Future Generation Computer Systems 144 (2023) 105–116

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A package-aware scheduling strategy for edge serverless functions based on multi-stage optimization

Senjiong Zheng [a], Bo Liu [a,*], Weiwei Lin [b,c,**], Xiaoying Ye [d], Keqin Li [e]

[a] School of Computer Science, South China Normal University, GuangZhou, 510631, China
[b] School of Computer Science and Engineering, South China University of Technology, GuangZhou, 510006, China
[c] Peng Cheng Laboratory, Shenzhen, 518066, China
[d] Guangdong Neusoft Institute, Foshan, 528225, China
[e] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 24 November 2022
Accepted 16 February 2023
Available online 2 March 2023

Keywords:
Serverless function offloading
Dependency package awareness
Package caching strategy

ABSTRACT

Serverless computing offers a promising deployment model for edge IoT applications. However, serverless functions that rely on large libraries suffer from severe library loading latency when containerized, which is unfriendly to edge latency-sensitive applications. Most function offload strategies in edge environments ignore the impact of this latency. We also found that the measures taken by serverless platforms to reduce loading latency may not work in edge environments. To remedy that, this paper proposes a function offloading strategy to minimize loading latency, a new way to deeply integrate placement optimization with cache optimization. In this way, we first design a package caching policy suitable for edge environments based on the consistency of execution topology. Then a Double Layers Dynamic Programming algorithm (*DLDP*) is proposed to solve the problem of function offloading considering the dependent packages using a multi-stage progressive optimization approach. The caching policy is embedded in the scheduling algorithm through a phased optimization approach to achieve joint optimization. Extensive experiments on the cluster trace from Alibaba show that *DLDP* reduces the loading latency of packages by more than 97.84% and significantly outperforms four baselines in the application completion time by more than 55.67%.

1. Introduction

As the intelligent society grows, a huge number of Internet of Things (IoT) endpoints are coming into use [1]. These IoT devices and edge gateways are becoming a vital part of the Cloud-Edge-IoT computing continuum [2], and the data generated by these devices is escalating and is more sensitive to processing latency and privacy [3,4]. The traditional cloud computing model suffers from high latency, poor security, and poor privacy [5], which makes it challenging to meet the diverse needs of today's intelligent society for data processing. Edge computing technology [6] has emerged as a computing paradigm that performs computing at the network's edge, with the core idea of bringing computing closer to the data source.

Serverless computing is a promising paradigm with features such as function-as-a-service, event-triggered and fine-grained scaling that meet the needs of IoT applications [7–9], thus driving the trend towards extending serverless functions in edge computing. A serverless function is a set of code that implements a logical part of an application [10], which will be executed when an event is triggered. This event-driven nature [11,12] is suitable for handling bursty and unpredictable workloads at IoT endpoints [13] and is ideal as an execution model for IoT event and data processing [2]. Also, serverless computing has a more fine-grained resource expansion model [7] with on-demand allocation, which is friendly for resource-constrained edge devices.

Extending serverless functions in edge computing would be a very promising computing paradigm, and a lot of work [13–15] has done research in this area. Although the serverless paradigm presents opportunities, it also suffers from problems such as function cold start latency and underutilization of resources [16]. Functions that rely on large libraries suffer from severe cold start latency when containerized. Cold starts can take hundreds of milliseconds to seconds [17] and accumulate with the function chain [10], eventually leading to performance degradation. Loading large dependency libraries is a significant cause of cold start latency, and caching these packages in advance can effectively reduce the latency [8,18,19].

* Corresponding author.
** Corresponding author at: School of Computer Science and Engineering, South China University of Technology, GuangZhou, 510006, China.
E-mail addresses: 2021023258@m.scnu.edu.cn (S. Zheng), liugubin@scnu.edu.cn (B. Liu), linww@scut.edu.cn (W. Lin), yexiaoying@nuit.edu.cn (X. Ye), lik@newpaltz.edu (K. Li).

https://doi.org/10.1016/j.future.2023.02.013
0167-739X/© 2023 Elsevier B.V. All rights reserved.

Edge applications consist of more simple functions or microservices [10]. As the number of parts composing an application grows, the connections between functions become more complex. We usually use a directed acyclic graph [20] to represent the topology of an application. The task offloading strategies in edge computing have been studied for many years. Many studies are also devoted to offloading optimization of serverless functions in edge computing to reduce the application completion time. Still, they ignore the severe latency caused by loading dependency libraries during container instantiation. However, edge applications are more sensitive to latency, and it is essential to mitigate the latency caused by loading dependency packages when optimizing function offloading. At the same time, we found that the optimization measures taken by serverless platforms for dependency packages may not work in edge environments. Simply offloading the function to a node with high dependency package affinity will be ineffective due to the lack of consideration of the complex topology of the application. It is also not enough to only consider function placement optimization because it cannot effectively reduce the latency caused by loading packages. It is clear that there is a conflict between optimizing function placement and reducing loading delay when optimizing application completion time.

The difficulty of scheduling serverless functions in edge environments is that we need to add consideration of node affinity to the general service placement problem (SPP) while implementing an effective package caching policy suitable for edge environments. It is a complex scheduling model with multiple impact factors. In this paper, the general function offloading model and the offloading model considering the dependency package are modeled, respectively. We analyze the two system models' common potential structure and trait factors, then adopt a multi-stage progressive optimization method to solve the multi-influence factor problem. Specifically, we analyze the optimal substructure hidden in the task scheduling problem and propose an adjustable dynamic programming algorithm (*ADP*), which can perform planning based on different optimization scenarios. *ADP* is suitable for optimizing multiple models with progressive relations, and the execution topology generated based on different optimization scenarios is consistent. Through superposition, we design a Double Layers Dynamic Programming algorithm to solve the optimal scheduling problem of the final model (the function offloading model considering the dependency packets) in stages, and the second layer of dynamic programming further considers the package awareness. The major contributions of this work are summarized as follows:

- Unlike the objective optimization model, we model the function offloading problem as a complex multi-factor model and adopt a multi-stage optimization approach to optimize function offloading. This phased approach weakens the optimization conflicts between factors.
- We design a package caching policy (*DPWP*) based on the consistency of execution topology, which is suitable for execution scenarios in edge environments. Then a Double Layers Dynamic Programming algorithm is proposed to solve the function offloading problem. *DLDP* embeds the caching strategy into the scheduling algorithm to reduce the loading latency of packages and considers package awareness when scheduling functions, creating a joint optimization effect.
- Guided by the maximum flow augmenting path, we designed a data multiplexing method to speed up the data transmission.
- Extensive simulations demonstrate the feasibility of our proposed caching policy and the superiority of our proposed offloading strategy.

2. Related work

The service placement problem [21] is an important research topic in edge computing, focusing on offloading tasks to edge servers. There are many factors involved in offloading tasks. The complex topology of IoT applications, data dependencies between functions, and function-dependent libraries determine that we cannot arbitrarily offload functions to an edge node. Most studies model task scheduling in edge environments as a single-objective or multi-objective optimization model. The optimization objectives include application completion time (*Makespan*), energy consumption, deadlines, QoS violation rate, scheduling cost, resource utilization, etc. Task offloading strategies also vary for different demand scenarios. The list scheduling algorithm [22] is the more classical priority-based scheduling strategy, which sorts the tasks according to the defined priorities and then schedules the task with the highest priority. Some mathematical optimization methods, like integer programming [12,23], and constrained optimization [24,25], formulate the problem as a mathematical optimization model with constraints and then solve the problem mathematically for an optimal solution. Traditional heuristic and meta-heuristic methods aim to find a feasible solution to the problem within a reasonable time, and classical algorithms include genetic algorithms [26], particle swarm algorithms [27], etc. Some hyperheuristics algorithms combine multiple methods [28]. Recent articles have proposed new references like the Markov decision [29] and joint optimization methods [30] etc.

Cold start delay is a crucial factor affecting the completion time of scheduling in serverless platforms. Microservices that rely on large libraries start slowly and affect platform resilience [18] due to the time required to load these dependencies. Current research focuses on reducing loading latency by pre-caching dependency packages [31–33], package-aware scheduling [18], and reuse [8,34]. Jeon et al. [31] proposed a deep reinforcement learning-based dependency package caching strategy, which is learned from historical experience. The algorithm considers global incentives for cache hits and QoS violations and performs reinforcement learning on the caching agents of each edge node, driving them to cooperate to cache critical dependency packages at the hierarchical edge cloud. Aumala et al. [18] consider the dependency package affinity of edge nodes and schedule cloud functions to nodes with higher affinity to reuse execution environments with preloaded packages. Fuerst and Sharma [34] treat hot function containers as cache objects and reuse the container resources by managing the object cache to reduce the cold start time. The algorithm automatically scales the container resources based on the workload characteristics.

There has been a lot of research in the serverless computing domain dedicated to reducing the latency caused by loading function libraries. However, these measures may not work in the edge serverless environment due to the conflict between optimizing function placement and optimizing dependency package loading time. We differ by considering both the complex topology of the application and the caching of function dependency packages, exploring new ways to integrate function placement optimization with function packet-aware scheduling deeply.

3. Problem formulation

This section formulates a complete offloading model for the problem to be solved. Scheduling serverless functions in an edge environment involve numerous influencing factors and potential optimization conflicts. To sort out the connection of multiple independent variables, we decompose the problem into two models: a general offloading model and an offloading model considering dependent packages, which are two models with recursive relationships.

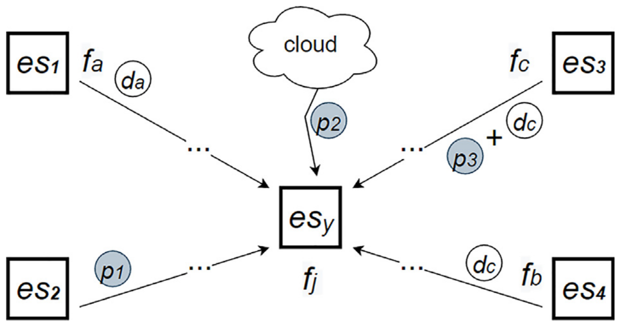

Fig. 1. *Example*1. a simple example of the model in Section 3.3: $f_j \equiv \{pkg = [p_1, p_2, p_3],\ pre = [f_a, f_b, f_c]\}$.

The application and data submitted by the user will be put into a set of heterogeneous edge servers for execution. In addition, there is a remote cloud data center that holds all the packages that the application may use. Dependency packages that are missing when executing functions will be downloaded from here. The edge servers adopt a non-preemptive execution strategy. Two conditions are required for function execution: all precursor tasks transfer data to the server where the function deployed, and the server has all the dependency packages needed for function execution.

3.1. Edge servers and workloads

Here $ES = \{es_1, es_2, \ldots, es_m\}$ denotes m edge servers. Since the servers are heterogeneous, each server has a different processing capacity, and the processing capacity of server es_y is P_y measured in *tflop/s*. The servers communicate with each other in a full duplex. If there is a communication link between servers es_x and es_y, it means they can transfer data to each other directly. if not, they need to transfer data through other servers. The maximum transmission capacity between servers es_x and es_y is denoted as $trans_{x,y}$, measured in *GB/s*.

The applications submitted by the user are usually a function chain composed of multiple functions (tasks) [10], which may have multiple entry or exit functions. We typically describe this relationship by an undirected connection graph: $G = \{Func, E\}$. The vertex set $Func = \{f_1, f_2, \ldots, f_n\}$ denotes the n functions composing the application, and $pf(j)$ indicates the number of floating point operations required by the function f_j. The edge $e_{ij} \in E$ describes the priority relationship among tasks, and $d(e_{ij})$ indicates the size of the data that function f_i need to transfer to function f_j.

Thus, the time taken to process the function f_j (deployed on server es_y) is

$$COST^{CPU}_{j,y} = \frac{pf(j)}{P_y} \qquad y = Loc(f_j) \tag{1}$$

The time taken by function f_i (deployed on server es_x) to transfer data to function f_j (deployed on server es_y) is

$$t(i,j) = \frac{d(e_{ij})}{\sigma_{x,y}} \qquad es_x = Loc(f_i),\quad es_y = Loc(f_j) \tag{2}$$

Data is transferred by multiplexing, $\sigma_{x,y}$ is the equivalent maximum bandwidth between servers es_x and es_y using multiplexing, we will introduce $\sigma_{x,y}$ in Section 4.3.

3.2. General function offloading model

In this model, we consider the general function (task) offloading problem and ignore the dependency package. The entry function f_{entry} can be deployed directly to the appropriate server for processing, so the earliest start time and earliest completion time of f_{entry} (deployed on server es_y) can be defined as

$$T^{est}_y\left(f_{entry}\right) = R_y \tag{3}$$

$$T^{fin}_y\left(f_{entry}\right) = R_y + COST^{CPU}_{entry,y} \tag{4}$$

Since the server adopts a non-preemptive execution policy, the server needs to finish processing all deployed tasks before starting a new task. R_y indicates the time when the server es_y finishes processing the last deployed task. For the non-entry function f_j, the function also needs to wait for its precursors to transfer data to the server where f_j is located before it is eligible for execution, and the earliest time when all the precursor data arrives is

$$T^{pre}(f_j) = \max_{f_i \in pre(j)} \left(T^{fin}(f_j) + t(i,j)\right) \tag{5}$$

Thus, the earliest start time and earliest completion time of the non-entry function f_j (deployed on the server es_y) can be computed from Eqs. (6) and (7) as follows:

$$T^{est}_y\left(f_j\right) = \max\left(T^{pre}\left(f_j\right),\ R_y\right) \tag{6}$$

$$T^{fin}_y\left(f_j\right) = T^{est}_y\left(f_j\right) + COST^{CPU}_{j,y} \tag{7}$$

3.3. Function offloading model with considering dependency packages

When considering the impact of function dependency packages on scheduling, the model becomes more complex, and more factors need to be weighed. In our model, precursor data and dependency packages can be transferred synchronously to compress the delay caused by downloading dependency packages. If a dependent package requested by function f_j exists on another server, it will be transferred to the server where f_j is deployed, just like the data. Those packages that are lacking in the edge server cluster will be downloaded from the remote data center (e.g., Fig. 1. *Example*1 : *cloud* $\rightarrow$ es_y). The time cost of remote download is

$$COST^{pkg} = \frac{size_{pkg\_lack}}{rl} \tag{8}$$

Where $size_{pkg\_lack}$ is the size of the packages to be downloaded from the remote data center, and rl is the download rate. When the server es_x where function f_i deployed transfers data to the server es_y where function f_j deployed, it may also transfer the dependent packages required by function f_j (e.g., Fig. 1. *Example*1 : $es_3 \rightarrow es_y$, $es_1 \rightarrow es_y$). The time spent on transferring data becomes

$$t'_x(i,j) = \frac{d\left(e_{ij}\right) + pkg\_size^j_{x,y}}{\sigma_{x,y}} \qquad es_x = Loc(f_i),\ es_y = Loc(f_j) \tag{9}$$

Where $pkg\_size^j_{x,y}$ denotes the size of the packages required by function f_j to be transferred from server es_x to server es_y. If es_x does not need to transfer data to es_y, then $pkg\_size^j_{x,y} = 0$. Sometimes the packages required by function f_j are located on some servers, but these servers do not need to transfer data to the server es_y (e.g., Fig. 1. *Example*1 : $es_2 \rightarrow es_y$). Let the number of these servers be $Z = \{z_1, z_2, \ldots\}$ and the time spent by es_{z_α} to transfer the packages is

$$t'_{z_\alpha}(i,j) = \frac{0 + pkg\_size^{j}_{z_\alpha,y}}{\sigma_{z_\alpha,y}} \quad (10)$$

For all packages that are transferred individually, the latest time they reach the server es_y can be expressed as

$$T^{pkg}\left(f_j\right) = \max_{z\in Z}\left(R_z + t'_z(i,j)\right) \quad (11)$$

In the model that considers packages, a function is ready for execution once all the precursor data reaches the server where the function is deployed, and the server has all the dependency packages required for execution. Therefore, the earliest start time of the entry function f_{entry}, and the non-entry function f_j (deployed in es_y) becomes

$$T^{est}_y\left(f_{entry}\right) = \max\left(T^{pkg}\left(f_{entry}\right), R_y + COST^{pkg}\right) \quad (12)$$

$$T^{est}_y\left(f_j\right) = \max\left(T^{pre}\left(f_j\right), T^{pkg}\left(f_j\right), R_y + COST^{pkg}\right) \quad (13)$$

Thus, the earliest completion time of a function (deployed on es_y) can be defined as

$$T^{fin}_y(f) = T^{est}_y(f) + COST^{CPU}_{\_,y} \quad (14)$$

3.4. Optimization objective formulation

Function scheduling aims to optimize the application completion time (*Makespan*), which is the maximum of the earliest completion times of all exiting functions (F_{exit}). We design the Dependency Package Window Policy to minimize the loading latency (δ) caused by dependency packages and minimize *Makespan* by finding the current optimal deployment location ($Deploy \equiv \{opt(f) \mid \forall f \in Func\}$) for each function. Thus, the optimization problem is formulated as

$$P : Minimize_{by\ DPWP,Deploy} Max\{T^{fin}(f) \mid \forall f \in F_{exit}\} \quad (15)$$

4. Algorithm design

Scheduling of functions is a multi-stage decision process. In this section, we analyze the optimal substructure hidden in the scheduling of tasks with dependencies, and then design a dynamic programming algorithm with adjustable optimization scenarios. The execution topology generated by the algorithm based on different optimization scenarios is consistent. We adopt the idea of multi-stage optimization and overlay two layers of the *ADP* algorithm for handling the scheduling models in Sections 3.2 and 3.3, respectively. Meanwhile, in the middle of the layers, we design the dependency package caching policy based on the consistency of *ADP*, which enhances the affinity of nodes for functions and provides support for the second layer of package-aware scheduling. The optimization scenarios of the two layers are also a recursive relationship, and the second layer further considers the dependent packet problem.

4.1. Multi-factor model

We model the serverless function scheduling problem in edge environments through the previous analysis as a multi-factor complex model. Among the many factors, application topology, function execution topology, dependency package, data dependencies, and edge environment configuration are the principal component factors. The scheduling of functions needs to satisfy the data dependencies between tasks. The choice of deployment nodes directly affects the data-transferring cost of the precursor tasks, and different edge nodes have different affinities for the functions. In most cases, node selection can only be optimal for one factor at a time. Therefore, data dependencies and dependency packages are a pair of conflicting factors when optimizing *Makespan*.

We decomposed the models to find the connection among the factors, as shown in Sections 3.2 and 3.3. The dependency package is the trait factor that distinguishes the two models. Application topology and function execution topology are common potential factors that act very similarly in both models, and they are the breakthroughs for our phased optimization. The data-transferring cost of precursor and the loading cost of dependency packages are two critical components of *Makespan* and are also the key to optimization.

4.2. Adjustable dynamic programming

An application's execution topology can affect the scheduling performance [35]. Function chains often have multiple entry or exit functions, and there is no strict execution sequence between the parallel tasks. This results in an application with many different execution topologies, and the performance of different topologies varies greatly.

We design an adjustable dynamic programming algorithm with adjustable optimization scenarios. *ADP* considers different combinations of impact factors and performs to optimize the function's earliest completion time. Running *ADP* will get the optimal deployment scheme for different scenarios, including functions' execution topology and the server location where each function is deployed.

4.2.1. Design of adjustable dynamic programming

Algorithm 1: Adjustable Dynamic Programming (*ADP*)

```
Input: The topological sequence Func of function chains, ES
Output: Optimal execution topology and opt(f)
1  for all $f_j \in Func$ do
2      if $f_j$ is an entry function then
3          continue
4      Get all precursors Prefunc of $f_j$
5      for $es_y \in ES$ do
6          Assume that $f_j$ is deployed in $es_y$
7          for $f_i \in Prefunc$ do
8              Calculate $opt(f_i)$ by Eq. (16), so that $f_j$ starts at
               the earliest time, this determines which server
               $f_i$ should be deployed on
9      Deployment of Prefunc
10     Update $T^{fin}(f_j)$ of $f_j$ on each $es \in ES$
```

4.2.2. Why choose dynamic programming

We choose dynamic programming for two main reasons.

(1) Dynamic programming can plan based on different recursive equations, which is suitable for our model decomposition. It has the advantage of optimal performance and simple execution.
(2) The dynamic programming algorithm we designed is adjustable for optimization scenarios. Different deployment schemes arise when considering different scenarios, but the execution topology of those schemes is consistent.

The second reason is crucial: the execution topology of the functions is invariant when stacking multiple layers of such algorithm. According to this consistency, we can cache the dependency packages required by the unexecuted function in advance to reduce the loading latency when they are executed.

4.2.3. Optimal deployment location for functions

Deployment of an application is a multi-stage decision process. The deployment of a function directly affects the completion time of that function, and also affects the deployment of subsequent functions. Each function needs to make the current optimal deployment decision to get the process's optimal substructure. Considering the function pair $f_i \rightarrow f_j$ with precedence relation, we get the earliest completion time $T_x^{fin}(f_i)$ of f_i on each server $es_x \in ES$ in line 10 of the code. If we want f_i to have the earliest completion time, then its deployment location is

$$opt'(f_i) = \left\{ es_x | \min_{es_x \in ES} T_x^{fin}(f_i) \right\}$$

In fact, $opt'(f_i)$ is not the optimal deployment location because the deployment choice of f_i affects the deployment of f_j. $opt(f_i)$ should make f_j have the earliest completion time. Thus, the better deployment option should be

$$\begin{aligned} &opt(f_i) = \left\{ es_x | \min_{es_x \in ES} T^{fin}(f_j) \right\} \\ &T^{fin}(f_j) = T_x^{fin}(f_i) + t(i,j) + COST_{j,\_}^{CPU} \end{aligned} \tag{16}$$

$opt(f_i)$ is exactly the optimal substructure of the process stage $\{funtions\ that\ had\ deployed, f_i, \ldots, f_j, \ldots\}$. When f_j is about to be deployed, we can determine the optimal deployment of its precursor f_i. lines 5–9 of the code use this idea to determine the deployment of the precursor of f_j.

Theorem 1. *The execution topology generated by ADP based on different optimization scenarios is consistent.*

Proof. $\{f_1, \ldots, f_j, \ldots, f_n\}$ is an arbitrary topological sequence of the given function chain, assume that the function f_j is currently being processed and $F = \{f_a, f_b, f_c, \ldots\} \rightarrow f_j$ is the precursor of the function f_j. Algorithm 1 determines the optimal deployment position of $\{f_a, f_b, f_c, \ldots\}$ one by one according to the recursive equations so that f_j has the earliest start time. The order $\{a, b, c, \ldots\}$ is constant in this process. This is true for f_j, and it is also true for functions after f_j. By induction, we can prove that given the sequence $\{f_1, \ldots, f_j, \ldots, f_n\}$, the *ADP* generates the same execution topology When the recursive equation (optimization scenario) changes. Experiments also verify the correctness of Theorem 1.

4.2.4. Foreseen sequence

We superimpose two layers of the *ADP* algorithm to optimize the offloading problem considering dependent packages in a phased manner. Since the execution topology of *ADP* is consistent, the execution topology generated by the first layer can be used as a foreseen sequence for the second layer.

4.3. Dependency package window cache policy

We design the Dependency Package Windowing Policy to solve the package caching problem based on the foresight of *ADP*, including warm-up and cache removal policies. The following two concepts are introduced first.

Definition 1. Server Dependency Package Window

We assume each server maintains a fixed-size package cache space, which is the server's dependency package window. Due to the large variety of dependent packages, the edge server cannot cache unlimited packages. When the size of the cached package exceeds the server's window capacity, packages that may not be used in the future will be deleted.

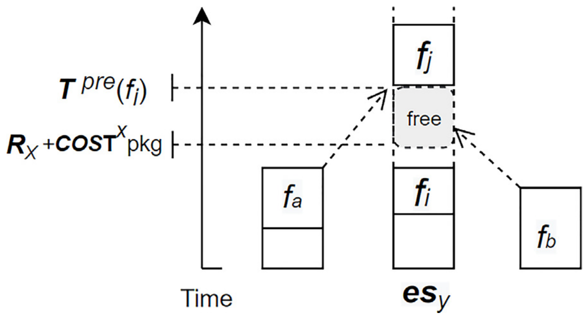

Fig. 2. Free gaps when executing function f_j.

Definition 2. Function Dependency Package Window

When executing the current task, we select $2 * w$ functions that will be executed one by one soon in the future, and the dependency packages they need form the three-level function dependency package window. The 1st-level window is packages required by the next function to be executed, and caching this window has the highest priority. The 2nd-level window consists of the packages required by the first w functions, which contains the packages that will be used soon. The 3rd-level window consists of the packages required by the first $2 * w$ functions, adding to the 2nd-level window the packages that may be used later.

4.3.1. Dependency package warm-up strategy

In the offloading model considering dependency packages, if the dependency package required by a function exists in another server, it can be transferred to the server where the function is deployed synchronously when transferring the data to compress the loading latency (as shown in Fig. 1. *Example* 1). During the second layer of dynamic programming, we know the dependency packages required by the next function to be executed based on the foreseen sequence. Caching these dependency packages in advance to the edge server cluster can reduce the loading latency during function execution.

When the server is waiting for the predecessor of a function to transfer data, there may be a free gap. We use this free gap to warm up the packages for the next function to be executed, as shown in Fig. 2.

Proof. Feasibility Analysis of free gap

Suppose the current execution function is f_i (deployed in es_x). When the package required by the f_i is not in edge servers, it will be downloaded from the remote cloud data center. Let the download time be $COST_i^{pkg}$. The server es_x finishes the last task at R_x, the data transfer completion time of f_i precursor tasks is $T^{pre}(f_i)$, if

$$R_x + COST_i^{pkg} < T^{pre}(f_i)$$

Then there is a free gap during the execution of f_i by the server es_x, the size is

$$T_i^{gap} = T^{pre}(f_i) - R_x - COST_i^{pkg} \tag{17}$$

We take advantage of the free gap when executing functions but with preconditions. Sufficient conditions for the execution of the warm-up policy are as follows:

(a) No impact on the normal execution speed of the edge server.
(b) Sufficient free gap to download the entire package.

4.3.2. Dependency package removal strategy

The lower-priority packages will be removed if there is no space left to cache new packages in the server dependency package window. The removal strategy is shown as follows.

(1) Check the packages one by one, and if the package is not in the 3rd-level Function Package Window, delete it until the server window can cache all new packages or traverse all packages.
(2) If there are still new packages not cached, do the same as (1) for the 2nd-level Function Package Window.
(3) If there are still new packages not cached, do the same as (1) for the 1st-level Function Package Window.

Algorithm 2: The Dependency Package Window Policy (*DPWP*)

```
Input: Foreseen Sequence FS, the currently executing
       function $f_i$ (deployed on $es_x$)
1 Generate the three-level FP_Win according to FS
2 Get the next two functions ($f_{i+1}$ and $f_{i+2}$) to be deployed
3 if $es_x$ has free time gap when executing $f_i$ then
4     Download the packages required by $f_{i+1}$ or $f_{i+2}$ and
      add to EP_Win of $es_x$ virtually
5 for all $p \in$ EP_Win do
6     if EP_Win is not enough and $p \notin$ 3rd_FP_Win then
7         Delete p
8 If EP_Win is still not enough, Executing lines 5-7 of code
  for 2nd_FP_Win, 1st_FP_Win in turn
9 Download the warm-up packages during the free gap
```

Definition 3. Dependency package loading latency

Dependency packages required by a function missing from the *ES* will be downloaded from the remote data center when the function is executed. The download time of these packages is part of the cold start latency, as the packages' download affects the processing schedule. Therefore, the dependency package loading latency of an application is defined as

$$\delta = \frac{SUM(\{\rho \mid \rho \in FuncWin \text{ and } \rho \text{ not in } ES\})}{rl} \tag{18}$$

Where *rl* is the remote data center download rate, *FuncWin* indicates the packages required by the functions included in the application.

Definition 4. Dependency package hit rate

The dependency package required by functions is considered to be warmed up successfully if it is in the edge server cluster *ES* during scheduling. In this case, the dependency packages and the precursor data can be transferred synchronously. Thus, the hit rate is defined as

$$\psi = \frac{SUM(\{\rho \mid \rho \in FuncWin \text{ and } \rho \text{ in } ES\})}{SUM(\{\rho \mid \rho \in FuncWin\})} \tag{19}$$

4.4. Multiplexed data transmission

The bandwidth of communication links in edge environments is usually limited. We considered dividing the data for multiplexing and then stitching them when all the data reaches the destination server. The existing study [25] finds all the simple paths between the source server and the destination server and then divide the data for multiplexing. However, this method has shortcomings. In most cases, the maximum transmission capacity between two points is less than the simple overlay of the transmission capacity of all shortest paths between two points. As shown in Fig. 3, the maximum transmission capacity of the link $\{es_1, es_2, es_3, es_4\}$ is 5, and the maximum transmission capacity of the link $\{es_1, es_3, es_4\}$ is 6, but the maximum transmission capacity between es_1 and es_4 is not $5+6=11$, This is because the two paths share the bandwidth of the link $\{es_3, es_4\}$.

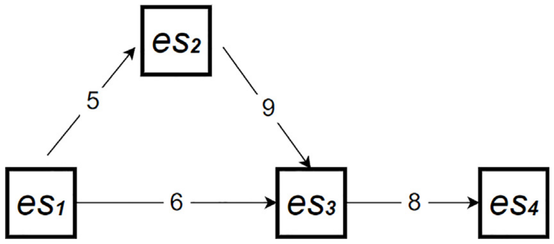

Fig. 3. Maximum transmission capacity.

Data multiplexing requires consideration of the maximum transmission capacity between two points and traffic balancing. We refer to the idea of the maximum flow augmentation path to solve data multiplexing. First, we find all the simple paths between the source server and the destination server, sort them by path length and remove the paths whose path length is larger than the shortest path length by more than ρ . This excludes paths that pass through multiple intermediate servers so that the traffic is concentrated around the shortest and maximum transmission capacity paths. Secondly, we sort the remaining paths according to the transmission time index of paths, with the smaller ones going forward.

Definition 5. Transmission time index

Suppose a simple path $Path = \{es_1, es_2, \ldots, es_s\}$ consists of s edge servers and the transmission time index of this path is defined as

$$\gamma = \sum_{i=1}^{s-1} \frac{1}{trans_{i,i+1}} \tag{20}$$

The time required to transfer α *GB* of data on this path can be calculated by

$$t = \alpha\gamma \tag{21}$$

Eq. (21) shows that the smaller the transmission time index γ of a path is, the less time it takes to transfer the same data. Ranking the paths with small γ forward is to rank the paths with large transmission capacity forward and also move the paths with small path length forward. Since the path length is smaller, the γ value will be relatively smaller.

A. Data multiplexing based on augmented path

To maximize the overall transmission capacity between two points, we check each path in turn of the path table ranked according to the transmission time index. For a simple path $Path = \{es_1, es_2, \ldots, es_s\}$, if all links $\{es_i, es_{i+1}\}$ have remaining bandwidth, it can act as an augmented path to increase the overall transmission capacity. The contributing bandwidth of this augmented path is

$$\omega = \min_{(es_i, es_{i+1}) \in Path} \left\{ BW^{left}(i, i+1) \right\} \tag{22}$$

$BW^{left}(i, i+1)$ is the remaining bandwidth of the link $\{es_i, es_{i+1}\}$. The sum of the contributing bandwidth contributed by all the augmentation paths is the equivalent maximum bandwidth between two points. Next, we need to divide the data so that the data on different augmentation paths can reach the target server at the same time. Suppose the source server es_x needs to transfer η *GB* data to the target server es_y. There are β augmentation paths between the servers es_x and es_y, and let the contributing bandwidth and length of these augmentation paths be

$$CBW = [bw_1, bw_2, \ldots, bw_\beta]^T$$
$$L = [l_1, l_2, \ldots, l_\beta]^T$$

Therefore, the weight vector of the transmission time index of the augmented paths can be expressed as

$$W = diag(\frac{l_1}{bw_1}, \frac{l_2}{bw_2}, \ldots, \frac{l_\beta}{bw_\beta})$$

Where $diag(\ldots)$ is the diagonal matrix, $\frac{l_\theta}{bw_\theta}$ is the transmission time index of augmentation path $Path_\theta$. Let the amount of data allocated on the augmented path $Path_\theta$ be d_θ, and the data vector of the augmented paths is

$$D = [d_1, d_2, \ldots, d_\beta]^T$$

By definition, there is the following relationship

$$\eta = SUM\left(\left[d_1, d_2, \ldots, d_\beta\right]\right) \tag{23}$$

$$WD = \left[t_1, t_2, \ldots, t_\beta\right]^T \tag{24}$$

Where t_θ is the time when the data transferred through the augmented path $Path_\theta$ reaches the target server. To make the overall arrival time the shortest, $t_1 = t_2 = \cdots = t_\beta$ (*). Solving the system of Eqs. (23) (24) (*) yields

$$\sigma_{x,y} = SUM\left(\sum_{c=1}^{\beta} bw_c\right) \tag{25}$$

$$D = \frac{\eta}{\sigma_{x,y}}\left[\, bw_1, bw_2, \ldots, bw_\beta \,\right]^T \tag{26}$$

$$t = t_\theta = d_\theta \frac{l_\theta}{bw_\theta} = \frac{\eta}{\sigma_{x,y}} \tag{27}$$

Where $\sigma_{x,y}$ is the equivalent maximum bandwidth between the server es_x and es_y, t is the time required to transfer η *GB* of data from es_x to es_y, and $t_\theta = d_\theta \frac{l_\theta}{bw_\theta}$ is the arrival time of data on the augmented path $Path_\theta$, which is calculated by Eq. (21)

4.5. Package-aware scheduling with embedded caching policy

We design a Double Layers Dynamic Programming algorithm to optimize function offloading by superimposing *ADP*. The two layers of *ADP* consider different optimization scenarios. *DLDP* embeds the caching policy into the scheduling algorithm by means of phased optimization to weaken the conflict between placement optimization and dependency package optimization.

The first layer of dynamic programming is devoted to optimizing the general function offload model without considering the latency in loading dependency packages. This layer considers the application topology and the edge environment configuration to find an optimal deployment solution for the current system model, then provides guidance for the optimization of common potential factors for both models. The optimal deployment location of the function for the general unloading model is shown in Eq. (16).

The second layer of dynamic programming solves the optimization problem of function offloading models considering the dependency package. This layer considers package awareness and package affinity of edge servers to schedule functions, taking full advantage of the foresight sequence to improve node affinity. The consistency of execution topology of dynamic programming provides guidance for package caching, which in turn increases the affinity of the edge nodes and supports the second layer of package-aware scheduling. This is a mutually reinforcing relationship.

With the dependency package window policy, most packages are already cached before function execution. When performing the second layer of dynamic programming, we can exclude edge servers with poor affinity and integrate all factors to find the optimal deployment of the function offloading model considering the dependency package.

Considering the same example $f_i \rightarrow f_j$, the optimal deployment position of the function f_i becomes

$$opt\,(f_i) = \left\{ es_x | \min_{es_x \in ES} T^{fin'}\left(f_j\right) \right\} \tag{28}$$

$$T^{fin'}\left(f_j\right) = T_x^{fin}\,(f_i) + t_x'\,(i,j) + COST_j^{pkg} + COST_{j,\_}^{CPU}$$

Algorithm 3: Double Level Dynamic Programming (*DLDP*)

```
   Input: The topological sequence $\boldsymbol{Func}$ of function chains,
          $\boldsymbol{ES}$
 1 Run ADP to solve the optimal offloading of the
    application without considering packages and output the
    Execution Sequence
 2 for all $f_j \in Func$ do
 3    if $f_j$ is an entry function then
 4       continue
 5    Get all precursors $Prefunc$ of $f_j$
 6    for $es_y \in ES$ do
 7       Assume that $f_j$ is deployed in $es_y$
 8       for $f_i \in Prefunc$ do
 9          Deploy $f_i$ virtually by Considering all factors,
             including the affinity of edge nodes, so that $f_j$
             starts at the earliest time
10    for $f \in Prefunc$ do
11       Deploy function $f$
12       Executing the caching policy DPWP
13    Update $T^{fin}(f_j)$ of $f_j$ on each $es \in ES$
14 Calculate the minimum Makespan of Func
```

5. Experimental validation

In this section, we conducted extensive experiments to evaluate the effectiveness and performance of the *DLDP* algorithm. The main evaluation metrics include application completion time, the loading latency and cache hit ratio of the dependency package. We also analyze the impact of different system configurations, workload types, and dependency scenarios on different scheduling strategies. Overall, when the number of edge servers exceeds 11, *DLDP* reduces the loading latency of the dependency package by more than 97.84%, and the package hit rate is over 99%. Within this context, *DLDP* achieves excellent performance and significantly outperforms four baselines in the application completion time by more than 55.67%. For diverse workloads and edge environments, *DLDP* continues to perform consistently.

5.1. Experiment setup

The experiments were implemented in Python 3.7 on a Windows 10 computer with an Intel Core i5-9400F CPU and 16.00 GB RAM.

5.1.1. Workload

The simulation is conducted based on the real cluster trace from Alibaba [36], which contains information from more than 20,000 different DAGs. We performed preliminary processing to select more than 3000 applications (DAGs) to test the performance of different algorithms. Also, 100 applications (DAGs), each containing a different number of functions, were selected to test the ability of the algorithms to handle different types of workloads.

5.1.2. Parameter configuration

Numerous factors affect application completion time, and we use the control variable method to analyze the impact of different variables on the performance of each algorithm. The impact factors tested include the number of edge servers, workload type, capacity of server cache window, type and size of dependency package, servers' performance, and link bandwidth. When testing the impact of workload type and specific configurations of edge servers, the number of edge servers is fixed at 12.

5.1.3. Baseline algorithm

The research in this paper, "Reducing the latency of loading the dependency packages while optimizing function placement" is essentially a task (function) offloading study for edge computing. We have selected four relatively advanced offloading algorithms for comparison, which show good performance in handling tasks with priority order. The function offloading algorithms in edge computing lack consideration of loading packages, and we improved the four baseline algorithms to consider dependency packages. We aim to evaluate the adaptability of our package caching strategy in the edge environment and the effectiveness achieved by *DLDP* in handling serverless function offloading and reducing loading latency. The baseline algorithm is as follows.

(1) PASS [30]: *PASS* is a priority-based allocation and selection algorithm dedicated to solving the problem of placing serverless functions with dependencies in edge computing. The algorithm considers the complex topology of the application and the priority order of the parallel-executable functions to priority ranking. It also selects the appropriate data transaction method (direct transmission or remote transmission) for functions with different fan-out degrees during the allocation process to reduce the time cost of transmitting data
(2) Fixdoc [20]: *Fixdoc* models the application as a DAG, then considers various constraints on task offloading and the configuration of functions on the server for optimal planning. Depending on the edge configuration, the algorithm aims to find the optimal placement for each function, to minimize the application completion time.
(3) HEFT [22]: Heterogeneous Earliest Completion Time (*HEFT*) is a high-performance, low-complexity task scheduling algorithm for heterogeneous environments. *HEFT* first defines the priority of the task and then selects the task with the highest priority for processing at each step to minimize the earliest completion time. Tasks adopt an insertion-based allocation approach in the scheduling process.
(4) Greedy [18,37]: For each task, the Greedy algorithm always schedules the task to the synthetically optimal server for processing. Existing offloading studies [18] on serverless platforms consider dependency package awareness and scheduling tasks to nodes with high affinity. We improve the Greed algorithm by evaluating the package affinity of edge nodes in scheduling.

The server windows of *PASS*, *Fixdoc*, *HEFT* and *Greedy* use the Least Recently Used (*LRU*) cache removal strategy and take the loading latency of dependency packages into account.

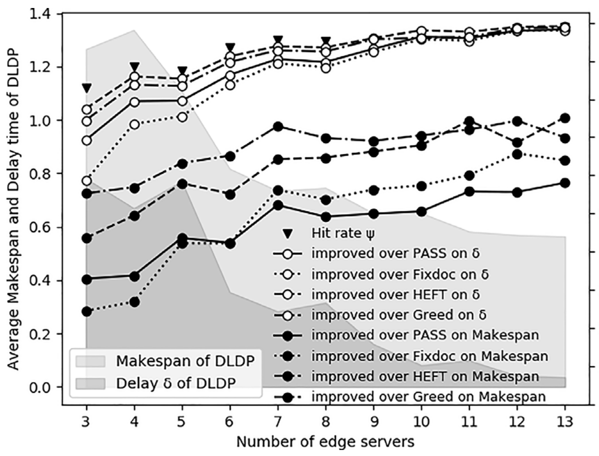

Fig. 4. Performance of *DLDP* and the improvement (%) of *DLDP* over four baseline algorithms.

5.2. Experimental results

We show the experimental results of each algorithm on the cluster track from Alibaba and analyze the main reasons for the improvement of the *DLDP* algorithm.

5.2.1. Overall performance

Fig. 4 shows the performance of processing all DAGs with a cluster containing a different number of edge servers. Left-vertical coordinates indicate the time. Right-vertical coordinates indicate the improvement (%) of *DLDP* over other baseline algorithms regarding the average completion time (*Makespan*) of all applications and the loading latency of packages (δ).

As the number of servers increases, the dependency package hit rate (ψ) gradually increases, and the loading latency (δ) rapidly decreases. Hence, the average completion time (*Makespan*) of *DLDP* keeps falling. Thanks to this, the performance improvement (%) of *DLDP* over the baseline algorithm rises steadily. Two reasons for the reduction in *Makespan* are given. One is more servers are available for processing tasks, and the other is that the loading latency is reduced due to more dependency packages being cached in advance. The *DLDP* algorithm performs well in caching dependency packages, with a hit rate of over 99% and a reduction in loading latency of over 97.84% when the number of servers exceeds 11. This shows that our package caching strategy works well and is suitable for scheduling scenarios in edge environments. Due to the excellent caching performance, the application completion time is reduced by more than 55.67% over the four baselines (*PASS* 55.67%, *Fixdoc* 65.78%, *HEFT* 68.61%, and *Greed* 74.34%), which is the result of the joint optimization with the package-aware scheduling and the caching strategy.

5.2.2. Sensitivity analysis

Figs. 6–12. shows the performance of each algorithm with different system configurations when the number of edge servers is 12. The Left-vertical coordinate indicates the average application completion time, and the Right-vertical coordinate indicates the performance improvement (%) of *DLDP* over the other four baseline algorithms.

A. Workload type

Complex large applications (containing more functions) take more time to schedule. The more complex the application topology is, the more time it should take. Figs. 5–6 shows that the average *Makespan* rises faster when the number of functions in

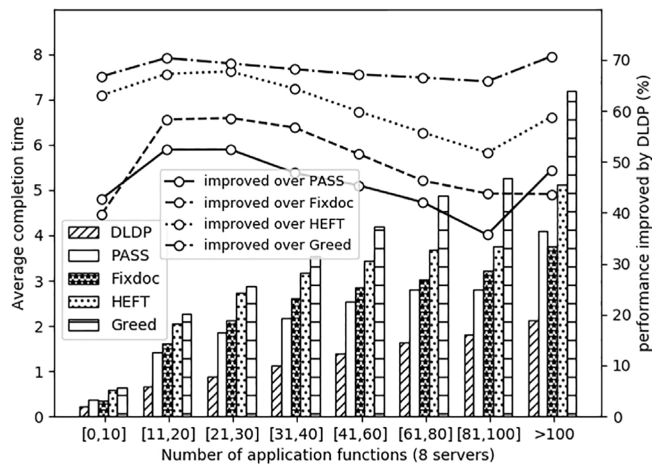

Fig. 5. Average completion time under different number of functions (8 servers).

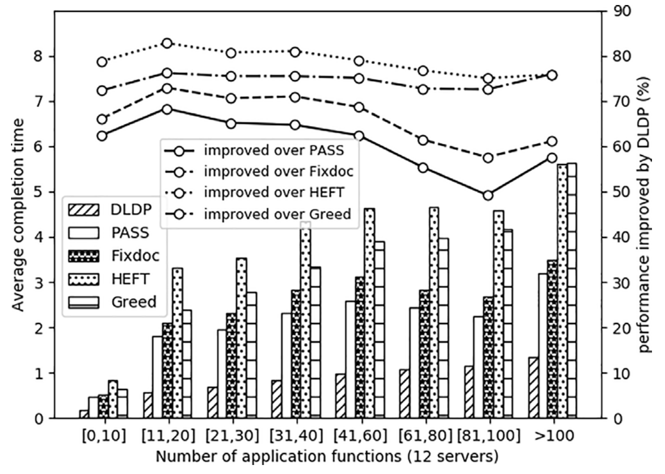

Fig. 6. Average completion time under different number of functions (12 servers).

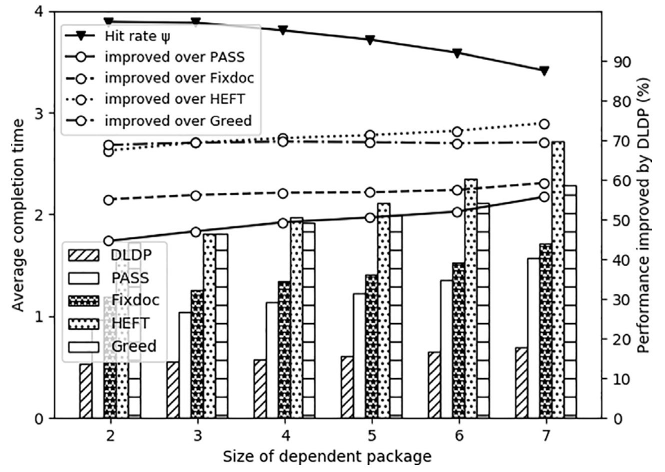

Fig. 7. Average completion time under different size of dependent package.

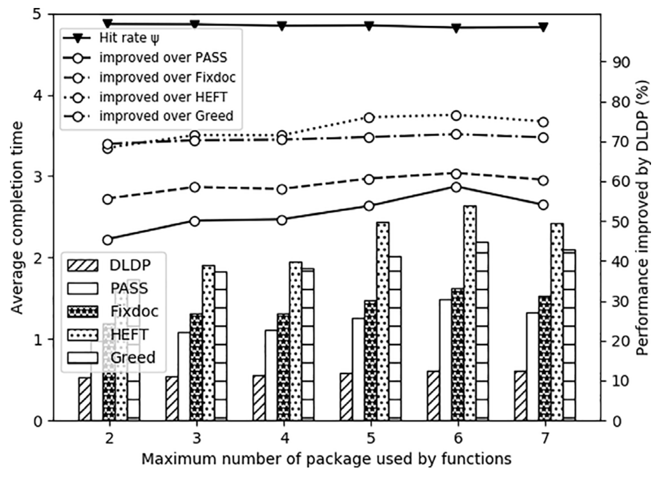

Fig. 8. Average completion time under different number of packages used by functions.

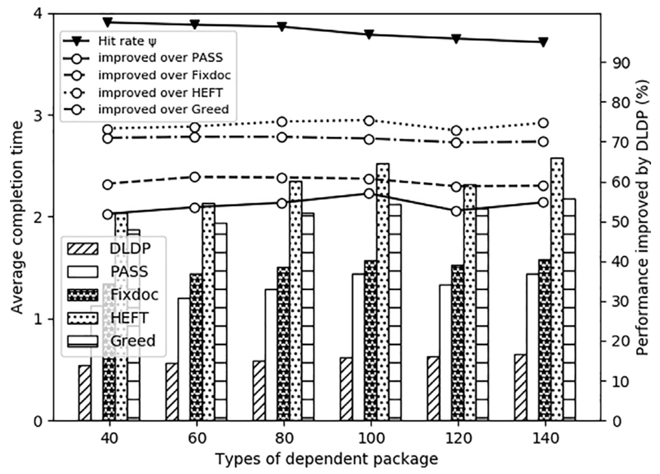

Fig. 9. Average completion time under different number of package types.

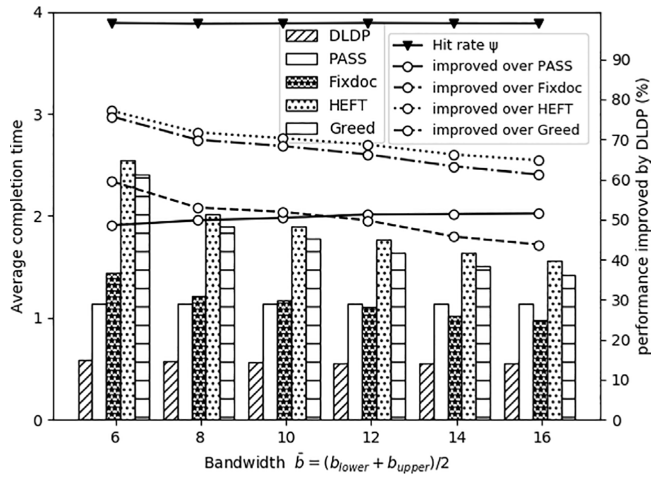

Fig. 10. Average completion time under different link bandwidth.

the application is 0 to 40 and rises slowly when the number is 40 to 100. Sporadic small workloads also take up a lot of server time. A single function's average processing time decreases instead when large workloads are arranged reasonably, illustrating that a suitable scheduling strategy is critical. When handling applications containing different numbers of functions, *DLDP* has a great improvement over other algorithms, especially when the number is from 10 to 60, 100 or more. This shows that *DLDP* adapts to different types of workloads. When resources are limited, the

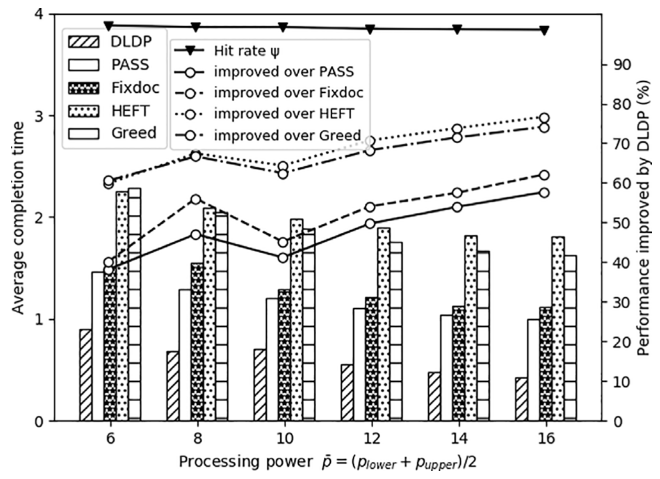

Fig. 11. Average completion time under different processing power of edge server.

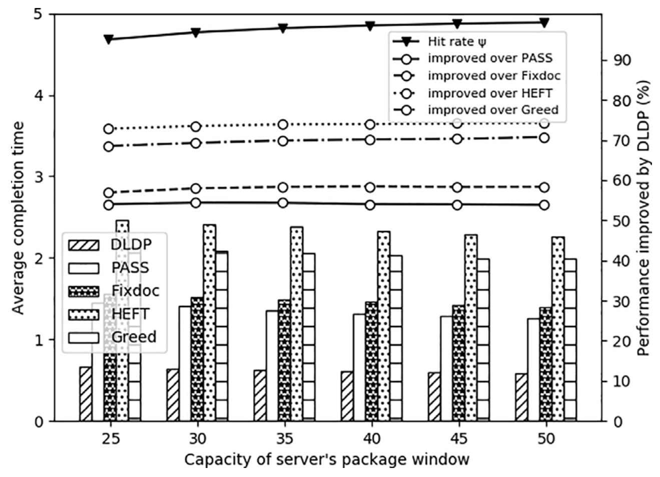

Fig. 12. Average completion time under different capacity of server's package window.

Greed strategy performs poorly and is unsuitable for handling large workloads.

B. Function dependency package

Figs. 7–8 shows that the application completion time increases significantly when the average size of the dependency packages required by the function becomes larger. It takes more time to load large dependency packages. Also, DLDP's hit rate of dependency packages tends to decrease, which shows that it is more challenging to cache large dependency packages, and this is one of the reasons why large dependency packages cause severe latency. The average number of dependencies of functions can also lead to higher latency. With the same edge environment configuration and application topology, functions that depend on more packages are more likely to download packages from remote data centers and take more time to load packages. *DLDP* performs well in these two aspects, with a steady improvement over the other algorithms, indicating that the caching strategy is successful and adapts to diverse dependency scenarios. In particular, the application completion time of *DLDP* remains almost the same when the average number of dependencies increases since *DLDP* keeps the cache hit ratio at a high level. Therefore, there is no need to spend extra time loading dependency packages. In terms of the types of dependency packages, all algorithms perform better. The average scheduling time of each algorithm fluctuates but has no rising trend when the total number of the kinds of packages increases, as shown in Fig. 9.

C. Edge server configuration

The performance of 8 and 12 servers handling different workloads (as shown in Figs. 5–6) shows that in resource-constrained edge environments, the performance gap between different scheduling strategies becomes more apparent, and the time required to schedule workloads with different sizes varies more. It means that resource-constrained environments are more stringent for scheduling. With limited resources in the edge environment, the scheduling strategy should consider the resource-constrained situation more. Figs. 10–11 shows the effects of communication link bandwidth and server processing capacity on scheduling performance. The processing capacity affects the task processing speed, and the communication capacity involves the data transmission. Both significantly impact the scheduling completion time, especially the link communication capacity. Generally speaking, *DLDP* performs steadily and has outstanding performance under the limited processing capability and link bandwidth. Under limited communication capacity, *DLDP* and *PASS* perform outstandingly, with only slight variation in application completion time due to the optimization of both algorithms for the data transfer method. The greedy strategy only considers the optimal placement in the current situation and lacks reasonable planning in the resource-constrained case. Fig. 12 shows the impact of different server window sizes on the application completion time, and we can see that the cache space constraint does not significantly affect our caching strategy.

6. Conclusion

In this paper, we discuss the great potential of the serverless edge computing paradigm, which has inspired research on scheduling serverless functions in edge computing. Functions that rely on large packages suffer from severe loading latency when containerized. Considering package caching is essential for edge latency-sensitive tasks. Our proposed Double Layers Dynamic Programming algorithm can effectively plan the deployment location of functions. Combined with the Dependency Package Windowing Policy, most of the required dependency packages are cached before function execution. The loading latency is reduced by 97.87% compared to other strategies. In the future, we will continue to focus on scheduling optimization and data transfer optimization in serverless edge computing and propose more approaches with practical value.

CRediT authorship contribution statement

Senjiong Zheng: Methodology, Software, Writing – original draft. **Bo Liu:** Methodology, Writing – review & editing. **Weiwei Lin:** Methodology, Writing – review & editing, Funding acquisition. **Xiaoying Ye:** Investigation, Writing – review & editing. **Keqin Li:** Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work is supported by Key-Area Research and Development Program of Guangdong Province (2021B0101420002), National Natural Science Foundation of China (62072187), Guangdong Major Project of Basic and Applied Basic Research (2019B030302002), the Major Key Project of PCL (PCL2021A09), Guangdong Marine Economic Development Special Fund Project (GDNRC[2022]17) and Guangzhou Development Zone Science and Technology Project (2021GH10, 2020GH10).

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet Things J. 3 (2016) 637–646.
[2] N. Ferry, R. Dautov, H. Song, Towards a model-based serverless platform for the cloud-edge-IoT continuum, in: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid, IEEE, 2022, pp. 851–858.
[3] A.M. Alwakeel, An overview of fog computing and edge computing security and privacy issues, Sensors 21 (24) (2021) 8226.
[4] A. Rasheed, P.H.J. Chong, I.W.H. Ho, X.J. Li, W. Liu, An overview of mobile edge computing: Architecture, technology and direction, KSII Trans. Internet Inf. Syst. 13 (2019) 4849–4864.
[5] L.A. Haibeh, M.C. Yagoub, A. Jarray, A survey on mobile edge computing infrastructure: Design, resource management, and optimization approaches, IEEE Access 10 (2022) 27591–27610.
[6] K. Cao, Y. Liu, G. Meng, Q. Sun, An overview on edge computing research, IEEE Access 8 (2020) 85714–85728.
[7] M.S. Aslanpour, A.N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S.S. Gill, R. Gaire, S. Dustdar, Serverless edge computing: Vision and challenges, in: 2021 Australasian Computer Science Week Multiconference, 2021, pp. 1–10.
[8] G.R. Russo, A. Milani, S. Iannucci, V. Cardellini, Towards QoS-Aware function composition scheduling in Apache OpenWhisk, in: 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events, PerCom Workshops, IEEE, 2022, pp. 693–698.
[9] G.A.S. Cassel, V.F. Rodrigues, R. da Rosa Righi, M.R. Bez, A.C. Nepomuceno, C.A. da Costa, Serverless computing for Internet of Things: A systematic literature review, Future Gener. Comput. Syst. 128 (2022) 299–316.
[10] S. Lee, D. Yoon, S. Yeo, S. Oh, Mitigating cold start problem in serverless computing with function fusion, Sensors 21 (24) (2021) 8416.
[11] M. Kiener, M. Chadha, M. Gerndt, Towards demystifying intra-function parallelism in serverless computing, in: Proceedings of the Seventh International Workshop on Serverless Computing, WoSC7 2021, 2021 pp. 42–49.
[12] L. Pan, L. Wang, S. Chen, F. Liu, Retention-aware container caching for serverless edge computing, in: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, 2022, pp. 1069–1078.
[13] P. Gackstatter, P.A. Frangoudis, S. Dustdar, Pushing serverless to the edge with WebAssembly runtimes, in: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid, IEEE, 2022, pp. 140–149.
[14] V. Kjorveziroski, S. Filiposka, V. Trajkovik, Serverless platforms performance evaluation at the network edge, in: International Conference on ICT Innovations, Springer, 2022, pp. 160–172.
[15] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, R. Prodan, A serverless real-time data analytics platform for edge computing, IEEE Internet Comput. 21 (2017) 64–71.
[16] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M.M. Swift, Peeking behind the curtains of serverless platforms, in: USENIX Annual Technical Conference, 2018, pp. 133–146.
[17] P. Silva, D. Fireman, T.E. Pereira, Prebaking functions to warm the serverless cold start, in: Proceedings of the 21st International Middleware Conference, 2020, pp. 1–13.
[18] G. Aumala, E.F. Boza, L. Ortiz-Avilés, G. Totoy, C.L. Abad, Beyond load balancing: Package-aware scheduling for serverless platforms, in: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID, 2019, pp. 282–291.
[19] J. Manner, M. Endreß, T. Heckel, G. Wirtz, Cold start influencing factors in function as a service, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion, 2018 pp. 181–188.
[20] L. Liu, H. Tan, S.H.-C. Jiang, Z. Han, X. Li, H. Huang, Dependent task placement and scheduling with function configuration in edge computing, in: 2019 IEEE/ACM 27th International Symposium on Quality of Service, IWQoS, 2019, pp. 1–10.
[21] F.A. Salaht, F. Desprez, A. Lèbre, An overview of service placement problem in fog and edge computing, ACM Comput. Surv. 53 (2020) 1–35.
[22] H.R. Topcuoglu, S. Hariri, M. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distributed Syst. 13 (2002) 260–274.
[23] S. Hu, G. Li, Dynamic request scheduling optimization in mobile edge computing for IoT applications, IEEE Internet Things J. 7 (2020) 1426–1437.
[24] A. Brogi, S. Forti, QoS-aware deployment of IoT applications through the fog, IEEE Internet Things J. 4 (2017) 1185–1192.
[25] S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, A.Y. Zomaya, Dependent function embedding for distributed serverless edge computing, IEEE Trans. Parallel Distrib. Syst. 33 (10) (2021) 2346–2357.
[26] A.A. Al-Habob, O.A. Dobre, A. garcía Armada, Sequential task scheduling for mobile edge computing using genetic algorithm, in: 2019 IEEE Globecom Workshops, GC Wkshps, 2019, pp. 1–6.
[27] S. Ma, S. Song, L. Yang, J. Zhao, F. Yang, L. Zhai, Dependent tasks offloading based on particle swarm optimization algorithm in multi-access edge computing, Appl. Soft Comput. 112 (2021) 107790.
[28] R. Xie, D.-S. Gu, Q. Tang, T. Huang, F. Yu, Workflow scheduling using hybrid PSO-GA algorithm in serverless edge computing for the Internet of Things, in: 2022 IEEE 95th Vehicular Technology Conference, VTC2022-Spring, 2022, pp. 1–7.
[29] S. Wang, R. Urgaonkar, M. Zafer, T. He, K.S. Chan, K.K. Leung, Dynamic service migration in mobile edge computing based on Markov decision process, IEEE/ACM Trans. Netw. 27 (2019) 1272–1288.
[30] Y. Li, D. Zeng, L. Gu, K. Wang, S. Guo, On the joint optimization of function assignment and communication scheduling toward performance efficient serverless edge computing, in: 2022 IEEE/ACM 30th International Symposium on Quality of Service, IWQoS, 2022, pp. 1–9.
[31] H. Jeon, S. Shin, C. Cho, S. Yoon, Deep reinforcement learning for QoS-aware package caching in serverless edge computing, in: 2021 IEEE Global Communications Conference, GLOBECOM, 2021, pp. 1–6.
[32] P. Vahidinia, B.J. Farahani, F.S. Aliee, Cold start in serverless computing: Current trends and mitigation strategies, in: 2020 International Conference on Omni-Layer Intelligent Systems, COINS, 2020, pp. 1–7.
[33] E. Oakes, L. Yang, K. Houck, T. Harter, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Pipsqueak: Lean Lambdas with large libraries, in: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW, 2017, pp. 395–400.
[34] A. Fuerst, P. Sharma, FaasCache: Keeping serverless computing alive with greedy-dual caching, in: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 386–400.
[35] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Topology-aware serverless function-execution scheduling, 2022, arXiv preprint arXiv:2205.10176.
[36] Alibaba cluster trace program, 2022, https://github.com/alibaba/clusterdata, Online.
[37] L. Ma, Y. Lu, F. Zhang, S. Sun, Dynamic task scheduling in cloud computing based on greedy strategy, in: ISCTCS, 2012.

Senjiong Zheng received the B.S. degree in 2020 from the School of Computer Science, South China Normal University, Guangzhou, China, where he is currently working toward the M.S. degree in software engineering. His research interests include edge computing and serverless computing.

Bo Liu is currently a professor in the School of Computer Science at South China Normal University. His research interests include cloud storage technology, cloud computing and big data technology.

Weiwei Lin received his B.S. and M.S. degrees from Nanchang University in 2001 and 2004, respectively, and the Ph.D. degree in Computer Application from South China University of Technology in 2007. He has been serving as visiting scholar at Clemson University from 2016 to 2017. Currently, he is a professor in the School of Computer Science and Engineering, South China University of Technology. His research interests include distributed systems, cloud computing, big data computing and AI application technologies. He has published more than 100 papers in refereed journals and conference proceedings. He has been the reviewers for many international journals, including TC, TCYB, TSC, TCC, Information Sciences, Future Generation Computer Systems, etc. He is a senior member of CCF and a member of the IEEE.

Xiaoying Ye received her Bachelor degree in Computer Science from University of Electronic Science and Technology of China in 2006 and Master degree in Business Administration from Guilin University of Technology in 2014. Her research interests mainly include big data and software engineering.

Keqin Li is a SUNY Distinguished Professor of Computer Science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber–physical systems, heterogeneous computing systems, big data computing, high performance computing, CPU–GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored over 870 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds nearly 70 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 5 most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an associate editor of the ACM Computing Surveys and the CCF Transactions on High Performance Computing. He has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing. He is an IEEE Fellow and an AAIA Fellow. He is also a Member of Academia Europaea (Academician of the Academy of Europe).