JOINT VIRTUAL MACHINE PLACEMENT AND MIGRATION IN DYNAMIC POLICY-DRIVEN DATA CENTERS

A Thesis

Presented

to the Faculty of

California State University Dominguez Hills

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Hugo Flores J Lucas

Fall 2018

ACKNOWLEDGEMENTS

I would first like to thank Montserrat Hansack, without her support I would not have been able to complete my Masters. I would also like to thank my thesis advisor, Dr. Bin Tang, for all the help and feedback he has given me these past three years and for all the help writing the paper on which this thesis is based on. Lastly I would like to thank my committee members, Dr. Mohsen Beheshti and Dr. Liudong Zuo, for helping me write this thesis.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

PAGE

ABSTRACT

Policy-Driven Data Centers (PDDCs) are data centers in which all Virtual Machine (VM) traffic must traverse a sequence of Middleboxes (MBs). These MB *policies* help guarantee a given level of security and performance for all applications hosted in the PDDC at the cost of increased network traffic and energy consumption. This thesis proposes a new VM optimization framework called VM^{2}P (Virtual-Machine Migration in PDDCs) which, when given an existing PDDC with existing VM placements and policies, will migrate VMs within a PDDC in order to minimize energy consumption while still maintaining all policy constraints. This thesis will also show how the VM placement problem VMP^2 (Virtual-Machine Placement in PDDCs), the problem of distributing VMs into an empty PDDC such that energy consumption is minimized, is actually a special case of VM^{2}P. The performance of VM^{2}P and VMP^2 are evaluated via simulations wherein the algorithms will compete with state-of-the-art policy-agnostic algorithms.

CHAPTER 1

INTRODUCTION

Middleboxes (MBs), also known as "network functions" (NFs), are intermediary network devices such as firewalls and load balancers that perform functions on network traffic other than packet forwarding (Carpenter & Brim, 2002). In recent years, MBs have been introduced into cloud data centers to improve the security and performance of cloud VM applications (Sherry et al., 2012). MBs are usually either purpose-built hardware placed inside the data center (Wang, Qian, Xu, Mao & Zhang, 2011) or they are implemented as software, or as VMs, running on commodity hardware (Gember-Jacobson et al., 2014). Popular examples of MBs include firewalls, intrusion detection systems (IDSs), intrusion prevention systems (IPSs), load balancers, and network address translators (NATs).

In particular, *data center policies* are established in data centers that demand VM traffic traverse a chain of MBs (Joseph, Tavakoli, & Stocia, 2008). Figure 1 shows an example of a possible data center policy. Traffic generated at VM_1 first goes through a firewall, then a load balancer, and lastly a cache proxy before it arrives at VM_2. In doing this, this policy can filter out malicious traffic, divert trusted traffic to avoid network congestion, and finally cache the content to share with other cloud users in the data center. Given the ever-increasing demands for security and performance from diverse cloud user applications, data center policies have become a unified part of the Service

Level Agreement (SLA) of data centers. We refer to such cloud data centers as *policy-driven data centers* (PDDCs).

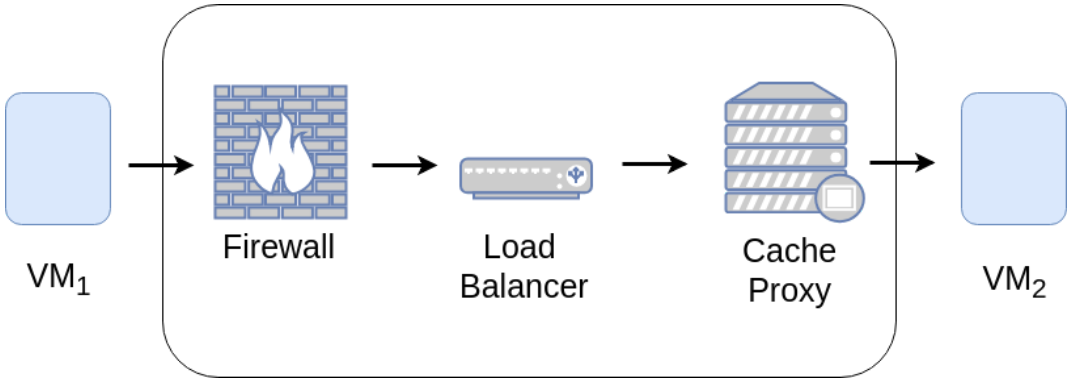


Figure 1. An example of a data center policy.

VM placement (Mann, 2015), which maps VMs to their physical hosts (often called Physical Machines (PMs)), and VM migration (Zhang, Liu, Fu, & Yahyapour, 2018), which moves VMs from one host to another, are popular and effective techniques for network operators to optimize resource expenditures within PDDCs, load balance network traffic, and increase fault tolerance. However despite their similarities VM placement and migration are mostly studied independently by the research community (Mann, 2015; Zhang et al., 2018). This separation may be due to the fact that each strategy attempts to optimize different PDDC parameters. While VM placement strategies primarily attempt to optimize *total communication cost* amongst VMs (e.g. energy cost, data access delay, and bandwidth usage) (Alicherry & Lakshman, 2013; Meng, Pappas, & Zhang, 2010; Li, Wu, Tang, & Lu, 2014; Cohen, Lewin-Eytan, Naor, & Raz, 2013), VM migration strategies mainly attempt to optimize the *total migration cost*

of VMs (e.g. total migration time, downtime, network traffic, and service degradation during live VM migration) (Wang, Li, Zhang, & Jin, 2015; Duong-Ba, Nguyen, Bose, & Tran, 2014; Cui, Yang, Xiao, Wang, & Yan, 2017). As achieving one goal often compromises the other, there is a need for the joint optimization of VM placement and migration in order to achieve optimal resource utilization in cloud data centers. This is especially true for PDDCs as VM communication along MB chains can generate more network traffic and consume more network bandwidth and energy than VM communications in traditional cloud data centers.

In this thesis we provide a new algorithmic framework called VM^{2}P (Virtual-Machine Migration in PDDCs) that jointly optimizes VM placement and migration in PDDCs whose intra-VM traffic can fluctuate. In this model a set of distinct MBs and communicating VM pairs have already been deployed in a PDDC. However, in response to some change to PDDC traffic, there is a need to migrate the VMs in order properly handle the changes in traffic. VM^{2}P integrates communication cost optimization and migration cost optimization into one scheme and therefore achieves joint optimization. We explore the applications of VM^{2}P using two types of PDDC policies: ordered and unordered. Under an *ordered policy* all VM traffic must traverse a given series of MBs in a specific order. In this case, a minimum-cost flow based *policy-aware* VM migration algorithm is used that can solve the migration problem optimally. Under an *unordered policy* all VM traffic must go through a given set of MBs but can do so in any order. In this case the migration problem is NP-hard and thus there does not exist an

efficient polynomial algorithm. Instead, a 2-approximation policy-aware VM migration algorithm is used to migrate VMs. For both ordered and unordered policies, the VM^{2}P framework can be periodically applied to respond to changes in traffic. An example highlighting the difference between ordered and unordered policies can be found in Figure 2.

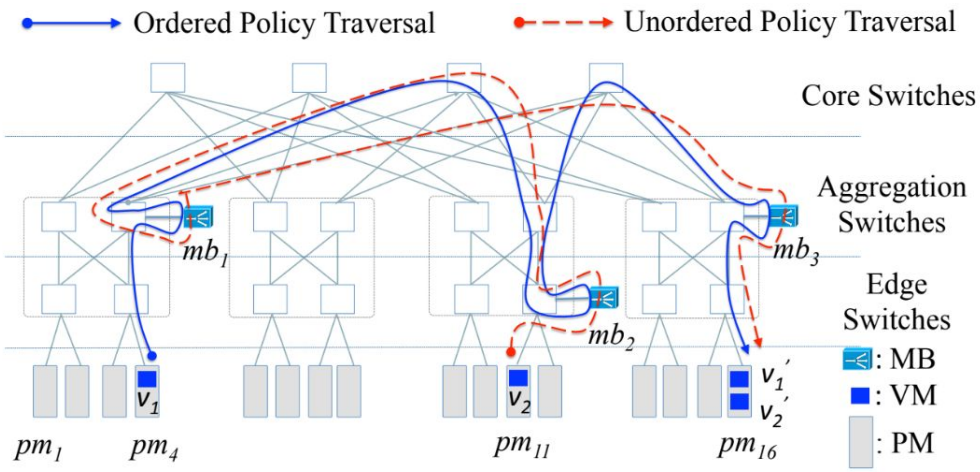


Figure 2. A PDDC with a $k = 4$ Fat Tree topology and two VM pairs.

In this thesis we also study a new VM placement problem called VMP^2: virtual-machine placement in PDDCs. Given a set of MBs already deployed in a PDDC and a data center policy that communicating VM pairs must satisfy, VMP^2 determines how to place a new set of VMs onto empty PMs such that the total communication cost between all VM pairs is minimized. This thesis will show that VMP^2 is a special case of VM^{2}P and therefore techniques designed for VM^{2}P can be adapted to solve VMP^2. Furthermore, by taking advantage of the discrete structures implicit in VMP^2, it is

possible to design a more time efficient policy-aware VM placement algorithm while achieving the same level of performance as the VM migration algorithms.

Via extensive simulations we compare these algorithms with state-of-the-art VM placement and VM migration algorithms. For VM placement we compare VMP^2 with a well-known traffic-aware VM placement algorithm that is oblivious to data center policies (Meng, Pappas, & Zhang, 2010). We show that VMP^2 consistently outperforms it in both ordered and unordered data center policies under different PDDC parameters. For VM migration we compare VM^2P with an algorithm called PLAN (Cui, Tso, Pezaros, Jia, & Zhao, 2017), a policy-aware and network-aware VM management scheme. We show that VM^2P outperforms it for both ordered and unordered policies.

CHAPTER 2

SYSTEM MODEL AND RELATED WORK

System Model

We adopt a fat tree (Al-Fares, Loukissas, & Vahdat, 2008) topology, a popular data center configuration, in order to illustrate the problem and its algorithmic solution. However as the problem and its algorithms must be applicable to any data center topology, we model a PDDC as an undirected general graph $G(V, E)$. Here, $V = V_p \cup V_s$ where $V_p = \{pm_1, pm_2, ..., pm_{|V_p|}\}$ is the set of all PMs and V_s is the set of all switches. E is the set of edges, each connecting either one switch to another switch or a switch to a PM. Figure 2 shows a PDDC of 16 PMs with $k = 4$ where k is the number of ports each switch has.

A set of m either hardware or software based MBs, denoted as $M = \{mb_1, mb_2, ..., mb_m\}$, have been deployed inside a PDDC, with mb_j being installed at switch $sw(j) \in V_s$. For each MB in the PDDC we adopt the commonly used *bump-off-the-wire* design (Joseph, Tavakoli, & Stocia, 2008). This approach takes the dedicated MB hardware out from the physical data path and uses a policy-aware switching layer to explicitly redirect traffic to off-path MBs. Figure 2 shows three MBs mb_1, mb_2, and mb_3 installed to switches using this design.

There are l VM pairs $P = \{(v_1, v'_1), (v_2, v'_2), ..., (v_l, v'_l)\}$ that have already been placed into the PMs of the PDDC. Here we focus on pairwise VM communication as most traffic in cloud data centers is between pairs of VMs (Meng et al., 2010). For any VM pair (v_i, v'_i), its index i is $1 \leq i \leq l$ and we refer to v_i as the *source* VM and v'_i as the *destination* VM. Similarly, the PMs where v_i and v'_i are placed are referred to as the *source* and *destination* PMs respectively. Denote the *communication frequency vector* as $\vec{\lambda} = \langle \lambda_1, \lambda_2, ..., \lambda_l \rangle$, where the index of λ_i is $1 \leq i \leq l$, is the number of communications taking place between v_i and v'_i per unit time. In a dynamic PDDC, the communication frequencies among VM pairs are constantly changing and thus $\vec{\lambda}$ is not a constant vector.

Let $V_m = \{v_1, v'_1, v_2, v'_2, ...v_l, v'_l\}$. One unit amount of PDDC resources is needed to create and execute each VM. Here each unit *resource* is an aggregated characterization of all the hardware resources (i.e., CPU, memory, storage, and bandwidth) needed to create and execute VMs. We leave the more general case where different VMs could need different amount of resources as future work. The *resource capacity* of the i-th PM is denoted as $m(i)$, which means the i-th PM has $m(i)$ *resource slots* where each slot can be used to create and execute one VM. As there are $2 * l$ VMs that each require one resource slot, it must be the case that $\Sigma_{i \in V_p} m(i) \geq 2 * l$. A summary of all the notation used can be found in Table 1.

Energy consumption in any cloud data center, which includes servers, cooling, and networking, is still a big concern (Armbrust et al., 2010). Although the servers and

cooling generally comprise most of the power consumption, studies show that network devices including various switches, routers, and links, can comprise nearly 50% of the overall power consumption in a data center (Abts, Marty, Wells, Klausler, & Liu, 2010). This is especially true for traffic-intensive data centers wherein energy consumption in networking is comparable to the energy consumption of servers (Cohen et al., 2013). We therefore focus on the networking component of energy consumption in PDDCs.

Table 1

Summary of Notation

Notation	**Meaning**	**Notation**	**Meaning**
E	set of all edges	m	number of MBs
V	set of all vertices	$sw(j)$	switch that connects to mb_j
V_p	set of all PMs	P	set of all VM pairs
V_s	set of all switches	(v_i, v'_i)	the i-th VM pair
V_m	set of all VMs	l	number of VM pairs
pm_i	the i-th PM	$\vec{\lambda}$	the set of VM pair communication frequencies
$m(i)$	resource capacity of i-th PM	λ_i	communication frequency of the i-th VM pair
M	set of all MBs	k	number of ports for each PM in a fat-tree topology
mb_j	the j-th MB in a policy chain		

Following (Meng et al., 2010), we model the *communication cost* of any VM pair as the product of the number of links a message sent from the source to the destination VM must traverse inside PDDC and the communication frequency of the VM pair. Let $c(i,j)$ denote the minimum energy cost between any PM (or switch) i and j. We also model the *migration cost* of migrating any VM as being proportional to the number of links a VM must traverse to get from one PM to another. Thus for any VM v migrated from pm_i to another pm_j, its migration cost is equal to $\mu * c(i,j)$ where μ is referred to as the *migration coefficient* and is a weight parameter that quantifies the trade-off between VM communication cost and VM migration cost. The migration coefficient depends on factors such as VM sizes and available network bandwidth and thus must be adjusted by PDDC operators depending on the current PDDC configuration.

Depending on the application requirements, some data center policies require that VM traffic to go through MBs in a strict order. For example in the data center policy shown in Figure 1, as security takes precedence over performance for many cloud applications VM traffic must go through the firewall first for a security check, then the load balancer, and finally the cache proxy for performance improvement. We refer to such policies as *ordered policies* and denote them as $(mb_1, mb_2, ..., mb_m)$. On the other hand as some MB functions are mostly independent from one another, many data center policies are considered satisfied as long as all the MBs in the policies are visited by VM traffic regardless of the order visited. We refer to such policies as *unordered policies* and denote them as $\{mb_1, mb_2, ..., mb_m\}$. An example of such a policy is given in (Li & Qian,

2016) wherein the authors demonstrate that for traffic monitoring a passive monitor MB can be placed before or after a deep packet inspector MB.

Concretely, refer to Figure 2 and assume $\vec{\lambda} = \langle 1, 2 \rangle$. Here (v_1, v'_1) must traverse an ordered policy, denoted by (mb_1, mb_2, mb_3), resulting in communication cost of 10. On the other hand (v_2, v'_2) must traverse an unordered policy, denoted by $\{mb_1, mb_2, mb_3\}$, resulting in communication cost of 16. We refer to the switch where the first MB visited is installed in as the *ingress switch* and the switch where the last MB is installed in as the *egress switch*. Under an ordered policy the ingress switch is always $sw(1)$ and the egress switch is always $sw(m)$. Under an unordered policy, no such constraint exists.

Related Work

VM placement and migration have each been studied intensively. For VM placement, Meng et al. (Meng et al., 2010) designed one of the first *traffic-aware* VM placement algorithms wherein VMs with large communication frequencies are assigned to the same PMs or PMs in close proximity. Alicherry and Lakshman (Alicherry & Lakshman, 2013) designed both optimal and approximation algorithms that place VMs in data centers such that data access latencies are minimized while still satisfying system constraints. Li et al. (Li et al., 2014) studied VM placement that aimed to reduce data center network costs as well as the cost caused by the utilization of physical machines. For VM migration, Wang et al. (Wang et al., 2015) studied how to schedule and allocate

network resources to migrate multiple VMs at the same time and they also designed a fully polynomial time approximation algorithm. Li et al. (Li et al., 2016) developed an energy-efficient VM migration and server consolidation algorithm based on modified particle swarm optimization methods. However, none of work above considers the joint optimization of VM placement and migration.

To the best of our knowledge the only two works that *explicitly* explore joint VM placement and migration optimization are (Duong-Ba et al., 2014) and (Cui et al., 2017). (Duong-Ba et al., 2014) optimizes the sum of VM migration and communication costs as well as server energy cost. It proves that the problem is NP-hard and provides a heuristic algorithm. In contrast, by focusing on the sum of VM migration and communication cost, we are able to design optimal and approximation algorithms therefore providing performance guarantees for VM placement and migration. (Cui et al., 2017) assumed that data center topologies are adaptive, with reconfigurable wireless links or optical circuit switches, and proposed a VM migration algorithm with a constant approximation ratio. Although this a promising approach, its practicality is yet to be demonstrated due to the performance hindrance caused by wireless links and the physical properties of data centers (Ghobadi et al., 2016). We thus focus on traditional data center networks with electrical packet switches arranged in a multi-tier topology. Furthermore, none of the above VM placement and migration research considered data center policies and thus falls short of maximizing the performance and security guarantees gained via the deployment of MBs inside PDDCs.

PACE (E. Li et al., 2013) was one of the first to study the *policy-aware* VM placement problem. It places a sequence of application requests into the cloud, each associated with a prize, a number of compute resources, and MB instances needed. The goal is to maximize the total prize amount of the allocated application requests. However, it only considers one type of MB and thus does not consider the type of policy chains addressed in this paper. The work most related to this thesis is from Cui et al. (Cui et al., 2017) which is one of the first papers that studied policy-aware VM migration. It considered *multiple ordered* policies wherein different VM pairs could take different policies. They showed the problem to be NP-hard and presented utility-based heuristic algorithms as possible solutions. They assumed that migration costs are measured and provided by the hypervisor hosting the VM and therefore the costs can be treated as constants for the optimization. In contrast, by considering the number of hops each VM migrates as the migration cost, this paper's model is network-topology aware and more accurately represents the delay or energy consumption of network traffic induced by VM migration. Unlike their work, which just provided heuristic algorithms for ordered-policies, this thesis considers both ordered and unordered policies and provides optimal, approximate, and heuristic algorithms.

There is another line of orthogonal research that studied the so called MB/VNF placement or migration problem (Cui, Cziva, Tso, & Pezaros, 2016; Zhang et al., 2013; Liu, Li, Zhang, Su, & Jin, 2017; Bhamare et al., 2017). Given VM placements and policy specifications in the cloud data center, it determines the optimal locations to place the

MBs or VNFs such that the performance is optimized. While this is a promising approach to improve service chaining performance, we note that many MBs in cloud data centers and enterprise networks are still purpose-built hardware installed and configured manually by network operators~\cite{unharmful}. As such, once physically deployed, these MBs are not easy to move around to adjust to the dynamic network traffic. In contrast, being mature technologies, VM placement and migration studied in this paper are more flexible.

CHAPTER 3

VM^{2}P: VM MIGRATION IN A PDDC

VM^{2}P assumes that all VM pairs have already been placed inside a PDDC via some algorithm dependant on taking the value of $\vec{\lambda}$ at some particular moment. Such an initial placement is done by a VM *placement function* $p : V_m \rightarrow V_p$, i.e. VM $v \in V_m$ is placed in PM $p(v) \in V_p$. We will solve the VM placement problem and obtain an optimal or approximate p in the next chapter. Generally speaking however, VM^{2}P will work for any initial placement of VMs.

Due to dynamic traffic changes in PDDCs, $\vec{\lambda}$ will constantly change. This means that any placement computed by a VM placement algorithm that takes into consideration the communication frequencies of the VM pairs may not be optimal if any value in the communications vector changes. It could be the case that a previously frequently communicating VM pair now communicates rarely and thus occupies resource slots that would be better suited for highly communicative VM pairs. As VM migration consumes energy in PDDC, the objective of VM^{2}P is to migrate VMs in order to minimize the total energy consumption of VM migration and VM communication. Below we formulate and solve VM^{2}P for both ordered and unordered policies respectively.

Ordered Policies

Problem Formulation

Under an ordered policy, for any VM pair communication the ingress switch is always $sw(1)$ and the egress switch is always $sw(m)$. Given *any* initial VM placement function p, we denote the *total communication cost* of all l VM pairs as $C_c(p)$. Thus:

$$C_c(p) = \sum_{i=1}^{l} \lambda_i * \sum_{j=1}^{m-1} c(sw(j), sw(j+1)) + \sum_{i=1}^{l} \lambda_i * (c(p(v_i), sw(1)) + c(sw(m), p(v'_i)))$$

Next we define a VM *migration function* as $m : V_m \rightarrow V_p$, meaning that VM $v \in V_m$ will be migrated from $p(v) \in V_p$ to $m(v) \in V_p$. Here it need not be the case that v is always migrated, $m(v) = p(v)$ can be true. Let $C_m(m)$ denote the *total migration cost* of all the VM pairs. Thus:

$$C_m(m) = \mu * \sum_{i=1}^{l} (c(p(v_i), m(v_i)) + c(p(v'_i), m(v'_i)))$$

Let $C_t(m)$ denote the *total migration and communication cost* of all the VM pairs *after* VM migration m. Thus $C_t(m) = C_m(m) + C_c(m)$ which means:

$$C_t(m) = \sum_{i=1}^{l} \lambda_i * \sum_{j=1}^{m-1} c(sw(j), sw(j+1)) + \sum_{i=1}^{l} (\mu * c(p(v_i), m(v_i)) + \lambda_i * c(m(v_i), sw(1))) + \sum_{i=1}^{l} (\mu * c(p(v'_i), m(v'_i)) + \lambda_i * c(sw(m), m(v'_i)))$$

The objective of VM^{2}P is to find a VM migration m that minimizes $C_t(m)$ while satisfying resource constraint of PMs: $|\{v \in V_m | m(v) = i\}| \leq m_i, \ \forall i \in V_p$. One thing to note from the equations above is when using an ordered policy, we only need to minimize the sum of the cost from the source VM to the ingress switch and the

destination VM to the egress switch as the MB traversal cost is constant for all VMs. We will now show that VM^{2}P in a PDDC is equivalent to the minimum cost flow (MFC) problem (Ahuja, Magnanti, & Orlin, 1993) in a properly transformed flow network. We do this as the minimum cost flow problem has many solutions which are efficient and optimal.

Minimum Cost Flow Algorithm

Let $G = (V, E)$ be a directed graph. Denote the capacity of edge $(u, v) \in E$ as $c(u, v)$. Denote the cost of edge $(u, v) \in E$ as $d(u, v)$. There exists a source node $s \in V$ that has a supply of amount b. There is also a sink node $t \in V$ with amount b as demand. Denote a flow on edge (u, v) as $f(u, v)$, $f : E \rightarrow \mathbb{R}^+$. Any flow $f(u, v)$ is subject to the following constraints:

1. Capacity constraint:

   $f(u, v) \leq c(u, v), \ \forall (u, v) \in E$.

2. Flow conservation constraint:

   $\Sigma_{u \in V} f(u, v) = \Sigma_{u \in V} f(v, u)$ for each $v \in V$.

The goal of the MCF problem is to find a flow function f such that the total cost of the flow $\Sigma_{(u,v) \in E}(d(u, v) * f(u, v))$ is minimized. The MCF problem can be solved efficiently by many combinatorial algorithms (Ahuja et al., 1993). In this paper, we adopt the scaling push-relabel algorithm proposed by Goldberg (Goldberg, 1997), which works well over a wide range of problem classes. For any flow network, the algorithm has the

time complexity of $O(a^2 * b * log(a * c))$, where a, b, and c are the number of nodes, number of edges, and maximum edge capacity in the flow network respectively.

As shown in Figure 3, we first transform the PDDC $G(V, E)$ into a flow network $G(V', E')$ following the steps below:

1. $V' = \{s\} \cup \{t\} \cup V_m \cup V_p$, where s is the source node and t is the sink node in the flow network
2. $E' = \{(s, v) : v \in V_m\} \cup \{(v, pm_j) : v \in V_m, pm_j \in V_p\} \cup \{(pm_j, t) : pm_j \in V_p\}$. Note that this is a complete bipartite graph between V_m and V_p.
3. For each edge (s, v), set its capacity as one and its cost to zero. For each edge (pm_j, t), set its capacity as m_j (the resource capacity of pm_j) and its cost to zero.
4. For each edge (v_i, pm_j), $v_i \in V_m$, $pm_j \in V_p$, set its capacity to one and its cost as $\mu * c(p(v_i), pm_j) + \lambda_i * c(pm_j, sw(1))$. For each edge (v'_i, pm_j), set its capacity to one and its cost as $\mu * c(p(v'_i), pm_j) + \lambda_i * c(pm_j, sw(m))$.
5. Set the supply of the source and the demand of the sink node to $2 * l$.

Figure 4(a) shows a small fat tree PDDC with $k = 2$. There are two PMs pm_1 and pm_2, two edge switches, two aggregation switches, and one core switch. Each PM has two resource slots; the four resource slots are $\{rs_1, rs_2, rs_3, rs_4\}$. There is an

ordered policy (mb_1, mb_2), with mb_1 installed on edge switch $sw(1)$ and mb_2 on aggregation switch $sw(2)$. There are two VM pairs (v_1, v'_1) and (v_2, v'_2) with communication frequency of one-hundred and one respectively. v_1 and v_2 are initially placed on pm_1 while v'_1 and v'_2 are initially on pm_2. For this example, we will assume $\mu = 1$.

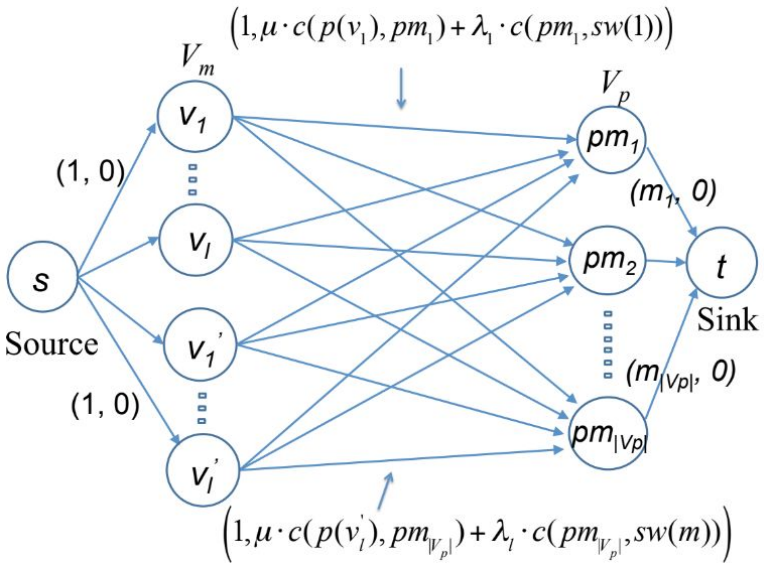


Figure 3. Summary of the PDDC to MFC transformation.

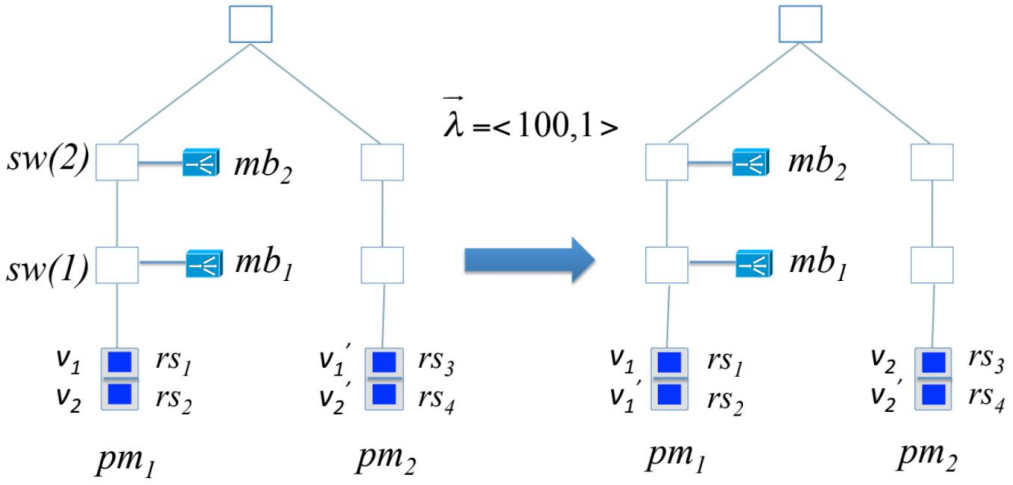


Figure 4. A working example for VM^{2}P.

Creating an MCF representation of this problem, see Figure 6, and solving it results in v_1 and v'_1 migrating to pm_1 and v_2 and v'_2 migrate to pm_2. This migration results in a total communication cost of $100 + 11 + 206 + 4 + 101 = 422$ (note here that 101 is the communication cost between ingress switch $sw(1)$ and egress switch $sw(2)$). This cost can also be double checked against Figure 4(b). Because v_1 is initially located in pm_1 and v'_2 in pm_2, only v'_1 and v_2 actually migrate. Without migration, the total communication cost is $6 * (100 + 1) = 606$. Therefore even though VM migration costs energy, selective migration of VM pairs can save energy via communication cost reductions.

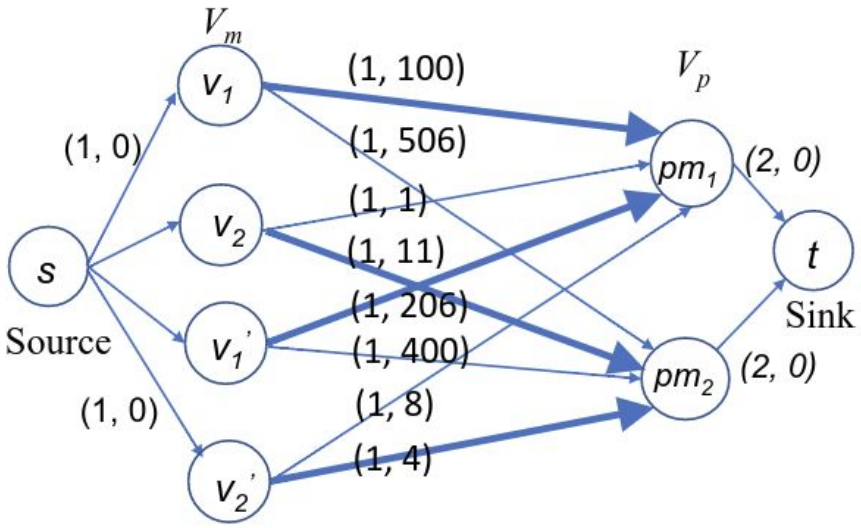


Figure 5. Example MCF transformation.

Theorem I

VM^{2}P under an ordered policy in $G(V, E)$ is equivalent to the minimum cost flow problem in $G'(V', E')$.

Proof. First we show that with the above transformation that sending $2 \cdot l$ amount of flow from s to t ensures that each of the $2 \cdot l$ VMs can be migrated to a PM. In particular since the amount of supply at s is $2 \cdot l$ (Step 5), since the capacity of each edge (s, v) is one (Step 3), and since there are $2 \cdot l$ VMs in V_m, a valid flow of $2 \cdot l$ amount from s to t must consist of one amount on edge (s, v_1), one amount on (s, v_2), ..., one amount on edge (s, v_l), one amount on (s, v'_1), ..., and finally one amount on edge (s, v'_l). Now, since the capacity on each edge (v, pm_j) is one (Step 4), according to flow conservation one amount of flow must come out of any edge v and go into exactly one of the PM pm_j. Thus each VM is migrated to exactly one PM.

Next, we show the above VM migration assignment does not violate the capacity constraint of any PMs. Since the edge capacity of edge (pm_j, t) is m_j (Step 3), no more than m_j amount of flow comes out of each node $mb_j \in V_p$. This guarantees that each PM pm_j will not store more than m_j VMs which satisfies the capacity constraint of each PM.

Finally for the cost note that the edge cost of (v_i, pm_j) is $\mu \cdot c(p(v_i, pm_j) + \lambda'_i \cdot c(pm_j, sw(1))$, which is equal to the migration and communication energy cost for v_i. The cost of edge (v'_j, pm_j) is $\mu \cdot c(p(v'_i), pm_j) + \lambda_i' \cdot c(pm_j, sw(m))$, which is equal to the migration and communication energy cost for v'_i. All other edges in

the flow network have zero cost. This indicates that only the VM migration and communication costs are considered in the minimum cost flow. The minimum cost flow algorithm gives the minimum cost of sending $2 \cdot l$ amount of flow from s_0 to t_o, showing that the corresponding VM migration and communication cost obtained is indeed minimum.

State of the Art VM Migration

Cui et al. (Cui et al., 2017) proposed a policy-aware VM management scheme called PLAN. The core idea behind their techniques is the *utility* of a VM migration. Utility is defined as the VM's communication cost reduction due to migration minus the cost of the migration (Cui et al., 2017, Definition 1). The goal of PLAN is to find a migration scheme that maximizes the *total utility* of all VMs. PLAN is a greedy algorithm (Cui et al., 2017, Algorithm One) that works in rounds. In each round PLAN determines which VM can be migrated to a PM with capacity such that the migration has the highest utility possible any VM yet to be migrated. This continues until all the VMs are either migrated or no migration exists that increases utility.

PLAN, however, is also a heuristic algorithm that does not offer any performance guarantee. Thus it is not clear how well it can perform at all times. We state in Lemma I that its goal is equivalent to our goal of minimizing the total communication and migration cost in VM^2P. Thus these two algorithms can be adequately compared later in this thesis in order to measure the effectiveness of our algorithms.

Lemma I

Minimizing total communication and migration cost $C_t(m)$ is equivalent to maximizing total utility in PLAN.

Proof. Under migration function m, the *utility* of migrating a source VM v_i from its current PM $p(v_i)$ to another PM $m(v_i)$ is defined as the reduction of its communication cost to the ingress switch minus the cost of migrating the VM. Similarly, we can define the utility of migrating a destination VM as the reduction of its communication cost to the egress switch minus the cost of migration. Denote the utility of VM v as $u(v)$. Thus:

$$u(v_i) = \lambda_i \cdot (c(p(v_i), sw(1)) - c(m(v_i, sw(1))) - \mu \cdot c(p(v_i, m(v_i))$$

$$u(v'_i) = \lambda_i \cdot (c(p(v'_i), sw(m)) - c(m(v'_i, sw(m))) - \mu \cdot c(p(v'_i, m(v'_i))$$

We can also defined the *total utility* of migrating all VMs under migration *m* as:

$$U^m = \sum_{i=1}^{l} (u(v_i) + u(v'_i))$$

As stated before the total migration and communication cost after migration is equal to: $C_t(m) = C_m(m) + C_c(m)$. This means that minimizing $C_t(m)$ is equivalent to maximizing $C_c(p) - C_t(m)$ which means:

$$C_c(p) - C_t(m) = \sum_{i=1}^{l} \lambda_i \cdot (c(p(v_i), sw(1)) + c(sw(m), p(v'_i)) - c(m(v_i), sw(1)) -$$

$$c(sw(m), m(v'_i))) - \mu \sum_{v \in V_m} c(p(v), m(v)) = U^m$$

Unordered Policies

Problem Formulation

Under an unordered policy we must define a VM *migration function* $m : V_m \rightarrow V_p$ as well as an MB *traversal function* $\pi^i : [1, 2, ..., m] \rightarrow [1, 2, ..., m]$ for each VM pair (v_i, v'_i). The traversal function is a permutation function indicating that after VM migration the j^{th} MB that (v_i, v'_i) visits is $mb_{\pi^i(j)}$. Let $\vec{\pi} = \langle \pi^1, \pi^2, ... \pi^l \rangle$ and let $C_t(m, \vec{\pi})$ denote the *total migration and communication cost* of all the VM pairs with VM migration m and MB traversal $\vec{\pi}$. Thus:

$$C_t(m, \vec{\pi}) = \sum_{i=1}^{l} \mu \cdot (c\,(p(v_i), m(v_i)) + c\,(p(v'_i), m(v'_i))) +$$

$$\sum_{i=1}^{l} \lambda \cdot (\sum_{j=1}^{m-1} c\,(sw(\pi^i(j)), sw(\pi^i(j+1))) + c\,(m(v_i), sw(\pi^i(1))) + c\,(sw(\pi^i(m)),\ m(v'_i)))$$

The first and second terms in the equation above are the total migration cost and total communication cost of all the VM pairs, respectively. The objective of VM^{2}P under an unordered policy is to find an m and a $\vec{\pi}$ to minimize $C_t(m, \vec{\pi})$ while satisfying the resource constraints of all PMs. We will show that VM^{2}P is NP-hard even for one pair of VMs and propose an approximation algorithm that achieves a total energy cost at most twice that of the optimal configuration.

Theorem II

Even when there is only one pair of VMs (v_1, v'_1) to migrate (i.e., $l = 1$), VM^{2}P is NP-hard.

Proof. We reduce a variation of the *s-t traveling salesman path problem* (TSPP) (Hoogeveen, 1991), which is an NP-hard problem, down to the special case of an

unordered policy in VM^{2}P. Given a complete undirected graph $K = (V_K, E_K)$ with a pair of prespecified vertices $s, t \in V_K$ and edge cost $c : E_K \rightarrow \mathbb{R}^+$ satisfying the *triangle inequality* $c(u, v) \leq c(u, v) + c(v, w)$ for all $u, v, w \in V_K$, the TSPP finds the cheapest *Hamiltonian path* that starts at s, visits each vertex exactly once, and ends at t. When $s = t$, the TSPP becomes the well-known *traveling salesman problem* (TSP) (Cormen, Leiserson, Rivest, & Stein, 2009), which finds the cheapest *Hamiltonian cycle* that starts at s, visits each vertex exactly once, and then returns to s. For clarification, by variation of the TSPP we mean that nodes s and t each have a cost and thus the *cost of the Hamiltonian path or cycle* also includes the costs of nodes s and t.

Given an instance PDDC graph $G(V_p \cup V_s, E)$ where $pm_i \in V_p$ has a cost $c(pm_i)$, we create $|V_p| \cdot (|V_P| + 1)/2$ instances of complete graphs $K^{i,j} = (V_k{}^{i,j}, E^{i,j}{}_k)$, $1 \leq i, j, \leq |V_p|$. Here $V^{i,j}{}_K = \{pm_i, pm_j, sw(1), sw(2), ..., sw(m)\}$ with $pm_i, pm_j \in V_p$. For edge $(u, v) \in E^{i,j}{}_k$, its cost $c(u, v)$ is the cost of the shortest path connecting u and v in G. $c(pm_i)$ is the migration cost of v_1 from its current PM $p(v_1)$ to pm_i and $c(pm_j)$ is the migration cost of v'_1 from its current PM $p(v'_1)$ to pm_j. If $K^{a,b}$ gives the minimum cost Hamiltonian path then migrating v_1 to pm_a and v'_1 to pm_b must be the minimum-cost migration for VM pair (v_1, v'_1) in G, and vice versa.

Figure 6 shows the three complete graphs $K^{1,1}$, $K^{1,2}$, and $K^{2,2}$ that are transformed from the linear PDDC graph G in Figure 4(a) while only considering (v_1, v'_1). $K^{1,2}$ gives the minimum cost Hamiltonian path among the three complete

graphs. It starts from pm_1 and visits $sw(1)$, $sw(2)$, and ends at pm_2 with cost of six. Therefore, migrating v_1 to pm_1 and v'_1 to pm_2 gives the minimum total migration and communication cost of six for (v_1, v'_1). As v_1 is initially located at pm_1 and v_2 at pm_2, both VMs do not need to migrate.

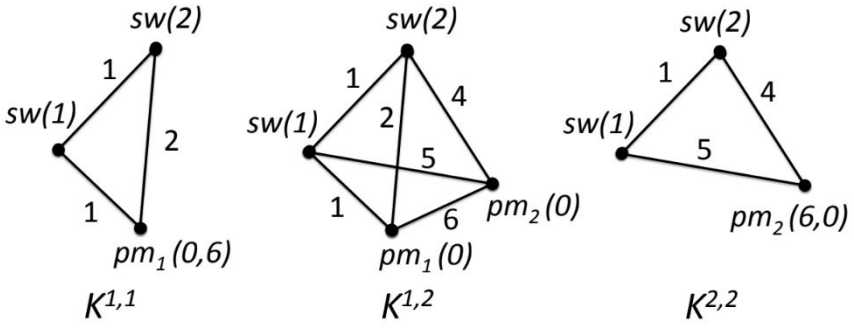


Figure 6. Complete graphs generated from PDDC example figure.

Algorithms

The VM migration algorithm under an unordered policy (Algorithm One) works as follows. For each VM pair (v_i, v'_i) (assuming $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_l$), the algorithm tries to find a target PM such that the total migration cost and communication cost of this pair is minimized. To do that, it constructs $|V_p| \cdot (|V_P + 1|)/2$ complete graphs $K^{i,j}$. For each complete graph the algorithm:

1. Computes its minimum spanning tree.
2. Computes a walk that starts with pm_i, visits all the vertices using each edge *at most twice*, and stops at pm_j.
3. Calculates and stores the cost of the walk including $c(pm_i)$ and $c(pm_j)$.
4. Finds the $K^{i,j}$, which we call $K^{a,b}$, that results in the minimum cost.

5. Migrates v_i to pm_a and v'_i to pm_b.

This continues until all the l VM pairs have been migrated. Constructing any of the $|V_p| \cdot (|V_P + 1|)/2$ $K^{i,j}$ takes $O(m^3)$, computing its MST takes $O(m^2)$, finding a walk takes $O(m)$, and (at least in fat tree topology) the number of switches is bounded by $O(|V_p{}^{2/3}|)$. Therefore if $m = O(|V_p{}^{2/3}|)$, the time complexity of Algorithm One is $O(l \cdot |V_p{}^{2/3}| \cdot m) = O(l \cdot |V_P|^4)$. A description of Algorithm One can be found below.

Example Two

Using the same PDDC as in Figure 4, Algorithm One will migrate v_1 and v_2 to pm_1 and v'_1 and v'_2 to pm_2, which means all the VMs stay in their initial placement. The total migration and communication cost is $101 \cdot 6 = 606$.

Theorem III

Algorithm One achieves a two-approximation when $l = 1$.

Proof. Let the pair of PMs that store (v_1, v'_1) be (pm_a, pm_b) and let W be the walk from pm_a to pm_b in the MST in the complete graph $K^{a,b}$. Let W^* denote the optimal walk from pm_a to pm_b in $K^{a,b}$. The cost of the MST computed in line 10 of Algorithm One is a lower bound on the cost of the optimal walk, $c(MST) \leq c(W^*)$. Since the walk W visits all vertices using each edge of the MST at most twice, $c(W) \leq 2 \cdot c(MST)$. Therefore we have $c(W) \leq 2 \cdot C(W^*)$.

Algorithm 1: VM^{2}P Algorithm for Unordered Policy.
Input: A PDDC with unordered policy $\{mb_1, mb_2, ..., mb_m\}$,
VM pairs P with placement p, $V_p = \{pm_i\}$, $m(i)$.
Output: A migration m and its total energy cost $C_t(m, \overrightarrow{\pi})$.
0. $m = \phi$, $C_t(m, \overrightarrow{\pi}) = 0$, $k = 0$
1. **while** $(k \leq l)$ //not all VM pairs are migrated yet
2. $a = 1$, $b = 1$, $c_{min} = \infty$;
3. **for** $(i = 1; i \leq |V_p|; i++)$
4. **if** $(m(pm_i) == 0)$ **break**;
5. **for** $(j = i; j \leq |V_p|; j++)$
6. **if** $(m(pm_j) == 0)$ **break**;
7. **if** $(i == j \wedge m(s_j) \leq 1)$ **break**;
8. $V_K^{i,j} = \{pm_i, pm_j, sw(1), sw(2), ...sw(m)\}$;
9. Construct complete graph $K^{i,j} = (V_K^{i,j}, E_K^{i,j})$;
10. Compute a minimum spanning tree MST for $K^{i,j}$;
11. Compute a walk W from pm_i to pm_j on MST by visiting all vertices using each edge *at most twice*, and calculate the cost $c_{i,j}$ of W;
12. $c(pm_i) = \mu \cdot c(p(v_i), pm_i)$,
$c(pm_j) = \mu \cdot c(p(v_i^{'}), pm_j)$;
13. $c_{i,j} = \lambda_k \cdot c_{i,j} + c(pm_i) + c(pm_j)$;
14. **if** $(c_{i,j} < c_{min})$
$a = i$, $b = j$, $c_{min} = c_{i,j}$;
15. **end for;**
16. **end for;**
17. $m = m \cup \{(pm_a, pm_b)\}$;
18. $C_t(m, \overrightarrow{\pi})$ += c_{min};
19. $m(pm_a)--$, $m(pm_b)--$;
20. $k++$; // the next VM pair
21. **end while;**
22. **RETURN** m and $C_t(m, \overrightarrow{\pi})$.

Figure 7. Pseudocode for Algorithm One.

CHAPTER 4

VMP^2: VM PLACEMENT IN A PDDC

When new cloud applications are initially submitted and created as VMs, cloud providers must optimize the usage of cloud resources by carefully allocating VMs to PMs. We study this VM placement problem and call it VMP^2 (Virtual Machine Placement in PDDCs). We show that VMP^2 is a special case of VM^{2}P under both ordered and unordered policies. Therefore the algorithms outlined in the previous sections can be used here as well. However, we will also design specialized algorithms for the VM placement problem in this section that are more time-efficient than the migration algorithms while still being optimal (for ordered policies) or two-approximate (for unordered policies).

Ordered Policies

Problem Formulation

The *total communication cost* of all the l VM pairs under VM placement p is denoted as $C_c(p)$ (Equation 1). The objective of VMP^2 is to find a placement p that minimizes $C_c(p)$ while satisfying the resource constraints of all PMs.

Theorem IV

Under an ordered policy, VMP^2 is a special case of VM^{2}P when $\mu = 0$.

Proof. When we plug in $\mu = 0$ into Equation 2 we get the following:

$$C_t(m) = \sum_{i=1}^{l} \lambda_i \cdot \sum_{j=1}^{m-1} c(sw(j), sw(j+1)) + \sum_{i=1}^{l} \lambda_i (c(m(v_i), sw(1)) + c(sw(m), m(v'_i)))$$

Replacing m in the equation above with p, which essentially means we're finding a placement rather than a migration, yields $C_t(m) = C_t(p) = C_c(p)$.

As the VM placement problem is a special case of the migration problem, the MCF algorithm proposed earlier solves VM placement optimally. However by taking advantage of the unique characteristics of the problem in a PDDC, we are able to design a more time efficient algorithm below.

Algorithms

The key to minimizing the communication cost of a VM pair is to find a resource slot as close as possible to the ingress switch for the source VM v_i and a resource slot as close as possible to the egress switch for the destination VM v'_i. Recall that there are $\Sigma_{i \in V_p} m_i \geq 2 \cdot l$ resource slots in the PDDC. For the purposes of our algorithm, each resource slot will now have an ID, an *ingress cost*, and an *egress cost*. A resource slot's *ingress* and *egress cost* is the number of hops a message would have to make to travel from the slot to the ingress and egress switch respectively.

Definitions. We refer to the *l* resource slots that will contain all *l* of the VM pairs in the PDDC as the *ingress* and *egress resource sets* (IRS and ERS). Therefore any slot in the IRS cannot appear in the ERS and vice versa. The cost of the IRS or ERS is the sum of the ingress and egress costs of each slot respectively. A pair of IRS and ERS are optimal if the sum of their costs in minimized. The following algorithm, Algorithm Two,

attempts to find an IRS and ERS, denoted as (I, E), such that their costs is minimized. The complexity of this algorithm is $O(|V_p| \cdot m_{avg} \cdot log(|V_p| \cdot m_{avg}) + l^2)$ where m_{avg} is the average resource capacity of the PMs.

Algorithm 2: VMP^2 Algorithm for Ordered Policy.
Input: A PDDC with ordered policy $(mb_1, mb_2, ..., mb_m)$, VM pairs P, $V_p = \{pm_i\}$, $m(i)$.
Output: A placement p and the total comm. cost $C_c(p)$.
Notations: $\mathcal{I}[i].id$, $\mathcal{I}[i].dist$, $\mathcal{E}[i].id$, $E[i].dist$: ID and cost of the i^{th} resource slot in $\mathcal{I}$ and $\mathcal{E}$.

0. $i = 1$, $j = 1$, $k = 1$, $C_c(p) = 0$, $\mathcal{I}$= ϕ (empty set), $\mathcal{E}$= ϕ, $p = \phi$;
1. Assign all resource slots in the PDDC unique IDs;
2. Sort them in ascending order of their costs to ingress switch $sw(1)$ (and egress switch $sw(m)$), store the first $2l$ resource slots and their costs in an array A (and B);
3. **while** $(k \leq l)$
4. **while** $(A[i].id \in \mathcal{E})$ $i++$;
5. **while** $(B[j].id \in \mathcal{I})$ $j++$;
6. **if** $(A[i].id \neq B[j].id)$
7. $\mathcal{I}[k].id = A[i].id, \mathcal{I}[k].dist = A[i].dist$;
8. $\mathcal{E}[k].id = B[j].id, \mathcal{E}[k].dist = B[j].dist$;
9. $i++$; $j++$;
10. **else**
11. **while** $(A[i+1].id \in \mathcal{E})$ $i++$;
12. **while** $(B[j+1].id \in \mathcal{I})$ $j++$;
13. **if** $(A[i+1].dist \leq B[j+1].dist)$
14. $\mathcal{I}[k].id = A[i+1].id, \mathcal{I}[k].dist = A[i+1].dist$;
15. $\mathcal{E}[k].id = B[j].id, \mathcal{E}[k].dist = B[j].dist$;
16. i += 2; $j++$;
17. **else**
18. $\mathcal{I}[k].id = A[i].id, \mathcal{I}[k].dist = A[i].dist$;
19. $\mathcal{E}[k].id = B[j+1].id, \mathcal{E}[k].dist = B[j+1].dist$;
20. $i++$; j += 2;
21. **end if;**
22. **end if;**
23. $k++$;
24. **end while;**
25. $a = \sum_{j=1}^{m-1} c\big(sw(j), sw(j+1)\big)$;
26. **for** $(1 \leq i \leq l)$
27. Place v_i at resource slot $\mathcal{I}[i].id$;
28. Place $v_i^{'}$ at resource slot $\mathcal{E}[i].id$;
29. $p = p \cup \{(\mathcal{I}[i].id, \mathcal{E}[i].id)\}$;
30. $C_c(p)$ += $\lambda_i * (\mathcal{I}[i].dist + a + \mathcal{E}[i].dist)$;
31. **end for;**
32. **RETURN** p and $C_c(p)$.

Figure 8. Pseudocode for Algorithm Two.

Example Three. Consider the same scenario outlined in Example One, but instead of VM migration we now try to place two sets of VM pairs into a PDDC: (v_1, v'_1) and (v_2, v'_2). Figure 9(a) shows a placement calculated with Algorithm Two. As pm_1 is closer to both mb_1 and mb_2 than pm_2, it could be that $A = B = \{rs_1, rs_2, rs_3, rs_4\}$. However Algorithm Two should create $I.id = \{rs_1, rs_3\}$ and $E.id = \{rs_2, rs_4\}$ meaning that VMs (v_1, v'_1) are placed in pm_1 while (v_2, v'_2) are placed in pm_2. The total communication cost for this configuration is $100 \cdot 4 + 1 \cdot 10 = 410$.

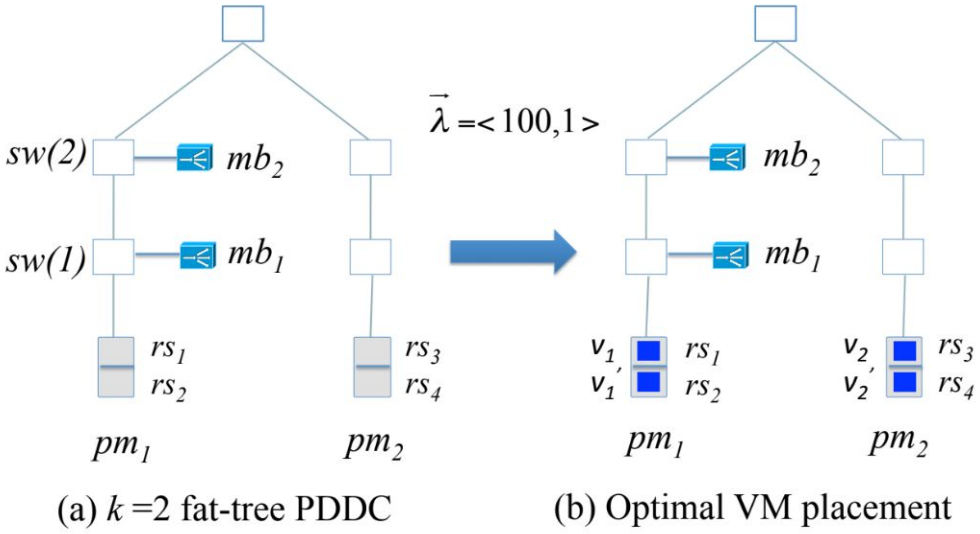


Figure 9. Example for VMP^2.

Theorem V. Algorithm Two finds the VM placement that minimizes the total communication cost for the l VM pairs.

Proof. Assume that Algorithm Two is not optimal and there exists an optimal algorithm called O. Therefore there must exist a problem instance such that the VM placements resulting from both algorithms are different. Let's assume that r, $1 \leq t \leq l$, is the smallest index at which the pair of resource slots store a different pair of VMs for Algorithm Two and O for such an instance. There are two possible cases in this situation.

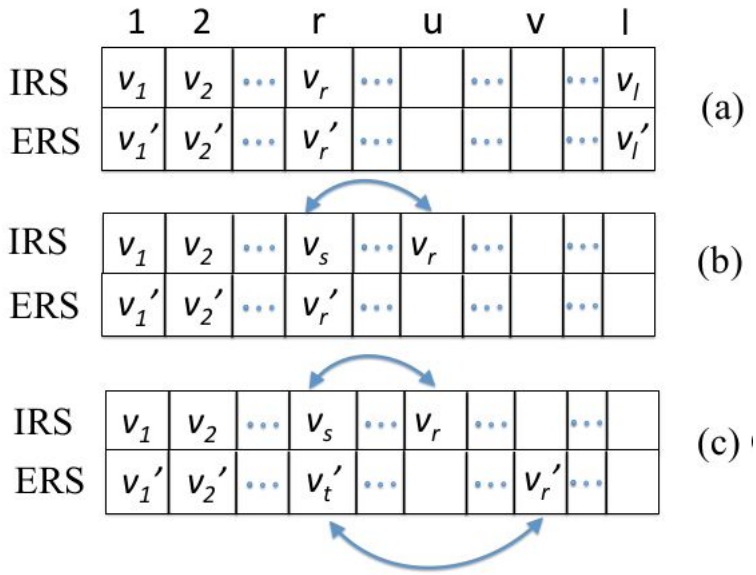


Figure 10. Proof aid for (a) is the order for Algorithm One, (b) is the ordering for Case 1, and (c) is the ordering for Case 2.

Case 1: only one of the two resource slots at *r* differ, i.e. either $I[r].id$ or $E[r].id$ stores a different VM but not both as in Figure 10(b). We denote the VMs placed by O at slot *r* as (v_r, v_s) and those placed by Algorithm Two as (v_r, v'_r). We denote the index where v'_r was placed by O as *u* where *r* < *u*. Because VMs are placed by Algorithm Two in decreasing order with respect to communication frequency, $\lambda_s \leq \lambda_r$. This means that if

we were to switch the positions of v_s and v'_r in the ordering from O, we would reduce the communication cost as the slot at *u* is farther from ingress/egress switch than the slot at *r*. By placing the VM with the higher frequency closer, we reduce the communication cost of the PDDC and thus contradict the notion that O is optimal.

Case 2: both resource slots $I[r].id$ and *E[r].id* store different VMs for both algorithms as in Figure 10(c). Like before the VMs placed by Algorithm Two (v_r, v'_r) at *r* must have a higher communication frequency than those placed by O in the same index. This means that O placed (v_r, v'_r) in slots farther away from either the ingress or egress switches. If we were to swap (v_r, v'_r) back to index *r* in O's ordering we would thus be reducing the communication cost as we are placing VMs with higher communication frequencies in better slots. This is a contradiction as O is suppose to be optimal.

In both cases we are able to swap resources in Optimal to further reduce the energy cost. This contradicts the notion that Optimal is optimal and Algorithm Two is not optimal, therefore Algorithm Two is optimal.

Unordered Policies

Problem Formulation

For unordered policies, VMP^2 needs to find both a placement function *p* and the optimal MB traversal path for each VM pair. With the help of an *MB traversal function* $\pi^i : [1, 2, ..., m] \rightarrow [1, 2, ..., m]$ for a VM pair (v_i, v'_i), we denote the j^{th} MB that a VM

pair must visit as $mb_{\pi^i(j)}$. Given both a placement p and a traversal function π^i, we denote the energy cost for a VM pair as $c^{p,\pi^i}{}_i$. Thus:

$$c^{p,\pi^i}{}_i = \lambda_i \cdot c(p(v_i, sw(\pi^i(1))) + \lambda_i \sum_{j=1}^{m-1} c(sw(\pi^i(j)), sw(\pi^i(j+1))) + \lambda_i \cdot c(sw(\pi^i(m)), p(v_i'))$$

Let $\vec{\pi} = \langle \pi^1, \pi^2, ..., \pi^l \rangle$. The objective of VMP^2 for an unordered policy is to minimize the total communication cost $C_c(p, \vec{\pi}) = \sum_{i=1}^{l} c_i^{p,\pi^i}$ such that the resource constraints of the PDDC PMs are not violated. Below we show that for unordered policies VMP^2 is a special case of VM^{2}P when $\mu = 0$. We then design a VM placement algorithm, Algorithm Three, that is more efficient than Algorithm One while still being two-approximate.

Theorem VI

For an unordered policy, VMP^2 is a special case of VM^{2}P when $\mu = 0$.

Proof. Plug in $\mu = 0$ into Equation Three:

$$C_t(m, \vec{\pi}) = \sum_{i=1}^{l} \lambda \cdot (\sum_{j=1}^{m-1} c\ (sw(\pi^i(j)), sw(\pi^i(j+1))) + c\ (m(v_i), sw(\pi^i(1))) + c\ (sw(\pi^i(m)),\ m(v'_i)))$$

When $\mu = 0$ this equation becomes the total communication cost of a VM placement algorithm.

Algorithms

The difference between Algorithm One and Algorithm Three is that in Algorithm Three we only need to create $|V_p| \cdot (|V_p| + 1)/2$ complete graphs one time and we only

need to sort them with respect to the cost of their Hamiltonian path one time. The running time of Algorithm Three is thus $O(|V_p|^2 \cdot (m^3 + log|V_p|) + l) = O(|V_p|^4 + l)$.

Example Four

Using the same PDDC as in Figure 9, Algorithm Three works as follows. We first sort the PM pairs by the walk cost: $X = \{(1,1,4),(1,2,6),(2,2,10)\}$. Next (v_1, v'_1) are placed into pm_1 while (v_2, v'_2) are placed into pm_2 because (v_1, v'_1) has a higher communication frequency than (v_2, v'_2) and pm_1 has the most efficient walk. Thus the total communication cost is $100 \cdot 4 + 1 \cdot 10 = 410$.

Theorem VII

Algorithm Three achieves a two approximation when $l = 1$.

Proof. The first VM pair in X is (s_1, t_1) (see line 11 of Algorithm Three). Let $pm_a = s_1$, let $pm_b = t_1$, and let W^* denote the optimal walk from pm_a to pm_b in the complete graph $K^{a,b}$. The cost of the MST computed is a lower bound on the cost of the optimal walk, $c(MST) \leq C(W^*)$. Since the walk W found in Algorithm Three visits all vertices using each edge in the MST at most twice, $c(W) \leq 2 \cdot c(MST)$. Therefore we have $c(W) \leq 2 \cdot c(W^*)$.

Algorithm 3: VMP^2 Algorithm for Unordered Policy.
Input: A PDDC with unordered policy $\{mb_1, mb_2, ..., mb_m\}$, VM pairs P, $V_p = \{pm_i\}$, $m(i)$.
Output: A placement p and the total comm. cost $C_c(p, \overrightarrow{\pi})$.

0. $X = \phi$; // stores each PM pair and the cost of the walk
1. **for** $(i = 1; i \leq |V_p|; i++)$
2. **for** $(j = i; j \leq |V_p|; j++)$
3. $V_K^{i,j} = \{pm_i, pm_j, sw(1), sw(2), ...sw(m)\}$;
4. Construct complete graph $K^{i,j} = (V_K^{i,j}, E_K^{i,j})$;
5. Compute a minimum spanning tree MST for $K^{i,j}$;
6. Compute a walk W from pm_i to pm_j on MST by visiting all vertices using each edge *at most twice*, and calculate the cost $c(i, j)$ of W.
7. $X = X \cup \{(i, j, c(i, j))\}$;
8. **end for;**
9. **end for;**
10. Sort X in non-descending order of $c(i, j)$.
Let $X = \{(s_1, t_1, c(s_1, t_1)), (s_2, t_2, c(s_2, t_2)), ...\}$;
11. $i = 1$, $j = 1$; // the i^{th} VM pair $(v_i, v_i^{'})$ is placed at the j^{th} PM pair (s_i, t_j) in X;
$p = \phi$, $C_c(p, \overrightarrow{\pi}) = 0$;
12. **while** $(i \leq l)$ //not all VM pairs are placed yet
13. **while** $\big(m(s_j) \geq 1 \wedge m(t_j) \geq 1\big)$
14. Place v_i at PM s_j, place $v_i^{'}$ at PM t_j;
15. $p = p \cup \{(s_j, t_j)\}$;
16. $C_c(p, \overrightarrow{\pi})$ += $\lambda_i * c(s_j, t_j)$;
17. $m(s_j)--$, $m(t_j)--$;
18. $i++$;
19. **if** $(i > l)$ **break**;
20. **end while;**
21. $j++$; // the next available PM pairs
22. **end while;**
23. **RETURN** p and $C_c(p, \overrightarrow{\pi})$.

Figure 11. Pseudocode for Algorithm Three.

CHAPTER 5

PERFORMANCE EVALUATION

Simulation Setup

We investigate the performance of our policy-aware VM placement and migration algorithms in this chapter. We will refer to ordered policy Algorithm Two as *Optimal* and to the unordered policy Algorithm Three as *Approximation*. Our simulation will consider a PDDC with a $k = 8$ Fat Tree topology and 128 PMs. In the VMP^2 simulations we vary two parameters:

1. The number of MBs: $m = 1, 3, 5$
2. The number of VM Pairs: $l = 500, 1000, 1500, 2000$

The communication frequency of each VM pair will always be in the range $[1, 1000]$. For all simulation plots, each bar represents the average of 20 independent runs with error bars indicating the 95% confidence interval.

Traffic Aware VM Placement

Meng et al. (Meng et al., 2010) proposed a traffic-aware VM placement algorithm that optimizes the placement of VMs into PMs. The authors observed that VMs with a large amount of communication traffic should be assigned to PMs as close as possible, ideally even the same PM. We refer to the algorithm outlined in this paper as *TrafficAware*. We use this algorithm in the upcoming section as a benchmark for own algorithms.

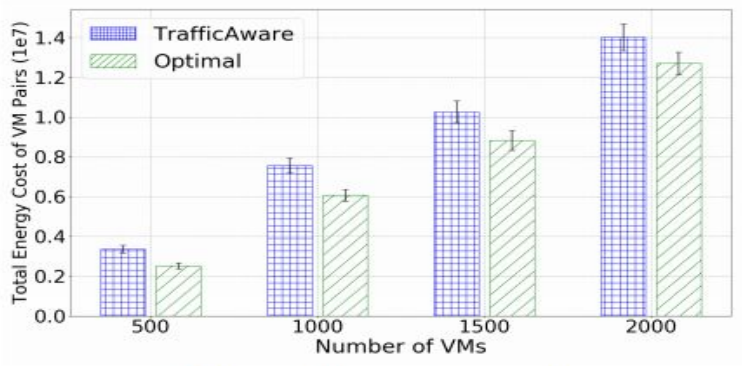
TrafficAware is policy-oblivious though, not taking into account the policies present in the PDDC. In an ordered policy TrafficAware considers all the VM pairs in their descending order of communication frequencies, and places each VM pair to the PM that is closest to the ingress switch until all the VM pairs are placed. In unordered-policy, it works as Algorithm Three while only considering the case of $pm_1 = pm_2$, as TrafficAware always places each VM pair in the same PM if possible. We compare our algorithms with TrafficAware for both ordered- and unordered-policies, and show that our algorithms consistently outperform TrafficAware.

VMP^2 Results

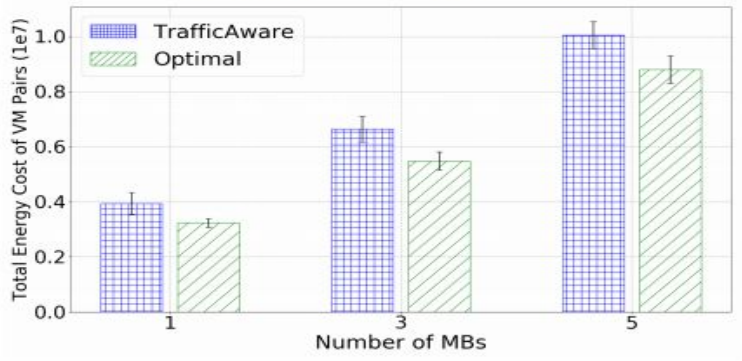
Ordered Policies

For our first simulation, we vary the number of VM pairs l from 500 to 2,000, in increments of 500, while maintaining the number of MBs constant at $m = 3$. Here we also keep the resource capacity of each PM at $rc = 40$. The results of this scenario are outlined in Figure 12(a). From these results it is obvious that as the number of VM pairs increases the total energy cost for both algorithms increases but the cost of the Optimal algorithm is always at least $15 - 20\%$ below that of the TrafficAware algorithm. In Figure 12(b) we see the results of varying the number of MBs while keeping $l = 1000$ and $rc = 40$. Again we see here how important it is to keep policies in mind when placing VMs as the Optimal algorithm outperforms the TrafficAware algorithm by an average of 15%. Lastly in Figure 12(c) we see how well Optimal performs when the number of resource slots vary per physical machine. Unlike the other scenarios where the

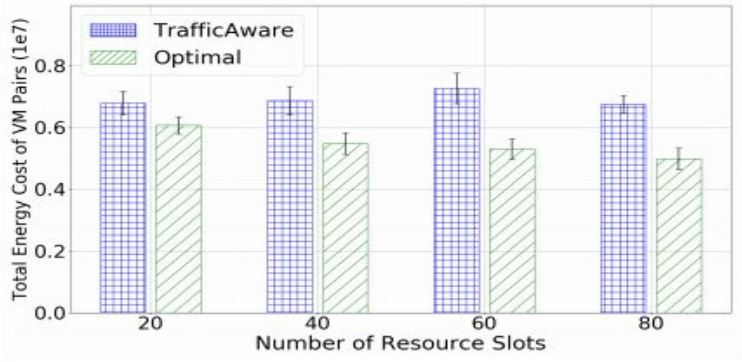
gap between Optimal and TrafficAware stays relatively the same, in this scenario the performance of Optimal relative to Trafficware increases as more resource slots become available.



(a) Varying l. $m = 3$, $rc = 40$.



(b) Varying m. $l = 1000$, $rc = 40$.



(c) Varying rc. $l = 1000$, $m = 3$.

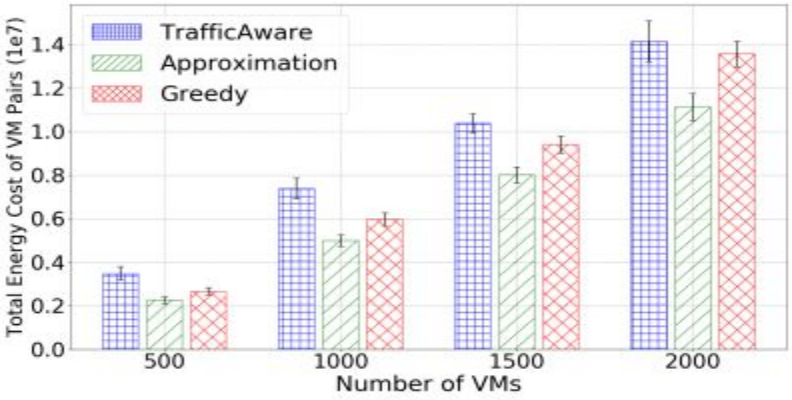
Figure 12. Simulation results for VMP^2 for ordered policies.

Unordered Policies

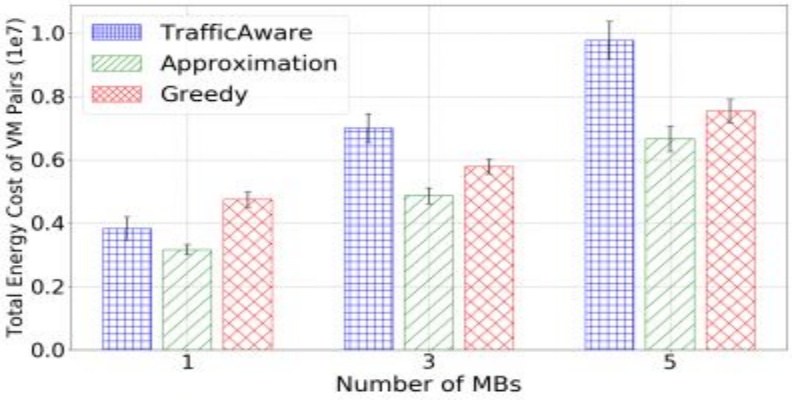
In an unordered policy, the ingress and egress switches are not constant for every VM pair as the traversal of the MB set can vary from pair to pair. Therefore for the case of unordered policies, we devise another benchmark algorithm to compare the performance of the Approximation algorithm. The *Greedy* algorithm is a new algorithm that places the first VM in a pair into the resource slot that is closest to an MB in the policy chain. Next, the algorithm calculates the shortest path through the remaining MBs in the policy chain and places the other VM in the pair into the resource slot closest to the last MB visited. One can see how this algorithm performs against the TrafficAware and Approximation algorithms in Figure 13(a) and Figure 13(b). It is obvious that Approximation outperforms TrafficAware and Greedy for all cases.

Comparison

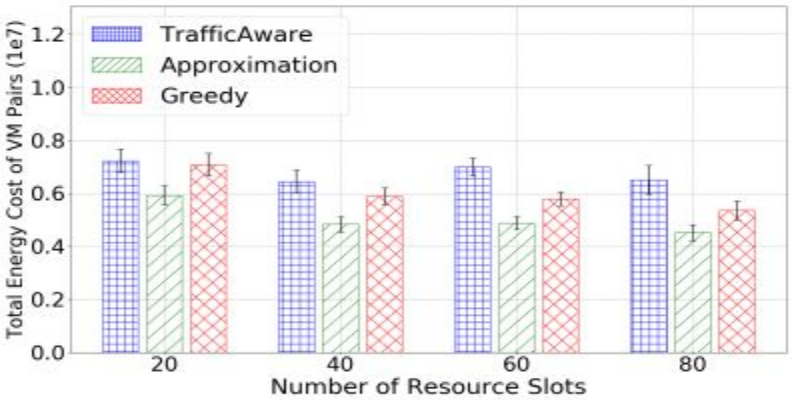
We compare the energy consumptions for all VM pairs for both ordered and unordered policies. We observe that unordered policies yield less energy costs than ordered policies for all VM pairs. We define the *performance difference* (PD) as the energy cost difference between ordered and unordered policies divided by energy cost of the ordered policy. In general, unordered policies cost around 20-30% less energy than ordered policies as shown in Figure 14. This is because unlike ordered policies, wherein each VM must traverse the MBs in a specific order, unordered policies allow VM pairs to



(a) Varying l. $m = 3$, $rc = 40$.



(b) Varying m. $l = 1000$, $rc = 40$.



(c) Varying rc. $l = 1000$, $mb = 3$.

Figure 13. Simulation results for VMP[2] for unordered policies

choose the order of the MBs traversal in order to reduce their energy cost. Note that as TrafficAware is totally oblivious to policies, its performance stays relatively constant.

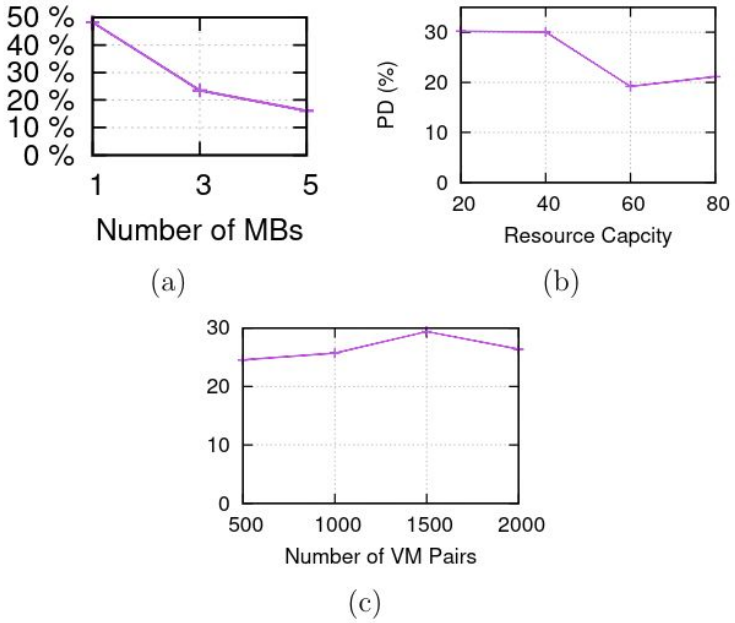


Figure 14. Performance difference plot between ordered and unordered VMP^2.

VM^{2}P Results

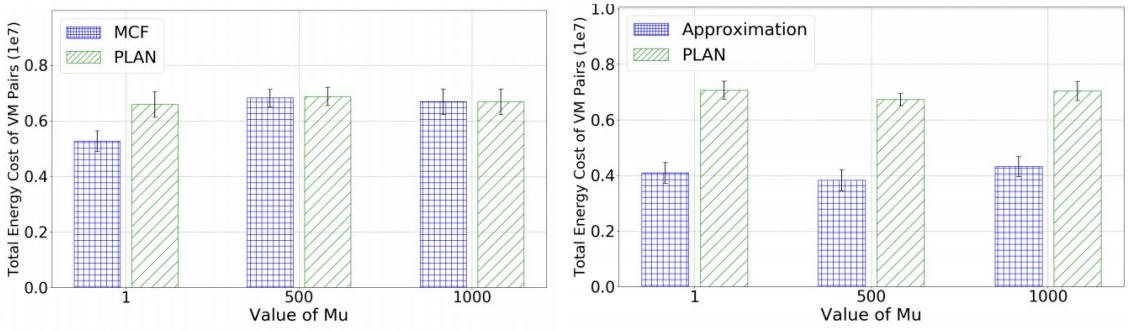


Figure 15. Simulation results for VM^{2}P.

Finally we compare our VM migration algorithms with PLAN for both ordered and unordered policies. We consider 1000 VM pairs and 3 MBs in a $k = 8$ data center where each PM has resource capacity of 70. We vary the migration coefficient μ from 1, 500, to 1000, which is comparable to the range of VM communication frequencies. Figure 15(a) shows that under an ordered policy, our MCF-based optimal migration algorithm performs much better for μ equal to 1 and only marginally better for all other values. Figure 15(b) shows the under an unordered policy, our approximation algorithm outperforms PLAN by at least 40%. This indirectly validates the optimality and approximately of our designed VM migration algorithms.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis we proposed an algorithmic framework called VM^2P, that jointly optimizes VM placement and VM migration in PDDCs. PDDCs have become important infrastructure for cloud computing as the MB-based policies provide the cloud user applications with security and performance guarantees. In particular, we uncover a suite of new algorithmic problems that migrate and place VMs inside PDDCs while respecting the existing policies and also minimize the total energy consumption of the VM applications. We solved VM^2P by designing both optimal and approximation algorithms under ordered and unordered policies, respectively, and show that VM placement is a special case of VM^2P. For future work, we are working on real data traces from production data centers to further validate our algorithms. We will also study if the optimality and approximability of our algorithms still hold when different VMs require different amount of resources. Finally we will focus on network function virtualization, when both MBs and VMs can be placed easily inside PDDCs, to design a holistic and synergistic MB and VM placement approach to achieve ultimate energy-efficiency in PDDCs.

REFERENCES

REFERENCES

Abts, D., Marty, M. R., Wells, P. M., Klausler, P., & Liu, H. (2010, 06). Energy proportional datacenter networks. *ACM SIGARCH Computer Architecture News, 38*(3), 338. doi:10.1145/1816038.1816004

Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network flows: Theory, algorithms, and applications*. Prentice-Hall.

Al-Fares, M., Loukissas, A., & Vahdat, A. (2008). A scalable, commodity data center network architecture. *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication - SIGCOMM '08*. doi:10.1145/1402958.1402967

Alicherry, M., & Lakshman, T. V. (2013, 04). Optimizing data access latencies in cloud systems by intelligent virtual machine placement. *2013 Proceedings IEEE INFOCOM*. doi:10.1109/infcom.2013.6566850

Armbrust, M., Stoica, I., Zaharia, M., Fox, A., Griffith, R., Joseph, A. D., . . . Rabkin, A. (2010, 04). A view of cloud computing. *Communications of the ACM, 53*(4), 50. doi:10.1145/1721654.1721672

Bhamare, D., Samaka, M., Erbad, A., Jain, R., Gupta, L., & Chan, H. A. (2017, 04). Optimal virtual network function placement in multi-cloud service function chaining architecture. *Computer Communications, 102*, 1-16. doi:10.1016/j.comcom.2017.02.011

Carpenter, B., & Brim, S. (2002, 02). Middleboxes: Taxonomy and Issues.

doi:10.17487/rfc3234

Cohen, R., Lewin-Eytan, L., Naor, J. S., & Raz, D. (2013, 04). Almost optimal virtual machine placement for traffic intense data centers. *2013 Proceedings IEEE INFOCOM*. doi:10.1109/infcom.2013.6566794

Cormen, T. H., & Leiserson, C. E. (2009). *Introduction to algorithms, 3rd edition*.

Cui, L., Cziva, R., Tso, F. P., & Pezaros, D. P. (2016, 04). Synergistic policy and virtual machine consolidation in cloud data centers. *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. doi:10.1109/infocom.2016.7524354

Duong-Ba, T., Nguyen, T., Bose, B., & Tran, T. (2014, 12). Joint virtual machine placement and migration scheme for datacenters. *2014 IEEE Global Communications Conference*. doi:10.1109/glocom.2014.7037154

Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., & Akella, A. (2014). OpenNF. *Proceedings of the 2014 ACM Conference on SIGCOMM - SIGCOMM '14*. doi:10.1145/2619239.2626313

Goldberg, A. V. (1992). *An efficient implementation of a scaling minimum-cost flow algorithm*. Stanford University, Dept. of Computer Science.

Hoogeveen, J. A. (1990). *Analysis of Christofides' heuristic: Some paths are more difficult than cycles*. Centrum voor Wiskunde en Informatica.

Joseph, D. A., Tavakoli, A., & Stoica, I. (2008). A policy-aware switching layer for data centers. *Proceedings of the ACM SIGCOMM 2008 Conference on Data*

Communication - SIGCOMM '08. doi:10.1145/1402958.1402966

Li, H., Zhu, G., Cui, C., Tang, H., Dou, Y., & He, C. (2015, 07). Energy-efficient migration and consolidation algorithm of virtual machines in data centers for cloud computing. *Computing, 98*(3), 303-317. doi:10.1007/s00607-015-0467-4

Li, L. E., Liaghat, V., Zhao, H., Hajiaghayi, M., Li, D., Wilfong, G., . . . Guo, C. (2013, 04). PACE: Policy-Aware Application Cloud Embedding. *2013 Proceedings IEEE INFOCOM*. doi:10.1109/infcom.2013.6566849

Li, X., Wu, J., Tang, S., & Lu, S. (2014, 04). Let's stay together: Towards traffic aware virtual machine placement in data centers. *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. doi:10.1109/infocom.2014.6848123

Li, X., & Qian, C. (2016, 01). A survey of network function placement. *2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. doi:10.1109/ccnc.2016.7444915

Liu, J., Li, Y., Zhang, Y., Su, L., & Jin, D. (2017, 07). Improve Service Chaining Performance with Optimized Middlebox Placement. *IEEE Transactions on Services Computing, 10*(4), 560-573. doi:10.1109/tsc.2015.2502252

Mann, Z. Á. (2015, 08). Allocation of Virtual Machines in Cloud Data Centers—A Survey of Problem Models and Optimization Algorithms. *ACM Computing Surveys, 48*(1), 1-34. doi:10.1145/2797211

Meng, X., Pappas, V., & Zhang, L. (2010, 03). Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. *2010 Proceedings IEEE*

INFOCOM. doi:10.1109/infcom.2010.5461930

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., & Sekar, V. (2012, 09). Making middleboxes someone else's problem. *ACM SIGCOMM Computer Communication Review, 42*(4), 13. doi:10.1145/2377677.2377680

Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., & Shenker, S. (2004). Middleboxes no longer considered harmful. OSDI'04 Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, 6, 15-15. Retrieved September 29, 2018, from https://dl.acm.org/citation.cfm?id=1251269.

Wang, H., Li, Y., Zhang, Y., & Jin, D. (2015, 04). Virtual machine migration planning in software-defined networks. *2015 IEEE Conference on Computer Communications (INFOCOM)*. doi:10.1109/infocom.2015.7218415

Wang, Z., Qian, Z., Xu, Q., Mao, Z., & Zhang, M. (2011, 10). An untold story of middleboxes in cellular networks. *ACM SIGCOMM Computer Communication Review, 41*(4), 374. doi:10.1145/2043164.2018479

Xiao, S., Cui, Y., Wang, X., Yang, Z., Yan, S., & Yang, L. (2016, 11). Traffic-aware virtual machine migration in topology-adaptive DCN. *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. doi:10.1109/icnp.2016.7784441

Zhang, F., Liu, G., Fu, X., & Yahyapour, R. (2018). A Survey on Virtual Machine Migration: Challenges, Techniques, and Open Issues. *IEEE Communications*

Surveys & Tutorials, 20(2), 1206-1243. doi:10.1109/comst.2018.2794881

Zhang, Y., Beheshti, N., Beliveau, L., Lefebvre, G., Manghirmalani, R., Mishra, R., . . . Tatipamula, M. (2013, 10). StEERING: A software-defined networking for inline service chaining. *2013 21st IEEE International Conference on Network Protocols (ICNP)*. doi:10.1109/icnp.2013.6733615

APPENDICES

APPENDIX A

ALGORITHM ONE CODE

```python
from __future__ import print_function
from ortools.graph import pywrapgraph


class MCF:

    """Algorithm One

    Creates a matrix representation of a graph topology in order to solve the MCF problem using Google OR tools.
    """

    def __init__(self, graph_topology, virtual_machine_pairs, mu):
        """
        Constructor.

        :param graph_topology:          the Topology object used to simulate a PDDC
        :param virtual_machine_pairs:   a list of VirtualMachinePair objects already placed inside the PDDC
        :param mu:                      the value of the MU migration parameter
        """
        self.graph_topology = graph_topology
        self.virtual_machine_pairs = virtual_machine_pairs
        self.mu = mu

        self.start_nodes = []
        self.end_nodes = []
        self.capacities = []
        self.unit_costs = []
        self.supplies = []

    def build_graph(self):
        """
        Assigns each PM and VM an index in order to create a MCF representation of the VMP2 and VM2P problem.

        :return: None
        """
        start_nodes = []
        end_nodes = []
        capacities = []
        unit_costs = []
        supplies = []

        sink_node_number = 2 * len(self.virtual_machine_pairs) + len(self.graph_topology.physical_machines) + 1
        first_mb = self.graph_topology.middleboxes[0]
        last_mb = self.graph_topology.middleboxes[-1]

        # Source to SourceVMs
        source_vm_index_start = source_vm_index_end = 1
        for _ in self.virtual_machine_pairs:
            start_nodes.append(0)
            end_nodes.append(source_vm_index_end)
            capacities.append(1)
            unit_costs.append(0)

            source_vm_index_end += 1
        source_vm_index_end -= 1

        # Source to DestinationVMs
        destination_vm_index_start = destination_vm_index_end = source_vm_index_end + 1
        for _ in self.virtual_machine_pairs:
            start_nodes.append(0)
            end_nodes.append(destination_vm_index_end)
            capacities.append(1)
            unit_costs.append(0)

            destination_vm_index_end += 1
        destination_vm_index_end -= 1

        # PMs to Sink
        pm_index_start = pm_index_end = destination_vm_index_end + 1
        for pm in self.graph_topology.physical_machines:
            start_nodes.append(pm_index_end)
            end_nodes.append(sink_node_number)
            capacities.append(pm.get_remaining_capacity())
            unit_costs.append(0)

            pm_index_end += 1
```

```
        # Source VMs to Hosts
        for vm_pair, vm_index_value in zip(self.virtual_machine_pairs, range(source_vm_index_start,
                                                                             source_vm_index_end + 1)):

            source_vm = vm_pair.get_pair_vms()[0]
            for pm, pm_index_value in zip(self.graph_topology.physical_machines, range(pm_index_start,
                                                                                       pm_index_end + 1)):

                vm_host_migration_cost = self.graph_topology.calculate_distance(source_vm.virtual_machine_host, pm)
                ingress_cost = self.graph_topology.calculate_distance(pm, first_mb.parent_switch)
                total_cost = (self.mu * vm_host_migration_cost) + (vm_pair.new_frequency * ingress_cost)

                start_nodes.append(vm_index_value)
                end_nodes.append(pm_index_value)
                capacities.append(1)
                unit_costs.append(int(total_cost))

        # Destination VMs to Hosts
        for vm_pair, vm_index_value in zip(self.virtual_machine_pairs, range(destination_vm_index_start,
                                                                             destination_vm_index_end + 1)):

            dst_vm = vm_pair.get_pair_vms()[1]
            for pm, pm_index_value in zip(self.graph_topology.physical_machines, range(pm_index_start,
                                                                                       pm_index_end + 1)):
                vm_host_migration_cost = self.graph_topology.calculate_distance(dst_vm.virtual_machine_host,
                                                                                pm)
                egress_cost = self.graph_topology.calculate_distance(pm, last_mb.parent_switch)
                total_cost = (self.mu * vm_host_migration_cost) + (vm_pair.new_frequency * egress_cost)

                start_nodes.append(vm_index_value)
                end_nodes.append(pm_index_value)
                capacities.append(1)
                unit_costs.append(int(total_cost))

        for ii in range(0, sink_node_number + 1):
            if ii == 0:
                supplies.append(2 * len(self.virtual_machine_pairs))
            elif ii == sink_node_number:
                supplies.append(-2 * len(self.virtual_machine_pairs))
            else:
                supplies.append(0)

        self.start_nodes = start_nodes
        self.end_nodes = end_nodes
        self.capacities = capacities
        self.unit_costs = unit_costs
        self.supplies = supplies

    def get_cost(self):
        min_cost_flow = pywrapgraph.SimpleMinCostFlow()
        for i in range(0, len(self.start_nodes)):
            min_cost_flow.AddArcWithCapacityAndUnitCost(self.start_nodes[i], self.end_nodes[i], self.capacities[i],
                                                        self.unit_costs[i])

        for i in range(0, len(self.supplies)):
            min_cost_flow.SetNodeSupply(i, self.supplies[i])

        if min_cost_flow.Solve() == min_cost_flow.OPTIMAL:
            return min_cost_flow.OptimalCost()
        else:
            raise ValueError("Something went wrong!")
```

APPENDIX B

ALGORITHM TWO CODE

```
import vm_placement.utils


class MemorySlot:

    """
    This class represents a resource slot inside a PM inside a PDDC.
    """

    def __init__(self, number, physical_machine, cost):
        """
        Constructor.

        :param number:              the resource slot number, unique inside each PM
        :param physical_machine:    the PhysicalMachine object this slot belongs to
        :param cost:                the cost, be it ingress or egress, assoicated with this
                                    object
        """
        self.slot_number = number
        self.slot_parent = physical_machine
        self.slot_cost = cost

        self.virtual_machine = None

    def __eq__(self, other):
        return (other.slot_parent == self.slot_parent) and (other.slot_number == self.slot_number)


class Optimal:

    """
    This class represents Algorithm Two.
    """

    def __init__(self, graph_topology, virtual_machine_pairs):
        self.graph_topology = graph_topology
        self.virtual_machine_pairs = virtual_machine_pairs

    def allocate_pairs(self):
        ingress_slots = self.create_memory_slots(0)
        egress_slots = self.create_memory_slots(-1)

        egress_list, ingress_list = [], []
        egress_index, ingress_index = 0, 0
        for vm_pair in self.virtual_machine_pairs:
            src_vm, dst_vm = vm_pair.get_pair_vms()

            # Find the next free ingress slot
            while ingress_slots[ingress_index] in ingress_list:
                ingress_index += 1

            # Find the next free egress slot
            while egress_slots[egress_index] in egress_list:
                egress_index += 1
```

```
            # If the two slots are completely different, place the vms
            if ingress_slots[ingress_index] != egress_slots[egress_index]:
                i_slot, e_slot = ingress_slots[ingress_index], egress_slots[egress_index]
                ingress_to_remove, egress_to_remove = ingress_index, egress_index
            else:
                # Find next pair of slots and determine which is the most efficient
                next_ingress_index, next_egress_index = ingress_index + 1, egress_index + 1
                while ingress_slots[next_ingress_index] in ingress_list:
                    next_ingress_index += 1
                while egress_slots[next_egress_index] in egress_list:
                    next_egress_index += 1

                if ingress_slots[next_ingress_index].slot_cost <=
 egress_slots[next_egress_index].slot_cost:
                    # The next ingress slot is more efficient, place src vm into that slot
                    i_slot, e_slot = ingress_slots[next_ingress_index], egress_slots[egress_index]
                    ingress_to_remove, egress_to_remove = next_ingress_index, egress_index
                else:
                    # The next egress slot is more efficient, place dst vm into that slot
                    i_slot, e_slot = ingress_slots[ingress_index], egress_slots[next_egress_index]
                    ingress_to_remove, egress_to_remove = ingress_index, next_egress_index

            i_slot.virtual_machine = src_vm
            e_slot.virtual_machine = dst_vm

            ingress_list.append(i_slot)
            egress_list.append(e_slot)

            self
.remove_used_slots(ingress_to_remove, egress_to_remove, ingress_slots, egress_slots)

        self.place_vms_by_slot(egress_list)
        self.place_vms_by_slot(ingress_list)

    def get_cost(self):
        return vm_placement.utils.get_cost(self.graph_topology, self.virtual_machine_pairs)

    @staticmethod
    def place_vms_by_slot(slot_map):
        for slot in slot_map:
            host = slot.slot_parent
            host.add_new_virtual_machine(slot.virtual_machine)

    @staticmethod
    def remove_used_slots(ingress_index, egress_index, ingress_slots, egress_slots):
        i_slot = ingress_slots[ingress_index]
        e_slot = egress_slots[egress_index]

        ingress_slots.remove(i_slot)
        egress_slots.remove(i_slot)

        ingress_slots.remove(e_slot)
        egress_slots.remove(e_slot)
```

```
    def create_memory_slots(self, mb_index):
        switch = self.graph_topology.middleboxes[mb_index].parent_switch
        ordered_hosts = sorted(self.graph_topology.physical_machines,
                               key=lambda x: self.graph_topology.calculate_distance(vertex_one=x,
                                                                                    vertex_two=
switch))

        slot_list = []
        for host in ordered_hosts:
            for i in range(0, host.get_remaining_capacity()):
                cost = self.graph_topology.calculate_distance(vertex_one=host, vertex_two=switch)
                slot_list.append(MemorySlot(number=i, physical_machine=host, cost=cost))
        return slot_list
```

APPENDIX C

ALGORITHM THREE CODE

```python
import igraph
import copy


class MBPath:
    def __init__(self):
        self.path_sequence = []
        self.path_count_map = {}

        self.is_path_valid = True
        self.last_vertex = None

    def add_vertex(self, vertex):
        self.path_sequence.append(vertex)
        self.last_vertex = vertex

        self.path_count_map[vertex.identifier] = self.path_count_map.get(vertex.identifier, 0) + 1
        self.update_validity()

    def is_complete(self, start_vertex, end_vertex, mb_list):
        if not self.is_path_valid:
            return False
        if self.path_sequence[0].identifier != start_vertex.identifier:
            return False
        if self.path_sequence[-1].identifier != end_vertex.identifier:
            return False
        for mb in mb_list:
            if self.path_count_map.get(mb.identifier) is None:
                return False
        return True

    def update_validity(self):
        for key in self.path_count_map.keys():
            if self.path_count_map[key] > 3:
                self.is_path_valid = False
                break
```

```python
    def cost_of_path(self, graph):
        cost = 0
        for i in range(1, len(self.path_sequence)):
            past_vertex = graph.vs.find(name=self.path_sequence[i-1].identifier)
            current_vertex = graph.vs.find(name=self.path_sequence[i].identifier)

            link = graph.es.select(_between=([past_vertex.index], [current_vertex.index]))
            cost += link["weight"][0]
        return cost


class PMTuple:
    def __init__(self, source_host_index, destination_host_index, traversal_cost):
        self.src_index = source_host_index
        self.dst_index = destination_host_index
        self.cost = traversal_cost


class Approximation:
    def __init__(self, graph_topology, virtual_machine_pairs):
        self.graph_topology = graph_topology
        self.virtual_machine_pairs = virtual_machine_pairs

        self.total_cost = 0

    def allocate_pairs(self):
        pm_tuple_set = self.construct_pm_tuple_set()
        pm_tuple_set.sort(key=lambda t: t.cost)

        self.virtual_machine_pairs.sort(key=lambda x: x.frequency, reverse=True)
        for vm_pair in self.virtual_machine_pairs:
            src_vm, dst_vm = vm_pair.get_pair_vms()

            while True:
                tuple_obj = pm_tuple_set[0]
                src_index, dst_index, cost = tuple_obj.src_index, tuple_obj.dst_index, tuple_obj.cost
                src_host, dst_host = self.graph_topology.physical_machines[src_index],\
                                     self.graph_topology.physical_machines[dst_index]
```

```
            if src_host.can_fit_virtual_machine(src_vm) and dst_host.can_fit_virtual_machine(dst_vm):
                src_host.add_new_virtual_machine(src_vm)
                dst_host.add_new_virtual_machine(dst_vm)

                self.total_cost += (vm_pair.frequency * cost)
                break
            else:
                pm_tuple_set.pop(0)

    def get_cost(self):
        return self.total_cost

    def construct_pm_tuple_set(self):
        pm_set = self.graph_topology.physical_machines
        pm_tuple_set = []
        for i in range(0, len(pm_set)):
            for j in range(0, len(pm_set)):
                if i != j:
                    host_one, host_two = pm_set[i], pm_set[j]

                    host_pair_graph = self.construct_pm_mb_graph(host_one, host_two)
                    spanning_tree = host_pair_graph.spanning_tree(weights=host_pair_graph.es['weight'],
                                                                  return_tree=True)

                    traversal_cost = self.calculate_pm_mb_graph_cost(host_one, host_two, spanning_tree)
                    pm_tuple_set.append(PMTuple(i, j, traversal_cost))

        return pm_tuple_set

    def calculate_pm_mb_graph_cost(self, start_host, end_host, graph):
        path_list = []

        start_path = MBPath()
        start_path.add_vertex(start_host)
        path_list.append(start_path)

        while True:
            current_path = path_list.pop()
            if current_path.is_path_valid:
                current_vertex = graph.vs.find(current_path.last_vertex.identifier)
```

```
            adjacent_vertices = graph.neighbors(current_vertex)

            for child_index in adjacent_vertices:
                child_vertex, child_asset = graph.vs[child_index], graph.vs[child_index]['asset']
                new_path = copy.deepcopy(current_path)
                new_path.add_vertex(child_asset)

                if new_path.is_complete(start_host, end_host, self.graph_topology.middleboxes):
                    return new_path.cost_of_path(graph)
                else:
                    path_list.append(new_path)

def construct_pm_mb_graph(self, host_one, host_two):
    complete_graph = igraph.Graph()

    # Add in vertices for hosts
    complete_graph.add_vertex(name=host_one.identifier)
    complete_graph.vs.select(name=host_one.identifier)["asset"] = host_one
    complete_graph.add_vertex(name=host_two.identifier)
    complete_graph.vs.select(name=host_two.identifier)["asset"] = host_two

    # Add in vertices for MBs
    for mb in self.graph_topology.middleboxes:
        complete_graph.add_vertex(name=mb.identifier)
        complete_graph.vs.select(name=mb.identifier)["asset"] = mb

    # Link each hosts to each MB
    for host in [host_one, host_two]:
        for mb in self.graph_topology.middleboxes:
            link_weight = self.graph_topology.calculate_distance(host, mb.parent_switch)
            complete_graph.add_edge(source=host.identifier, target=mb.identifier, weight=link_weight)

    # Link hosts to each other
    link_weight = self.graph_topology.calculate_distance(host_one, host_two)
    complete_graph.add_edge(source=host_one.identifier, target=host_two.identifier, weight=link_weight)

    # Link MBs to each other
    for i in range(0, len(self.graph_topology.middleboxes)):
        for j in range(i + 1, len(self.graph_topology.middleboxes)):
            mb_1, mb_2 = self.graph_topology.middleboxes[i], self.graph_topology.middleboxes[j]
            link_weight = self.graph_topology.calculate_distance(mb_1.parent_switch, mb_2.parent_switch)
```

```
            complete_graph.add_edge(source=mb_1.identifier, target=mb_2.identifier, weight=link_weight)

    return complete_graph

def traversal_complete(self, current_path, start_host, end_host):
    if current_path[0] == start_host and current_path[-1] == end_host:
        for mb in self.graph_topology.middleboxes:
            if mb not in current_path:
                return False
        return True
    else:
        return False
```

APPENDIX D

GRAPH TOPOLOGY CONSTRUCTION CODE

```
import math
import random

import igraph

from utils import graph_components


class Node(object):
    def __init__(self, mb, cost):
        self.mb = mb
        self.cost = cost

        self.children = []

    def add_child(self, obj):
        self.children.append(obj)


class Topology:

    def __init__(self, k_value, num_middleboxes, pm_capacity):
        self.k_value = k_value
        self.num_middleboxes = num_middleboxes
        self.pm_capacity = pm_capacity

        self.physical_machines = []
        self.edge_switches = []
        self.agg_switches = []
        self.core_switches = []
        self.middleboxes = []

        self.topology_graph = None

    def create_topology(self):
        fat_tree_topology = igraph.Graph()

        n_pms_per_pod = n_core_switches = int(math.pow(self.k_value/2, 2))
        n_agg_switches_per_pod = n_edge_switches_per_pod = n_connections = self.k_value / 2

        # Create physical machines
        for _ in range(0, n_pms_per_pod * self.k_value):
            pm = graph_components.PhysicalMachine(maximum_capacity=self.pm_capacity)

            fat_tree_topology.add_vertex(name=pm.identifier)
            fat_tree_topology.vs.select(name=pm.identifier)["PM"] = pm
            fat_tree_topology.vs.select(name=pm.identifier)["TYPE"] = "PM"
            self.physical_machines.append(pm)

        # Create edge switches
        for _ in range(0, int(n_edge_switches_per_pod * self.k_value)):
            edge_switch = graph_components.PhysicalSwitch("EDGE")

            fat_tree_topology.add_vertex(name=edge_switch.identifier)
            fat_tree_topology.vs.select(name=edge_switch.identifier)["edge_switch"] = edge_switch
            fat_tree_topology.vs.select(name=edge_switch.identifier)["TYPE"] = "E"
            self.edge_switches.append(edge_switch)

        # Create aggregate switches
        for _ in range(0, int(n_agg_switches_per_pod * self.k_value)):
            agg_switch = graph_components.PhysicalSwitch("AGG")

            fat_tree_topology.add_vertex(name=agg_switch.identifier)
            fat_tree_topology.vs.select(name=agg_switch.identifier)["agg_switch"] = agg_switch
            fat_tree_topology.vs.select(name=agg_switch.identifier)["TYPE"] = "AGG"
            self.agg_switches.append(agg_switch)
```

```
# Create core switches
        for _ in range(0, n_core_switches):
            core_switch = graph_components.PhysicalSwitch("CORE")

            fat_tree_topology.add_vertex(name=core_switch.identifier)
            fat_tree_topology.vs.select(name=core_switch.identifier)["core_switch"] = core_switch
            fat_tree_topology.vs.select(name=core_switch.identifier)["TYPE"] = "CORE"
            self.core_switches.append(core_switch)

        # Link hosts with edge switches
        current_pm_index = 0
        for edge_switch in self.edge_switches:
            for _ in range(0, int(n_connections)):
                current_pm = self.physical_machines[current_pm_index]
                fat_tree_topology.add_edge(source=edge_switch.identifier, target=current_pm.identifier)
                current_pm.set_parent_edge_switch(edge_switch)

                current_pm_index += 1

        # Link agg switches with edge switches
        edge_switch_start_index, n_pod_connections = 0, 0
        for agg_switch in self.agg_switches:
            for i in range(0, int(n_edge_switches_per_pod)):
                fat_tree_topology.add_edge(source=agg_switch.identifier,
                                           target=self.edge_switches[int(edge_switch_start_index + i)].identifier)

            n_pod_connections += 1
            if n_pod_connections >= int(n_edge_switches_per_pod):
                edge_switch_start_index += n_edge_switches_per_pod
                n_pod_connections = 0

        # Link core switches to agg switches
        edge_number, agg_switch_index = 0, 0
        for core_switch in self.core_switches:
            for i in range(0, self.k_value):
                fat_tree_topology.add_edge(source=core_switch.identifier,
                                           target=self.agg_switches[agg_switch_index + int(i * n_connections)].
                                           identifier)

            edge_number += 1
            if edge_number >= int(n_edge_switches_per_pod):
                agg_switch_index += 1
                edge_number = 0

        self.topology_graph = fat_tree_topology
        self.create_middleboxes()

    def create_middleboxes(self):
        current_parent_switches = []
        switch_list = self.agg_switches + self.core_switches + self.edge_switches
        random.shuffle(switch_list)

        while len(current_parent_switches) < self.num_middleboxes:
            random_switch = switch_list[random.randint(1, len(switch_list) - 1)]
            if random_switch not in current_parent_switches:
                mb = graph_components.Middlebox(parent_switch=random_switch)
                random_switch.assign_middlebox(mb)

                self.middleboxes.append(mb)
                current_parent_switches.append(random_switch)

    def calculate_distance(self, vertex_one, vertex_two):
        return self.topology_graph.shortest_paths_dijkstra(source=vertex_one.identifier,
                                                           target=vertex_two.identifier)[0][0]
```

```
def calculate_in_order_mb_traversal(self):
        total_cost = 0

        current_mb = self.middleboxes[0]
        for next_mb in self.middleboxes[1:]:
            total_cost += self.calculate_distance(vertex_one=current_mb.parent_switch,
                                                  vertex_two=next_mb.parent_switch)
            current_mb = next_mb

        return total_cost

    def get_mb_closest_to_host(self, host):
        best_mb, distance = None, float('inf')
        for mb in self.middleboxes:
            d = self.calculate_distance(host, mb.parent_switch)
            if d < distance:
                distance = d
                best_mb = mb
        return best_mb, distance

    def calculate_unordered_mb_traversal(self, start_mb):
        root_node = Node(start_mb, 0)
        self.middleboxes.remove(start_mb)

        self.build_mb_tree(root_node, self.middleboxes)
        best_path, path_cost = self.traverse_mb_tree(root_node)

        self.middleboxes.append(start_mb)
        return best_path[-1].mb, path_cost

    def traverse_mb_tree(self, root_node):
        if len(root_node.children) == 0:
            return [root_node], root_node.cost
        else:
            node_path, current_cost = None, float('inf')
            for child in root_node.children:
                nodes, cost = self.traverse_mb_tree(child)
                if cost < current_cost:
                    node_path = nodes
                    current_cost = cost
            return ([root_node] + node_path), (root_node.cost + current_cost)

    def build_mb_tree(self, parent_node, unvisited_mbs):
        for ii in range(0, len(unvisited_mbs)):
            new_node = Node(unvisited_mbs[ii], self.calculate_distance(parent_node.mb.parent_switch,
                                                                       unvisited_mbs[ii].parent_switch))

            self.build_mb_tree(new_node, unvisited_mbs[:ii] + unvisited_mbs[ii+1:])
            parent_node.add_child(new_node)

    def reset_topology(self):
        for host in self.physical_machines:
            host.reset_physical_machine()

    def shortest_first_mb_path(self, start_host):
        indices = []

        current_node, total_cost = start_host, 0
        while len(indices) < len(self.middleboxes):

            shortest_len, next_node, next_node_index = float('inf'), None, -1
            for ii, mb in enumerate(self.middleboxes):
                if ii not in indices:
                    cost = self.calculate_distance(current_node, mb.parent_switch)

                    if shortest_len > cost:
                        shortest_len, next_node, next_node_index = cost, mb, ii

            indices.append(next_node_index)
            current_node, total_cost = next_node.parent_switch, total_cost + shortest_len

        return self.middleboxes[-1], total_cost
```