Online DAG Scheduling with On-Demand Function Configuration in Edge Computing

Liuyan Liu, Haoqiang Huang, Haisheng Tan*,
Wanli Cao, Panlong Yang, and Xiang-Yang Li

School of Computer Science and Technology, University of Science and Technology of China, Hefei, China
{hstan,plyang,xiangyangli}@ustc.edu.cn,
{liuliuy,PB1998,cwl233}@mail.ustc.edu.cn

Abstract. Modern applications in mobile computing become increasingly complex and computation intensive. Task offloading from mobile devices to the cloud is more and more frequent. Edge Computing, deploying relatively small-scale edge servers close to users, is a promising cloud computing paradigm to reduce the network communication delay. Due to the limited capability, each edge server can be configured with only a small amount of functions to run corresponding tasks. Moreover, a mobile application might consist of multiple dependent tasks, which can be modeled and scheduled as Directed Acyclic Graphs(DAGs). When an application request arrives online, typically with a deadline specified, we need to configure the edge servers and assign the dependent tasks for processing. In this work, we jointly tackle on-demand function configuration on edge servers and DAG scheduling to meet as many request deadlines as possible. Based on list scheduling methodologies, we propose a novel online algorithm, named `OnDoc`, which is efficient and easy to deploy in practice. Extensive simulations on the data trace from Alibaba (including more than 3 million application requests) demonstrate that `OnDoc` outperforms state-of-the-art baselines consistently on various experiment settings.

Keywords: Edge Computing · DAG Scheduling · Function Configuration · Online Algorithm.

1 Introduction

With the development of cloud computing [1,2], resource-limited mobile devices can signicantly expand their capability by offloading computational-intensive tasks to a remote cloud. However, due to the long distance between mobile devices and the remote cloud, serious communication delay is inevitable. It is even worse when a large amount of data needs to be transferred in some applications. Such long latency cannot meet the demand of delay-sensitive applications with

* Haisheng Tan is the Corresponding Author. The first two authors have equal contribution.

strict deadlines, e.g., Automatic Drive and VR applications. To mitigate this problem, *edge computing* is proposed as a new cloud computing paradigm [3,4]. By deploying relative small-scale edge servers at the edge of the Internet (e.g., wireless access points), edge computing can provide nearby rich computing resources to the mobile users so that the propagation delay can be decreased significantly.

The resource and computation ability of edge servers are relatively constrained compared with a remote cloud. We should carefully configure the functions (i.e. computation modules of applications) that an edge server supports. At the same time, modern applications might consist of multiple dependent functions whose dependency can be modeled as DAGs (Seen in Fig. 1). Furthermore, in edge computing, mobile users come online and generate requests on some applications, which are typically latency-sensitive and require an immediate response. Assigning requests deadline is common practice in real-time systems to guarantee the quality of service. Therefore, it is of great importance to study how to configure edge servers and schedule the tasks to each server so that as many request deadlines can be satisfied as possible.

Task offloading [5–10] has been a major research topic because of its difficulty and importance. A notable work by Topcuoglu et al. [6] aimed to make suitable decisions to improve performance by reducing makespan. Inspired by list scheduling schemes, they proposed an insertion-based heuristic algorithm called HEFT [6]. Minimizing the overall cost is also a practical objective in task offloading. Neto et al. [7] tried to reduce the energy consumed by mobile devices and designed a flexible and innovative computation offloading framework. Sundar et al. [8] considered the execution and communication cost jointly to minimize the total cost subject to the application deadline. By appropriately allocating the application deadline among individual tasks, the tasks were scheduled in a greedy manner. All aforementioned works assumed that the edge servers can serve any requests assigned, which is impractical. Due to the limited resource(e.g. memory and CPU), an edge server cannot configure all the functions simultaneously.

The scheduling of dependent tasks represented as a DAG, called *DAG scheduling* for short, has also been extensively studied [6,11–14]. Based on their techniques, the existing DAG scheduling schemes can be classified into three categories [15,16]: *list scheduling* [6,17], *cluster-based scheduling* [11,18] and *task duplication-based scheduling* [15,19]. However, all these work focused on scheduling tasks of one DAG in offline manner. In the real scenario in edge computing, users' requests come in arbitrary order and time and we cannot know any information about these requests until their arrivals. To tackle the scheduling problem in this manner, we need an online algorithm.

A common practical method for function configuration is on-demand configuration (e.g., Azure Functions [20]): when a task is assigned to some edge server without the corresponding function for it, the edge server will first copy the function image from the remote cloud and deploy it locally to serve the task. In this work, we consider on-demand function configuration and DAG scheduling jointly in edge computing. Our objective is to satisfy as many request deadlines as possible. The problem, even its special case, has been proved to be NP-Hard [8]. Based on *list scheduling* methodology, we then propose an efficient and effective **on**line **D**AG scheduling with **o**n-demand function **c**onfiguration algorithm, named `OnDoc`.

Our main contributions can be highlighted as follows:

– We formulate the online function configuration and DAG scheduling problem in edge computing where multiple application requests, with specific deadlines, are generated online in arbitrary time and order. The objective is maximizing the number of requests that satisfy the deadlines.
– We design a scheduling algorithm to meet as many request deadlines as possible. To the best of our knowledge, this is the first work that studies function configuration and DAG scheduling jointly in an online manner in edge computing. Our algorithm `OnDoc` maintains multiple task scheduling lists to reduce idle time of edge servers dramatically. Besides, `OnDoc` is easy to implement and does not introduce large scheduling overhead.
– We conduct extensive emulations under Alibaba data trace [21]. The evaluation results show that our online algorithm, `OnDoc`, can adapt well to different network environments and perform consistently better than the heuristic baselines on various experiment settings, e.g., the number of requests satisfying their deadline by `OnDoc` can be at least **1.9**$\times$ that of the baselines.

The rest of this paper is organized as follows. In Sec. II, we present the model and problem formulation. In Sec. III, we present `OnDoc` in detail. Sec. IV illustrates our simulation results, and Sec. V concludes the whole work.

2 System Model and Problem Definition

2.1 Network

The network consists of heterogeneous edge servers and a remote cloud, denoted as $\mathcal{S} = \{s_1, s_2, ..., s_m\}$. Each edge server s_i has a limited capacity C_i, which means that the multi-dimension resources (e.g. CPU, I/O, and storage) available in s_i can maximally configure a number of C_i functions (i.e., deploying functions and serving the corresponding tasks) simultaneously. Note that there is a special server s_m to represent the remote cloud, where we assume it has enough resources to configure all functions. The data rate between server s_i and s_j is denoted as $d_{i,j}$. We set $d_{i,j} = d_{j,i}$ and $d_{i,j} = +\infty$ if $i = j$.

2.2 Application and Request

There are multiple applications in our edge computing system and each request invokes one of them with initial data. Each application is modeled by a directed acyclic graph (DAG) $G(\mathcal{V}, \mathcal{E}, \mathcal{W})$, to depict the dependence among all the functions. Here, $\mathcal{V}$ is the set of nodes denoting the tasks, $\mathcal{E}$ is the set of directed links defining the dependence among tasks, and the set $\mathcal{W}$ depicts the amount of required data transferred from the predecessor task to the successor on each link. For instance, a link (v_i, v_j) with weight $w_{i,j}$ specifies that there is $w_{i,j}$ amounts of data transferred from task v_i to v_j. Hence, v_j cannot start before the data transfer is finished. The computation and communication of one task cannot overlap. Then if two tasks are placed at different servers s_x and s_y, the communication delay, i.e., $w_{i,j}/d_{x,y}$, need to be taken into consideration.

Each request is submitted to the edge system from one edge server termed *the initial server*, which will call for an application denoted by a DAG. Given the DAG of a request, a task without any predecessor tasks is called *the entry task* and a task without any successor tasks is called *the exit task*. For ease of presentation, we let the *exit (entry) tasks* all connect to a *pseudo exit (entry) task* which does not take any processing time or resources, so that there will be exactly one pseudo exit(entry) task in each DAG.The weight of links adjacent to the pseudo exit task is the amount of the output data produced by the exit tasks. For the pseudo entry task, the weight of the additional links is the amount of initial data received by each entry task.The pseudo entry and exit task must be processed in the initial server of the request, which means that initial data must be transferred from and the result must be sent back to the initial server.

2.3 Function Configuration

Without loss of generality, we assume that each type of task exactly maps to one function. We define a function $map : V \rightarrow F$. V is the set of all tasks and F is the set of all functions. For any $v \in V$ and $f \in F$, $map(v) = f$ means that task v is to be processed by function f. To process task v_j on server s_i, s_i must have configured the corresponding function $f_j = map(v_j)$ locally. Specifically, when task v_j is assigned to edge server s_i without function f_j, s_i has to suffer configuration time, denoted as $\mathcal{C}_{i,f_j}$, to download the function from the remote cloud and deploy it. Each deployed function on a edge server can process one task at one moment, which means that the queuing delay is considered. Recall that we assume the remote cloud has configured all functions, where all tasks assigned can be processed at their arrival. An edge server can configure a new function directly as long as it has enough capacity. Otherwise, we need to release some configured functions for the new one.

2.4 Problem Formulation

Here, we consider a series of requests arriving *online* in arbitrary time and order, denoted as $\mathcal{R} = \{r_1, r_2, ...\}$. Each request r_k calls for one application in the edge system with a deadline L_k. Except for the parameters of the network and the DAGs of all applications, we cannot know any information of a request before its arrival, e.g., the application it calls for, the amount of initial data, the initial edge server, and its deadline. Let v_i^k denote the i-th task of request r_k. The processing time for v_i^k at server s_j is $p_{i,j}^k$, which can be known at the request's arrival. The assumption is practical since that we can estimate the processing time well based

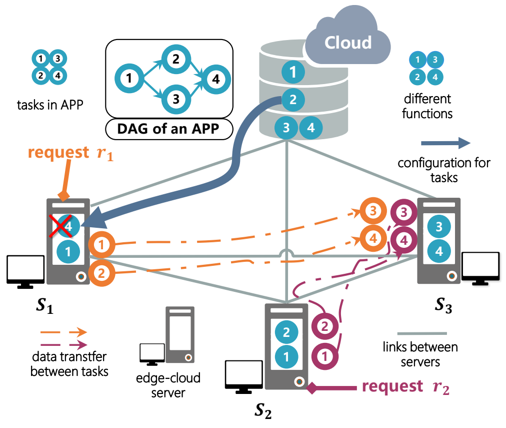


Fig. 1. A simple case of system model

on the previous record. Note that since the initial data of each request might be different, the processing time of the same task from different requests of the same application might be different. When r_k arrives at server s_j in time $t_k^{\uparrow}$, we decide where to process each task (called *task assignment*). Here, if task v_i^k is assigned to a server s_j, we should first configure the corresponding function $map(v_i^k)$ if s_j does not hold it. If a task is assigned to the remote cloud, we can process it as soon as it arrives at the cloud. As the granularity of tasks is relatively small, we do not consider task preemption to avoid extra processing overhead. That is to say, once one task is started, it will be continuously processed until its completion. The completion of the exit task indicates the completion of the request, which should be before the deadline. Under the aforementioned model, our goal is to satisfy as many request deadlines as possible.

A simple example of our model is illustrated in Fig. 1. Specifically, there is an edge-cloud system containing 3 edge servers and a remote cloud. The capacity of edge servers is all set to 2, while the cloud holds all functions. Two requests, r_1 and r_2, call for the same application with their own initial data arriving at server s_1 and s_2, respectively. We take r_1 as an example, whose task 1 and 2 are assigned to edge server s_1, and task 3 and 4 to s_2. s_1 then need download function 2 from the remote cloud and choose to drop the existing function 4. Besides, task 2 can be processed only if function 2 has been deployed on s_1 and its predecessor node (i.e., task 1) has been finished.

3 Algorithms

We describe the details of `OnDoc` (Alg. 1) in this section. `OnDoc` is a variant list scheduling scheme, which remains simplicity and efficiency of many prevalent list scheduling schemes of DAG scheduling. The main idea of list scheduling is to define priorities of tasks and assign tasks to the server in priority order. Besides, we have to modify the function configuration of each edge server simultaneously due to the limited capacity. Hence, `OnDoc` is composed of three parts: priority-calculating strategy, task-assigning strategy, and function-configuring strategy. We next present these strategies specifically.

Priority-calculating strategy: When addressing DAG scheduling problem with list scheduling strategies, a useful method that can prioritize the tasks efficiently and simply is significant. K. Shin et al. [16] classifies task priorities applied by most list scheduling heuristics into three types:*S-level, B-level, T-level*. *S-level*, called static level, is the longest path from the task to the exit task with computation cost taken into consideration only. *B-level* is also calculated from bottom(exit task) to top(entry task). The difference between *S-level* and *B-level* is that communication cost is taken into consideration by *B-level* as well. *T-level*, namely, is the sum of computation cost and communication cost of the longest path from the entry task to the concerned task. After computing task priorities, we can prioritize these tasks corresponding to the decreasing(increasing) order of *S-level, B-level*(*T-level*). However, all these task priorities are static and can only prioritize tasks from one request. The challenge remaining is that we have to schedule tasks from multiple requests concurrently in most cases and we cannot employ these task priorities to determine the priority of tasks from different requests.

To tackle the aforementioned challenge, we maintain multiple task scheduling lists $Q = \{l_1, l_2, ...\}$ rather than follow the idea of list scheduling methodology to merge tasks form all request to construct one prioritized list. Q is the set of prioritized lists of requests. Let Q be empty initially. When $t = t_k^{\uparrow}$, request r_k arrives at initial server s_j with deadline L_k, we employee the *B-level* as task priorities to get a scheduling list l_k of r_k and insert l_k into Q instantly (Line 1–4 in Alg. 1). Every time when we are scheduling, we only take the head of all the scheduling lists in Q into consideration. To choose an appropriate task to assign, we first determine the target server based on the *Task-assigning strategy* for all candidate tasks. Then leaded by the idea to reduce the idle time of edge servers, we choose the task that can start execution at earliest time in its target server among all the candidate tasks to assign (Line 6–7). Each scheduling process ends with the chosen task is already starting execution in its target server, then the chosen task is deleted from its scheduling list and the next scheduling begins. If the scheduling list of request r_k is empty or request exceeds the corresponding deadline, we delete it from Q (Line 9–10).

Task-assigning strategy: We greedily assign each task to the server which can finish it as early as possible besides the pseudo exit(entry) task whose target sever is initial server under our model. Let $EFT_{i,j}^k$ denote the $\underline{\text{e}}$arliest $\underline{\text{f}}$inish $\underline{\text{t}}$ime of task v_i^k on server s_j and $EST_{i,j}^k$ be the $\underline{\text{e}}$arliest $\underline{\text{s}}$tart $\underline{\text{t}}$ime corresponding. As illustrated above, $EFT_{i,j}^k$ and $EST_{i,j}^k$ are two most important attributes. We follow the steps below to calculate them.

When determining $EFT_{i,j}^k$, it is obvious that we should obey two constraints: precedence constraint and capacity constraint. Precedence constraint means that tasks cannot start execution before the data of its precursors transferring to server s_j. Capacity constraint means that tasks have to wait until server s_j has spare capacity. To articulate the two constraints, we formulate them as following.

For ease of formula, we record status of edge servers all the time. For instance, we maintain a set $A_j = \{(v'_1, r'_1), (v'_2, r'_2), ..., (v'_n, r'_n)\}$(without pseudo entry task) for server s_j, when task v_i^k is assigned to server s_j, we insert a tuple (i, k) to A_j. Without loss of generality, we assume that

$$EFT^{r'_1}_{v'_1,j} \geqslant EFT^{r'_2}_{v'_2,j} \geqslant ... \geqslant EFT^{r'_n}_{v'_n,j}$$

and denote $A_j(m) = \{(v'_1, r'_1), (v'_2, r'_2), ..., (v'_m, r'_m)\}$. For convenience, if $m \geq |A_j|$, we fill the set up with $(m - |A_j|)$ virtual tuples (v_{entry}, r_k).

Then we define an indicator binary variable as following:

$$x_{i,j} = \begin{cases} 0 & \text{if i = j} \\ 1 & \text{if otherwise} \end{cases} \tag{1}$$

For (pseudo) entry task v_{entry}, we have

$$EST^k_{v_{entry},j} = EFT^k_{v_{entry},j} = t_k^{\uparrow}, \forall j \in S \tag{2}$$

The *precedence constraint* is denoted as

$$EST^k_{i,j} \geqslant \max_{i' \in pre(i,k)} EFT^k_{i',tar(v^k_{i'})} + \frac{w^k_{i',i}}{d_{tar(v^k_{i'}),j}} \tag{3}$$

Here, $tar(v^k_{i'})$ denotes the target server of $v^k_{i'}$ and $pre(i, k)$ represents the set of predecessor tasks of v^k_i. Also, the communication delay is included in the inequation (3) and $w^k_{i',i}$ denotes the amount of data transfer from $v^k_{i'}$ to v^k_i.

The *capacity constraint* is

$$EST^k_{i,j} \geqslant \min_{i',r' \in A_j(C_j)} EFT^{r'}_{i',j} + \mathcal{C}_{j,map(v^k_i)} * x_{map(v^k_i)map(v^{r'}_{i'})} \tag{4}$$

As a result, $EST^k_{i,j}$ can be computed as below:

$$\begin{aligned} EST^k_{i,j} = \max\{ & \max_{i' \in pre(i,k)} EFT^k_{i',tar(v^k_{i'})} + \frac{w^k_{i',i}}{d_{tar(v^k_{i'}),j}}, \\ & \min_{i',r' \in A_j(C_j)} EFT^{r'}_{i',j} + \mathcal{C}_{j,map(v^k_i)} * x_{map(v^k_i),map(v^{r'}_{i'})}\} \end{aligned} \tag{5}$$

Obviously, we have:

$$EFT^k_{i,j} = EST^k_{i,j} + p^k_{i,j} \tag{6}$$

Then we tentatively enumerate tasks' EFT in each server to determine their target server. For any task v^k_i (i is not entry or exit), we have $tar(v^k_i) = \min_{s_j \in \mathcal{S}} EFT^k_{i,j}$.

Function-configuring strategy: In Eqn. (5), when $x_{map(v^k_i),map(v^{r'}_{i'})}$ equals to 1, it means that we have to configure the corresponding function to allow the task to begin executing. First, we have to check whether there is enough capacity for configuration. If so, we just take up the spare space simply, if not, we need to decide which function will be replaced. In `OnDoc`, to allow the configuration to start as soon as possible, we always choose the function which can be replaced at the earliest time to drop. If there are many candidate functions that can be replaced at the same time, we choose one uniformly random(Line 8).

Algorithm 1: `OnDoc`

```
1  set $Q \leftarrow \emptyset$;
   /* Thread for maintaining $Q$                                    */
2  if new request $r$ arrives then
3      $l \leftarrow$ scheduling list of $r$;
4      $Q \leftarrow Q \cup \{l\}$;
   /* Thread for assigning tasks and configuring functions          */
5  while $Q \neq \emptyset$ do
       /* $H$ consists of head of each scheduling list in $Q$       */
6      Construct set $H$ ;
7      We assign task $v^*$ to server $tar(v^*)$
8      Configure $tar(v^*)$ with corresponding function before $v^*$ executed;
9      if $\exists l \in Q$ and $l = \emptyset$ then
10         delete $l$ from $Q$;
```

4 Performance Evaluation

In this section, we conduct extensive emulations to evaluate the performance of `OnDoc` based on an 8-day data trace from Alibaba [21] and compare our algorithm with two heuristic baselines. The emulation results show that the number of requests satisfying their deadline by `OnDoc` can be at least **1.9**$\times$ that of the baselines. Besides, we investigate the impact on `OnDoc` of various parameter settings and demonstrate that the performance of `OnDoc` is stably better the baselines.

4.1 Experiment Settings

Workloads: We use an 8-day data trace from Alibaba [21] as our workload. The data trace contains more than 3 million jobs (called requests of applications in this work) with the DAG information and the start time. We randomly choose multiple requests consecutively each day to get 8 different sets and take the start time of each request as its release time. Besides, to be more consistent with the real scenario of edge computing, where requests need to be processed in real time, we scale all the processing time to milliseconds. We conduct experiments on 8 request sets independently and analyze the average performance.

Default Setting: We simulate an edge computing system with 5 heterogeneous edge servers and one remote cloud. The available capacity of edge servers is limited to 3, while the remote cloud has all functions preloaded. The configuration time for each function is set to 500ms. And the data transmission time from server to the cloud, which is by default set as 20 times of that between two edge server, is set $[100, 2000]$ms. And we conduct experiments to study the impact of these parameters. If not specified explicitly, the processing time in the remote cloud is set $0.75\times$ (in average value) of that in edge servers [5] as the remote cloud typically has more resources than edge servers.

4.2 The heuristic Baselines

Due to the lack of work jointly studying online DAG scheduling and function configuration, we adopt two classical schemes as our baselines, which are

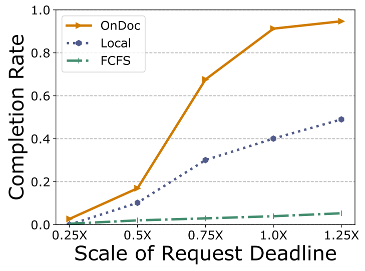
– **Local Heuristic (Local)**: when an application request arrived at **initial server**, the local heuristic will process it in it. Precisely, we limit the target server of each task to its initial edge server in Alg 1. This baseline makes sense to verify that task assignment of one request should be considered even with induced communication cost.
– **First-Come-First-Serve (FCFS)**: *FCFS* is a popular scheduling policy which is commonly used by the methods based on queuing theory [5]. We implement it by converting multiple task scheduling lists to a single list with respect to the releasing order and always assigning the task in the head of the list to its target server.

Moreover, the two baselines employ the same task priorities and function configuration strategy as `OnDoc`.

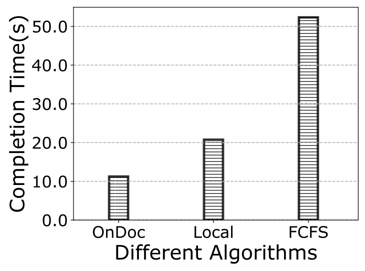
4.3 Experiment Results

In this part, we first illustrate the overall performance. The result shows that `OnDoc` outperforms other baselines dramatically. The number of deadlines which are satisfied is at least $\mathbf{1.9}\times$ of that of the baselines under default setting. Furthermore, we conduct multi-group experiments to study the influences of different settings of various parameters (i.e., the offload overhead to the cloud and the capacity of the edge servers).

The Overall Performance. Fig. 2 demonstrates the performance of all algorithms on the workloads from Alibaba. We scale the deadline of each request from 0.25x to 1.25x of the original value in the default setting with other parameters remaining as the default value. The figure Fig. 2(a) depicts that the performance of all algorithms get better with the deadlines increasing. Meanwhile, `OnDoc` outperforms the baselines dramatically. Under the default setting, the number of requests that satisfy their deadline in `OnDoc` is $\mathbf{1.9}\times$, $51.6\times$ that of *Local* and *FCFS* respectively. Furthermore, the makespan, the gap between the release time of the first request and the completing time of the lasted completed request, of our scheduling is minimum, which is a surprising byproduct shown by Fig. 2(b).

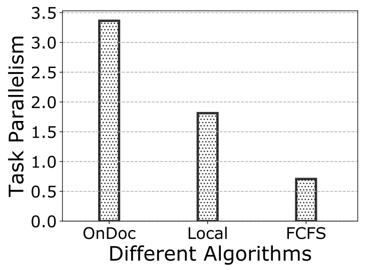


(a) Proportion of satisfied deadlines

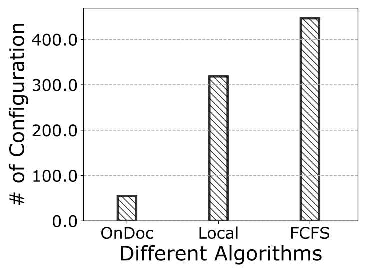


(b) Makespan

Fig. 2. Overall performance of different algorithms



(a) Task parallelism



(b) # of function configuration

Fig. 3. Task parallelism and # of function configuration of different algorithms

To understand why `OnDoc` outperforms the baselines, we conduct some further analysis. 1)From the perspective of the parallelism between tasks from different or even the same request, Fig. 3(a) demonstrates that `OnDoc` can exploit the parallelism well. Task parallelism is defined as the average number of running tasks at every moment. Fig. 3(a) depicts that task parallelism of `OnDoc` is the highest. It means that `OnDoc` utilizes the resources of edge servers to process more tasks concurrently than the baselines. Hence, `OnDoc` makes use of computing resources more sufficiently. Meanwhile, it exploits the parallelism between tasks better. 2)Configuration time is relatively larger compared with processing time and communication cost of data transfer between edge servers. The communication time from edge servers to the remote cloud is the same order of magnitude as configuration time if the amount of transferring data is large. Thus, an appropriate trade-off between the long distance communication and configuration plays a significant role. Fig. 3(b) shows that the number of function configuration by `OnDoc` is dramatically less than the baselines. Further analysis, which depicts 39.5% of all tasks are assigned to the remote cloud, illustrates that `OnDoc` utilizes the remote cloud to decrease the configuration cost without inducing notable communication cost. Almost all the tasks assigned to remote cloud request for a relatively small amount of data, which means that `OnDoc` assigns tasks with a large amount of input data to edge servers to avoid notable communication delay. Meanwhile, `OnDoc` employs cloud to mitigate edge servers' pressure to avoid repeating configuration.

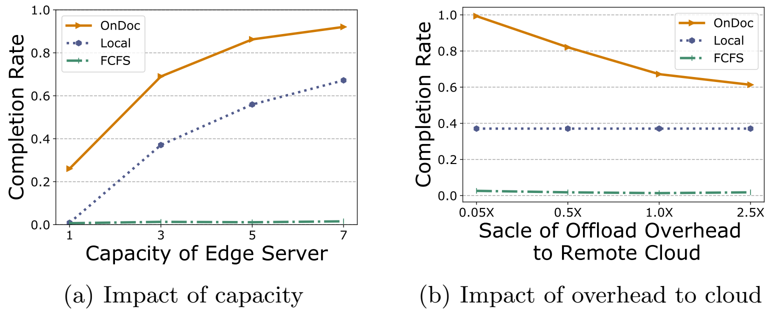


Fig. 4. The proportion of requests satisfying deadlines with different settings of the capacity of edge servers and the overhead to the cloud

Sensitivity Analysis. We also conduct abundant experiments to investigate the impact of different settings. Fig. 4(a) demonstrates the performance of all algorithms under different capacity setting. `OnDoc` and *Local* are affected significantly because more capacity means more computing resources. The number of function replacing decreases due to the increase in capacity. On the one hand, some repeating configurations are avoided. On the other hand, more capacity can support more task execution simultaneously, which can exploit the parallelism between tasks better. To study the impact of communication time to the remote cloud, we scale it from 0.05× to 2.5× of the default value. Fig. 4(b) illustrates when communication time is 0.05×, `OnDoc` can finish all requests before their deadline. Since the communication time to cloud equals to that between edge server, the cloud can provide powerful computing resources (i.e., infinite capacity, no function configuration time, faster task processing) without inducing more communication cost than edge servers. It is easily understood that *Local* is not affect by the overhead for it only uses edge servers. *FCFS* performs badly and is not sensitive to the above parameters since its task assignment policy is the main bottleneck constraining performance. The requests needing large processing time will block the execution of all requests that released after it.

5 Conclusion

In this work, we study task scheduling with function on-demand configuration in edge computing where requests for some application with specific deadlines and initial data arrive online in arbitrary time and order. We construct a general model for this problem with the goal to meet as many request deadlines as possible. An efficient heuristic algorithm, named `OnDoc`, is proposed, which is the first online algorithm to solve this problem to the best of our knowledge. Specifically, `OnDoc` is based on list scheduling schemes and can be implemented easily in practice. Besides, extensive experiments validate that `OnDoc` has a stable and superior performance compared with the baselines.

References

1. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: 6th International Proceedings on Computer systems, pp. 301–314. ACM (2011). https://doi.org/10.1145/1966445.1966473
2. Zhao, Y., Liu, X., Qiao, C.: Job scheduling for acceleration systems in cloud computing. In: International Proceedings on ICC, pp. 1–6. IEEE (2018). https://doi.org/10.1109/icc.2018.8422110
3. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The case for vm-based cloudlets in mobile computing. IEEE pervasive Computing (4), 14–23 (2009). https://doi.org/10.1109/mprv.200964
4. Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., et al.: Edge-centric computing: Vision and challenges.ACM SIGCOMM CCR **45**(5), 37–42 (2015). https://doi.org/10.1145/2831347.2831354
5. Tan, H., Han, Z., Li, X.Y., Lau, F.C.: Online job dispatching and scheduling in edge-clouds. In: 36th International Proceedings on INFOCOM, pp. 1–9. IEEE (2017). https://doi.org/10.1109/infocom.2017.8057116
6. Topcuoglu, H., S. Hariri., Wu, M.: Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE TPDS **13**(3), 260–274 (2002). https://doi.org/10.1109/71.993206
7. Neto, J.L.D., Yu, S.Y., Macedo, D.F., Nogueira, M.S., Langar, R., Secci, S.: ULOOF: A user level online offloading framework for mobile edge computing. IEEE TMC **17**(11), 2660–2674 (2018). https://doi.org/10.1109/tmc.2018.2815015
8. Sundar, S., Liang, B.: Offloading dependent tasks with communication delay and deadline constraint. In: 37th International Proceedings on INFOCOM, pp. 37–45. IEEE (2018). https://doi.org/10.1109/infocom.2018.8486305
9. Zhang, W., Wen, Y., Wu, D.O.: Energy-efficient scheduling policy for collaborative execution in mobile cloud computing. In: 32th International Proceedings on INFO-COM, pp. 190–194. IEEE (2013). https://doi.org/10.1109/infcom.2013.6566761
10. Guo, H., Liu, J., Zhang, J.: Efficient Computation Offloading for Multi-Access Edge Computing in 5G HetNets. In: International Proceedings on ICC, pp. 1–6. IEEE (2018). https://doi.org/10.1109/icc.2018.8422238
11. Palis, M.A., Liou, J.C., Wei, D.S.L.: Task clustering and scheduling for distributed memory parallel architectures. IEEE TPDS **7**(1), 46-55 (1996). https://doi.org/10.1109/71.481597
12. Darbha, S., Agrawal, D.P.: Optimal scheduling algorithm for distributed-memory machines. IEEE TPDS **9**(1), 87–95 (1998). https://doi.org/10.1109/71.655248
13. Sakellariou, R., Zhao, H.: A hybrid heuristic for DAG scheduling on heterogeneous systems. In: 18th International Proceedings on IPDPS, pp. 111. IEEE (2004). https://doi.org/10.1109/ipdps.2004.1303065
14. Deng, M., Tian, H., Fan, B.: Fine-granularity based application offloading policy in cloud-enhanced small cell networks. In: International Proceedings on ICC, pp. 638–643). IEEE (2016). https://doi.org/10.1109/iccw.2016.7503859
15. He, K., Meng, X., Pan, Z., Yuan, L., Zhou, P.: A Novel Task-Duplication Based Clustering Algorithm for Heterogeneous Computing Environments. IEEE TPDS **30**(1), 2–14 (2019). https://doi.org/10.1109/tpds.2018.2851221
16. Shin, K., Cha, M., Jang, M., Jung, J., Yoon, W., Choi, S.: Task scheduling algorithm using minimized duplications in homogeneous systems. Elsevier JPDC, **68**(8), 1146–1156 (2008). https://doi.org/10.1016/j.jpdc.2008.04.001

17. Liu, G.Q., Poh, K.L., Xie, M.: Iterative list scheduling for heterogeneous computing. Elsevier JPDC **65**(5), 654-665 (2005). https://doi.org/10.1016/j.jpdc.2005.01.002
18. Ali, J., Khan, R.Z.: Optimal task partitioning model in distributed heterogeneous parallel computing environment. AIRCC IJAIT **2**(6): 13 (2012). https://doi.org/10.5121/ijait.2012.2602
19. He, K., Zhao, Y.: A new task duplication based multitask scheduling method. In: 5th International Proceedings on Grid and Cooperative Computing, pp. 221–227. IEEE (2006). https://doi.org/10.1109/gcc.2006.13
20. Azure Functions, https://azure.microsoft.com/en-us/services/functions
21. Alibaba trace, https://github.com/alibaba/clusterdata. 2018