Scheduling Parallel Migration of Virtualized Services under Time Constraints in Mobile Edge Clouds

Peiyue Zhao and György Dán
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden. E-mail: {peiyue|gyuri}@kth.se

***Abstract*—Migrating virtualized services (VSs) in mobile edge clouds is essential for maintaining service quality under mobility, for optimizing resource utilization, and for responding to incidents. We consider migrating VSs with heterogeneous resource requirements from a source placement to a target placement under a time constraint, while maintaining service continuity as much as possible. We formulate the VS migration problem as an integer programming problem, and propose an efficient algorithm to compute sequences of migration actions. The algorithm is based on a graphical representation of the VS dependencies, and constructs a collection of acyclic directed hypergraphs with bounded length. We evaluate our algorithm in realistic scenarios and compare it to the optimal solution and to a baseline algorithm. Extensive simulations show that our algorithm achieves near-optimal performance, and is computationally efficient and scalable.**

I. Introduction

Service virtualization benefits end users by reducing capital costs and by enabling flexible resource scaling, with performance isolation, high reliability and security. Examples of Virtualized Services (VSs) can be network functions, content caches, primitives of computational offloading for Internet of Things (IoT) devices and for in-network data analytics [1]–[4].

A promising architecture for hosting VSs for latency critical and high bandwidth applications is Mobile Edge Computing (MEC). As a key enabler for 5G, MEC provides distributed computation and storage resources in the proximity of end users, with high performance and low latency. A fundamental primitive for resource management when hosting VSs in MEC is scheduling VS migration. VS migration is necessary for MEC node maintenance, and it also facilitates load balancing and performance optimization under shifting workloads caused by user mobility. Representative emerging use cases are the control and monitoring of unmanned aerial vehicles (UAVs) and self-driving cars. VS migration also enhances resilience, for example, in case of cyber attacks or device failures a VS can be migrated to a MEC node that operates normally to recover service availability.

A critical requirement for scheduling VS migration within MEC is to make fast reaction to workload shifts and incidents. Existing works on migration scheduling mainly consider data center clouds, which serve large areas with slowly shifting workloads, and thus they mostly focus on maintaining service availability, without considering the migration time constraints. Compared to data centers, MEC nodes serve relatively small areas and thus the workloads may shift fast, which requires fast VS migration so as to accommodate user mobility and to satisfy QoE requirements.

Fast VS migration within MEC is constrained by the requirement of maintaining service continuity during migration and by the available computing resources. This is because migrating VSs while maintaining the service continuity requires extra computing resources, and thus some of the VSs may need to be turned off to make resources available for fast migration. Additionally, during migration a VS transits through multiple phases, in each of which a VS has different resource requirements and service availability. Therefore computing an optimal solution requires joint consideration of the available resources, the load of the system, the phases of migration, and the availability requirements of VSs. These factors together make VS migration under time constraints extremely challenging.

In this paper we address the problem of scheduling the migration of VSs under time constraint, with the objective of maximizing the service continuity during VS migration. We consider a set of MEC nodes with computational capacity for hosting a set of VSs. Each VS requires some computational resources for providing a service, and is associated with a value according to the service it provides. The VSs are initially placed according to a source placement, and they need to be migrated to a target placement within a time constraint. We formulate the VS migration scheduling (VMS) problem as an integer programming (IP) problem, and propose an efficient heuristic for solving the VMS problem. We prove that the heuristic terminates in a finite number of iterations. By simulations we benchmark our algorithm against the optimal solution and against a baseline algorithm. Numerical results show that our algorithm achieves near-optimal performance, and is efficient and scalable.

The rest of the paper is organized as follows. Section II reviews the related work and Section III introduces the system model and problem formulation. Section IV presents our solution to the VMS problem and Section V presents the numerical results. Finally, Section VI concludes the paper.

This work was partly funded by the MSB CERCES project.

II. Related Work

Closely related to our work is the area of Virtual Machine (VM) migration. VM migration is recognized as a key enabler for improving performance optimization and energy efficiency for various cloud architectures. [5] considers VM migration for server consolidation in data centers, and proposes a framework to compute a VM placement and to schedule VM migration subject to server capacity. The solution relies on solving constraint programming problems by an optimization solver, and does not consider the complexity. Authors in [6] consider the problem of scheduling VM migration to satisfy the requirement of the VMs in terms of resources, security, and communication, but the migration time minimization is not considered.

Related to our work are recent works on VM migration scheduling for migration time and downtime minimization [7]–[11]. Due to the complexity of modeling VM migration, those works propose solutions based on heuristics or approximations. Authors in [7] address the problem of minimizing the total migration time of VMs while considering energy consumption. The problem is solved by using an optimization solver with the assistance of a proposed heuristic. [8] proposes a heuristic for minimizing VM migration time and downtime in data centers. The proposed heuristic limits parallel VM migration to VMs that utilize different network links and different hosting servers, even if there are sufficient servers and network resources. [12] considers maximizing the transmission rate for VM migration in a Software Defined Network (SDN), and thus indirectly minimizes the total migration time. The paper formulates a mixed integer linear programming (MILP) problem, and reduces it to a maximum multicommodity flow problem by ignoring the bandwidth constraints, for which an approximation algorithm is proposed.

An important issue in VM migration is to handle migration deadlocks, which is explicitly considered in [9], [10]. Authors in [9] propose a heuristic for scheduling VM migration so as to minimize the migration time and downtime, under the constraints of network topology and bandwidth capacity. The proposed solution resolves a migration deadlock by migrating the VM with the minimal migration cost to a temporary location. [10] formulates the VM migration problem as an Integer Linear Problem (ILP), and provides a heuristic that migrates VMs in descending order of their weight, based on the dependencies among the VMs. In case of a migration deadlock, the algorithm interrupts the VM with the lowest weight.

For considering the impact of VM migration on the service availability of VMs, cost minimization based VM migration can be adopted. Authors in [11] compute the VM migration cost based on the memory consumption of a VM and they formulate a constraint satisfaction programming problem to minimize the overall VM migration cost. However, the importance of different VMs or migration time constraints are not considered.

Closest to our work is [13], which considers scheduling the migration of VSs with homogeneous resource requirements, and proposes a solution based on a top-down approach, which decomposes the migration dependency graph. Our work goes beyond [13] substantially by considering VSs with heterogeneous resource requirements. Due to this heterogeneity, the dependencies between VSs form a hypergraph, which requires a completely different solution approach. In this paper, we propose a bottom-up approach that constructs a hypergraph from an empty graph, opposite to the top-down approach. Numerical results show that for migrating VSs with heterogeneous resource requirements, our bottom-up approach outperforms the top-down approach based solution significantly, while the top-down approach provides lower execution time when the number of VSs is low.

Table I: Decision variables and the states of VSs

$x_{m,k,i}$	$e_{m,k,i}$	State
1	0	Starting
1	1	Running
0	0	Powered off

III. System Model and Problem Formulation

A. System Model

We consider a set $\mathcal{F} = \{f_1, f_2, \ldots, f_{|\mathcal{F}|}\}$ of VSs, each of which provides some service. The VSs are placed on a set $\mathcal{M}$ of MEC nodes. We consider that a VS instance on a MEC node can be in one of three states.

- *Powered off* is the state when a VS instance is turned off, and hence it cannot provide service.
- *Running* is the state when a VS instance is providing service. A VS outage occurs when there is no instance of a VS in running state. A VS instance in the running state can be turned off immediately, in which case it enters the powered off state.
- *Starting* is the state when a VS instance is turned on and is performing a set of operations to get ready for the running state, for example, loading the VS image from storage, configuring runtime environment, and setting up network connections. We denote by t_s the time that a VS spends in the starting state, and consider that a VS instance cannot provide service in the starting state.

Since a VS instance can enter the powered off state immediately, while it takes a VS instance t_s time to enter the running state, we divide time into slots of length t_s to facilitate the scheduling. A VS can be turned on or off at the beginning of each slot. We will refer to these time slots as migration rounds, and we denote by T the target number of migration rounds, i.e. the total time available for migration. Furthermore, we denote by v_k the value of the service provided by VS $f_k \in \mathcal{F}$ per migration round in the running state.

We use two binary variables to jointly indicate the state of a VS instance. In migration round i the binary variable $x_{m,k,i}$ indicates whether an instance of f_k is placed on MEC node $m \in \mathcal{M}$, and the binary variable $e_{m,k,i}$ further indicates whether an instance of VS f_k is in the running state on MEC node m. There is a unique mapping between the value of the decision variables and the states of the VSs, as shown in Table I. To make sure that a VS makes valid transitions among

different states, a set of constraints has to be satisfied. First, if VS f_k is in the running state on MEC node m in time slot i, it should be placed on MEC node m during the same time slot,

$$x_{m,k,i} - e_{m,k,i} \geq 0, \ \forall m, k, i. \tag{1}$$

Moreover, a VS instance has to be placed on a MEC node m for one time slot to enter the running state, hence

$$x_{m,k,i-1} \geq e_{m,k,i}, \forall m, k, i. \tag{2}$$

Also, if VS f_k is placed on MEC node m in the time slot $i-1$, it can be either in the running state or in the powered off state on m in time slot i,

$$x_{m,k,i-1} + x_{m,k,i} - e_{m,k,i} \leq 1, \ \forall m, k, i. \tag{3}$$

We denote by $\mathbf{x}_i = (x_{1,1,i}, \ldots, x_{|\mathcal{M}|,|\mathcal{F}|,i})$ the placement of the VSs in migration round i. We denote by $\mathbf{x}_s$ and $\mathbf{x}_t$ the source placement and the target placement of the VSs, respectively; thus we have

$$\mathbf{x}_0 = \mathbf{x}_s, \text{and } \mathbf{x}_T = \mathbf{x}_t, \tag{4}$$

which ensure the VSs to be in the running state after migration.

We also consider that VS migration is constrained by computational resources. We denote by ω_m the amount of computational resources on MEC node m for hosting VSs, and we consider that VS f_k requires $r_k \in \mathbb{Z}_{\geq 0}$ computational resources in the starting or in the running state. A VS instance in the powered off state does not require any computational resources, thus

$$\sum_{k=1}^{|\mathcal{F}|} x_{m,k,i} r_k \leq \omega_m, \ \forall m, i. \tag{5}$$

For the sake of simplicity, we assume that both r_k and ω_k are integers. This assumption is reasonable considering that the number of virtual CPUs (e.g., cores) is an integer, and bare metal partitioning hypervisors, such as Jailhouse [14] allocate an integer number of virtual CPUs to each process to ensure isolation.

B. Problem Formulation

We are now ready to formulate the VMS problem. The VMS problem is to compute a migration schedule for migrating VSs from a source placement $\mathbf{x}_s$ to a target placement $\mathbf{x}_t$ within T migration rounds, with the objective of maximizing the total value of the VS instances in the running state during the migration.

$$\begin{aligned}
\underset{x_{m,k,i}, e_{m,k,i}}{\text{maximize}} \quad & \sum_{i=1}^{T} \sum_{k=1}^{|\mathcal{F}|} \sum_{m \in \mathcal{M}} v_k e_{m,k,i} \\
\text{subject to} \quad & (1) - (5) \\
& \sum_{m} (x_{m,k,i} - e_{m,k,i}) \leq 1, \ \forall k, i \\
& \sum_{m} e_{m,k,i} \leq 1, \ \forall k, i \\
& x_{m,k,i}, e_{m,k,i} \in \{0, 1\}, \ \forall m, \ k, \ i \\
& T \in \mathbb{N}
\end{aligned} \tag{P1}$$

The additional constraints in (P1) enforce that each VS can have at most one instance in the running state and at most one instance in the starting state in each migration round. These two constraints ensure that the potential interruption of a VS is not compensated by running multiple instances of a VS in another migration round.

If the constraint matrix of problem (P1) is totally unimodular, the linear relaxation of (P1) has optimal solutions correspond to integral optimal solutions of (P1). However, as the variables in (5) have coefficients $r_k \in \mathbb{Z}_{\geq 0}$, the constraint matrix of (P1) is not totally unimodular in general. Due to the complexity of (P1), it is infeasible to solve even moderate sized instances of the VMS problem by optimization solvers. Therefore we propose an efficient heuristic to solve the VMS problem.

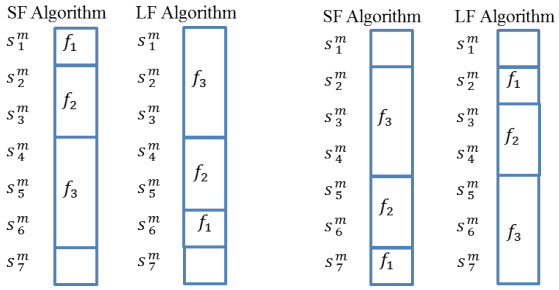
IV. The Migration Dependence Hypergraph Construction Algorithm

In this section we propose an efficient heuristic to solve the VMS problem. First, we analyze the source and the target placement of the VSs to create a dependency hypergraph $\mathcal{G}$, which consists of cyclic and acyclic paths. We then observe that a migration schedule can be built based on acyclic paths, which motivates us to construct a hypergraph that consists of acyclic paths only and we use it to generate a migration schedule that satisfies the migration time constraint. In what follows we describe each step in detail.

A. Creating and Assigning MEC Slots

To begin with, for each MEC node m we create a set S_m of MEC slots that correspond to the resources on MEC node m with $|S_m| = \omega_m$. We denote by $s_n^m \in S_m$ the n^{th} MEC slot of MEC node m. We then assign each VS f_k to a set of r_s MEC slots on its source MEC node M_k^s and to a set of r_s MEC slots on its target MEC node M_k^t, referred to as the set of source slots σ_k^s and the set of target slots σ_k^t, respectively. When assigning source and target slots for VSs, a straightforward way could be to assign MEC slots arbitrarily. However, as we will show later, the MEC slot assignment affects the total value of VSs in the migration schedule, and thus we propose to assign MEC slots as follows.

For each MEC node m, we consider two algorithms for assigning MEC slots S_m as source slots. The first algorithm orders the VSs on MEC node m in ascending order of their resource requirement r_k, and assigns to each VS the available MEC slots with the lowest indices. We refer to this as the Smallest First (SF) algorithm. The second algorithm orders the VSs on MEC node m in descending order of their resource requirement r_k, and assigns to each VS the available MEC slots with the lowest indices. We refer to this as the Largest First (LF) algorithm. Figure 1 (a) illustrates the source slot assignment for 3 VSs, with $r_1 = 1$, $r_2 = 2$, $r_3 = 3$, and $M_1^s = M_2^s = M_3^s = m$. Since f_1 has the lowest resource requirement, f_1 is assigned to σ_1^m by the SF algorithm. On the contrary, MEC slots $\{\sigma_1^m, \sigma_2^m, \sigma_3^m\}$, which have the lowest indices, are assigned to VS f_3 by the LF algorithm, as f_3 has the highest resource requirement. Similarly, we consider



(a) Source Assignment (b) Target Assignment

Figure 1: The MEC slot assignment for 3 VSs with $r_1 = 1$, $r_2 = 2$, and $r_3 = 3$.

Algorithm 1: Building the Dependency Hypergraph

Input : $\mathcal{F}$

Create a hypergraph $\mathcal{G} = (\mathcal{F}, \mathcal{E}, (q_e)_{e\in\mathcal{E}})$, where $\mathcal{E} = \emptyset$.

for $\forall f_k \in \mathcal{F}$ **do**

  if $D(f_k) \neq \emptyset$ **then**

    $e = (f_k, D(f_k))$

    $\mathcal{E} = \mathcal{E} \cup \{e\}$ and $q_e = 0$

Output : $\mathcal{G}$

the SF and the LF algorithms for the target slot assignment, but for the target slot assignment we assign to each VS the available MEC slots with the highest indices. An example of the target slot assignment is shown in Figure 1 (b), for the same set of VSs as in Figure 1 (a) and assuming that the VSs use m as their target MEC node. Observe that the algorithm for the source and the target slot assignment can be chosen independently, but as we will show later, source assignment with the SF algorithm minimizes the number of hyperedges in the dependency hypergraph.

B. Building the Dependency Hypergraph

In general if the target MEC slots σ_k^t of VS f_k are not assigned to any VS in $\mathbf{x}_s$ then f_k can be migrated immediately. Otherwise, f_k has to wait for the VSs on σ_k^t to be migrated, or for them to be interrupted. To create a dependency hypergraph, we start with defining the dependency relation between two VSs as follows.

Definition 1. *For any two VSs f_k and f_j, we say that f_k depends on f_j if at least one of the target slots of f_k is a source slot of f_j, that is, $\sigma_k^t \cap \sigma_j^s \neq \emptyset$, and we denote the dependence of f_k on f_j by $f_k \to f_j$.*

We define $D(f_k) = \{f_j : f_k \to f_j, \forall f_j \in \mathcal{F}\}$ and $D^{-1}(f_k) = \{f_j : f_j \to f_k, \forall f_j \in \mathcal{F}\}$ as the set of VSs that f_k depends on and the set of VSs that depends on f_k, respectively. To describe the dependency relations among the VSs, we create the weighted directed hypergraph $\mathcal{G} = (\mathcal{F}, \mathcal{E}, (q_e)_{e\in\mathcal{E}})$, which we call the dependency hypergraph. Each vertex in $\mathcal{G}$ corresponds to a VS, and each hyperedge $e \in \mathcal{E}$ is an ordered pair of two disjoint sets of VSs,

$$e = (\mathcal{F}_e^t, \mathcal{F}_e^h), \tag{6}$$

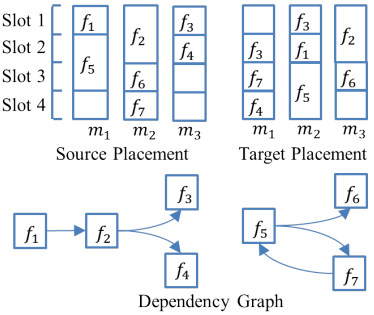


Figure 2: The dependency hypergraph of a sample system with 3 MEC nodes and 7 VSs.

where $\mathcal{F}_e^t$ is the tail of e and $\mathcal{F}_e^h$ is the head of e. The edges are built based on the dependency relations among the VSs: we form a hyperedge $e = (f_k, D(f_k))$ if $D(f_k) \neq \emptyset$. Therefore each hyperedge in $\mathcal{G}$ has only one tail but can have multiple heads. For a hyperedge e, we denote by q_e its weight, which is initially set to 0; we will show how to update q_e later. Algorithm 1 summarizes the steps of building the dependency hypergraph. In the dependency hypergraph $\mathcal{G}$, we define paths as follows.

Definition 2. *In the dependency hypergraph $\mathcal{G} = (\mathcal{F}, \mathcal{E}, (q_e)_{e\in\mathcal{E}})$, a path $P \in \mathcal{G}$ is a weakly connected subgraph with $P = (\mathcal{F}_p, \mathcal{E}_p)$. We say that P is cyclic, if there exists a set of VSs $\{f_1, \ldots, f_N\} \in \mathcal{F}_P$ such that $f_k \to f_{k+1}\ \forall k < N$, and $f_N \to f_1$. Otherwise, P is acyclic and we define the length of P as the number of the constituent VSs of the longest simple path on P, denoted by $L(P)$.*

For convenience, we denote by $P(f_k)$ the path that f_k is on. Among the two types of paths, we are interested in acyclic paths, as we will show in the next subsection that an acyclic path can be used for scheduling the migration of its constituent VSs.

C. Building Migration Schedules from Acyclic Paths

In an acyclic path the VSs are related to each other according to their locations. We define the set $F(f_k)^s = \{f_j : f_k \in \mathcal{F}_e^t, f_j \in \mathcal{F}_e^h, e \in \mathcal{E}_p\}$ of VSs that have an incoming edge from f_k as the successors of f_k. Similarly, the set $F(f_k)^p = \{f_j : f_j \in \mathcal{F}_e^t, f_k \in \mathcal{F}_e^h, e \in \mathcal{E}_p\}$ of VSs that have outgoing edges to f_k are the predecessors of f_k. Moreover, we define f_k as the tail of a path if $F(f_k)^p = \emptyset$, and define f_k with $F(f_k)^s = \emptyset$ as the head of a path. Figure 2 shows a sample dependency hypergraph of a system with $|\mathcal{M}| = 3$, $\omega_m = 4\ \forall m \in \mathcal{M}$, and $|\mathcal{F}| = 7$. The path induced by $\{f_1, f_2, f_3, f_4\}$ is acyclic with $F(f_2)^s = \{f_3, f_4\}$ and $F(f_2)^p = \{f_1\}$, and the path induced by $\{f_5, f_6, f_7\}$ is cyclic.

Consider now an acyclic path P. To avoid VS interruption, VSs on P shall be migrated according to a topological order such that f_k will not be migrated until VSs in $F(f_k)^s$ are migrated. Since P is acyclic, a topological order of P exists and can be computed by Algorithm 7.2 in [15] with complexity $\mathcal{O}(|\mathcal{E}_P| + |\mathcal{F}_P|)$. Let us define t_k^m to be the time slot when f_k is migrated from its source slots to its target slots. As we next show, t_k^m can be computed easily.

Algorithm 2: Compute Migration Schedule

Input : A path $P = (\mathcal{F}_P, \mathcal{E}_P)$
Sort $\mathcal{F}_P$ according to its topological order
for $f_k \in \mathcal{F}_P$ **do**
  if $F(f_k)^s = \emptyset$ **then**
    Set $t_k^m = 1$
  else
    Set $t_k^m = \max(\{t_j^m : f_j \in F(f_k)^s\}) + 1$
Output : t_k^m

Lemma 1. *t_k^m can be calculated as*

$$t_k^m = \begin{cases} 1, & \text{if } F(f_k)^s = \emptyset \\ max(\{t_j^m : f_j \in F(f_k)^s\}) + 1 & \text{if } F(f_k)^s \neq \emptyset \end{cases} \quad (7)$$

Proof. Let us consider a VS f_k that is to be migrated. If f_k has no successor then f_k can be migrated immediately without waiting and therefore $t_k^m = 1$. Otherwise, f_k waits for all of its successors to be migrated. Upon migration, an instance of $f_j \in \mathcal{F}(f_k)^s$ enters the starting state at time slot t_j^m on MEC slots σ_j^t and then it enters the running state at time slot $t_j^m + 1$. The instance of f_j running on source slots σ_j^s can be turned off at time slot $t_j^m + 1$ to make resource available for the migration of f_k, thus (7) holds. □

The steps to schedule the migration of VSs on path P are summarized in Algorithm 2. In Algorithm 2, VS $f_k \in P$ cannot be interrupted by its predecessors on path P, but f_k still can be interrupted if VSs $D^{-1}(f_k)$ are located on different paths. As an example, consider that $f_j \in D^{-1}(f_k)$, and f_k and f_j are on different paths. f_k will be interrupted if the migration of f_j is executed before the migration of f_k is finished, which is $t_j^m - t_k^m < 1$. Let us denote by t_k^i the number of time slots that f_k is interrupted, we can then write

$$t_k^i = max(\{t_j^m - t_k^m + 1 | f_j \in D^{-1}(f_k)\} \cup \{0\}). \quad (8)$$

We can use t_k^i to calculate the sum interruption cost of the VSs on path P as

$$C(P) = \sum_{f_k \in \mathcal{F}_P} t_k^i c_k, \quad (9)$$

and we refer to $C(P)$ as the cost of path P. Moreover, we can also use t_k^m to calculate the length of an acyclic path P by the following result.

Lemma 2. *The length of path P equals to the number of time slots to migrate the constituent VSs of P.*

Proof. First, we prove by contradiction that t_k^m is the number of VSs on the longest simple path that starts with f_k. Let us consider a VS f_1 with $t_1^m = u$. Since f_1 is migrated at time slot t_1^m and the migration of each VS takes one time slot, there exits a simple path P_1 consists of VSs $\mathcal{F}_{P_1} = \{f_1, f_2, \ldots, f_u\}$, with $f_j \in F(f_{j-1})^s, \forall j = 2, \ldots, u$. Clearly, the length of P_1 is t_1^m.

Let us assume that the longest simple path that starts with f_1 is P_2, and $\mathcal{F}_{P_2} = \{f_1', f_2', \ldots, f_v'\}$, with $f_1' = f_1$ and $f_j' \in F(f_{j-1}')^s, \forall j = 2, \ldots, v$. If $L(P_2) > t_1^m$, it indicates that f_2' will be interrupted by f_1, which violates the derivation of t_1^m. If $L(P_2) < t_1^m$, the assumption that P_2 is the longest simple path that starts with f_1 is contradicted. Therefore, $L(P_2) = t_1^m$ and then the length of P can be calculated as

$$L(P) = \max(\{t_k^m : f_k \in \mathcal{F}_P\}), \quad (10)$$

which is also the number of time slots to migrate the constituent VSs of P. □

Inspired by the observations above, a hypergraph that consists of a set of disjoint acyclic paths $\mathcal{P}_{ac}$ can be used to build a migration schedule that solves the VMS problem if $\mathcal{P}_{ac}$ has the following properties :

1) Complete : $\bigcup_{P \in \mathcal{P}_{ac}} \mathcal{F}_p = \mathcal{F}$.
2) Disjoint : $\mathcal{F}_P \cap \mathcal{F}_{P'} = \emptyset, \ \forall P, P' \in \mathcal{P}_{ac}$.
3) Constrained : $L(P) \leq T, \forall P \in \mathcal{P}_{ac}$.

D. Migration Dependence Hypergraph Construction Algorithm

A natural way to find the set $\mathcal{P}_{ac}$ would be to start from $\mathcal{G}$ and remove edges from it iteratively until the remaining paths are acyclic and satisfy the length requirement. However, this requires to classify the paths on the hypergraph and to update their lengths and costs whenever an edge is removed. The complexity of doing so can increase exponentially as the number of VSs increases. For example, using the adjacency matrix of the dependency graph $\mathcal{G}$ to identify all the cyclic paths requires to compute the $|\mathcal{F}|^{th}$ power of a $|\mathcal{F}| \times |\mathcal{F}|$ matrix [16]. Instead, we propose the migration dependency hypergraph construction (DHC) algorithm, which works as follows.

Initialization: We initialize the DHC algorithm with the hypergraph $\mathcal{G}_{ac}^{(0)} = (\mathcal{F}, \emptyset)$, i.e., the empty graph. As the following result shows, the graph $\mathcal{G}_{ac}^{(0)}$ is a feasible solution for the VMS problem, and thus a valid hypergraph to start from.

Lemma 3. *$\mathcal{G}_{ac}^{(0)} = (\mathcal{F}, \emptyset)$ is a feasible solution for the VMS problem.*

Proof. Since the set of hyperedges on $\mathcal{G}_{ac}^{(0)}$ is empty, VSs on $\mathcal{G}_{ac}^{(0)}$ are disjoint. Then each VS forms a path $P(f_k) = \left(\mathcal{F}_{P(f_k)}, \mathcal{E}_{P(f_k)}\right)$ with $\mathcal{F}_{P(f_k)} = \{f_k\}$ and $\mathcal{E}_{P(f_k)} = \emptyset$. Therefore the initial set of disjoint paths is $\mathcal{P}_{ac}^{(0)} = \{P(f_k) : f_k \in \mathcal{F}\}$, and for any two VSs $f_k, f_j \in \mathcal{F}$, paths $P(f_k)$ and $P(f_j)$ are disjoint, which satisfies the disjointness property. Clearly, $\bigcup_{P \in \mathcal{P}_{ac}^{(0)}} \mathcal{F}_p = \bigcup_{f_k \in \mathcal{F}} \mathcal{F}_{P(f_k)} = \mathcal{F}$, and thus the completeness property is satisfied. Finally, note that $L(P) = |\mathcal{E}_P| = 1 \ \forall P \in \mathcal{P}_{ac}^{(0)}$, and thus $\mathcal{G}_{ac}^{(0)}$ is constrained. □

Finding feasible edges: Let us denote by $\mathcal{G}_{ac}^{(n)} = (\mathcal{F}, \mathcal{E}_{ac}^{(n)})$ the hypergraph constructed after n iterations, and by $\mathcal{P}_{ac}^{(n)}$ the set of disjoint paths on graph $\mathcal{G}_{ac}^{(n)}$. $P_{f_k}^{(n)}$ indicates the path that f_k belongs to in graph $\mathcal{G}_{ac}^{(n)}$. In iteration $n > 0$, we add an edge $e \in \mathcal{E} \setminus \mathcal{E}_{ac}^{(n-1)}$ to $\mathcal{E}_{ac}^{(n-1)}$, and thus $\mathcal{E}_{ac}^{(n)} = \mathcal{E}_{ac}^{(n-1)} \cup \{e\}$.

Nevertheless, we have to ensure that $\mathcal{E}_{ac}^{(n)}$ is a feasible solution, which we do by restricting the selection to feasible edges, defined as follows.

Definition 3. *Let edge $e \in \mathcal{E} \setminus \mathcal{E}_{ac}^{(n-1)}$. We say that e is a feasible edge for $\mathcal{E}^{(n-1)}$ if $G_{ac}^{(n)} = (\mathcal{F}, \mathcal{E}_{ac}^{(n-1)} \cup \{e\})$ is a feasible solution to the VMS problem.*

To derive the conditions for e to be a feasible edge for $\mathcal{E}_{ac}^{(n-1)}$, let us denote by $\mathcal{P}_e^{(n-1,t)} = \{P_{f_k}^{(n-1)} : f_k \in \mathcal{F}_e^t\}$ and $\mathcal{P}_e^{(n-1,h)} = \{P_{f_k}^{(n-1)} : f_k \in \mathcal{F}_e^h\}$ the set of paths that contains the tail and heads of e on graph $\mathcal{G}_{ac}^{(n-1)}$. Adding an edge e to $\mathcal{E}_{ac}^{(n-1)}$ connects its tail and heads, and merges paths $\mathcal{P}_e^{(n-1,t)}$ and $\mathcal{P}_e^{(n-1,h)}$ to a new path $P_e^{(n)} = (\mathcal{F}_{P_e^{(n)}}, \mathcal{E}_{P_e^{(n)}})$, with $\mathcal{F}_{P_e^{(n)}} = \{\mathcal{F}_P : P \in \mathcal{P}_e^{(n-1,t)} \cup \mathcal{P}_e^{(n-1,h)}\}$ and $\mathcal{E}_{P_e^{(n)}} = e \cup \{\mathcal{E}_P : P \in \mathcal{P}_e^{(n-1,t)} \cup \mathcal{P}_e^{(n-1,h)}\}$. Consequently,

$$\mathcal{P}_{ac}^{(n)} = \left(\mathcal{P}_{ac}^{(n-1)} \setminus \left(\mathcal{P}_e^{(n-1,t)} \cup \mathcal{P}_e^{(n-1,h)}\right)\right) \cup P_e^{(n)}. \quad (11)$$

Now we are ready to prove the conditions for an edge e to be feasible for $\mathcal{E}_{ac}^{(n-1)}$.

Lemma 4. *An edge $e \in \mathcal{E} \setminus \mathcal{E}_{ac}^{(n-1)}$ is feasible for $\mathcal{E}_{ac}^{(n-1)}$ if and only if $P_e^{(n)}$ is acyclic and $L(P_e^{(n)}) \leq T$.*

Proof. First, we prove the sufficiency of the conditions. Since $\mathcal{F}_{P_e^{(n)}} = \{\mathcal{F}_P : P \in \mathcal{P}_e^{(n-1,t)} \cup \mathcal{P}_e^{(n-1,h)}\}$, adding e to $\mathcal{E}_{ac}^{(n-1)}$ does not change the constituent VSs of the paths in $\mathcal{P}_{ac}^{(n)}$ and thus completeness is satisfied. Since paths $\mathcal{P}_e^{(n-1,t)}$ and $\mathcal{P}_e^{(n-1,h)}$ are disjoint with $\mathcal{P}_{ac}^{(n-1)} \setminus (\mathcal{P}_e^{(n-1,t)} \cup \mathcal{P}_e^{(n-1,h)})$, and edge e is only incident to paths $\mathcal{P}_e^{(n-1,t)}$ and $\mathcal{P}_e^{(n-1,h)}$, $P_e^{(n)}$ is disjoint with paths $\mathcal{P}_{ac}^{(n-1)} \setminus (\mathcal{P}_e^{(n-1,t)} \cup \mathcal{P}_e^{(n-1,h)})$. Thus by (11) adding e to $\mathcal{E}_{ac}^{(n-1)}$ ensures disjointness. Therefore, for e to be feasible for $\mathcal{E}_{ac}^{(n-1)}$ it is sufficient that $P_e^{(n)}$ is acyclic and $L(P_e^{(n)}) \leq T$.

To show necessity, observe that if adding e to $\mathcal{E}_{ac}^{(n-1)}$ is feasible, all the paths in $\mathcal{P}_{ac}^{(n)}$ are acyclic and satisfy the constrained condition. Since $P_e^{(n)} \in \mathcal{P}_{ac}^{(n)}$, $P_e^{(n)}$ is acyclic and $L(P_e^{(n)}) \leq T$. □

Algorithm 3 uses the conditions in Lemma 4 to verify whether an edge e is feasible for $\mathcal{E}_{ac}^{(n-1)}$. The algorithm assumes that e is feasible for $\mathcal{E}_{ac}^{(n-1)}$ until it finds that e fails the conditions in Lemma 4. Lines 2-8 check if $P_e^{(n)}$ is cyclic. For edge e, if the tail VS $\mathcal{F}_e^t$ and a head VS $f_j \in \mathcal{F}_e^h$ are not on the same path of $\mathcal{G}_{ac}^{(n-1)}$, the algorithm can instantly confirm that $P_e^{(n)}$ is not a cyclic path that contains $\mathcal{F}_e^t$ and f_j, since $\mathcal{P}_e^{(n-1,t)}$ and $P_{f_j}^{(n-1)}$ are disjoint (Lines 3-4). However, if $\mathcal{F}_e^t$ and f_j are already on the same path at iteration $n-1$, the algorithm needs to go through Lines 5-8. Since edge e forms a path from $\mathcal{F}_e^t$ to f_j, if there exists a path from f_j to $\mathcal{F}_e^t$, $P_e^{(n)}$ would be cyclic according to Definition 2. This check can be performed by making use of the function $A(f_k) = \left(\bigcup_{f_j \in F(f_k)^p} A(f_j)\right) \cup F(f_k)^p$, which returns the set of VSs that can reach VS f_k before e is connected. If $P_e^{(n)}$ is cyclic, e is not feasible and then the algorithm terminates. Otherwise, the algorithm checks if the length of $P_e^{(n)}$ is larger than T by (10) (Lines 9-10). Note that (10) requires the migration time of the constituent VSs of $P_e^{(n)}$. Observe that the successors of VSs in $\mathcal{F}_e^h$ remain the same when e is connected, and the migration times of those VSs also remain the same. Therefore, we only need to update the migration times of the constituent VSs of $\mathcal{F}_e^t$ by (7).

Algorithm 3: Feasibility Check

```
   Input  : An edge e
1  Result=True
2  for f_j ∈ F_e^h do
3      if P_e^(n-1,t) ≠ P_{f_j}^(n-1) then
4          Continue
5      else
6          if f_j ∈ A(F_e^t) then
7              Result=False
8              Terminate
9  if L(P_e^(n)) > L then
10     Result=False
   Output : Result
```

Recomputing edge weights: Finally, if edge e is feasible for $\mathcal{E}_{ac}^{(n-1)}$, we can update the weight of e as the reduction of interruption cost when adding e to $\mathcal{E}_{ac}^{(n-1)}$,

$$q_e = \sum_{P \in (\mathcal{P}_e^{(n-1,t)} \cup \mathcal{P}_e^{(n-1,h)})} C(P) - C(P_e^{(n)}). \quad (12)$$

At iteration n among the feasible edges of $\mathcal{E}_{ac}^{(n-1)}$ we choose to add the edge e with the highest positive weight to $\mathcal{E}_{ac}^{(n-1)}$. Let us consider now what happens after this. Since $\mathcal{E}_{ac}^{(n)}$ affects the structure of $\mathcal{G}_{ac}^{(n)}$, the feasibility and the weight of edges in $\mathcal{E} \setminus \mathcal{E}_{ac}^{(n)}$ need to be updated. The following results show that we only need to update the feasibility and weight for a small subset of $\mathcal{E} \setminus \mathcal{E}_{ac}^{(n)}$.

Lemma 5. *Let $e, e' \in \mathcal{E} \setminus \mathcal{E}_{ac}^{(n-1)}$. If e' and $P_e^{(n)}$ are disjoint, then the feasibility and weight of e and e' are independent.*

Proof. First, we prove by contradiction that paths $P_{e'}^{(n)}$ and $P_e^{(n)}$ are disjoint. Since $e, e' \in \mathcal{E} \setminus \mathcal{E}_{ac}^{(n-1)}$, the paths $\mathcal{P}_e^{(n-1,t)}$, $\mathcal{P}_e^{(n-1,h)}$, $\mathcal{P}_{e'}^{(n-1,t)}$ and $\mathcal{P}_{e'}^{(n-1,h)}$ are in the set $\mathcal{P}_{ac}^{(n-1)}$ and thus are disjoint. If $P_{e'}^{(n)}$ is incident to $P_e^{(n)}$, then there exists a path $P \in \mathcal{P}_e^{(n-1,t)} \cup \mathcal{P}_e^{(n-1,h)}$ and there exists a path $P' \in \mathcal{P}_{e'}^{(n-1,t)} \cup \mathcal{P}_{e'}^{(n-1,h)}$ such that $\mathcal{F}_P \cap \mathcal{F}_{P'} \neq \emptyset$, which contradicts the disjoint condition.

Therefore, $P_{e'}^{(n)}$ and $P_e^{(n)}$ are disjoint, which indicates that the feasibility and weight of e' are independent of whether e will be added to $\mathcal{E}_{ac}^{(n-1)}$. This proves the lemma. □

From Lemma 5, the following corollary is immediate.

Corollary 1. *Consider that an edge e is added to $\mathcal{E}_{ac}^{(n-1)}$ at iteration n, and let $\mathcal{E}_u^{(n+1)} = \{e' : (\mathcal{F}_{e'}^t \cup \mathcal{F}_{e'}^h) \cap \mathcal{F}_{P_e^{(n)}} \neq \emptyset, \forall e' \in \mathcal{E} \setminus \mathcal{E}_{ac}^{(n)}\}$. Then the set of edges whose feasibility and weight have to be updated is exactly $\mathcal{E}_u^{(n+1)}$.*

Algorithm 4: DHC Algorithm

Input : $\mathcal{F}$, $\mathbf{x}_s$, $\mathbf{x}_t$

1 Execute the source and target assignment algorithms
2 Build $\mathcal{G} = (\mathcal{F}, \mathcal{E}, (q_e)_{e\in\mathcal{E}})$ by Algorithm 1
3 Initialize $\mathcal{G}_{ac}^{(0)} = (\mathcal{F}, \mathcal{E}_{ac}^{(0)})$ with $\mathcal{E}_{ac}^{(0)} = \emptyset$, and $\mathcal{P}_{ac}^{(0)}$
4 n=0
5 Set $\mathcal{E}_u^{(1)} = \mathcal{E}$ and $t_k^m = 1\ \forall f_k$
6 **while** $(\mathcal{E} \setminus \mathcal{E}_{ac}^{(n)}) \neq \emptyset$ **do**
7 n=n+1
8 **for** $e \in \mathcal{E}_u^{(n)}$ **do**
9 **if** *e is feasible by Algorithm 3* **then**
10 Update q_e by (12)
11 **else**
12 $\mathcal{E} = \mathcal{E} \setminus e$
13 Let $e \in argmax\{q_{e'} | e' \in \mathcal{E} \setminus \mathcal{E}_{ac}^{(n-1)}\}$
14 **if** $q_e > 0$ **then**
15 $\mathcal{E}_{ac}^{(n)} = \mathcal{E}_{ac}^{(n-1)} \cup \{e\}$
16 Update $P_{ac}^{(n)}$ by (11)
17 Update $t_k^m\ \forall\ f_k \in \mathcal{P}_e^t$ by Algorithm 2
18 Update $A(f_k)\ \forall\ f_k \in \mathcal{P}_e^{(n-1,h)}$
19 Update $\mathcal{E}_{ac}^{(n+1)}$ by Corollary 1
20 **else**
21 **Break**

Output : $\mathcal{P}_{ac}^{(n)}$ and t_m^k

We are now ready to summarize the steps of the DHC algorithm, shown in Algorithm 4. Line 1 assigns the MEC slots according to the source and target assignment algorithms, and line 2 builds the dependency hypergraph $\mathcal{G}$ by Algorithm 1. Then line 3 initializes the solution $\mathcal{G}_{ac}^{(0)}$ and initializes the set of acyclic paths $\mathcal{P}_{ac}^{(0)}$ accordingly. The set $\mathcal{E}_u^{(1)}$ is initialized as $\mathcal{E}$ on line 5. Line 9 updates the feasibility of $e \in \mathcal{E}_u^{(n)}$ for $\mathcal{E}_{ac}^{(n-1)}$. If e is infeasible, it will be removed from $\mathcal{E}$, otherwise, line 10 updates its weight q_e. After the weight of the edges in $\mathcal{E}_u$ are updated, line 13 finds the edge e with the highest weight. If q_e is positive, the algorithm adds e to $\mathcal{E}_{ac}^{(n-1)}$ and then updates $\mathcal{P}_{ac}^{(n)}$, t_k^m, $A(f_k)$ and $\mathcal{E}_u^{(n+1)}$ accordingly (Lines 15-19). The algorithm terminates when $\mathcal{E} \setminus \mathcal{E}_{ac}^{(n)} = \emptyset$ or $max(q_e) \leq 0\ \forall e \in \mathcal{E} \setminus \mathcal{E}_{ac}^{(n-1)}$. The algorithm results in a hypergraph $\mathcal{G}_{ac}^{(n)}$ that consists of a set of disjoint acyclic paths $\mathcal{P}_{ac}^{(n)}$ with length bounded by T, and t_k^m indicates when VS f_k should be migrated.

The following result shows that the DHC algorithm terminates in a finite number of iterations.

Lemma 6. *The DHC algorithm terminates in at most $|\mathcal{F}|$ iterations.*

Proof. Observe that in each iteration n the DHC algorithm adds one edge into $\mathcal{E}_{ac}^{(n-1)}$ until termination. Since there is at most one hyperedge for each VS, $|\mathcal{E}| \leq |\mathcal{F}|$. Therefore the DHC algorithm terminates in at most $|\mathcal{F}|$ iterations. □

In the DHC algorithm the set of edges $\mathcal{E}$ determines the set of edges that the solution $\mathcal{E}_{ac}^{(n)}$ can contain, and it depends on the MEC slot assignment. In the following we show that our proposed MEC slot assignment can help to improve the performance of the DHC algorithm.

Proposition 1. *The source slot assignment with the SF algorithm when using the DHC algorithm induces the minimal number of edges in $\mathcal{E}$.*

Proof. Let us consider a MEC node m that is the source node of a set $\mathcal{F}_s$ of VSs, and is the target node of a set $\mathcal{F}_t$ of VSs. We need to assign $\omega_s = \sum_{f_k\in\mathcal{F}_s} r_k$ and $\omega_t = \sum_{f_k\in\mathcal{F}_t} r_k$ MEC slots as source slots and as target slots, respectively. The MEC slot assignment in the DHC algorithm assigns the MEC slots $s_1^m, s_2^m, \ldots, s_{\omega_s}^m$ as source slots to VSs, and assigns MEC slots $s_{\omega_m-\omega_t+1}^m, \ldots, s_{\omega_m}^m$ as target slots. We refer to a MEC slot as a shared slot if it is assigned to VSs both as a source slot and as a target slot. Clearly, a hyperedge is formed for each VS that uses at least one shared slot as a source slot.

When $\omega_s + \omega_t \leq \omega_m$, no shared slot will be induced by the DHC algorithm and thus no hyperedges. On the contrary, when $\omega_s + \omega_t > \omega_m$, the minimal number of shared slots is $\omega_s + \omega_t - \omega_m$, and the shared MEC slots will be $s_{\omega_m-\omega_t+1}^m, \ldots, s_{\omega_s}^m$. Consider now the algorithms for source slot assignment. Among those, the SF algorithm assigns the shared slots to the VSs with the highest computational resource requirements as source slots, and hence occupies the shared slots with minimal number of VSs. Consequently, the number of edges in $\mathcal{E}$ is minimized. □

V. Numerical Results

In the following we use simulations to evaluate the performance of the DHC algorithm in terms of total service value, efficiency and scalability. We simulated MEC nodes with $\omega_m = 10\ \forall m$ to host VSs. The computational resource requirement r_k of each VS is between 1 and 3, representing the resource requirement of small, medium, and large sized VSs. In the simulations we considered two distributions of r_k to simulate systems with different constituent VSs. The first distribution is $\mathcal{D}_1$ with $P(r_k = 1) = P(r_k = 2) = P(r_k = 3) = \frac{1}{3}$, and the second distribution is $\mathcal{D}_2$ with $P(r_k = 1) = P(r_k = 3) = 0.25$ and $P(r_k = 2) = 0.5$. The service value v_k of VS f_k is chosen uniform at random on $[1, 50]$. In the simulations we uniformly select the source node M_k^s and the target node M_k^t of VS $f_k \in \mathcal{F}$ for each VS among the MEC nodes with available resources.

To benchmark the DHC algorithm, we compare its performance with the optimal solution and with a baseline algorithm, referred to as the migration dependency hypergraph decomposition (DHD) algorithm. The optimal solution is obtained by using the MILP solver of Matlab. The DHD algorithm is adopted from a heuristic algorithm for scheduling the migration of VSs with homogeneous resource requirements in [13]. The DHD algorithm first creates the dependency hypergraph $\mathcal{G}$ and assigns the MEC slots as in the DHC algorithm. Then the DHD algorithm breaks each simple cycle in $\mathcal{G}$ by removing the incoming edges of the VS with the lowest service value v_k on each cycle. Finally, by a local search based approach the

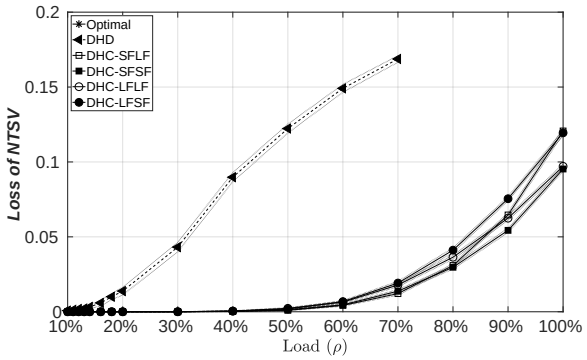


Figure 3: Loss of NTSV vs. load, $|\mathcal{M}|$=80, T=4.

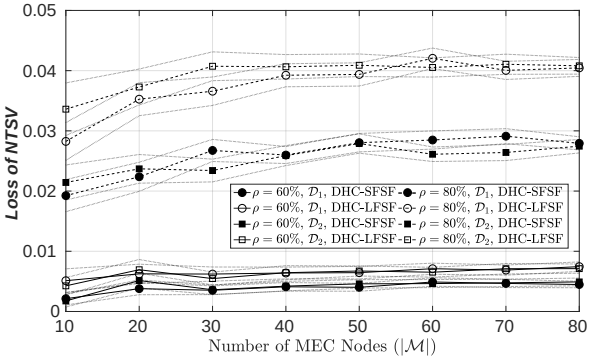


Figure 4: Loss of NTSV vs. number of MEC nodes ($\mathcal{M}$) for $\mathcal{D}_1$ and $\mathcal{D}_2$.

DHD algorithm cuts each path P with $L(P) > L$ into a set of paths whose length are bounded by T.

In the simulations, we implemented the DHC algorithm with all four combinations of the MEC slot assignment algorithms. We concatenate the abbreviations of the source and the target slot assignment algorithms to denote different implementations. For example, we refer as DHC-SFSF to the DHC algorithm that uses the SF algorithm for both source slot and target assignment, while we refer as DHC-SFLF to the DHC algorithm that uses the SF algorithm for source slot assignment and uses the LF algorithm for target slot assignment. The results shown are the averages of 100 simulations, and the confidence intervals are at the 95% confidence level.

A. Service Value Performance

We first evaluate the service value performance of the DHC algorithm. To compare the service value of different simulation scenarios on a common scale, we compute the normalized total service value (NTSV) as follows,

$$\text{NTSV} = \frac{\sum_{i=1}^{T} \sum_{f_k \in \mathcal{F}} \sum_{m \in \mathcal{M}} v_k e_{m,k,i}}{T \sum_{f_k \in \mathcal{F}} v_k}, \tag{13}$$

and we define 1-NTSV as the loss of NTSV, which measures the service interruption during VS migration. To measure the overall computational resource consumption of a system, we define the average load ρ,

$$\rho = \frac{\sum_{f_k \in \mathcal{F}} r_k}{\sum_{m \in \mathcal{M}} \omega_m}. \tag{14}$$

Figure 3 shows the loss of NTSV of six algorithms as a function of the load (ρ) , for a system with $|\mathcal{M}|= 80$, $T = 4$, and $r_k \sim \mathcal{D}_1$. The results show that the optimal solution can only be computed for very low load values, and the different variants of the DHC algorithm achieve near-optimal performance for those problem instances. The results also show that the DHC algorithm outperforms the DHD algorithm significantly: the performance gap increases as the load increases, and when the load is 70% the loss of NTSV for the DHC-SFSF algorithm is 90% less than that for the DHD algorithm. This is because the DHC algorithm considers the global impact of adding each edge to the solution, while DHD breaks each simple cycle locally and thus loses accuracy and performance as the load increases.

Figure 3 also shows that among the variants of the DHC algorithm, the DHC-SFSF algorithm achieves the lowest loss of NTSV for most of the scenarios, while the DHC-LFSF algorithm leads to the highest loss of NTSV. This shows the importance of choosing the right combination of the MEC slot assignment algorithms. Though Proposition 1 shows that the SF algorithm for source slot assignment induces the minimal number of edges in the dependency graph $\mathcal{G}$, the performance gap between the DHC-SFSF and the DHC-SFLF algorithms shows that the target slot assignment also has impact on the performance. As we will show later, evaluating the performance of the DHC-SFSF and the DHC-LFSF algorithms for different scenarios shows interesting observations, and thus we focus on those two algorithms in the following figures.

In what follows we investigate the impact of the distribution of r_k on the performance of the DHC algorithm. Figure 4 shows the loss of NTSV for the DHC-LFSF and the DHC-SFSF algorithms as a function of the number of MEC nodes $|\mathcal{M}|$, for various load values and distributions for r_k. The figure allows us to assess scaling with the number of MEC nodes, and shows that contrary to intuition, the loss of NTSV shows an increasing trend with the number of MEC nodes for a given load. The reason is that for the same load the system with more MEC nodes generally has more VSs and thus has more cyclic paths and acyclic paths with length larger than T, for which the DHC algorithm induces more VS interruptions to compute a feasible migration schedule. The results also show that the distribution of r_k has limited impact on the DHC-SFSF and the DHC-LFSF algorithms, and therefore we use $\mathcal{D}_1$ in the rest of the simulations.

To evaluate the impact of the time constraint T, Figure 5 shows the loss of NTSV for the DHC-LFSF and the DHC-SFSF algorithms as a function of T with $r_k \sim \mathcal{D}_1$. The results show that the loss of NTSV decreases as T increases, with a decreasing marginal gain. This is because a larger T allows the DHC algorithm to include more edges on a path to avoid VS interruptions. It is also interesting to observe that the performance gap between the DHC-LFSF and the DHC-SFSF algorithms for $\rho = 80\%$ is larger than that for $\rho = 60\%$. This is because as the load increases there are more VSs that form

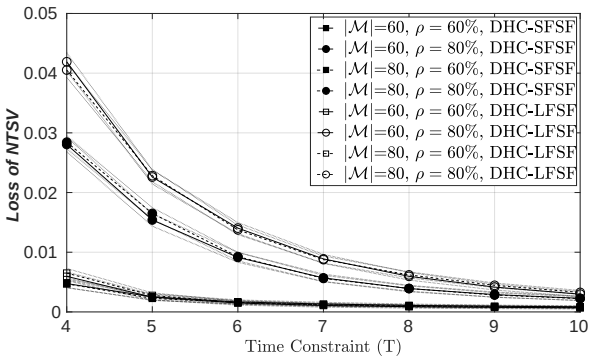


Figure 5: Loss of NTSV vs. time constraint (T) for combinations of $|\mathcal{M}|$ and load.

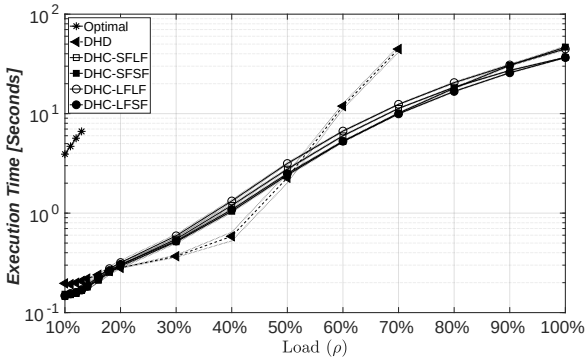


Figure 6: Running time vs. load, $|\mathcal{M}|$=80, T=4.

dependency relations, and thus it becomes critical for the DHC algorithm to start from a good assignment of the MEC slots.

B. Efficiency and Scalability

Figure 6 compares the execution time of the six algorithms as a function of the load (ρ), for the same scenarios as in Figure 3. Compared to the optimal solution, the variants of the DHC algorithm have orders of magnitude lower execution time. Note that when the load is higher than 13%, the coefficient matrix of the problem instances exceeds the memory limit of the optimization solver, and thus we are unable to obtain the optimal solution. It is interesting to see that when the load is between 20% and 50%, the DHD algorithm has lower execution time than the DHC algorithm, while the DHD algorithm has a higher execution time when the load exceeds 50%. This is because when the load is low the dependency hypergraph $\mathcal{G}$ contains few cyclic paths and few paths with length larger than T, and thus the DHD algorithm can compute a solution relatively fast. However, as the load increases, it becomes more time consuming to find all the acyclic paths and to cut the paths with lengths longer than T. Finally, it is important to note that for loads beyond 70% the DHD algorithm is not scalable anymore due to the increased time for finding all the cyclic paths. Furthermore, Figure 6 shows that when the load is high the DHC-SFSF algorithm outperforms the DHC-SFLF algorithm at the cost of a higher execution time. This observation shows that there is a tradeoff between the execution time and the service value performance.

Overall, the results above show that the DHC algorithm is an effective and efficient solution for the VMS problem, and the DHC algorithm scales well as the size and load of the system increase.

VI. Conclusion

We have proposed an algorithm for solving the problem of migration scheduling of virtualized services with heterogeneous resource requirements for maximizing service availability under time constraints. The proposed algorithm follows a bottom-up approach and iteratively constructs a hypergraph for generating a parallel migration schedule. Extensive simulations show that our proposed algorithm provides near-optimal performance and significantly outperforms a baseline algorithm that is based on a top-down approach, with high efficiency and scalability. An interesting extension of our work is to consider the problem of migration scheduling of virtualized services that need multiple types of resources, which would require to create a dependency graph per resource type and then to jointly consider those.

References

[1] S. Abdelwahab, B. Hamdaoui, M. Guizani, and T. Znati, "Replisom: Disciplined tiny memory replication for massive IoT devices in LTE edge cloud," *IEEE IoT Journal*, vol. 3, no. 3, pp. 327–338, 2016.

[2] X. Liu, J. Zhang, X. Zhang, and W. Wang, "Mobility-Aware Coded Probabilistic Caching Scheme for MEC-Enabled Small Cell Networks," *IEEE Access*, vol. 5, pp. 17 824–17 833, 2017.

[3] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts*, vol. 19, no. 3, pp. 1628 – 1656, 2017.

[4] P. Zhao and G. Dán, "A Benders Decomposition Approach for Resilient Placement of Virtual Process Control Functions in Mobile Edge Clouds," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 4, pp. 1460 – 1472, 2018.

[5] F. Hermenier, J. Lawall, and G. Muller, "Btrplace: A flexible consolidation manager for highly available applications," *IEEE Trans. Dependable Secure Comput.*, vol. 10, no. 5, pp. 273–286, 2013.

[6] S. Al-Haj and E. Al-Shaer, "A formal approach for virtual machine migration planning," in *Proc. of IEEE CNSM*, 2013, pp. 51–58.

[7] V. Kherbache, E. Madelaine, and F. Hermenier, "Scheduling live migration of virtual machines," *IEEE Trans. Cloud Comput.*, 2017, early access.

[8] M. F. Bari, M. F. Zhani, Q. Zhang, R. Ahmed, and R. Boutaba, "CQNCR: optimal VM migration planning in cloud data centers," in *Proc. of IFIP Networking*, 2014, pp. 1–9.

[9] T. K. Sarker and M. Tang, "Performance-driven live migration of multiple virtual machines in datacenters," in *Proc. of IEEE International Conference on Granular Computing (GrC)*, 2013, pp. 253–258.

[10] K. Onoue, S. Imai, and N. Matsuoka, "Scheduling of parallel migration for multiple virtual machines," in *Proc. of IEEE AINA*, 2017, pp. 827–834.

[11] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: a consolidation manager for clusters," in *Proc. of International Conference on Virtual Execution Environments*, 2009, pp. 41–50.

[12] H. Wang, Y. Li, Y. Zhang, and D. Jin, "Virtual machine migration planning in software-defined networks," *IEEE Trans. Cloud Comput.*, 2017, early access.

[13] P. Zhao and G. Dán, "Time constrained service-aware migration of virtualized services for mobile edge computing," in *Proc. of ITC*, 2018, pp. 64–72.

[14] V. Sinitsyn, "Jailhouse," *Linux Journal*, no. 252, p. 2, 2015.

[15] H. Bhasin, *Algorithms - Design and Analysis*. Oxford University Press, 2015.

[16] D. M. Himmelblau, "Decomposition of large scale systems-I.Systems composed of lumped parameter elements," *Chemical Engineering Science*, vol. 21, no. 5, pp. 425–438, 1966.