Placement is not Enough: Embedding with Proactive Stream Mapping on the Heterogenous Edge

Hailiang Zhao†, Shuiguang Deng†, Zijie Liu†, Zhengze Xiang‡, and Jianwei Yin†
†College of Computer Science and Technology, Zhejiang University
‡College of Computer Science and Technology, Zhejiang University City College

Abstract—Edge computing is naturally suited to the applications generated by Internet of Things (IoT) nodes. The IoT applications generally take the form of directed acyclic graphs (DAGs), where vertices represent interdependent functions and edges represent data streams. The status quo of minimizing the makespan of the DAG motivates the study on optimal function placement. However, current approaches lose sight of proactively mapping the data streams to the physical links between the heterogenous edge servers, which could affect the makespan of DAGs significantly. To solve this problem, we study both function placement and stream mapping with data splitting simultaneously, and propose the algorithm DPE (Dynamic Programming-based Embedding). DPE is theoretically verified to achieve the global optimality of the embedding problem. The complexity analysis is also provided. Extensive experiments on Alibaba cloud trace dataset show that DPE significantly outperforms two state-of-the-art joint function placement and task scheduling algorithms in makespan by 43.19% and 40.71%, respectively.

I. INTRODUCTION

Nowadays, widely used stream processing platforms, such as Apache Spark, Apache Flink, Amazon Kinesis Streams, etc., are designed for large-scale data-centers. These platforms are not suitable for real-time and latency-critical applications running on widely spread Internet of Things (IoT) nodes. By contrast, near-data processing within the network edge is a more applicable way to gain insights, which leads to the birth of edge computing. However, state-of-the-art edgy stream processing systems, for example, Amazon Greengrass, Microsoft Azure IoT Edge and so on, do not consider that how the dependent functions of the IoT applications distributed to the resource-constrained edge. To address this limitation, works studying function placement across the distributed edge servers spring up [1]–[3]. In these works, the IoT application is structured as a service function chain (SFC) or a directed acyclic graph (DAG) composed of interdependent functions, and the placement strategy of each function is obtained by minimizing the makespan of the application, under the trade-off between node processing time and cross-node communication overhead.

However, when minimizing the makespan of the application, state-of-the-art approaches only optimize the placement of functions, while passively generate the stream mapping. Here the stream mapping refers to mapping the input/output streams to the physical links between edge servers. The passivity here means the routing path between each placed function is not optimized but generated automatically through SDN controllers. Nevertheless, for the heterogenous edge, where each edge server has different processing power and each link has various throughput, better utilization of the stream mapping can result in less makespan even though the corresponding function placement is worse. This phenomenon is illustrated in Fig. 1. The top half of this figure is an undirected connected graph of six edge servers, abstracted from the physical infrastructure of the heterogenous edge. The numbers tagged in each node and beside each link of the undirected graph are the processing power (measured in flop/s) and throughput (measured in bit/s), respectively. The bottom half is a SFC with three functions. The number tagged inside each function is the required processing power (measured in flops). The number tagged beside each data stream is the size of it (measured in bits). Fig. 1 demonstrates two solutions of function placement. The numbers tagged beside nodes and links of each solution are the time consumed (measured in second). Just in terms of function placement, solution 1 enjoys lower function processing time (2.5s < 4s), thus performs better. However, the makespan of solution 2 is 1.5s lesser than solution 1 because the path which solution 2 routes through possess a higher throughput.

Fig. 1. Two function placement solutions with different stream mappings.

The above example implies that different stream mappings can significantly affect the makespan of the IoT applications. It enlightens us to take the stream mapping into consideration proactively. In this paper, we name the combination of function placement and stream mapping as function embedding. Moreover, if stream splitting is allowed, i.e., the output data stream of a function can be splitted and route on multiple paths, the makespan can be decreased further. This phenomenon is captured in Fig. 2. The structure of this figure is the same as Fig. 1. It demonstrates two function embedding
solutions with stream splitting allowed or not, respectively. In solution 2, the output stream of the first function is divided into two parts, with 2 bits and 3 bits, respectively. Correspondingly, the time consumed on routing are 3s and 2.5s, respectively. Although the two solutions have the same function placement, the makespan of solution 2, which is calculated as $1 + \max\{3, 2.5\} + 1 + 1.5 + 1 = 7.5$s, is 4.5s lesser than solution 1. In practice, segment routing (SR) can be applied to split and route data stream to different edge servers by commercial DNS controllers and HTTP proxies [4].

The remainder of the paper is organized as follows. In Sec. I [1] we present the system model and formulate the problem. In Sec. III [6] we present the proposed algorithms. Performance guarantee and complexity analysis are provided in Sec. IV. The experiment results are demonstrated in Sec. V. In Sec. VI [19] we review related works on functions placement on the heterogenous edge. Sec. VII concludes this paper.

II. SYSTEM MODEL

Let us formulate the heterogenous edge as an undirected connected graph $G \triangleq (\mathcal{N}, \mathcal{L})$, where $\mathcal{N} \triangleq \{n_1, ..., n_N\}$ is the set of edge servers and $\mathcal{L} \triangleq \{l_1, ..., l_L\}$ is the set of links. Each edge server $n \in \mathcal{N}$ has a processing power $\psi_n$, measured in flop/s while each link $l \in \mathcal{L}$ has the same uplink and downlink throughput $b_l$, measured in bit/s.

A. Application as a DAG

The IoT application with interdependent functions is modeled as a DAG. The DAG can have arbitrary shape, not just linear SFC. In addition, multi-entry multi-exit is allowed. We write $(\mathcal{F}, \mathcal{E})$ for the DAG, where $\mathcal{F} \triangleq \{f_1, ..., f_Q\}$ is the set of $Q$ interdependent functions listed in topological order. $\forall f_i, f_j \in \mathcal{F}, i \neq j$, if the output stream of $f_i$ is the input of its downstream function $f_j$, a directed link $e_{ij}$ exists. $\mathcal{E} \triangleq \{e_{ij} | \forall f_i, f_j \in \mathcal{F}\}$ is the set of all directed links. For each function $f_i \in \mathcal{F}$, we write $c_i$ for the required number of floating point operations of it. For each directed link $e_{ij} \in \mathcal{E}$, the data stream size is denoted as $s_{ij}$ (measured in bits).

B. Dependent Function Embedding

We write $p(f_i) \in \mathcal{N}$ for the chosen edge server which $f_i$ is to be placed on, and $\mathcal{P}(e_{ij})$ for the set of paths from $p(f_i)$ to $p(f_j)$. Obviously, for all path $\varrho \in \mathcal{P}(e_{ij})$, it consists of links from $\mathcal{L}$ without duplication. For a function pair $(f_i, f_j)$ and its associated directed link $e_{ij} \in \mathcal{E}$, the data stream can be splitted and route through different paths from $\mathcal{P}(e_{ij})$. $\forall \varrho \in \mathcal{P}(e_{ij})$, let us use $z_{\varrho}$ to represent the allocated data stream size for path $\varrho$. Then, $\forall e_{ij} \in \mathcal{E}$, we have the following constraint:

$$\sum_{\varrho \in \mathcal{P}(e_{ij})} z_{\varrho} = s_{ij}. \quad (1)$$

Notice that if $p(f_i) = p(f_j)$, i.e., $f_i$ and $f_j$ are placed on the same edge server, then $\mathcal{P}(e_{ij}) = \emptyset$ and the routing time is zero. Fig. 2 gives a working example. The connected graph in it has four edge servers and five links, from $l_1$ to $l_5$. The two squares represents the source function $f_i$ and the destination function $f_j$. From the edge server $p(f_i)$ to the edge server
p(f_i), e_{ij} routes through three paths with data size of 3 bits, 2 bits, and 1 bits, respectively. In this example, p_{ij} = 6. On closer observation, we can find that two data streams route through l_1. Each of them is from path q_1 and q_2 with 3 bits and 2 bits, respectively.

![Diagram](image)

**Fig. 3. A working example for stream splitting.**

**C. Involution Function of Finish Time**

Let us use $T(p(f_i))$ to denote the finish time of $f_i$ on edge server $p(f_i)$. Considering that the functions of the DAG have interdependent relations, for each function pair $(f_i, f_j)$ where $e_{ij}$ is defined, $T(p(f_j))$ should involve according to

$$T(p(f_j)) = \max_{e_{ij} \in E} \left( T(p(f_i)) + t(e_{ij}) \right) + t(p(f_j)),$$

where $t(e_{ij})$ is the routing time of the directed link $e_{ij}$ and $t(p(f_j))$ is the processing time of $f_j$ on edge server $p(f_j)$. Corresponding to (2), for each entry function $f_i$,

$$T(p(f_i)) = t(p(f_i)) + r_{p(f_i)},$$

where $r_{p(f_i)}$ stores the beginning time for processing $f_i$. If $f_i$ is the first function which is scheduled on server $p(f_i)$, $r_{p(f_i)}$ is zero. Otherwise, it is the finish time of the last function which is scheduled on server $p(f_i)$.

For all $e_{ij} \in E$, $t(e_{ij})$ is decided by the paths in $P(e_{ij})$. Specifically, when $f_i$ and $f_j$ are not placed on the same edge server,

$$t(e_{ij}) = \max_{e \in P(e_{ij})} \sum_{l \in E} z_{l_f} b_l,$$

where $1 \{ \cdot \}$ is the indicator function. (4) means that the routing time is decided by the smallest branch of data streams. Solution 2 in Fig. 2 is an example.

For all $f_i \in F$, $T(p(f_i))$ is decided by the processing power of the chosen edge server $p(f_j)$:

$$t(p(f_i)) = \frac{c_j}{\psi_{p(f_j)}}.$$

To describe the earliest makespan of the DAG, inspired by (1), we add a dummy tail function $f_{Q+1}$. As a dummy function, the processing time of $f_{Q+1}$ is set as zero whichever edge server it is placed on. That is,

$$t(p(f_{Q+1})) = 0, \forall p(f_{Q+1}) \in N.$$

Besides, each destination function needs to point to $f_{Q+1}$ with a link weight of its output data stream size. We write $F_{dst} \subset F$ for the set of destination functions and $\mathcal{F}_{src}$ for the set of entry functions of the DAG. In addition, we write $\mathcal{E}_{dmy}$ for the set of new added links which point to $f_{Q+1}$, i.e.,

$$\mathcal{E}_{dmy} \doteq \{ e_{i,Q+1} \forall f_i \in \mathcal{F}_{dst} \}.$$

As such, the DAG is updated as $(\mathcal{F}', \mathcal{E}')$, where $\mathcal{F}' \doteq \mathcal{F} \cup \{ f_{Q+1} \}$ and $\mathcal{E}' \doteq \mathcal{E} \cup \mathcal{E}_{dmy}$.

**D. Problem Formulation**

Our target is to minimize the makespan of the DAG by finding the optimal $p(f_i)$, $P(e_{ij})$, and $z_{l_f}$ for all $f_i \in \mathcal{F}'$, $e_{ij} \in \mathcal{E}'$, and $p \in P(e_{ij})$. Let us use $T^*(p(f_i))$ to represent the earliest finish time of $f_i$ on edge server $p(f_i)$. Further, we use $T(\mathcal{F}', \mathcal{E}', p(f_{Q+1}))$ to represent the earliest makespan of the DAG when $f_{Q+1}$ is placed on $p(f_{Q+1})$. Obviously, it is equal to $T^*(p(f_{Q+1}))$. The dependent function embedding problem can be formulated as:

$$\textbf{P} : \min_{\forall p(f_i), \forall P(e_{ij}), \forall z_{l_f}} \left[ T(\mathcal{F}', \mathcal{E}', p(f_{Q+1})) \right]$$

s.t. (1),

$$z_{l_f} \geq 0, \forall \varnothing p \in P(e_{ij}), \forall e_{ij} \in \mathcal{E}'.$$

**III. ALGORITHM DESIGN**

**A. Finding Optimal Substructure**

The set of function embedding problems are proved to be NP-hard. As a special case of these problems, $\textbf{P}$ is NP-hard, too. Because of the dependency relations of fore-and-aft functions, the optimal placement of functions and optimal mapping of data streams cannot be obtained simultaneously. Nevertheless, we can solve it by finding its optimal substructure.

Let us dig deeper into the involution equation (2). We can find that

$$T(p(f_j)) = \max_{e_{ij} \in E'} \left( T(p(f_i)) + t(e_{ij}) + t(p(f_j)) \right)$$

because $t(p(f_j))$ has no impact on $\max_{e_{ij} \in E'} t(e_{ij})$. Notice that $\mathcal{E}$ is replaced by $\mathcal{E}'$. Then the following expression holds:

$$T^*(p(f_j)) = \max_{e_{ij} \in E'} \left\{ \min_{p(f_i), P(e_{ij}), z_{l_f}} \right\} \left( T^*(p(f_i)) + t(e_{ij}) + t(p(f_j)) \right).$$

Besides, for all the entry functions $f_i \in \mathcal{F}_{src}$, $T^*(p(f_i))$ is calculated by (3) without change.

With (10), for each function pair $(f_i, f_j)$ where $e_{ij}$ exists, we define the sub-problem $\textbf{P}_{sub}$:

$$\textbf{P}_{sub} : \min_{p(f_i), P(e_{ij}), z_{l_f}} \phi_{ij} \doteq T^*(p(f_i)) + t(e_{ij}) + t(p(f_j))$$

s.t. (1), (8).

In $\textbf{P}_{sub}$, $p(f_j)$ is fixed. We need to decide where $f_i$ should be placed and how $e_{ij}$ is mapped. Based on that, $T^*(p(f_j))$ can be updated by $\max_{e_{ij} \in E'} \min_{p(f_i), P(e_{ij}), z_{l_f}} \phi_{ij}$. As a result, $\textbf{P}$ can be solved optimally by calculating the earliest finish time of each function in turn.
The analysis is captured in Algorithm 1, i.e. Dynamic Programming-based Embedding (DPE). Firstly, DPE finds all the simple paths (paths without loops) between any two edge servers \( n_1 \) and \( n_j \), i.e. \( \mathcal{P}(e_{ij}) \) (Step 2 ~ Step 6). It will be used to calculate the optimal \( z^*_\phi \) and \( \mathcal{P}^*(e_{ij}) \). For all the functions \( f_i \) who directly point to \( f_j \), DPE firstly fixes the placement of \( f_j \). Then, for the function pair \((f_i, f_j)\), \( \mathcal{P}_{\text{sub}} \) is solved and the optimal solution is stored in \( \Theta \) (Step 18 ~ Step 19). If \( \mathcal{T}^*(f_i) \) has been decided beforehand, \( \Phi^*_{ij} \) can be directly obtained by finding \( \mathcal{P}^*(e_{ij}) \) and \( z^*_\phi \) between \( \mathcal{T}^*(f_i) \) and \( n \). The optimal transmission cost is stored in \( f^*(e_{ij}) \) (Step 11 ~ Step 14). If the statement holds for \( f_i \in \mathcal{F}^\prime \) when \( f_i \) is a predecessor of multiple functions. Step 12 is actually a sub-problem of \( \mathcal{P}_{\text{sub}} \) with \( f^*(f_i) \) fixed. Notice that the calculation of the finish time of the entry functions \( \Phi \) is solved and the optimal solution is stored in \( \sub \). Notice that the calculation of the finish time of the entry functions and the other functions are different (Step 16 and Step 21, respectively). At last, the global minimal makespan of the DAG is \( \min_{p(f_{Q+1})} T^*(p(f_{Q+1})) \). The optimal embedding of each function can be retrieved from \( \Theta \).

**Algorithm 1: DP-based Embedding (DPE)**

**Input:** \( G \) and \( (\mathcal{F}^\prime, \mathcal{E}^\prime) \)

**Output:** Optimal value and corresponding solution

```
\( \Theta \leftarrow \emptyset \)

for each \( n_i \in \mathcal{M} \) do
  for each \( n_j \in \mathcal{M} \land n_j \neq n_i \) do
    Find all the simple paths between \( n_i \) and \( n_j \)
  end for
end for

for \( j = |\mathcal{F}_{\text{src}}| + 1 \ to \ Q + 1 \) do
  for each \( n \in \mathcal{N} \) do
    \( p(f_j) \leftarrow n \) /* Fix the placement of \( f_j \)
    for each \( f_i \in \{f_i | e_{ij} \text{ exists}\} \) do
      if \( \mathcal{T}^*(f_i) \) has been decided then
        \( \Phi^*_{ij} \leftarrow T^*(\mathcal{T}^*(f_i)) + T^*(e_{ij}) + t(p(f_j)) \)
        goto Step 19
      end if
      if \( f_i \in \mathcal{F}_{\text{src}} \) then
        \( \forall p(f_i) \in \mathcal{N}, \text{update } T^*(p(f_i)) \) by \( f \)
      end if
      Obtain the optimal \( \Phi^*_{ij}, \mathcal{T}^*(f_i), \mathcal{P}^*(e_{ij}) \), and \( z^*_\phi \) by solving \( \mathcal{P}_{\text{sub}} \)
      \( \Theta \) \( \cdot \) append \( \{p^*(f_i)\} \times \{\mathcal{P}^*(e_{ij})\} \times \{z^*_\phi | g \in \mathcal{P}^*(e_{ij})\} \)
    end for
    Update \( T^*(p(f_j)) \) by \( f \)
  end for
end for
return \( \min_{p(f_{Q+1})} T^*(p(f_{Q+1})) \) and \( \Theta \)
```

**B. Optimal Proactive Stream Mapping**

Now the problem lies in that how to find all the simple paths and solve \( \mathcal{P}_{\text{sub}} \) optimally. For the former, we propose a Recursion-based Path Finding (RPF) algorithm, which will be detailed in Sec. [III-B1]. For the latter, when \( p(f_j) \) is fixed, the value of \( t(p(f_i)) \) is known and can be viewed as a constant for \( \mathcal{P}_{\text{sub}} \). Besides, from Step 12 of DPE we can find that, for all \( f_i \in \{f_i \in \mathcal{F}^\prime - \mathcal{F}_{\text{src}} | e_{ij} \text{ exists}\} \), \( \forall p(f_i) \in \mathcal{N}, \mathcal{T}^*(p(f_i)) \) is already updated in the last iteration. Thus, for solving \( \mathcal{P}_{\text{sub}} \), the difficulty lies in that how to select the optimal placement of \( f_i \) and the optimal mapping of \( e_{ij} \). It will be detailed in Sec. [III-B2].

1) **Recursion-based Path Finding:** We use \( \mathcal{P}(n_i, n_j, \mathcal{M}) \) to represent the set of simple paths from \( n_i \) to \( n_j \) where no path goes through nodes from the set \( \mathcal{M} \subseteq \mathcal{N} \). The set of simple paths from \( n_i \) to \( n_j \) we want, i.e. \( \mathcal{P}(e_{ij}) \), is equal to \( \mathcal{P}(n_i, n_j, \emptyset) \). \( \forall n \in \mathcal{N} \), let us use \( \mathcal{A}(n) \) to represent the set of edge servers adjacent to \( n \). Then, \( \mathcal{P}(n_i, n_j, \mathcal{M}) \) can be calculated by the following recursion formula:

\[
\mathcal{P}(n_i, n_j, \mathcal{M}) = \left\{ J(\emptyset, n_i) \bigcup_{m \in \mathcal{S}} \mathcal{P}(m, n_j, \mathcal{M} \cup \{n_i\}) \right\},
\]

where \( \mathcal{S} \triangleq \mathcal{A}(n_i) - \mathcal{M} \cup \{n_i\} \). \( J(\emptyset, n_i) \) is a function that joins the node \( n_i \) to the path \( \emptyset \) and returns the new joint path \( \emptyset \cup \{n_i\} \). The analysis in this paragraph is summarized in Algorithm 2, i.e. Recursion-based Path Finding (RPF) algorithm.

Before calling RPF, we need to initialize the global variables. Specifically, \( \Omega \) stores all the simple paths, which is initialized as \( \emptyset \), \( \mathcal{V} \), as the set of visited nodes, is initialized as \( \emptyset \). \( \mathcal{V} \) is allocated for the storage of current path, which is also initialized as \( \emptyset \). Whereafter, by calling RPF(\( n_i, n_{ij}, \mathcal{M} \)), all the simple paths between \( n_i \) and \( n_j \) are stored in \( \Omega \). RPF(\( n_i, n_{ij}, \mathcal{M} \)) is used to replace Step 4 of DPE.

**Algorithm 2: Recursion-based Path Finding (RPF)**

**Input:** \( n_i, n_j, \) and \( \mathcal{M} \)

1 /* global variables \( n_i, n_j, \mathcal{V}, \emptyset, \mathcal{G} \), and \( \Omega \) can be visited by RPF. Before calling it, \( \mathcal{V}, \emptyset, \) and \( \Omega \) are set as \( \emptyset \). */

2 if \( n == n_j \) then
  \( \Omega \).append(\( J(\emptyset, n) \)) // Store the path \( J(\emptyset, n) \)
else
  \( \mathcal{V}.\)push(\( n \)); \( \mathcal{V}.\)add(\( n \))
  for each \( n' \in \mathcal{A}(n) - \mathcal{V} \) do
    \( \text{RPF}(n_i, n', \mathcal{M}) \) // Recursive call
  end for
  \( \mathcal{V}.\)pop(); \( \mathcal{V}.\)delete(\( n \))
end if

In the following, we calculate the optimal value of \( z_\emptyset \) for each simple path \( \emptyset \in \Omega \).

2) **Optimal Data Splitting:** Since both \( p(f_i) \) and \( p(f_j) \) are fixed, \( T^*(p(f_i)) \) and \( t(p(f_i)) \) are constants. Therefore,
solving $P_{sub}$ is equal to solving the following problem:

$$
P'_{sub} : \min_{\gamma \in \Omega : z_\gamma} \max_{P \in \Omega} \left\{ \left( \sum_{l \in \theta} \frac{1}{l_{\theta}} \right) \cdot z_\gamma \right\}$$

s.t. \( \sum_{\theta \in \Omega} z_\theta = \sum_{\gamma \in \Omega} z_\gamma \geq 0, \forall \theta \in \Omega. \) (11)

(11) is reconstructed from (1) and (8). To solve $P'_{sub}$, we define a diagonal matrix

$$A \triangleq \text{diag} \left( \sum_{l \in \theta} \frac{1}{b_{l}}, \sum_{l \in \phi_2} \frac{1}{b_{l_2}}, \ldots, \sum_{l \in \Omega} \frac{1}{b_{l_\Omega}} \right).$$

Obviously, all the diagonal elements of $A$ are positive real numbers. The variables that need to be determined can be written as $z \triangleq [z_{\theta_1}, z_{\theta_2}, \ldots, z_{\theta_\Omega}]^T \in \mathbb{R}^{\Omega}$. Thus, $P'_{sub}$ can be transformed into

$$P_{norm} : \min_z \|Az\|_\infty$$

s.t. \( 1^T z = s_{ij}, \) \( z \geq 0. \) (12)

$P_{norm}$ is an infinity norm minimization problem. By introducing slack variables $\tau \in \mathbb{R}$ and $y \in \mathbb{R}^{\Omega}$, $P_{norm}$ can be transformed into the following linear programming problem:

$$P'_{norm} : \min_{z, y, \tau} \tau$$

s.t. \( \sum_{\theta \in \Omega} z_\theta = s_{ij}, \) \( Az + y = \tau \cdot 1, \) \( z \geq 0. \) (13)

$P'_{norm}$ is feasible and its optimal objective value is finite. As a result, simplex method and dual simplex method can be applied to obtain the optimal solution efficiently.

However, simplex methods might be time-consuming when the scale of $G$ increases. In fact, we can find that the optimal objective value of $P_{norm}$ is

$$\min_z \|Az\|_\infty = \frac{s_{ij}}{\sum_{k=1}^{\Omega} 1/A_{k,k}},$$

if and only if

$$A_{u_1,u_2}z^{(u)} = A_{v_1,v_2}z^{(v)}, 1 \leq u \neq v \leq |\Omega|,$$

where $z^{(u)}$ is the $u$-th component of vector $z$ and $A_{u_1,u_2}$ is the $u$-th diagonal element of $A$. Detailed proof of this result is provided in Sec. IV-A. Base on (13), we can infer that the optimal variable $z^* > 0$, which means that $\forall \theta \in \Omega, z^*_\theta \neq 0$. Therefore, the assumption in Sec. III-B1 is not violated and the optimal $P^*(e_{ij})$ is equivalent to $\Omega$.

Up to now, when $p(f_i)$ is fixed, we have calculate the optimal $P^*(e_{ij})$ and $z^*_\theta$ for all paths $\theta \in P^*(e_{ij})$. The analysis in this subsection are summarized in Algorithm 3. Optimal Stream Mapping (OSM) algorithm. In OSM, $P_{(m)}$ is the $m$-th objective value of $P_{sub}$ by taking $p(f_i) = m$. Similarly, $\Omega(p^*(f_i))$ is the $p^*(f_i)$-th set of simple paths obtained by RPF. For at most $|N_f|$ choices of $p(f_i)$, OSM calculates the optimal objective value (Step 1 $\sim$ Step 5). The procedure is executed in parallel (with different threads) because intercoupling is nonexistent. Then, OSM finds the best placement of $f_i$ and returns the corresponding $P^*(e_{ij})$, $z^*$. OSM is used to replace Step 18 of DPE.

![Fig. 4. Embedding of a DAG with the DPE algorithm.](image-url)

Fig. 4 demonstrates an example on how PDE works. The top left portion of the figure is a DAG randomly sampled from the Alibaba cluster trace, where all the functions are named in the manner of topological order. $\forall f_i \in F, c_i$ is set as 1. The top right is the edge server cluster $G$. The bottom demonstrates how the functions are placed and scheduled by DPE.

### Algorithm 3: Optimal Stream Mapping (OSM)

**Input:** $G$, $(F', \mathcal{E}')$, and $p(f_i)$

**Output:** The optimal $\Phi$, $P^*(f_i)$, $P^*(e_{ij})$, and $z^*$

1. for each $m \in N$ do in parallel
2. $p(f_i) \leftarrow m$
3. /* Obtain the $m$-th optimal $\Phi$ by (13) */
4. $\Phi^{(m)} \leftarrow \frac{s_{ij}}{\sum_{k=1}^{\Omega} 1/A_{k,k}} + T^*(p(f_i)) + t(p(f_j))$
5. end for
6. $p^*(f_i) \leftarrow \arg\min_{m \in N} \Phi^{(m)}$
7. $P^*(e_{ij}) \leftarrow \Omega(p^*(f_i))$
8. Calculate $z^*$ by (12) and (14) with $A = A(p^*(f_i))$
9. return $\Phi(p^*(f_i))$, $p^*(f_i)$, $P^*(e_{ij})$, and $z^*$
Theorem 1. **Optimality of DPE** For a DAG \( (\mathcal{F}, \mathcal{E}) \) with a given topological order, DPE can achieve the global optimality of \( \mathbf{P} \) by replacing the Step 4 of it with RPF and Step 18 of it with OSM.

**Proof.** We firstly prove that OSM solves \( \mathbf{P}_{sub} \) optimally. Recall that solving \( \mathbf{P}_{sub} \) is equal to solving \( \mathbf{P}_{\prime sub} \), and \( \mathbf{P}_{\prime sub} \) can be transformed into \( \mathbf{P}_{norm} \). For \( \mathbf{P}_{norm} \), we have

\[
\| \mathbf{A} \mathbf{z} \|_\infty \overset{\Delta}{=} \max_k \{ |A_{k,k}z^{(k)}| \} = \lim_{x \to \infty} \sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x },
\]

because \( \forall k, A_{k,k} > 0, z^{(k)} \geq 0 \). According to the AM–GM inequality, the following inequality always holds:

\[
\sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x } \geq \sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x },
\]

iff \([14]\) is satisfied. It yields that \( \forall x > 0, \)

\[
\sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x } \geq \sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x },
\]

Multiply both sides of \([16]\) by \( \sqrt[\Omega]{\Omega} \), we have

\[
\sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x } \geq \sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x }.
\]

By taking the limit of \([17]\), we have

\[
\| \mathbf{A} \mathbf{z} \|_\infty \geq \lim_{x \to \infty} \sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x },
\]

Combining with \([12]\) and \([14]\), the right side of \([18]\) is actually a constant. In other words,

\[
\min_{\mathbf{z}} \| \mathbf{A} \mathbf{z} \|_\infty = \lim_{x \to \infty} \sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x } = \sqrt[k=1]{ \sum_{k=1}^{[\Omega]} (A_{k,k}z^{(k)})^x } \Rightarrow \text{with \([12]\)}.
\]

The result shows that \([13]\) and \([14]\) are the optimal objective value and corresponding optimal condition of \( \mathbf{P}_{sub} \), respectively, if the topological ordering is given and regarded as an known variable. The theorem is immediate from the optimality of DP.

B. **Complexity Analysis**

In this subsection, we analyze the complexity of the proposed algorithms in the worst case, where \( \mathcal{G} \) is fully connected.

1) **Complexity of RPF:** Let us use \( \kappa(i, j) \) to denote the flops required to compute all the simple paths between \( n_i \) and \( n_j \). If \( \mathcal{G} \) is fully connected,

\[
\kappa(1, N) = (1 + \kappa(2, N)) + (1 + \kappa(3, N))
\]

\[
+ ... + (1 + \kappa(N - 1, N)) + 1
\]

\[
= (N - 1) + (N - 2) \cdot \kappa(2, N).
\]

To simplify notations, we use \( \kappa_i \) to replace \( \kappa(i, N) \). We can conclude that \( \forall i \in \{1, ..., N - 1\}, \)

\[
\kappa_i = (N - i) + (N - i - 1) \cdot \kappa_{i+1}.
\]

Based on \([19]\), we have

\[
\kappa_1 = (N - 2)! \cdot \sum_{i=1}^{N-1} \frac{N-i}{(N-i-1)!}
\]

\[
= (N - 2)! \cdot \sum_{i=1}^{N-1} \frac{\kappa_i}{i!}.
\]

which is the maximum flops required to compute all the simple paths between any two edge servers. Before calculating the complexity of RPF, we prove two lemmas.

**Lemma 1.** \( \forall N \geq 7 \) and \( N \in \mathbb{N}^+, N! > N^3(N + 1) \).

**Proof.** The proof is based on induction. When \( N = 7 \), \( N! = 5040 > N^3(N + 1) = 2744 \). The lemma holds. Assume that the lemma holds for \( N = q, \) i.e., \( q! > q^3(q + 1) \) (induction hypothesis). Then, for \( N = q + 1 \), we have

\[
(q + 1)! = (q + 1) \cdot q! > (q + 1)^2 q^3.
\]

Notice that the function \( g(q) = \left( \frac{1}{2} + \frac{2}{q} + \frac{3}{q^2} \right)^{-1} \) monotonically increases when \( q \in \mathbb{N}^+ - \{1, 2\} \). Hence \( g(q) \geq g(3) = \frac{27}{25} > 1 \), and

\[
1 < \frac{q^3}{(q + 2)^2} < \frac{q^3}{(q + 1)(q + 2)}
\]

\[
\Rightarrow q^3 > (q + 1)(q + 2)
\]

\[
\Rightarrow (q + 1) \cdot q^3 > (q + 1)^3(q + 2).
\]

Combining \([21]\) with \([22]\), we have

\[
(q + 1)! > (q + 1)^2(q + 2),
\]

which means the lemma holds for \( q + 1 \). \( \square \)

**Lemma 2.** \( \forall N \geq 2 \) and \( N \in \mathbb{N}^+, \sum_{i=1}^{N-1} \frac{i^2}{i!} < 6 - \frac{1}{N} \).

**Proof.** We can verify that when \( N \in [2, 7] \cap \mathbb{N}^+ \), the lemma holds. In the following we prove the lemma holds for \( N > 7 \) by induction. Assume that the lemma holds for \( N = q, \) i.e., \( \sum_{i=1}^{q-1} \frac{i^2}{i!} < 6 - \frac{1}{q} \) (induction hypothesis). Then, for \( N = q + 1 \), we have

\[
\sum_{i=1}^{q} \frac{i^2}{i!} < 6 - \frac{1}{q} + \frac{q^2}{q!}.
\]

By applying Lemma \([11]\) we get

\[
\sum_{i=1}^{q} \frac{i^2}{i!} < 6 - \frac{1}{q + 1}.
\]
which means the lemma holds for $q + 1$.

Based on these lemmas, we can obtain the complexity of RPF, as illustrated in the following theorem:

**Theorem 2. Complexity of RPF** In worst case, where $G$ is a fully connected graph and $N \geq 2$, the complexity of OSM is $O((N-2)!)$.

**Proof.** According to Lemma 2

$$\lim_{N \to \infty} \sum_{i=1}^{N-1} \frac{i^2}{i!} < 6.$$ 

Hence $\lim_{N \to \infty} \kappa_1 < 6(N-2)! = O((N-2)!)$.

Finding all the simple paths between arbitrary two nodes is a NP-hard problem. To solve it, RPF is based on depth-first search. In real-world edge computing scenario for IoT stream processing, $G$ might not be fully connected. Even though, the number of edge servers is small. Thus, the real complexity is much lower.

2) **Complexity of DPE:** Notice that RPF is called by OSM $N$ times in parallel. It is easy to verify that the complexity of OSM is $O((N-2)!)$ in worst case, too. OSM is designed to replaced the Step 6 of DPE. Thus, we have the following theorem:

**Theorem 3. Complexity of DPE** In worst case, where $G$ is a fully connected graph and $N \geq 2$, the complexity of DPE is

$$\max \left\{ O(N!), O\left( |E| \cdot N \cdot |P^*(e_{ij})| \right) \right\}.$$ 

**Proof.** From Step 2 to Step 6, DPE calls RPF $N(N-1)$ times. Thus, the complexity of this part (Step 2 ~ Step 6) is $O(N!)$ according to Theorem 2. The average number of preorder functions for each non-source function $f \in \mathcal{F}' - \mathcal{F}'_{src}$ is $\frac{|E|}{Q \cdot |P^*|}$. As a result, in average, OSM is called $\frac{N}{Q \cdot |P^*|}$ times. In OSM, the step required the most flops is Step 8. If the variable substitution method is adopted, the flops required of this step is $2(|P^*(e_{ij})| - 1) + 1$. Thus, the complexity of Step 7 ~ Step 19 of DPE is $O\left( |E| \cdot |P^*| \cdot |P^*(e_{ij})| \right)$. The theorem is immediate by combining the two parts.

Although $O(N!)$ is of great order of complexity, $N$ is not too large in real-world edgy scenario. Even if it is not, as an offline algorithm, it is worth the sacrifice of runtime overhead in pursuit of global optimality.

V. EXPERIMENTAL VALIDATION

A. Experiment Setup

**IoT stream processing workloads.** The simulation is conducted based on Alibaba’s cluster trace of data analysis. This dataset contains more than 3 million jobs (called applications in this work), and 20365 jobs with unique DAG information. Considering that there are too many DAGs with only single-digit functions, we sampled 2119 DAGs with different size from the dataset. The distribution of the samples is visualized in Fig. 5. For each $f \in \mathcal{F}$, the processing power required and output data size are extracted from the corresponding job in the dataset and scaled to $[1, 10]$ giga flop and $[5, 15] \times 10^3$ kbit/s, respectively.

![Fig. 5. Data distribution sampled from the cluster trace.](image)

**Heterogenous edge servers.** In our simulation, the number of edge servers is 10 in default. Considering that the edge servers are required to formulate a connected graph, the impact of the sparsity of the graph is also studied. The processing power of edge servers and the maximum throughput of physical links are uniformly sampled from $[20, 40]$ giga flop and $[30, 80] \times 10^3$ kbit/s in default, respectively.

**Algorithms compared.** We compare DPE with the following algorithms.

- **FixDoc** [1]: FixDoc is a function placement and DAG scheduling algorithm with fixed function configuration. FixDoc places each function onto homogeneous edge servers optimally to minimize the DAG completion time. Actually, [1] also proposes an improved version, GenDoc, with function configuration optimized, too. However, for IoT streaming processing scenario, on-demand function configuration is not applicative. Thus, we only compare DPE with FixDoc.

- **Heterogeneous Earliest-Finish-Time (HEFT)** [8]: HEFT is a heuristic to schedule a set of dependent functions onto heterogeneous workers with communication time taken into account. Starting with the highest priority, functions are assigned to different workers to heuristically minimize the overall completion time. HEFT is an algorithm that stands the test of time.

B. Experiment Results

All the experiments are implemented in Python 3.7 on macOS Catalina equipped with 3.1 GHz Quad-Core Intel Core i7 and 16 GB RAM.

1) **Theoretical Performance Verification:** Fig. 6 illustrates the overall performance of the three algorithms. For different data batch, DPE can reduce 43.19% and 40.71% of the completion time on average over FixDoc and HEFT on 2119 DAGs. The advantage of DPE is more obvious when the scale of DAG is large because the parallelism is fully guaranteed. Fig. 7 shows the accumulative distribution of 2119 DAGs’ completion time. DPE is superior to HEFT and FixDoc on 100% of the DAGs. Specifically, the maximum completion of
DAG achieved by DPE is 1.24s. By contrast, only less 90% of DAGs’ completion time achieved by HEFT and FixDoc can make it. The results verify the optimality of DPE.

Fig. 6 and Fig. 7 verify the superiority of proactive stream mapping and data splitting. By spreading data streams over multiple links, transmission time is greatly reduced. Besides, the optimal substructure makes sure DPE can find the optimal placement of each function simultaneously.

Fig. 6. Average completion time achieved by different algorithms.

Fig. 7. CDF of completion time.

2) Scalability Analysis: Fig. 8 and Fig. 9 shows the impact of the scale of the heterogenous edge $G$. In Fig. 8, we can find that the average completion time achieved by all algorithms decreases as the edge server increases. The result is obvious because more idle servers are available, which ensures that more functions can be executed in parallel without waiting. For all data batches, DPE achieves the best result. It is interesting to find that the gap between other algorithms and DPE get widened when the scale of $G$ increases. This is because the available simple paths become more and the data transmission time is reduced even further. Fig. 8 also demonstrates the run time of different algorithms. The results show that DPE has the minimum time overhead.

Fig. 8. Average completion time under different number of edge servers.

Fig. 9. Average completion time under different sparsity of $G$.

Fig. 10. Average completion time under different processing power of servers.

VI. RELATED WORKS

In this section, we review related works on function placement and DAG scheduling at the network edge.

Studying the optimal function placement is not new. Since cloud computing paradigm became popular, it has been extensively studied in the literature [9] [10]. When bringing function placement into the paradigm of edge computing, especially power and throughput increase, the computation and transmission time achieved by all algorithms are reduced. Even so, DPE outperforms all the other algorithms, which verifies the robustness of DPE adequately.
for the IoT stream processing, different constraints, such as
the response time requirement of latency-critical applications,
availability of function instances on the heterogeneous edge
servers, and the wireless and wired network throughput, etc.,
should be taken into consideration. In edge computing, the
optimal function placement strategy can be used to maximize
the network utility \[1\], minimize the inter-node traffic \[12\],
minimize the makespan of the applications \[1–5\], or even
minimize the budget of application service providers \[13\].

The application is either modeled as an individual blackbox
or a DAG with complicated composite patterns. Considering
that the IoT stream processing applications at the edge usually
have interdependent correlations between the for- and-aft
functions, dependent function placement problem has a strong
relation with DAG dispatching and scheduling. Scheduling
algorithms for edgy computation tasks have been extensively
studied in recent years \[8\]–\[14\]. In edge computing, the
joint optimization of DAG scheduling and function placement
is usually NP-hard. As a result, many works can only achieve
a near optimal solution based on heuristic or greedy policy.
For example, Gedeon et al. proposed a heuristic-based solution
for function placement across a three-tier edge-fog-cloud
heterogeneous infrastructure \[17\]. Cat et al. proposed a greedy
algorithm for function placement by estimating the response
time of paths in a DAG with queue theory \[18\]. Although
FixDoc \[1\] can achieve the global optimal function placement,
the completion time can be reduced further by optimizing the
stream mapping.

VII. CONCLUSION

This paper studies the optimal dependent function embed-
ding problem. We first point out that proactive stream mapping
and data splitting could have a strong impact on the makespan
of DAGs with several use cases. Based on these observations,
we design the DPE algorithm, which is theoretically verified
to achieve the global optimality for an arbitrary DAG when the
topological order of functions is given. DPE calls the RPF and
the OSM algorithm to obtain the candidate paths and optimal
stream mapping, respectively. Extensive simulations based on
the Alibaba cluster trace dataset verify that our algorithms can
reduce the makespan significantly compared with state-of-the-
art function placement and scheduling methods, i.e., HEFT and
FixDoc. The makespan can be further decreased by finding the
optimal topological ordering and scheduling multiple DAGs
simultaneously. We leave these extensions to our future work.

REFERENCES

[1] L. Liu, H. Tan, S. H.-C. Jiang, Z. Han, X.-Y. Li, and H. Huang,
“Dependent task placement and scheduling with function configuration
in edge computing,” in Proceedings of the International Symposium on
Quality of Service, ser. IWQoS ’19, New York, NY, USA, 2019.
[2] S. Khare, H. Sun, J. Gascon-Samson, K. Zhang, A. Gokhale, Y. Barve,
A. Bhattacharjee, and X. Koutsoukos, “Linearize, predict and place:
Minimizing the makespan for edge-based stream processing of directed
acyclic graphs,” in Proceedings of the 4th ACM/IEEE Symposium on
Edge Computing, ser. SEC ’19, New York, NY, USA, 2019, p. 1–14.
[3] Z. Zhou, Q. Wu, and X. Chen, “Online orchestration of cross-edge
service function chaining for cost-efficient edge computing,” IEEE
Journal on Selected Areas in Communications, vol. 37, no. 8, pp. 1866–
1880, 2019.
[4] F. Duchene, D. Lebrun, and O. Bonaventure, “Srv6pipes: Enabling
in-network bystream functions,” Computer Communications, vol. 145,
p. 223 – 233, 2019.
[5] S. Vassilaras, L. Gkatziakis, N. Liakopoulos, I. N. Stiakogiannakis, M. Qi,
L. Shi, L. Liu, M. Debbah, and G. S. Paschos, “The algorithmic aspects
of network slicing,” IEEE Communications Magazine, vol. 55, no. 8,
p. 112–119, Aug 2017.
[6] I. Afosli, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, “Network
slicing and softwarization: A survey on principles, enabling technolo-
gies, and solutions,” IEEE Communications Surveys Tutorials, vol. 20,
no. 3, pp. 2429–2453, thirdquarter 2018.
[7] “Alibaba cluster trace program,” https://github.com/alibaba/clusterdata.
[8] H. Topcuoglu, S. Hariri, and Min-Yau Wu, “Performance-effective and
low-complexity task scheduling for heterogeneous computing,” IEEE
Transactions on Parallel and Distributed Systems, vol. 13, no. 3, pp.
260–274, 2002.
[9] G. T. Lakshmanan, Y. Li, and R. Strom, “Placement strategies for
internet-scale data stream systems,” IEEE Internet Computing, vol. 12,
no. 6, pp. 50–60, 2008.
[10] L. Tom and V. R. Bindu, “Task scheduling algorithms in cloud com-
umputing: A survey,” in Inventive Computation Technologies. S. Smys,
R. Bestak, and A. Rocha, Eds. Cham: Springer International Publishing,
2020, pp. 342–350.
[11] M. Leconte, G. S. Paschos, P. Mertikopoulos, and U. C. Kozat, “A
resource allocation framework for network slicing,” in IEEE INFOCOM
2018 - IEEE Conference on Computer Communications, 2018, pp.
2177–2185.
[12] J. Xu, Z. Chen, J. Tang, and S. Su, “T-storm: Traffic-aware online
scheduling in storm,” in 2014 IEEE 34th International Conference on
Distributed Computing Systems, 2014, pp. 535–544.
[13] L. Chen, J. Xu, S. Ren, and P. Zhou, “Spatio-temporal edge service
placement: A bandit learning approach,” IEEE Transactions on Wireless
Communications, vol. 17, no. 12, pp. 8388–8401, 2018.
[14] Y. Kao, B. Krishnamachari, M. Ra, and F. Bai, “Hermes: Latency
optimal task assignment for resource-constrained mobile computing,”
IEEE Transactions on Mobile Computing, vol. 16, no. 11, pp. 3056–
3069, 2017.
[15] S. Sundar and B. Liang, “Offloading dependent tasks with communica-
tion delay and deadline constraint,” in IEEE INFOCOM 2019 - IEEE
Conference on Computer Communications, 2018, pp. 37–45.
[16] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, “Dedas: Online
task dispatching and scheduling with bandwidth constraint in edge
computing,” in IEEE INFOCOM 2019 - IEEE Conference on Computer
Communications, 2019, pp. 2287–2295.
[17] J. Gedeon, M. Stein, L. Wang, and M. Muelhaeuser, “On scalable
in-network operator placement for edge computing,” in 2018 27th
International Conference on Computer Communication and Networks
(IEEE/CIC ICCCN), 2018, pp. 1–9.
[18] X. Cai, H. Kuang, H. Hu, W. Song, and J. Lu, “Response time aware
operator placement for complex event processing in edge computing,” in
Service-Oriented Computing, C. Pahl, M. Vukovic, J. Yin, and Q. Yu,
Eds. Cham: Springer International Publishing, 2018, pp. 264–278.

Fig. 11. Average completion time under different throughput of links.