PMs. 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 PMs in close vicinity that can possibly host our VI. We call this group of PMs 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 PMs 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 PMs 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: \( ar = cr - ur \), where \( cr \) indicates the capacity of the resource for the included PMs, and \( ur \) reflects how much of the resource is currently being used. We rank all neighborhoods using the following formula:

\[
\text{Score}_{\text{neigh}} = w_c \sum_{PM_i \in \text{neigh}} (w_{cpu}\alpha_{ci} + w_{ram}\alpha_{mi} + w_{disk}\alpha_{si}) + w_r \sum_{PL_i \in \text{neigh}} ab_i
\]

Each of the three resources (cores, RAM and disk) must have their individual availability calculated. Weights \( w_c \), \( w_r \) 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 PMs and the topology
vinfra: The VI to be placed, with its given VMs and VLs
size_limit: The maximum allowed PMs a neighborhood is allowed to contain
Output: true if application was fully placed, otherwise false
1: min_PMs:= amount of PMs a switch can accommodate
2: vList := sort(vinfra.virtualLinkList)
3: while true do
4: neigh := getNeighborhood(infra, min_PMs, size_limit)
5: if size_limit was reached then
6: break
7: else if Opportunity(vList, 0, neigh) then
8: return true
9: else if neigh includes the PMs of only a single switch then
10: min_PMs:= amount of PMs in a pod (request a pod)
11: else
12: min_PMs:= neigh.size + amount of PMs in a pod (request a merge of pods)
13: end if
14: end while
15: 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 VMs 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 VMs on each of our neighborhood’s PMs, 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 \((\text{size\_limit})\). Beyond that point, we assume that including more \(\text{PMs}\) in our search will most likely only expand the decision time without reaching a successful placement as we already include the most resource-free \(\text{PMs}\) 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 \(\text{size\_limit}\), the algorithm rejects the VI.

In the second phase of our algorithm, we attempt to place our \(\text{VI}\) in the \(\text{PMs}\) of the neighborhood returned from the first phase. For the placement decision we employ our \text{Opportunistic-fit} algorithm presented in Section V, providing it with the list of \(\text{PMs}\) 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 \(\text{PMs}\) in the neighborhood with 2 and 4 available cores respectively, but our \(\text{VI}\) included two \(\text{VMs}\) 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 \text{getNeighborhood} to return a bigger neighborhood. To this end we update and utilize the minimum amount of \(\text{PMs}\) the returned neighborhood should have \((\min_{\text{PMs}})\).

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 \(\text{PMs}\) with 32 cores, 64GBytes RAM and storage of 6TBybes, linked through 1Gbps Ethernet connections; this configuration was based on \text{IBM x3850 X5} Servers. Our \text{PortLand} switches are connected through 1Gbps links as well. Unless stated otherwise, the simulation is run on a 1,024 \(\text{PM}\) 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 \(w_{\text{cpu}}=w_{\text{ram}}=w_{\text{disk}}=0.33\). We also set the maximum allowed vicinity size to be 5-times the amount of the given \(\text{VI}\)'s \(\text{VMs}\), namely \(\text{size\_limit}=5 \times n_{\text{vms}}\). Lastly, we set the pre-determined amount of maximum allowed reverts for each \(\text{VL}\) 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, \(\text{VI}\) sizes and workloads and 3) examine core and aggregation network switch utilization in each algorithm for communication-heavy \(\text{VIs}\). We want to stress the importance of user-provided bandwidth needs, therefore we compared our \text{VIO} and \text{VIBES} algorithms with the \text{FFD} baseline approach which is network-agnostic.

The \(\text{VIs}\) used in our experiments display three different data flow topologies termed \text{Pipeline, Data Aggregation} and \text{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 \(\text{VIs}\):

1) \(80\%\) \text{Pipeline} \(\text{VIs}\) (Fig.3(a)) with 5 \(\text{VMs}\)
2) \(80\%\) \text{Data Aggregation} \(\text{VIs}\) (Fig.3(b)) with 14 \(\text{VMs}\)
3) \(80\%\) \text{Epigenomics} \(\text{VIs}\) (Fig.3(c)) with 20 \(\text{VMs}\)

As we request \(\text{VIs}\) for deployment, we also randomly select already deployed \(\text{VIs}\) for removal. The removed \(\text{VIs}\) are 30\% of the total \(\text{VIs}\) 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, \(\text{VI}\) size, and the bandwidth levels of the tested \(\text{VIs}\).

- \text{Overall network utilization and average path length}: In this experiment, we examine the network utilization while we gradually place additional \(\text{VIs}\). We choose to take snapshots of the network usage within our infrastructure at different \(\text{PM}\) load levels, as we keep adding \(\text{VIs}\). We term load of a \(\text{PM}\) 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 \text{FFD} by up to 3 times. The \(\text{VI}\) rejection rate established by the three algorithms remains almost identical for \text{FFD} and \text{VIO}, while \text{VIBES} displays up to 2\% higher rejection rate. The tendency of \text{VIBES} to reject more \(\text{VIs}\) 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 \text{VIBES} exhibiting up to 40\% less hops than \text{FFD}, and 20\% less hops against \text{VIO}. This experiment points into the fact that \text{VIBES} places nodes on groups of \(\text{PMs}\) more effectively than its two counterparts.

![Network Utilization per load level](image1.png)

**Fig. 4.** Network Utilization per load level

![Average path length](image2.png)

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

- \text{Scaling the bandwidth requirements of the \(\text{VIs}\)}: Here, we stress the network by increasing the average bandwidth required by the \(\text{VIs}\). We do so by increasing the bandwidth needs by a factor of 3. We gradually increase the amount of \(\text{VIs}\) placed until we reach 90\% resource utilization of the \(\text{PMs}\). Fig.6 shows the network utilization of core, aggregation, and edge switches for the 3 placement policies.

\text{FFD} displays high core and aggregation physical link (PL) usage, since it is treating each \(\text{VM}\) as an isolated entity instead of being part of a \(\text{VI}\). Compared to \text{FFD}, \text{VIO} and \text{VIBES} present 75\% and 95\% respectively less core switch PL usage.
In regards to the aggregation PLs, 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 PMs even though the other neighbouring PMs are heavily loaded. Consequently, some of the VMs of the VI will not be placed in PMs 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 PMs that makes sure we utilize the least amount of core/aggregation PLs. Therefore, even at high loads and bandwidth-demanding VIs, VIBES makes minimal use of core switches. We must also note that during high-bandwidth demand tests FFD was constantly bringing multiple edge PLs 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 PMs) while maintaining the default amount of VMs in the VIs (5 VMs for Pipeline, 14 VMs for Data Aggregation, and 20 VMs for Epigenomics). In order to get an accurate measurement of decision times for both the successful and the unsuccessful placement requests, we deploy VIs until we reach a 20% rejection rate. The decision time for successfully placed VIs are on average 50-80% less than the decision times for rejecting VIs. This is because rejecting a VI involves exploring a much larger search space that requires much time for reverting VIs 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 PMs ($k=16$), 3,456 PMs ($k=24$) and 8,192 PMs ($k=32$). 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 PMs 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 VMs that a VI includes, while keeping the infrastructure size to 1024 PMs. We select three group ranges for the VI sizes:
The placement of VI infrastructure while exploiting hints and the PortLand account user-provided constraints and network usage hints.

12 decision time of under seems to be largely affected by the rejection rate of

We will test our approach in other network topologies such as

VIBES algorithm and further reduce the network utilization.

The Epigenomics workflows have a pre-set topology that we scale by increasing the amount of VMs operating in parallel (spreading a job to more VMs 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 VIs.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we propose two algorithms that perform the placement of Vls made of several VMs 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 VMs. 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