Time Constrained Service-aware Migration of Virtualized Services for Mobile Edge Computing

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*—We consider the migration of virtualized services (VSs) in Mobile Edge Computing (MEC), so as to facilitate server maintenance, load balancing under mobility, improved energy efficiency and resource utilization, and incident response. We consider a set of VSs that has to be migrated from a source placement to a target placement, while maintaining service continuity as much as possible. We formulate the VS migration problem as an integer programming problem, and analyze its complexity. We propose an efficient iterative algorithm for computing when and in what order the VSs shall be migrated among the MEC nodes. We evaluate the proposed solution in terms of total service value, efficiency, and scalability. Extensive simulations show that our algorithm is computationally effective, and performs close to optimal.**

I. Introduction

Mobile Edge Computing (MEC) is an emerging technology that provides distributed computing and storage resources at the edge of cellular networks. Due to the proximity of computing resources to the end-users, MEC is a promising technology to cope with increasing bandwidth demands, and to satisfy the stringent delay and availability requirements of emerging real-time applications [1]. Notable examples of such applications are computational offloading for capacity-constrained Internet of Things (IoT) devices for smart city and smart home applications, mobile big data analytics, and augmented reality [2]–[6].

In these applications, virtualized MEC computing and storage resources are used for executing Virtualized Services (VSs) corresponding to end-user applications. VSs provide benefits to end users and to operators alike. They provide performance isolation, and enable to adapt the placement of VSs as a function of wireless bandwidth availability and in response to device mobility, e.g., in case of unmanned aerial vehicles (UAVs). At the same time, VSs allow MEC operators to load balance in response to spatial and temporal changes in application workloads, due to, e.g., mobility, and can facilitate improved energy efficiency and resource utilization [6], [7]. They also facilitate incident response and resilience, as upon an intrusion or a fault event VSs can be migrated to MEC nodes that are known to operate normally [8].

A fundamental prerequisite for adaptive MEC resource management is fast VS migration scheduling that avoids service degradation, and is able to react at the time scale at which workloads change. Existing works on migration scheduling were motivated by data center clouds, which serve large areas and user workloads that tend to change slower due to statistical multiplexing, and thus their focus has been mostly on maintaining service availability [9]–[14]. On the contrary, in the case of MEC, individual nodes will serve relatively small areas and wireless conditions can change rapidly, it will thus be essential for migration to be fast.

Maintaining service availability is, unfortunately, contradicting migration under time constraints. On the one hand, if MEC resources are scarce migration may require certain VSs to be shut down temporarily to meet the migration time constraints. On the other hand, even if MEC resources are abundant, migration may require VSs to be shut down for resolving deadlocks. The optimal solution depends on the amount of time available for migration, on MEC resource availability, and on the service availability requirements of VSs, which makes scheduling of migration particularly challenging under time constraints.

In this paper, we address the problem of scheduling VS migration with the objective of maximizing service availability subject to a migration time constraint. We consider a system in which a set of VSs has to be migrated from a source placement to a target placement, and each VS is associated with a value according to the importance of the service it provides. We formulate the time constrained VS migration (TCVM) problem as an integer programming (IP) problem, and analyze its complexity. Our problem formulation allows to explore trade-off between service availability and the time needed for migration. We analyze the limits of reducing the dimension of the problem, and we propose an efficient iterative algorithm for computing a migration schedule. Numerical results show that the proposed algorithm can achieve near-optimal performance with low computational complexity, and it can outperform the state of the art Wedelin heuristic for solving IP problems.

The rest of the paper is structured as follows. We review the related work in Section II and introduce the system model and problem formulation in Section III. We analyze the problem complexity in Section IV. We present our solution to the VS migration problem in Section V. In Section VI We provide numerical results, and we conclude the paper in section VII.

II. Related Work

Related to our work is the area of Virtual Machine (VM), which provides running environment for applications and ser-

This work was partly funded by the EU H2020 SUCCESS project, grant agreement No. 700416, and by the MSB CERCES project.

vices. The VM migration scheduling problem has attracted significant attention lately, for performance and energy efficiency, and for downtime minimization. Authors in [9] formulate the VM migration problem to find a migration schedule that satisfies security, dependency and performance requirements, but does not consider the minimization of the migration time and the downtime. [10] proposes a VM consolidation algorithm that computes the placement of VMs to reduce the number of running servers, and schedules the migrations to reach the new placement. The consolidation problem is formulated as a constraint programming problem with linear and binary variables, subject to the resource capacity of the servers. The migration schedule is then computed using off-the-shelf optimization solvers, without consideration of the computational complexity. Different from our work, the problem of migration deadlock is not considered, and the migration algorithm is targeted for the purpose of VM consolidation.

Related to our work are recent works on VM migration scheduling for migration time and downtime minimization [11]–[14]. Due to the complexity of the VM migration problem, the solution approaches in these works are based on heuristics. [11] addresses the VM migration scheduling problem for pre-copy based live migrations for multiple VMs, with consideration of the server capacity and the network bandwidth. The paper proposes a heuristic algorithm to minimize the total migration time and the downtime of the VMs. To handle the case where there is a migration deadlock, a heuristic deadlock handling algorithm is proposed. [12] considers maximizing the network bandwidth for VM migration, so as to minimize the total migration time indirectly. The paper assumes that a VM can be migrated only if the allocated bandwidth exceeds a pre-defined threshold, and as such the indirect approach results in an approximation.

[13] proposes a heuristic VM migration solution to minimize the total migration time and the downtime. The proposed heuristic limits parallel VM migration to the VMs that do not share the same network link, even if there are available resources on the servers, and does not address the problem of migration deadlock. [14] models the VM migration problem with an Integer Linear Problem (ILP), and provides a heuristic based on analyzing the dependencies of the VMs and the individual migration time. The problem of migration deadlock is addressed by interrupting the VM with the lowest migration time, but the migration time constraints are considered.

In order to minimize the interruption of VMs providing important services, cost minimization based VM migration can be used. [15] formulates the problem of minimum cost VM migration as a constraint satisfaction programming problem, using a cost model based on the size of the memory of the VMs, and hence it does not consider importance of different VMs or migration time constraints. To the best of our knowledge ours is the first work to consider migration scheduling under migration time constraints, with the objective of maximizing service availability.

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 to be placed on a set $\mathcal{M}$ of MEC nodes. Each VS provides some service, and we denote by v_k the value of the service provided by VS $f_k \in \mathcal{F}$. The value of a service corresponds to the importance of the service being provided. 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 shut down, and hence it does not provide service.
- *Running* is the state when a VS is providing service. A service experiences a service outage if there is no VS instance of it in the running state. A VS instance in the running state can be shut down immediately, and then it enters the powered off state.
- *Starting* is the state when a VS instance is preparing for the running state (e.g., loading and booting the VS image, initializing the VS instance and network connections). A VS instance can not provide service in the starting state. We denote by t_s the time it takes for a VS instance to enter the running state, i.e., the time it spends in the starting state.

Since a VS can be shut down immediately, without loss of generality we can divide time into slots of length t_s, and consider that a VS instance can enter the starting state or can be shut down at the beginning of a time 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. We use the binary variable $x_{m,k,i}$ to indicate whether an instance of VS f_k is placed on MEC node $m \in \mathcal{M}$ in migration round i, i.e., it is in the starting state or in the running state, and we define $\mathbf{x}_i = (x_{1,1,i}, \ldots, x_{|\mathcal{M}|,|\mathcal{F}|,i})$. We refer to $\mathbf{x}_i$ as the placement 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 $\mathbf{x}_0 = \mathbf{x}_s$, and $\mathbf{x}_T = \mathbf{x}_t$, which ensures the VSs to be in the running state after migration. For convenience, we denote by M_k^s and M_k^t the MEC node on which f_k is placed in $\mathbf{x}_s$ and $\mathbf{x}_t$, respectively.

We consider that a VS instance in the starting state or in the running state requires one unit of computing resource, and we denote by ω_m the number of VS instances that can be placed on a MEC node $m \in \mathcal{M}$. This assumption is reasonable, for example, if capacity is defined as the number of VMs or containers (such as Dockers) on a node, as industrial applications require isolation for performance and security reasons and have tight delay requirements; hence a single VS would be allocated per VM or container. A VS instance in the powered off state does not consume any computational resources, thus

$$\sum_k x_{m,k,i} \leq \omega_m, \ \forall m, i. \qquad (1)$$

We use the binary variable $e_{m,k,i}$ to indicate whether an instance of VS f_k is in the starting state on MEC node m in migration round i. The decision variables correspond to the states of the VSs, as shown in Table I. Clearly, a VS instance

Table I
DECISION VARIABLES AND THE STATES OF VSS

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

Table II
MIGRATION ACTIONS

$x_{m,k,i-1}$	$x_{m,k,i}$	Migration action of f_k w.r.t. m in round i
0	0	No migration action executed on m.
0	1	f_k is being started on m
1	0	f_k is shut down on m.
1	1	f_k is running on m. No migration action executed on m.

has to be placed on a MEC node m in order to enter the starting state, hence

$$x_{m,k,i} - e_{m,k,i} \geq 0,\ \forall m, k, i \tag{2}$$

The migration of VS f_k from a MEC node m to a MEC node m' involves starting an instance of VS f_k on MEC node m' at some round i' and shutting down the instance of VS f_k on MEC node m at some round i. If $i > i'$ then the instance on MEC node m' is in the running state by the time the instance on MEC node m is shut down, and hence the service of VS f_k is provided continuously. Otherwise, if $i \leq i'$ then there is an interruption in the service provided by VS f_k for $i' - i + 1$ rounds. Whether or not a VS f_k is migrated to or from a MEC node m in a migration round, the placement variables in two successive migration rounds $i-1$ and i have to satisfy

$$x_{m,k,i-1} - x_{m,k,i} + e_{m,k,i} \geq 0,\ \forall\ m,\ k,\ i > 0. \tag{3}$$

As shown in Table II, $x_{m,k,i-1} - x_{m,k,i} < 0$ if and only if f_k is being migrated to m in migration round i (and thus $e_{m,k,i} = 1$).

Moreover, if VS f_k is placed on MEC node m in iteration $i-1$, it can only be in the running state or in the powered off state on m in iteration i,

$$x_{m,k,i-1} + e_{m,k,i} \leq 1,\ \forall\ m,\ k,\ i > 0. \tag{4}$$

B. Problem formulation

We are now ready to formulate the time constrained VS migration (TCVM) problem. For a given source placement $\mathbf{x}_s$, target placement $\mathbf{x}_t$, and target number of migration rounds T, the TCVM problem is to maximize the total value of the VS instances in the running state while migrating the VSs,

$$\begin{aligned}
\underset{x_{m,k,i},e_{m,k,i}}{\text{maximize}}\quad & \sum_{i=1}^{T}\sum_{k=1}^{|\mathcal{F}|}\sum_{m\in\mathcal{M}} v_k\left(x_{m,k,i} - e_{m,k,i}\right)\\
\text{subject to}\quad & (1) - (4)\\
& \sum_{m}\left(x_{m,k,i} - e_{m,k,i}\right) \leq 1,\ \forall k, i\\
& \sum_{m} e_{m,k,i} \leq 1,\ \forall k, i\\
& \mathbf{x}_0 = \mathbf{x}_s, \mathbf{x}_T = \mathbf{x}_t,\\
& x_{m,k,i}, e_{m,k,i} \in \{0,1\},\ \forall m,\ k,\ i\\
& T \in \mathbb{N}
\end{aligned} \tag{P1}$$

The additional constraints in the problem formulation ensure that in every migration round there is at most one instance of each VS in the running state, and at most one instance of each VS in the starting state. 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.

IV. COMPLEXITY CONSIDERATION

Problem (P1) is an IP problem, and as such is computationally hard in general. Nonetheless, since (P1) has integer constraints and a constraint matrix with elements belonging to $\{-1, 0, 1\}$, if the constraint matrix of (P1) is totally unimodular, the linear relaxation of (P1) has an integral optimal solution that corresponds to the integer solutions of (P1). Unfortunately, our next result shows that the constraint matrix of (P1) is not totally unimodular in general.

Proposition 1. *The constraint matrix of the TCVM problem* (P1) *is not totally unimodular for $T \geq 3, |\mathcal{F}| \geq 3,\ |\mathcal{M}| \geq 2$, and $\omega_m \geq 2\ \forall m$. Hence, a linear relaxation of* (P1) *does not provide an optimal solution to* (P1).

Proof. To prove that the constraint matrix of the TCVM problem is not totally unimodular, it is sufficient to show that the constraint matrix has a submatrix whose determinant is neither ± 1 nor 0.

To show this, let A be the constraint matrix of a problem with $T = 3, |\mathcal{M}| = 2,\ \omega_1 = \omega_2 = 2$, and $|\mathcal{F}| = 3$. Let us now construct submatrix A_1 by taking the rows of A that correspond to constraints

$$x_{1,1,1} + x_{1,2,1} + x_{1,3,1} \leq 2, \tag{5}$$
$$x_{2,1,1} + x_{2,2,1} + x_{2,3,1} \leq 2, \tag{6}$$
$$-x_{1,3,1} + x_{1,3,2} + e_{1,3,2} \leq 0, \tag{7}$$
$$-x_{2,1,1} + x_{2,1,2} - e_{2,1,2} \leq 0, \tag{8}$$
$$x_{1,1,2} + x_{2,1,2} - e_{1,1,2} - e_{2,1,2} \leq 1, \tag{9}$$
$$-x_{2,3,1} + e_{2,3,1} \leq 0, \tag{10}$$
$$x_{1,3,1} + e_{2,3,1} \leq 0. \tag{11}$$

By taking the columns $3, 4, 12, 13, 16, 18$, and 21 of A_1, we get a 7×7 square submatrix A_2 of A_1,

$$A_2 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 1 & 0 & 0 & 0\\ -1 & 0 & 0 & 0 & 0 & 0 & -1\\ 0 & -1 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & -1 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}. \tag{12}$$

It is easy to verify that the determinant of A_2 is -2, and A_2 is a submatrix of A, which proves that A is not totally unimodular (Theorem 19.3 [16]).

Furthermore, as A is a submatrix of the constraint matrix for any problem with $T \geq 3, |\mathcal{F}| \geq 3,\ |\mathcal{M}| \geq 2$, and $\omega_m \geq 2\ \forall m$, the constraint matrix for any problem instance with $T \geq 3, |\mathcal{F}| \geq 3,\ |\mathcal{M}| \geq 2$ and $\omega_m \geq 2\ \forall m$ contains A_2 as a submatrix, and hence it is not totally unimodular. Thus, the

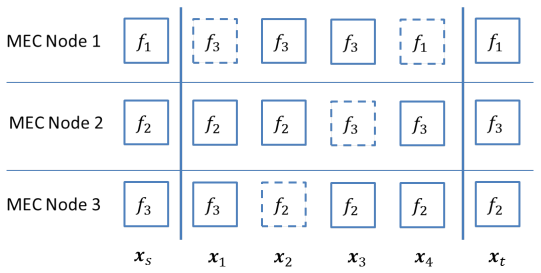

Figure 1. An optimal migration schedule for a system with 3 VSs. Solid frame indicates running state, dashed frame starting state.

linear relaxation of (P1) does not provide an optimal solution to (P1) (Corollary 19.2, [16]). $\square$

Motivated by these negative results, it may be tempting to try to reduce the size of a problem instance by only considering the set $\mathcal{F}_m = \{f_k | M_k^s \neq M_k^t, \forall f_k \in \mathcal{F}\}$ of the VSs that need to be migrated, and to ignore the set $\mathcal{F}_u = \{f_k | M_k^s = M_k^t, \forall f_k \in \mathcal{F}\}$ of the VSs that have the same placement in $\mathbf{x}_s$ and $\mathbf{x}_t$. Doing so reduces the number of VSs and MEC nodes to be considered, and may result in a new problem that can be solved optimally using LP relaxation. Unfortunately, as the below example shows, this approach may lead to a suboptimal result, as VSs in $\mathcal{F}_u$ may need to be interrupted to obtain an optimal solution.

Example 1. *Consider a system with $|\mathcal{F}| = 3$, $\mathcal{F}_u = \{f_1\}$, $v_2, v_3 \gg v_1$, $|\mathcal{M}| = 3$, and $\omega_m = 1$ $\forall m \in \mathcal{M}$. Let the migration time constraint to be $T = 4$. Since the VSs fill up the capacities of the MEC nodes, at least one of the VSs has to be interrupted during the migration. Every optimal migration schedule would interrupt f_1 only, as illustrated by one of the optimal schedules shown in Figure 1. The solid and the dashed frames indicate being in running and in starting states, respectively. Since v_1 is much lower than v_2 and v_3, f_1 is turned off in $\mathbf{x}_1$ such that MEC node 1 can host f_3 temporarily, and f_2 and f_3 can be migrated to their target MEC nodes without interruption in $\mathbf{x}_2$ and $\mathbf{x}_3$, respectively. Then in $\mathbf{x}_4$ VS f_1 is started again in MEC node 1.*

The following result shows that similar examples can only be constructed for $T \geq 3$.

Proposition 2. *For the TCVM problem, an optimal solution cannot interrupt a VS $f_j \in \mathcal{F}_u$ unless $T \geq 3$. Thus, for $T < 3$ it is sufficient to consider $\mathcal{F}_m$ for computing an optimal solution.*

Proof. Consider an optimal solution, in which a VS $f_j \in \mathcal{F}_u$ is interrupted to host $f_k \in \mathcal{F}_m$ on M_j^t temporarily, so as to avoid interrupting f_k. It takes one round to interrupt f_j and to make f_k start on M_j^t, and then it takes another round to make one instance of f_k start on M_k^t, while keeping one instance of f_k running on M_j^t. Finally, it takes one more round to make f_j start on M_j^t again. Therefore an optimal solution may interrupt a VS $f_j \in \mathcal{F}_u$ only if $T \geq 3$. Consequently, for $T < 3$ it is enough to consider $\mathcal{F}_m$ for computing an optimal solution. $\square$

For $T \geq 3$ it is not enough to consider $\mathcal{F}_m$, but as we show next, it is sufficient to consider at most $2|\mathcal{F}_m|$ VSs, instead of $|\mathcal{F}|$ VSs, to compute an optimal solution.

Proposition 3. *For a system with $\mathcal{F} = \mathcal{F}_m \cup \mathcal{F}_u$ and $T \geq 3$, it is sufficient to consider at most $2|\mathcal{F}_m|$ VSs for computing an optimal solution.*

Proof. It is clear that the proposition holds when $|\mathcal{F}_u| \leq |\mathcal{F}_m|$, since the maximal number of VSs that can be considered is $|\mathcal{F}| = |\mathcal{F}_m| + |\mathcal{F}_u| \leq 2|\mathcal{F}_m|$.

For $|\mathcal{F}_u| > |\mathcal{F}_m|$ we analyze the maximal number of VSs that need to be considered for computing an optimal solution in two cases.

1) For $T = 3$, each interrupted VS in $\mathcal{F}_u$ can be used for the migration of one VS in $\mathcal{F}_m$, and thus an optimal solution interrupts at most $|\mathcal{F}_m|$ VSs. Therefore it is sufficient to consider the $|\mathcal{F}_m|$ VSs with the lowest values in $\mathcal{F}_u$, and the VSs in $\mathcal{F}_m$, that is $2|\mathcal{F}_m|$ VSs in total.
2) For $T > 3$, each VS in $\mathcal{F}_u$ can be interrupted to host at least one VS in $\mathcal{F}_m$ for at least one round temporarily, and therefore the number of VSs that need to be considered for computing the optimal solution is at most as much as for $T = 3$. $\square$

We can thus reduce the number of VSs to be considered for computing an optimal solution by using Propositions 2 and 3. However, due to the complexity of the problem, it is infeasible to solve even moderate sized instances of the TCVM problem. Therefore in what follows we propose an efficient heuristic to solve the TCVM problem.

V. The Migration Dependence Graph Decomposition Algorithm

In this section we propose an efficient heuristic to solve the TCVM problem. First, we analyze the source and the target placement of the VSs to create a dependency graph $\mathcal{G}$. We then convert $\mathcal{G}$ into a graph that only consists of disjoint linear subgraphs. Finally, we cut the linear subgraphs into short subgraphs so as 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

As a first step, for each MEC node m we create a set S_m of MEC slots, with $|S_m| = \omega_m$. Each MEC slot can host one VS. We denote by $\mathcal{S} = \bigcup_{m \in \mathcal{M}} S_m$ the set of all MEC slots. We then assign each VS f_k to one MEC slot on its source MEC node M_k^s and to one MEC slot on its target MEC node M_k^t, referred to as the source slot σ_k^s and the target slot σ_k^t, respectively. Furthermore, we define the functions $F^s(s) = \{f_k : \sigma_k^s = s\}$ and $F^t(s) = \{f_k : \sigma_k^t = s\}$. We say that a MEC slot $s \in \mathcal{S}$ is available if $F^s(s) = F^t(s) = \emptyset$, and we denote by S_0 the set of available MEC slots.

B. Building the dependency graph

In general if the target slot σ_k^t of VS f_k is not assigned to any VS in $\mathbf{x}_s$ then f_k can be migrated immediately. Otherwise, f_k has to wait for the VS on σ_k^t to be migrated, or for it to

Algorithm 1: Building The Dependency Graph

```
   Input  : $\mathcal{F}$
1  Create a graph $\mathcal{G} = (\mathcal{F}, \mathcal{E})$, where $\mathcal{E} = \emptyset$.
2  for $\forall f_k \in \mathcal{F}$ do
3      if $F^s(\sigma_k^t) \neq \emptyset$ then
4          Add edge $(f_k, F^s(\sigma_k^t))$ to $\mathcal{E}$
   Output : $\mathcal{G}$
```

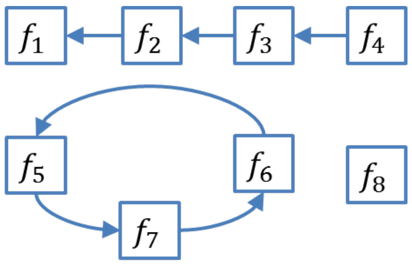

Figure 2. The dependency graph of a sample system with 8 VSs.

be interrupted. To create a dependency graph, 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 the target slot of f_k is the source slot of f_j, that is, $\sigma_k^t = \sigma_j^s$, and we denote the dependence of f_k on f_j by $f_k \to f_j$.*

To describe the dependency relations among the VSs, we create the directed graph $\mathcal{G} = (\mathcal{F}, \mathcal{E})$, which we call the dependency graph. Each vertex in $\mathcal{G}$ corresponds to a VS, and there is an edge $(f_k, f_j) \in \mathcal{E}$ if f_k depends on f_j. The dependency graph can be built in polynomial time using Algorithm 1, which goes through each VS to check if an edge needs to be added according to the function $F^s(s)$.

In the dependency graph $\mathcal{G}$, the VSs can be classified into four categories according to their location.

- A VS is a head VS if it has one incoming edge but no outgoing edge.
- A VS is an interior VS if it has one incoming edge and one outgoing edge.
- A VS is a tail VS if it has one outgoing edge but no incoming edge.
- A VS is isolated if it is not incident to any edge.

Observe that the dependency graph consists of a set $\mathcal{P}_a$ of disjoint linear subgraphs, and of a set $\mathcal{P}_c$ of disjoint circular subgraphs. We refer to the subgraphs in $\mathcal{P}_a$ and $\mathcal{P}_c$ as acyclic and as cyclic paths, respectively. Figure 2 shows a sample dependency graph of eight VSs. The subgraph induced by $\{f_1, f_2, f_3, f_4\}$ is an acyclic path, and f_1 and f_4 are the head and the tail VSs, respectively. The subgraph induced by $\{f_5, f_6, f_7\}$ is a cyclic path. Finally, f_8 is isolated.

An acyclic path can be used to generate a migration schedule that does not interrupt any VS, by migrating the VSs from their source slots to their target slots starting with the head VS following the edges in the opposite direction. Nevertheless, a cyclic path has to be converted into an acyclic path to be able to generate a migration schedule. We discuss this step in the following.

Algorithm 2: Breaking Cyclic Paths

```
    Input  : Cyclic path $P_c = \{f_1, f_2, \dots, f_{|P_c|}\}$
1   if $S_0 \neq \emptyset$ then
2       for $\forall f_k \in P_c$ do
3           if $\exists s \in \mathcal{S}_0 \cap \mathcal{S}_m$ where $m = M_k^t$ then
4               Set $\sigma_k^t = s$
5               $S_0 = S_0 \setminus s$
6               Break
7       Migrate $f_1$ temporarily to $s$
8       Migrate $f_1$ to $\sigma_1^t$ after $f_{|P_c|}$ has been migrated
9       $S_0 = S_0 \setminus s$
10  else
11      Let $k' = argmin\{v_k | f_k \in P_c\}$
12      $P_{ac} = \{f_{k'+1}, f_{k'+2}, \dots, f_{|\mathcal{F}_c|}, f_1, f_2, \dots, f_{k'}\}$.
    Output : $P_{ac}$
```

C. Breaking cycles

In what follows we propose an efficient algorithm to convert cyclic paths to acyclic paths. The pseudo-code of the algorithm is shown in Algorithm 2. The algorithm considers a cyclic path $P_c = \{f_1, f_2, \dots, f_{|P_c|}\} \in \mathcal{P}_c$,

$$f_1 \leftarrow f_2 \leftarrow \dots \leftarrow f_{|P_c|}, \text{ and} f_{|P_c|} \leftarrow f_1. \tag{13}$$

If there are available MEC slots ($S_0 \neq \emptyset$), the cyclic path P_c can be converted to an acyclic path without interrupting any VS in one of two ways. If there exists a slot $s \in S_0$ on M_k^t for any VS $f_k \in P_c$, the algorithm reassigns s as the target slot of f_k, which converts P_c into an acyclic path P_{ac}. Otherwise, if there is no available slot $s \in S_0$ on any of the target MEC nodes of the VS $f_k \in P_c$, then the algorithm first migrates VS f_1 temporarily to an empty slot s, and then migrates f_1 to its target slot σ_1^t after $f_{|P_c|}$ has been migrated. This way the algorithm converts the cyclic path to an acyclic path at the cost of adding an extra migration round.

Finally, if there is no available MEC slot ($S_0 = \emptyset$), the algorithm converts the cyclic path by interrupting the cheapest VS in P_c for $|P_c|$ rounds, as shown in Lines 11-12. The following theorem shows that the total value of the VSs interrupted by Algorithm 2 is minimal.

Theorem 1. *For a cyclic path P_c, the sum value of the VSs interrupted by the solution given by Algorithm 2 is minimal.*

Proof. It is clear that when S_0 is not empty, the cyclic path P_c can be converted to an acyclic path without interrupting any VS, and therefore the sum value of the interrupted VSs is 0, which is minimal.

Consider now that S_0 is empty, and assume that the algorithm converts P_c into a set $\mathcal{P}_{ac}$ of acyclic paths. Let us denote by $|P_{ac}|$ the length of path $P_{ac} \in \mathcal{P}_{ac}$, and by v_{ac} the value of the tail VS of P_{ac}. Then the sum value C of the interrupted VSs in $\mathcal{P}_{ac}$ during the migration is $C = \sum_{P_{ac} \in \mathcal{P}_{ac}} |P_{ac}| v_{ac}$.

Let us denote by $v^\star = min\{v_k | f_k \in P_c\}$. Then for any $\mathcal{P}_{ac}$ we have

$$C = \sum_{P_{ac}\in\mathcal{P}_{ac}} |P_{ac}| v_{ac}$$
$$\geq \sum_{P_{ac}\in\mathcal{P}_{ac}} |P_{ac}| v^\star = v^\star \sum_{P_{ac}\in\mathcal{P}_{ac}} |P_{ac}| = v^\star |P_c| \quad (14)$$

Therefore when $S_0 = \emptyset$, the minimal sum value of the interrupted VSs for converting P_c into $\mathcal{P}_{ac}$ is $v^\star|P_c|$. When $S_0 = \emptyset$ Algorithm 2 interrupts the VS $f_{k'}$ with $v_{k'} = v^\star$ for $|P_c|$ rounds at the cost of $v^\star|P_c|$, which is the minimal sum value of the interrupted VSs, and $\mathcal{P}_{ac} = \{P_{ac}\}$. □

D. Cutting long acyclic paths

After executing Algorithm 2 for each cyclic path $P_c \in \mathcal{P}_c$, all the paths are acyclic, but the length of some paths may exceed the migration time constraint T. We refer to such paths as long paths, and as a next step, we have to cut those into paths that are at most T long, referred to as short paths.

Let us consider an acyclic long path $P_l = \{f_1, f_2, \ldots, f_{|P_l|}\}$,

$$f_1 \leftarrow f_2 \leftarrow \ldots \leftarrow f_{|P_l|}.$$

Note that since the path is long, we have $T < |P_l|$. We use the decision variable $y_k \in \{0,1\}$ to denote whether P_l will be cut after $f_k \in P_l$, and we define $\mathbf{y} = \{y_1, y_2, \ldots, y_{|P_l|}\}$.

If we cut a long path into short subpaths, the tail VSs except $f_{|P_l|}$ will have to be interrupted for the duration of the whole migration period of the subpaths that they belong to. For example, if we cut P_l after f_3 (y_3=1), f_4 takes the slot of f_3 in the first migration round, and f_3 is started in the third round and starts providing service in the fourth round. Therefore f_3 is interrupted for three rounds, which is the length of the path.

To compute the duration of the interruption for each VS, we define the variable $y_{k,j} = \min\{y_k, \bar{y}_{k-1}, \bar{y}_{k-2}, \ldots, \bar{y}_j\}$, where $k > j$, and $\bar{y}_j$ is the negation of y_j. $y_{k,j} = 1$ if f_k is the tail VS on a subpath, and f_k and f_j are on the same subpath. If VS f_k and f_j are not on the same subpath, P_l must be cut at least once between f_j and f_k, and therefore there exists $\bar{y}_n = 0$ with $j \leq n \leq k-1$ and $y_{k,j} = 0$. Since the maximal length of a path is T, the maximal interruption period of any VS is $\left(\sum_{j=1}^{\min\{T-1,k-1\}} y_{k,j} + y_k\right)$ rounds. Therefore the interruption cost for solution $\mathbf{y}$ is

$$C(\mathbf{y}) = \left(\sum_{k=1}^{F-1} v_k \left(\sum_{j=1}^{\min\{T-1,i-1\}} y_{k,j} + y_k\right)\right). \quad (15)$$

We can thus formulate the problem of cutting a long path with the objective of minimizing the interruption cost as the following IP problem.

$$\underset{y_k, y_{k,j}, b_{k,j,r}}{\text{minimize}} \quad C(\mathbf{y}) \quad \text{(P2)}$$

$$\text{subject to} \quad \sum_{a=0}^{T-1} y_{k+a} \geq 1, \ \forall k \leq |P_l| - T \quad (16)$$

$$y_{k,j} > y_k - b_{k,j,0}, \ \forall k, j \quad (17)$$

$$y_{k,j} > 1 - y_k - b_{k,j,r}, \ \forall k, j, j \leq r < k \quad (18)$$

$$\sum_{r=0}^{j} b_{k,j,r} = j, \quad (19)$$

$$y_k, y_{k,j}, b_{k,j,r} \in \{0,1\} \ \forall k, j, j \leq r < k. \quad (20)$$

Constraint (16) enforces that every $T+1$ successive VSs on P must be on at least two different subpaths. Constraints (17)-(19) are the so called "big M constraints" according to the definition of $y_{k,j}$ (Section 3.2.4 in [17]), and $b_{k,j,r}$ are slack variables. Our next result shows that the long path cutting problem can be solved efficiently when the values of the VSs are homogeneous.

Theorem 2. *Consider an acyclic long path* $P_l = \{f_1, f_2, \ldots, f_{|P_l|}\}$ *and time constraint* $T < |\mathcal{P}_l|$. *If* $v_k = v_0 \ \forall f_k \in P_l$ *then there exist optimal solutions with cost* $v_0(|P_l| - T)$.

Proof. Without loss of generality, we assume that in a feasible solution of the long path cutting problem P_l has been divided into a set of subpaths $\mathcal{P}_s = \{P_1, P_2, \ldots, P_{|\mathcal{P}_s|}\}$. Since the VSs have the same value v_0, the total interruption cost C for the solution is

$$C = \sum_{P_p \neq P_{|\mathcal{P}_s|}} v_0 |P_p| = v_0 \sum_{P_p \neq P_{|\mathcal{P}_s|}} |P_p| = v_0 \left(|P_l| - |P_{|\mathcal{P}_s|}|\right). \quad (21)$$

Since the maximal length of subpath $P_{|\mathcal{P}_s|}$ is T, therefore the minimal cost $C^\star = v_0(|P_l| - T)$. Further we can construct the optimal solution as follows. First we cut P_l after the first $(|P_l| - T)$ VSs such that the last subpath contains the last T VSs of P_l. Then we cut the rest of P_l arbitrarily such that the length of each subpath is at most T. □

In what follows we use Theorem 2 for developing a heuristic for the general case, based on local search. The pseudo-code of the heuristic is shown in Algorithm 3. The heuristic first obtains a feasible solution by creating a set $\mathcal{P}_s$ of subpaths of length T starting from the last VS in P_l, and updates $\mathbf{y}$ accordingly. Then the heuristic iteratively goes through each VS f_k to check if the total interruption cost can be reduced by applying an operation to f_k. The set O_k of applicable operations for f_k depends on the location of f_k, and is shown in Table III. If such an operation exists, it applies the operation, otherwise it keeps the previous solution. The algorithm terminates when the total interruption cost cannot be further reduced.

E. Migration dependence graph decomposition algorithm

We are now ready to summarize the migration dependence graph decomposition (DGD) algorithm. The pseudo-code of the algorithm is shown in Algorithm 4. First we initialize the set $\mathcal{S}$ of MEC slots, $F^s(s)$ and $F^t(s)$ (Line 1). We then create a set S_m of MEC slots for each MEC node m, and append S_m to $\mathcal{S}$, as shown in Lines 2-4. We assign each VS f_k a source and a target slot and update the functions $F^s(s)$ and $F^t(s)$ (Lines 5-8). Next, we build the dependency graph $\mathcal{G}$ according to Algorithm 1 (Line 9), and convert the cyclic subgraphs of $\mathcal{G}$ to acyclic subgraphs using Algorithm 2 (Lines 10-12). Finally,

Algorithm 3: Cutting Long Paths

```
Input    : Acyclic long path $P_l = \{f_1, f_2, \dots, f_{|P_l|}\}$ and $T$
1 Create a set $\mathcal{P}_s$ of subpaths of length $T$ starting from the last VS in $P_l$, and update $\mathbf{y}$ accordingly.
2 Interruption cost $C = C(\mathbf{y})$
3 while $C$ can be reduced do
4     for $\forall f_k \in P_l$ do
5         for $\forall o \in O_k$ do
6             Apply the operation $o$ and update $\mathbf{y}$
7             $C' = C(\mathbf{y})$
8             if $C' < C$ then
9                 $C = C'$
10            else
11                Discard $o$ and restore $\mathbf{y}$
Output   : $\mathcal{P}_s$
```

Table III
AVAILABLE OPERATIONS FOR VS f_k

	Head	Interior	Tail	Isolated
Move f_k to the previous path	✓			✓
Move f_k to the next path			✓	✓
Make f_k isolated	✓	✓	✓	
Cut the path after f_k		✓		
Cut the path after f_{k-1}		✓		

we cut each long path P_l into a set $\mathcal{P}_s$ of short paths (Lines 13-15) to replace P_l in $\mathcal{P}_a$. The algorithm results in a set $\mathcal{P}_a$ of acyclic short paths, which can be used to generate a VS migration schedule within the time constraint T.

The following result shows that the DGD algorithm computes the optimal solution to the TCVM problem under certain conditions.

Theorem 3. *Consider a TCVM problem with $|\mathcal{M}| = |\mathcal{F}|$, $\omega_m = 1\ \forall m$, and $v_k = v_0\ \forall f_k$. Then the solution given by the DGD algorithm is optimal.*

Proof. We prove the theorem by contradiction. We denote by $\mathbf{x}_h$ the solution of the DGD algorithm, and assume that $\mathbf{x}_h$ is not optimal. When $\omega_m = 1\ \forall m$, the assignment σ_k^s and σ_k^t are unique for each VS f_k, and therefore the dependency graph $\mathcal{G}$ is unique. The suboptimality of $\mathbf{x}_h$ thus requires that the result of at least one of the Algorithms 2 and 3 is not optimal with respect to its inputs.

Since $|\mathcal{M}| = |\mathcal{F}|$ and $\omega_m = 1\ \forall m$, and each MEC slot must host one VS both in $\mathbf{x}_s$ and $\mathbf{x}_t$, we know that $S_0 = \emptyset$. Therefore, Algorithm 2 is optimal with respect to its input, according to Theorem 1. Furthermore, Theorem 2 shows that when the values of all VSs are homogeneous, Algorithm 3 is also optimal with respect to its input. Since both Algorithm 2 and 3 are optimal with respect to their inputs when $|\mathcal{M}| = |\mathcal{F}|$, $\omega_m = 1\ \forall m$, and the value of all the VSs are homogeneous, $\mathbf{x_h}$ must be optimal. □

Algorithm 4: DGD Algorithm

```
1 $\mathcal{S} = \emptyset$, $F^s(s) : \mathcal{S} \mapsto \{\emptyset\}$, and $F^t(s) : \mathcal{S} \mapsto \{\emptyset\}$
2 for $\forall\ m \in \mathcal{M}$ do
3     Create a set $S_m$ of MEC slots with $|S_m| = \omega_m$
4     $\mathcal{S} = \mathcal{S} \cup S_m$
5 for $\forall\ f_k \in \mathcal{F}$ do
6     Assign $f_k$ a source slot $\sigma_k^s$ and a target slot $\sigma_k^t$
7     Add $\sigma_k^s \mapsto f_k$ to function $F^s(s)$
8     Add $\sigma_k^t \mapsto f_k$ to function $F^t(s)$
9 Build dependency graph $\mathcal{G} = \mathcal{P}_c \cup \mathcal{P}_a$ by Algorithm 1
10 for $\forall\ P_c \in \mathcal{P}_c$ do
11     Convert $P_c$ into acyclic path $P_{ac}$ by Algorithm 2
12     $\mathcal{P}_a = (\mathcal{P}_a \backslash P_c) \cup P_{ac}$
13 for $\forall$ long path $P_l \in \mathcal{P}_a$ do
14     Cut $P_l$ into a set $\mathcal{P}_s$ of short paths by Algorithm 3
15     $\mathcal{P}_a = (\mathcal{P}_a \backslash P_l) \cup \mathcal{P}_s$
Output   : $\mathcal{P}_a$
```

Besides the complete DGD algorithm that is described in Algorithm 4, it is tempting to consider two variants of it. The first variant solves (P2) instead of using Algorithm 3 to cut the long paths, and thus can potentially provide a higher total service value than the complete DGD algorithm. The other variant is the DGD algorithm that skips the local search in Algorithm 3, and has a lower running time than the complete DGD algorithm. We refer to the former and the latter variant of the DGD algorithm as DGD-optPB and DGD-noLS algorithm, respectively. As we next show, the DGD-noLS algorithm is indeed computationally lightweight.

Theorem 4. *The DGD-noLS algorithm has worst case complexity $\mathcal{O}(|\mathcal{M}| + |\mathcal{F}|)$.*

Proof. It is easy to see that Lines 2-4 and Lines 5-8 in Algorithm 4 have complexity $\mathcal{O}(|\mathcal{M}|)$ and $\mathcal{O}(|F|)$, respectively. Furthermore, the DGD-noLS algorithm calls Algorithm 1, which has complexity $\mathcal{O}(|F|)$.

In terms of breaking a cyclic path P_c, the worst case is when $S_0 \neq \emptyset$ and Lines 2-9 in Algorithm 2 are executed. Since Lines 2-6 has complexity $\mathcal{O}(|P_c|)$, and Lines 7-9 has complexity $\mathcal{O}(1)$, the Algorithm 2 has worst case complexity $\mathcal{O}(|P_c|+1)$. When all the paths are cyclic, it takes $\mathcal{O}(\sum_{P_c \in \mathcal{P}_c}(|P_c|+1) \leq \mathcal{O}(2|\mathcal{F}|) = \mathcal{O}(|\mathcal{F}|)$ time to break all the cyclic paths.

Finally, the Line 1 of Algorithm 3 is executed for each long path, and has complexity $\mathcal{O}(|P_l|)$ to set $\mathbf{y}$. In the worst case that all the paths are long, cutting the long paths takes $\mathcal{O}(\sum_{P_l \in \mathcal{P}_c} |P_l|) = \mathcal{O}(|\mathcal{F}|)$ time. Therefore the DGD-noLS algorithm has worst case complexity $\mathcal{O}(|\mathcal{M}| + |\mathcal{F}|)$. □

VI. NUMERICAL RESULTS

In this section we evaluate the performance of the DGD algorithm in terms of efficiency, scalability, and total service

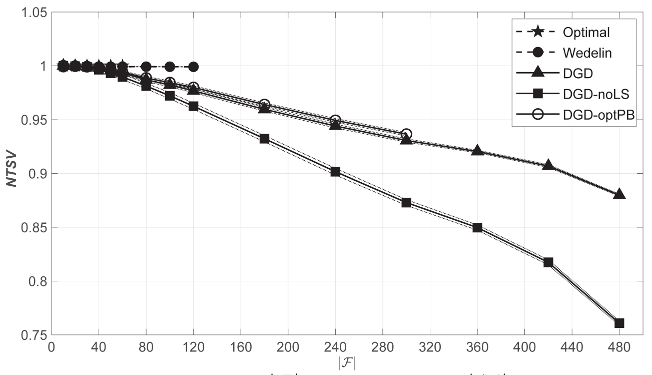

Figure 3. NTSV vs. $|\mathcal{F}|$ for a system with $|\mathcal{M}|$ =80.

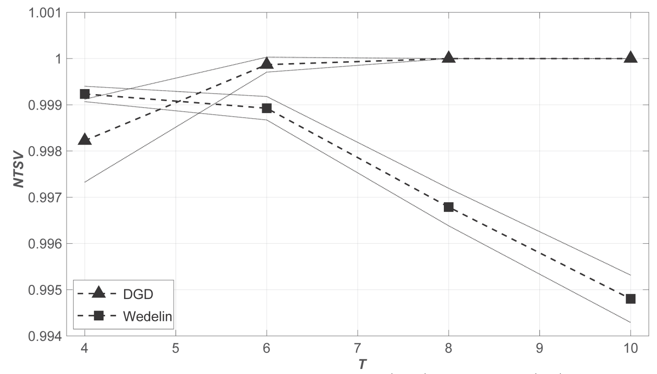

Figure 4. NTSV vs. T for a system with $|\mathcal{M}| = 80$ and $|\mathcal{F}| = 40$ for the DGD algorithm and the Wedelin heuristic.

value. We simulated systems with $\omega_m = 6, \forall m$ and v_k is a uniformly distributed integer variable on $[1, 50]$, representing the priority of f_k according to the importance of its real-time performance from low to high. For example, services with $v_k = 1$ could correspond to predictive maintenance applications for automotive and smart grid applications, while services with $v_k = 50$ could correspond to real-time state estimation and UAV control, corresponding to the highest importance. In each simulation the source node M_k^s and the target node M_k^t of VS $f_k \in \mathcal{F}$ are uniformly selected from the MEC nodes with free computing resources. The results shown are the averages of 100 simulations, and the confidence intervals are at the 95% confidence level.

To benchmark the DGD algorithm, we compare it with the optimal solution and the Wedelin heuristic. The optimal solution is obtained by using the MILP solver of the Matlab Optimization Toolbox, which is based on branch and bound. The Wedelin heuristic is a general heuristic for IPs in the following form,

$$\begin{aligned} \underset{\mathbf{z}}{\textbf{maximize}} \quad & \mathbf{cz} \\ \textbf{subject to} \quad & A\mathbf{z} = \mathbf{b} \end{aligned} \tag{P3}$$

where $\mathbf{z}$ is the vector of binary variables, and A is a matrix with elements belonging to $\{0, 1, -1\}$. The Wedelin heuristic is based on the Lagrangian relaxation of (P3), and modifies the coefficients of the objective function iteratively. The Wedelin heuristic is widely used to solve large scale IPs, for example the crew scheduling problem in the airline industry [18], [19].

A. Service value performance

We start with evaluating the service value performance of the DGD algorithm. To compare the service value over different scenarios on a common scale, we define the normalized total service value (NTSV) as follows,

$$\text{NTSV} = \frac{\sum_{i=1}^{T} \sum_{f_k \in \mathcal{F}} \sum_{m \in \mathcal{M}} v_k \left(x_{m,k,i} - e_{m,k,i}\right)}{T \sum_{f_k \in \mathcal{F}} v_k}. \tag{22}$$

Figure 3 shows the NTSV of the five algorithms as a function of the number $|\mathcal{F}|$ of VSs, for a system with $|\mathcal{M}| = 80$ and $T = 4$. The optimal solution, the Wedelin heuristic, and the DGD-optPB only scale up to $|\mathcal{F}| = 60$, $|\mathcal{F}| = 120$, and $|\mathcal{F}| = 300$, respectively, limited by their scalability. The figure shows that the DGD algorithm and the DGD-noLS algorithm perform close to the optimal solution when $|F| \le 60$. The performance gap between the DGD algorithm and the Wedelin heuristic is within three percent until the Wedelin heuristic is not scalable anymore. Furthermore, the results show that the DGD algorithm outperforms the DGD-noLS algorithm by more than ten percent when the number of VSs is high, and performs close to the DGD-optPB algorithm. This shows that the local search is an essential component of the DGD algorithm, and contributes significantly to maintaining a high NTSV. The results also show that the NTSV of the DGD algorithm decreases as $|\mathcal{F}|$ increases. However, note that as $|\mathcal{F}|$ increases, the optimal NTSV also decreases as a result of the decreasing free MEC capacity. As an extreme example, when $|\mathcal{F}| = 480$ it is easy to see that migrating any VS without any VS interruption is infeasible.

In what follows we compare the performance of the DGD algorithm and the Wedelin heuristic with respect to T. Figure 4 shows the NTSV of the DGD algorithm and of the Wedelin heuristic for a system with $|\mathcal{M}| = 80$ and $|\mathcal{F}| = 40$. The results show that the Wedelin heuristic performs better than the DGD algorithm when $T = 4$, which is the same as Figure 3 shows. However, the NTSV of the DGD algorithm increases as T increases, and the DGD algorithm outperforms the Wedelin heuristic when $T \ge 6$. This is because as the T increases, there are less VSs to be interrupted to cut long paths into short paths in the DGD algorithm. However, increasing the T increases the dimension of the IP of the TCVM problem, and thus the performance of the Wedelin heuristic degrades.

Figure 5 shows the NTSV of the DGD algorithm for four combinations of $|\mathcal{M}|$ and $|\mathcal{F}|$ with respect to T. The results show that increasing T increases the NTSV of the DGD algorithm, which is consistent with Figure 4. Comparison of the dashed curves also shows that for the same $|\mathcal{M}|$, the NTSV of the DGD algorithm decreases as $|\mathcal{F}|$ increases. This is because increasing $|\mathcal{F}|$ leads to a higher chance to get cyclic paths and long paths, and at the same time the number of available MEC slots decreases. A similar reasoning explains the phenomenon that for the same $|\mathcal{F}|$ the NTSV is higher when $|\mathcal{M}|$ is larger, as the curves with triangular markers show. Practically, the trade-offs above allow to tune the migration performance by configuring the system load (e.g., $|\mathcal{F}|$) and T.

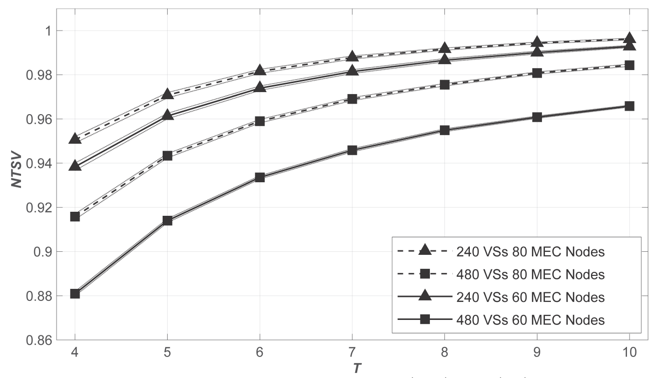

Figure 5. NTSV vs. T for combinations of $|\mathcal{M}|$ and $|\mathcal{F}|$ for the DGD algorithm.

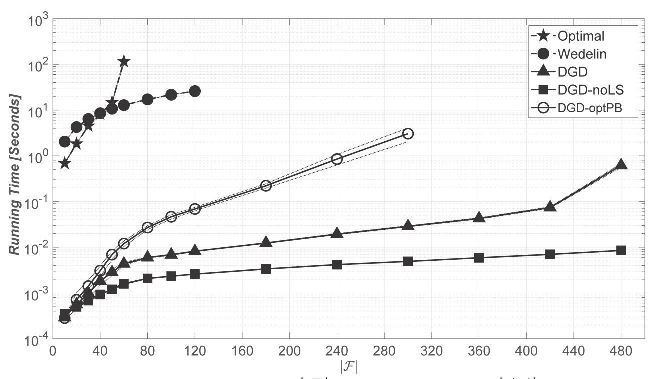

Figure 6. Running time vs. $|\mathcal{F}|$ for a system with $|\mathcal{M}| = 80$.

B. Efficiency and scalability

Figure 6 shows the running time of the four algorithms as a function of $|\mathcal{F}|$, for the same scenario as in Figure 3. Compared with the optimal solution and the Wedelin heuristic, the DGD algorithm and its two variants consume orders of magnitude lower running time, and scale significantly better. The running time of the optimal solution increases exponentially as $|F|$ increases. It is interesting to see that the optimal solution and the Wedelin heuristic intersect at $|F| = 40$. This is because when the dimension of the problem is relatively small, it is easier for the optimization solver to find the optimal solution, while the Wedelin heuristic still needs to go through the constraints and solve the problem iteratively. The scalability of the Wedelin heuristic is mainly limited by the fact that it requires the coefficient matrices of the problem instances to be stored. Furthermore, Figure 6 shows that the superior performance of the DGD-optPB algorithm comes at the cost of a significant increase of the running time. This is because the DGD-optPB algorithm obtains an optimal solution for breaking a long path by solving an IP. The DGD-noLS algorithm skips the local search to achieve a lower running time than the DGD algorithm at the cost of a lower NTSV. This observation allows to tune the computational complexity and the NTSV performance by choosing among the DGD algorithm and its variants.

Overall, the results show that the DGD algorithm is an effective, efficient, and scalable algorithm for solving the TCVM problem, and allows to balance between the solution quality, VS migration time, and the computation complexity of finding a solution.

VII. Conclusion

We have proposed an algorithm for solving the problem of scheduling virtualized services migration with the objective of maximizing service availability subject to a target migration time. The iterative algorithm we proposed builds and decomposes a migration dependency graph to generate a migration schedule. Extensive numerical results show that the proposed algorithm achieves near-optimal performance with low computational complexity, and scales significantly better than the state of the art Wedelin heuristic for solving binary programming problems. Furthermore, the numerical results show that the proposed algorithm allows to balance between the solution quality, migration time, and the computation complexity of finding a solution.

References

[1] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, 2017.

[2] 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.

[3] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, 2017.

[4] 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.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE IoT Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[6] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE IoT Journal*, 2017.

[7] Y. Wen, Z. Li, S. Jin, C. Lin, and Z. Liu, "Energy-efficient virtual resource dynamic integration method in cloud computing," *IEEE Access*, vol. 5, pp. 12 214–12 223, 2017.

[8] P. Zhao and G. Dán, "Resilient placement of virtual process control functions in mobile edge clouds," in *Proc. of IFIP TC6 Networking*, June, 2017.

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

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

[11] 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.

[12] X. Yao, H. Wang, C. Gao, F. Zhu, and L. Zhai, "VM migration planning in software-defined data center networks," in *Proc. of IEEE HPCC/SmartCity/DSS*, 2016, pp. 765–772.

[13] 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.

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

[15] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: a consolidation manager for clusters," in *Proc. of ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2009, pp. 41–50.

[16] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[17] E. K. Burke, G. Kendall *et al.*, *Search methodologies*. Springer, 2005.

[18] D. Wedelin, "An algorithm for large scale 0–1 integer programming with application to airline crew scheduling," *Annals of operations research*, vol. 57, no. 1, pp. 283–301, 1995.

[19] ——, "Revisiting the in-the-middle algorithm and heuristic for integer programming and the max-sum problem," 2013.