Efficient Serverless Function Scheduling in Edge Computing

Jiong Lou*, Zhiqing Tang†, Xinyu Lu*, Shijing Yuan*, Jie Li*, Weijia Jia†, and Chentao Wu*
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†Institute of Artificial Intelligence and Future Networks, Beijing Normal University, China
lj1994@sjtu.edu.cn, zhiqingtang@bnu.edu.cn,{gale13,2019ysj,lijiecs}@sjtu.edu.cn,jiawj@bnu.edu.cn,wuct@sjtu.edu.cn

***Abstract*—Serverless computing is a promising approach for edge computing since its inherent features, e.g., lightweight virtualization, rapid scalability, and economic efficiency. However, there are two challenges existing in serverless edge computing: significant cold start latency and request blocking. Previous studies have not successfully resolved these challenges, which affect the Quality of Experience. In this paper, we formulate the Serverless Function Scheduling (SFS) problem in resource-limited edge computing, aiming to minimize the average response time. To solve this intractable scheduling problem, we first consider a simplified offline form of the SFS problem and design a polynomial-time optimal scheduling algorithm. Inspired by this optimal algorithm, we propose an Enhanced Shortest Function First (ESFF) algorithm, including function creation and function replacement. To avoid frequent cold starts, ESFF selectively decides the initialization of new function instances when receiving requests. To deal with request blocking, ESFF judiciously replaces serverless functions based on the function weight at the completion time of requests. Extensive simulations based on real-world serverless request traces are conducted, and the results show that ESFF consistently and substantially outperforms existing baselines under different settings.**

***Index Terms*—Serverless function scheduling, Edge computing, Faas.**

I. Introduction

In edge computing, computation resources are deployed at the network edge to offer low-latency services [1]. However, the edge servers are still resource-limited compared with cloud servers. Besides, the frequent mobility of users generates highly dynamic workloads [2]. It is challenging to offer high-quality services for dynamic requests with limited edge resources. Serverless computing [3], a paradigm designed for dynamic short-lived computations, is promising for edge computing [4]. The serverless function is initialized on demand to avoid resource wastage. Besides, it is economical for it only charges for CPU time spent in request execution [5]. Fig. 1 depicts the overview of serverless edge computing, where various requests are released by mobile users and processed by edge servers.

Most of the current serverless platforms, e.g., OpenWhisk [6], Lambda [7] and Azure [8], are designed for cloud computing, unsuitable to resource-limited edge computing. They scale up serverless function instances when there is no idle instance for waiting requests [5]. Though the waiting time for idle instances may be eliminated by initializing new instances, this mechanism can easily bring function instance over-provision [9] and frequent cold starts [10] (i.e., the process of initializing a new function instance). These two problems result in delayed response and largely affect the Quality of Experience. Therefore, the response time of serverless edge computing should be optimized.

(Corresponding author: Jie Li)

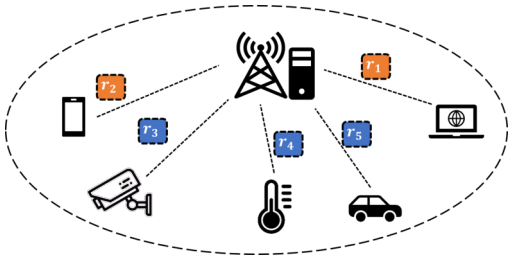
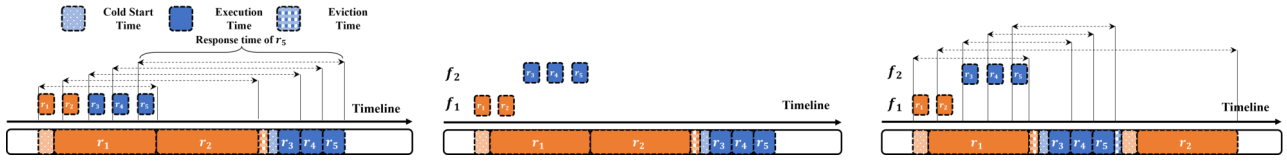


Fig. 1. The overview of serverless edge computing.

Most of existing studies about the serverless edge computing focus on data communication [11], function placement [12]–[14] and function cache [15]. They cannot properly resolve the severe problems of function instance over-provision and frequent cold starts. Recently, two schedulers have been designed to partially mitigate these problems, considering the scheduling for each single function [16], [17]. The OpenWhisk V2 is designed to separate the control flow and data flow [16]. LaSS [17] estimates the instance number for each function to meet the individual deadline based on queue theory but neglects the cold starts.

However, these studies still do not overcome the following challenges, which greatly increase the response time: (1) Due to the highly dynamic workloads and limited resources in edge computing, the serverless function instances will be frequently replaced, which prolongs the current request's response time and delays the future requests. (2) In resource-limited edge computing, short serverless function requests are easily blocked by long requests, which increases the average response time. Moreover, request bursts are very common for highly dynamic workloads in edge computing [18], making the request blocking worse. These challenges make the serverless function scheduling in edge computing much intractable and result in poor performance of existing methods. The key solutions for these two challenges are specifically designed serverless function creation and replacement policies.



(a) OpenWhisk (b) OpenWhisk V2 (c) ESFF

Fig. 2. An illustrative scheduling example of OpenWhisk, OpenWhisk V2 and ESFF. The requests with the same color belong to the same function. Two long requests r_1, r_2 of function f_1 arrive earlier than there short requests r_3, r_4, r_5 of function f_2. ESFF achieves the shortest average response time.

In this paper, we first formulate the Serverless Function Scheduling (SFS) problem in resource-limited edge computing, aiming to minimize the average response time, which is proved to be NP-hard. To efficiently solve this intractable scheduling problem, we first analyze a Simplified form of the SFS (SSFS) problem: offline scheduling of requests on an edge server that can execute a single function at once. Then, based on the optimal scheduling algorithm of SSFS, we propose an Enhanced Shortest Function First (ESFF) algorithm, considering the aforementioned challenges. ESFF consists of two sub-policies: (1) Function Creation Policy (FCP). FCP selectively initializes a new function instance at the request arrival time by judging whether the average response time can benefit from the function initialization. (2) Function Replacement Policy (FRP). FRP replaces functions at the request completion time according to the function weight, i.e., the scheduling urgency.

To the best of our knowledge, we are the first to formulate the Serverless Function Scheduling problem, fully considering the significant cold start latency and highly dynamic function workloads in resource-limited edge computing. The contributions of this paper are summarised as follows:

1) Firstly, we analyze a simplified problem SSFS and prove that the optimal scheduling of SSFS can be obtained in polynomial time.
2) Secondly, we design a lightweight scheduling algorithm to solve the SFS problem in an online manner. ESFF judiciously prioritizes each serverless function based on the execution time, the cold start time, and the number of waiting requests. It initializes and replaces new function instances based on function weights.
3) Finally, extensive simulations based on real-world serverless request traces [19] show that ESFF substantially and consistently outperforms existing baselines.

II. Motivation

In Fig. 2, we give a simple example to illustrate the motivation of serverless function scheduling. Fig. 2(a) and 2(b) generated by different scheduling algorithms show the same results. In Fig. 2(c), the proposed algorithm ESFF produces the best scheduling in terms of the average response time.

In Fig. 2(a), OpenWhisk processes the requests in the central queue based on the ascending order of their arrival time. As a result, short requests r_3, r_4 and r_5 are seriously blocked by long requests r_1 and r_2. In a real edge environment, the request bursts are very common [20], so many short requests will be blocked by long requests with the scheduling of OpenWhisk. In Fig. 2(b), OpenWhisk V2 maintains an individual queue for each function. If there are requests waiting in the queue, the required function instance will continue to process these requests. This design still results in blocking. In Fig. 2(b), when request r_1's execution is finished, request r_2 has already arrived at the edge server, waiting in the queue. Moreover, if multiple requests of the function f_1 arrive at the edge server before finishing the request r_2, the requests r_3, r_4, and r_5 will be blocked again, which is unreasonable and far from optimal.

In Fig. 2(c), ESFF also sends each request to its individual function queue. It differs from OpenWhisk V2 in that after finishing a request, a function instance can be selectively replaced by another function instead of processing the requests in its queue. Therefore, short requests will not be largely blocked by long ones, and ESFF achieves the best performance.

III. System Model and Problem Formulation

A. System Model

Request Model: In this paper, the serverless function requests are released by mobile users and arrive at the serverless platform over time. A request is denoted as $r_i \in \mathbf{R}$, where $\mathbf{R}$ is the set of requests. Request r_i's arrival time is denoted as t_i^a. The execution time of request r_i is denoted as t_i^e. Due to the dynamic nature of edge computing, the arrival time and execution time of each request are unknown in prior and hard to predict [21]. The start execution time and the completion time of the request r_i are denoted as t_i^s and t_i^c, respectively.

Serverless Function Model: The function set is denoted by $\mathbf{F} = \{f_1, f_2, \ldots, f_{|\mathbf{F}|}\}$. The function of request r_i is denoted by $l_i \in \mathbf{F}$. Before processing a request, the corresponding function instance should be initialized, named cold start. For a function f_j, the cold start latency is defined as t_j^l. In this paper, it is assumed that request execution cannot be interrupted. k_o^j represents the o-th instance of f_j, with two states: (1) Idle state, $state(k_o^j) = 0$, when waiting for requests. (2) Busy state, $state(k_o^j) = 1$, when processing a request. After finishing a request, the function instance turns idle, waiting for future requests. An idle function can be evicted to release the resources. For a function f_j, the eviction time is defined as t_j^v. The set of initialized instances of function f_j is defined as $\mathbf{K}^j = \{k_1^j, k_2^j, \ldots, k_{|\mathbf{K}^j|}^j\}$, which changes over time.

Edge Server Model: For simplicity, the serverless platform is assumed to be deployed on a resource-constrained edge server[1]. The resource capacity of the edge server is represented by the maximum number of function instances that can be executed concurrently, denoted as C.

[1]Multiple edge servers inter-connected by the ultra-low latency network can be modeled as a powerful edge server for the neglectable transmission time.

Algorithm 1: Enhanced Shortest Function First

if *A request r_i arrives at the edge server* **then**
  Invoke Algorithm 2;
if *A function instance k_o^j finishes execution* **then**
  Invoke Algorithm 3;

B. Problem Formulation

For each request r_i, it cannot be processed before arriving, i.e., $t_i^a < t_i^s$. As each request's processing cannot be interrupted, therefore, $t_i^c = t_i^s + t_i^e$. Before processing a request r_i, an instance of the required function l_i should be initialized and in the idle state.

At any time, the edge server can hold at most C function instances. Each function instance can process one request of its type at once [6]. When an instance of f_j is initialized without evicting any function instance, it requires t_j^l for cold start; When it is initialized by evicting an instance of function $f_{j'}$, it requires $t_{j'}^v$ for eviction and t_j^l for cold start.

The response time of a request r_i is $t_i^r = t_i^c - t_i^a$, including both the execution time and waiting time. The objective is to optimize the average response time of all requests:

$$T = \frac{1}{|\mathbf{R}|} \sum_{r_i \in \mathbf{R}} t_i^r. \tag{1}$$

The SFS problem is an online scheduling problem, i.e., making scheduling decisions without future information.

Theorem 1. *The SFS problem is NP-hard.*

Proof. The theorem is proved in [22]. □

IV. Simplified Form of Serverless Function Scheduling

Due to the NP-hardness of SFS, we consider a Simplified form of SFS (SSFS) and design an optimal scheduling algorithm, which can efficiently be extended for the SFS problem.

SSFS is simplified in the following aspects: (1) The edge server is unary, i.e., holding at most one initialized function instance at any time. (2) The requests of the same function have identical execution time, i.e., $t_i^e = t_{i'}^e = t_j$ if $l_i = l_{i'} = f_j$. t_j is the request f_j's execution time and also function f_j's execution time. (3) All requests arrive at the edge server at time 0. (4) All functions' request number, cold start time, eviction time, and execution time are known before scheduling. The request number of the function f_j is denoted as n_j.

Optimal scheduling algorithm of the SSFS problem (OSSFS): Since the edge server is unary, the scheduling of SSFS is equivalent to request ordering. Similar to shortest job first, the OSSFS algorithm has two steps. (1) Each function is associated with a weight, which is defined as $w_j = t_j + \frac{t_j^l + t_j^v}{n_j}$. (2) Each function's requests are processed continuously and processed in ascending order of function weights.

Theorem 2. *The OSSFS algorithm generates the optimal scheduling of the SSFS problem.*

Algorithm 2: Function Creation Policy

Input: $\overline{t_j^l}$, $\overline{t_j^v}$, $\overline{t_j^e}$, n_j^w, r_i, q_j, $\mathbf{F}$, $\mathbf{K}^j$
1 **if** $n_j^w = 0$ *and* $\sum_{k_o^j \in \mathbf{K}^j} state(k_o^j) > 0$ **then**
2 Process r_i in f_j's idle instance;
3 **else**
4 **if** $\sum_{f_j \in \mathbf{F}} |\mathbf{K}^j| < C$ **then**
5 Compute n_j^e based on Eq. (2);
6 **if** $n_j^e > 0$ **then**
7 Initialize an instance for function f_j;
8 **else if** $\sum_{f_j \in \mathbf{F}} |\mathbf{K}^j| = C$ **then**
9 Compute $\mathbf{S}$ based on Eqs. (3) and (4);
10 **if** $|\mathbf{S}| > 0$ **then**
11 $f_{j'} = \text{argmax}_{f_{j'} \in \mathbf{S}} \overline{t_{j'}^e}$;
12 Replace an idle instance of $f_{j'}$ with f_j;
13 q_j.join(r_i);

Proof. The theorem is formally proved in [22]. □

V. ESFF Algorithm

Inspired by the OSSFS algorithm, the Enhanced Shortest Function First (ESFF) algorithm is designed for online scheduling by considering multiple function instances and unknown future information. First, the average execution time $\overline{t_j^e}$, average cold start latency $\overline{t_j^l}$, and average eviction time $\overline{t_j^v}$ of f_j are computed according to the history. Then, the definition of function weight is modified to adapt to the SFS problem.

As shown in Algorithm 1, ESFF consists of two sub-policies: (a) Function Creation Policy (FCP), invoked when a new request r_i of function f_j arrives. FCP decides whether a new function instance should be initialized. (b) Function Replacement Policy (FRP), invoked when a request r_i is finished by an instance of function f_j. FRP decides whether this instance should be replaced by other function instances. At other time, each function instance continuously processes the requests waiting in its queue.

A. Function Creation Policy

When a request r_i of function f_j arrives, FCP is invoked to judge whether a new instance of f_j should be initialized. The total number of f_j's requests waiting in its queue q_j at r_i's arrival time is denoted as n_j^w.

Algorithm 2 depicts the pseudocode of FCP. The key idea of FCP is making scheduling decisions according to the current state of the system. If $n_j^w = 0$ (i.e., the queue q_j is empty) and there exists an idle instance of the function f_j, the request r_i is directly sent to an idle function instance for execution. In lines 4-7, if there is no idle function instance of f_j and there are enough resources for initializing a new function instance of f_j, FCP prefers to initialize a new function instance with free resources than replace an idle function instance. It assesses whether the total response time can benefit from the function instance initialization, i.e., at least one request will be

Algorithm 3: Function Replacement Policy

Input: k_o^j, $\overline{t_j^e}$, $\overline{t_j^v}$, $\mathbf{K}^j$, $\overline{t_j^l}$, $n_{j'}^w$

1 Compute w_j based on Eq. (5);
2 $\mathbf{S} = \{f_{j'} | n_{j'}^w > 0\}$;
3 $f_x = f_j$, $w_x = w_j$;
4 **if** $|\mathbf{S}| < 1$ **then**
5 **for** $j' \in \mathbf{S}$ **do**
6 Estimate $n_{j',j}^e$ based on Eq. (3);
7 Compute $w_{j'}$ based on Eq. (6);
8 **if** $w_{j'} < w_x$ **then**
9 $w_x = w_{j'}$, $f_x = f_{j'}$;
10 **if** $f_x \neq f_j$ **then**
11 Replace k_o^j with a new instance of function f_x;
12 **else if** $n_j^w > 0$ **then**
13 k_o^j processes the first request in q_j;
14 **else**
15 $state(k_o^j) = 1$;

processed by the new function instance. The number of f_j's waiting requests after the cold start latency $\overline{t_j^l}$ is estimated by:

$$n_j^e = n_j^w + 1 - \frac{\overline{t_j^l}|\mathbf{K}^j|}{\overline{t_j^e}}. \tag{2}$$

If $n_j^e > 0$, then it is likely that at least one request will be processed by the newly initialized function instance of f_j, so a new instance will be initialized; Otherwise, no function instance will be initialized. The request r_i will join the queue q_j in this and the following conditions. In lines 8-12, if there is no idle function instance of f_j and the resources are insufficient to initialize such instance, it assesses whether the total response time can benefit from replacing an idle function instance. The number of f_j's waiting requests after $f_{j'}$'s eviction time and f_j's cold start latency is estimated by:

$$n_{j,j'}^e = n_j^w + 1 - \frac{(\overline{t_j^l} + \overline{t_{j'}^v})|\mathbf{K}^j|}{\overline{t_j^e}}. \tag{3}$$

The candidate set of functions with enough eviction time that makes $n_{j,j'}^e > 0$ is computed by:

$$\mathbf{S} = \{f_{j'} | n_{j,j'}^e > 0 \; and \sum_{k_o^{j'} \in \mathbf{K}^{j'}} state(k_o^{j'}) > 0\}. \tag{4}$$

FCP chooses $f_{j'} \in \mathbf{S}$ with the largest $\overline{t_{j'}^e}$ and replaces an idle instance of $f_{j'}$ with a new instance of f_j. If $\mathbf{S}$ is empty, no function instance will be initialized.

B. Function Replacement Policy

When an instance k_o^j of the function f_j finishes a request, FRP is invoked to decide whether k_o^j should be replaced. Similar to the OSSFS algorithm, the function f_j is associated with a weight to represent its scheduling urgency, defined as:

$$w_j = \overline{t_j^e} + \frac{\overline{t_j^v}|\mathbf{K}^j|}{n_j^w}. \tag{5}$$

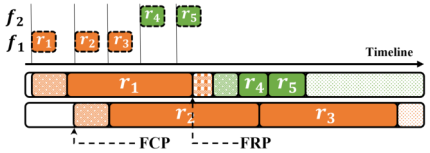


Fig. 3. The scheduling example of ESFF. FCP initializes a new function instance for the request r_2, and replaces the function instance for request r_4.

The item $\overline{t_j^l}$ is removed from the numerator in Eq. (5) for the function f_j is already initialized.

Algorithm 3 shows the pseudocode of FRP. The prime idea of FRP is trying to replace the existing function instance with a more urgent function based on the comparison of function weights. The scheduling decision made by FRP also depends on the current state of the system as follows. In lines 1-9, FRP finds the function that can benefit most from replacing the instance k_o^j. Since the future information of requests is unavailable, we only consider the current requests. The candidate function set is $\mathbf{S} = \{f_{j'} | n_{j'}^w > 0\}$. It means that there must exist requests waiting in the queue of candidate functions; Otherwise, there is no need for initializing a new function instance. Similar to Algorithm 2, the number of $f_{j'}$'s waiting requests, $n_{j',j}^e$, after evicting k_o^j and initializing a new instance of $f_{j'}$ is estimated based on Eq. (3). In line 7, the weight of each function in the candidate function set is:

$$w_{j'} = \overline{t_{j'}^e} + \frac{(\overline{t_{j'}^l} + \overline{t_{j'}^v})(|\mathbf{K}^j| + 1)}{n_{j',j}^e}. \tag{6}$$

Finally, the function $f_{j'} \in \mathbf{S}$ with the smallest $w_{j'}$ and $w_{j'} \leq w_j$ is selected to replace the function instance k_o^j. In lines 12-13, if there is no function satisfying the above requirements, the function instance k_o^j will not be replaced. If there is any request waiting in the queue q_j, k_o^j processes the first request at the head of the queue. Otherwise, k_o^j turns idle, $state(k_o^j) = 1$.

C. Scheduling Example

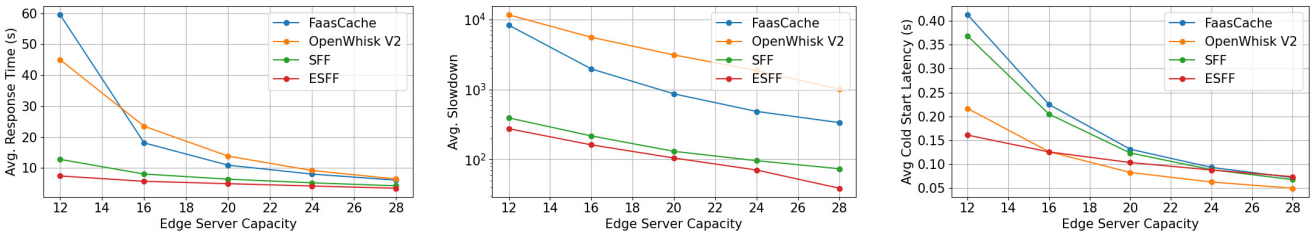
Fig. 3 is a scheduling example of ESFF. The edge server's capacity is two. Five requests of two functions arrive at the edge server at different times. Requests r_1, r_2 and r_3 are of function f_1 and requests r_4, r_5 are of function f_2. The request r_1 incurs a cold start at the beginning. When r_2 arrives, FCP decides to initialize a new instance of function f_1 to reduce r_2's waiting time. At the arrival time of r_3, all function instances are busy and the computation resources are insufficient, so r_3 joins the queue q_1. When r_1 is finished, FRP decides to initialize a new instance of function f_2 to replace the instance of function f_1 since $w_2 < w_1$ and $n_{j'}^w > 0$.

VI. Evaluation

A. Setups

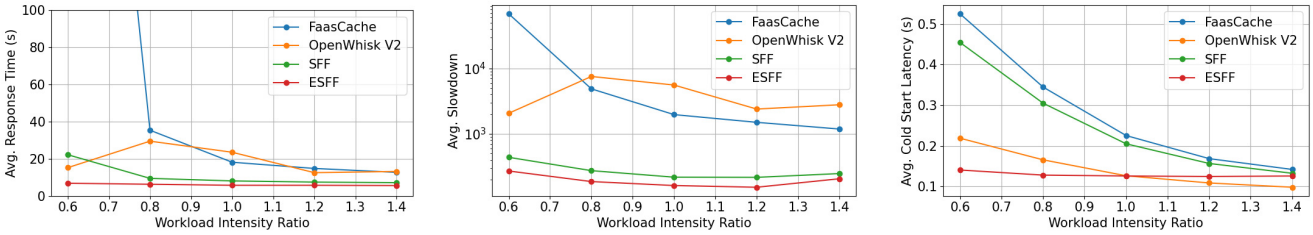
The ESFF algorithm, baseline algorithms, and the simulation environment are implemented in Python 3.6 on a desktop with an Intel Core i9-10900K 3.7 GHz CPU and 32GB RAM.

The serverless request traces are collected from a real cluster of Azure, containing 2.2×10^6 requests during two weeks [19]. The function, completion time, and execution time of each



(a) Average Response Time (b) Average Slowdown (c) Average Cold Start Time

Fig. 4. Average response time, slowdown and cold start time under different edge server capacities.



(a) Average Response Time (b) Average Slowdown (c) Average Cold Start Time

Fig. 5. Average response time, slow down and cold start time under different workload intensity ratios.

request are recorded. Limited by the measurement precision, some requests' execution time is recorded as 0, and is set to 1ms in the following experiments. We choose the first 6×10^5 requests as the simulation dataset.

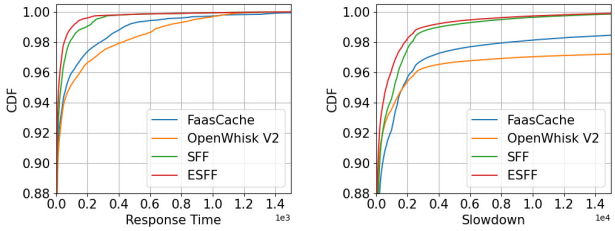
The default capacity of the resource-limited edge server is set to 16. Each function's cold start latency and eviction time are randomly chosen from [0.5,1.5] according to [23], since function details (e.g.,function codes and dependencies) are not included in the request traces [19].

We select three most related serverless function scheduling algorithms as baselines: (1) FaasCache [24]. It schedules requests sequentially based on the arrival time. When there is no idle function instance for the current request, FaasCache tries to initialize a new instance or replace another instance. (2) OpenWhisk V2 [16]. It makes requests of the same function wait in an individual queue. When the request at the queue head has waited for more than a fixed threshold (i.e., 100ms), a corresponding function instance is initialized. (3) Shortest Function First (SFF). The only difference between SFF and OpenWhisk is that SFF schedules requests sequentially based on the ascendant order of their average execution time.

The following two metrics are used to evaluate ESFF: (1) Average response time, $\frac{1}{|\mathbf{R}|}\sum_{r_i \in \mathbf{R}} (t_i^c - t_i^a)$. (2) Average slowdown, $\frac{1}{|\mathbf{R}|}\sum_{r_i \in \mathbf{R}} \frac{t_i^c - t_i^a}{t_i^e}$.

B. Comparison with Baselines

Impact of edge server capacity. In Fig. 4, ESFF is compared against baselines with respect to different edge server capacities. In Fig. 4(a), compared with the best baseline SFF, ESFF substantially reduces the average response time by 18% to 40%. When the edge server capacity increases, the function replacement times will be reduced for all scheduling algorithms. Therefore, the average cold start time is also reduced as shown in Fig. 4(c). Besides, with more resources, the edge server can better handle the request bursts, and then request blocking problem will be mitigated. FaasCache and OpenWhisk V2 have very poor performance under very limited resources since they are unaware of massive cold starts and request blocking. Fig. 4(b) proves that by mitigating the request blocking problem, ESFF successfully reduces the response time of short requests and achieves the best fairness.



(a) CDF of Response Time (b) CDF of Slowdown

Fig. 6. CDF of response time and slowdown.

Impact of workload intensity. In Fig. 5, the workload intensity ratio is defined to control the intervals between requests. In Fig. 5(a), ESFF achieves the lowest average response time under different workload intensity. With lower workload intensity ratios, the improvement of ESFF is more. In Fig. 5(b), we observe that the average slowdown of ESFF increases as the workload intensity ratio increases from 1.0 to 1.4. The reason is that with longer intervals, the function instances of short requests will be more likely replaced, which brings more cold starts.

The cumulative distribution function (CDF) of response time and slowdown is shown in Fig. 6. Compared with the baselines, the CDF curve of ESFF is always closer to the left, which means that the response time and slowdown of ESFF are consistently better. In particular, the P95 and P99 response time of ESFF are much better than other baselines.

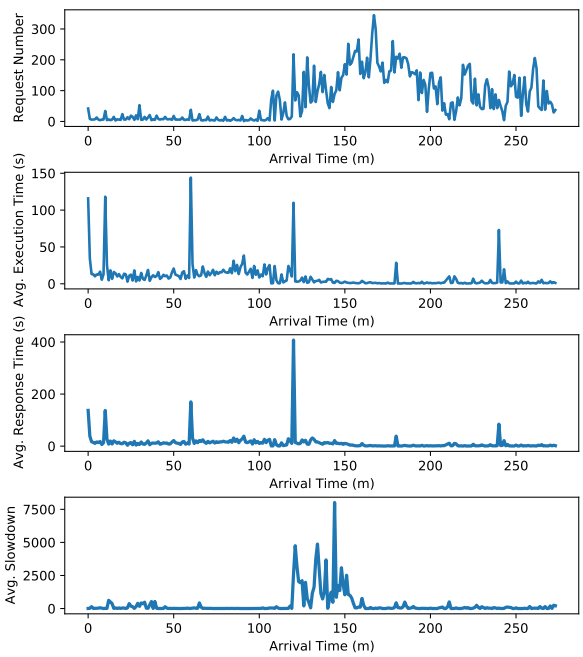


Fig. 7. Detailed scheduling results of ESFF over request arrival time.

C. Further Analysis

In Fig. 7, we select 20000 continuous arriving requests to show detailed results of ESFF on each minute. Three points can be observed in this experiment: (1) It is obvious that with longer execution time, the response time also increases. (2) More request numbers further prolong the response time since the longer waiting time (i.e., short requests are blocked by long requests). (3) A large number of long requests not only affect the current requests but also largely slow down the upcoming requests. From these observations, we learn that the request burst (i.e., request number and request size) leads to significant response time. To deal with request bursts, offloading long requests to the powerful cloud will be a better choice.

VII. Conclusion

In this paper, we formulated the SFS problem for resource-limited edge computing. We proposed a polynomial-time optimal scheduling algorithm for a simplified offline form of SFS. Then, we designed an Enhanced Shortest Function First (ESFF) algorithm. To avoid wasteful cold starts, ESFF selectively decides function instance initialization when requests arrive. To deal with dynamic workloads, ESFF judiciously replaces function instances based on function weights. Extensive simulations based on real-world traces show ESFF's substantial outperformance over existing baselines.

References

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

[2] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2333–2345, 2018.

[3] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[4] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: vision and challenges," in *Proceedings of the 2021 Australasian Computer Science Week Multiconference (ACSW)*, 2021, pp. 1–10.

[5] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 133–146.

[6] Apache openwhisk. [Online]. Available: https://openwhisk.apache.org

[7] Aws lambda. [Online]. Available: https://aws.amazon.com/lambda/

[8] Microsoft azure serverless functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions

[9] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Proceedings of the 21st International Middleware Conference (Middleware)*, 2020, pp. 280–295.

[10] S. Wu, Z. Tao, H. Fan, Z. Huang, X. Zhang, H. Jin, C. Yu, and C. Cao, "Container lifecycle-aware scheduling for serverless computing," *Software: Practice and Experience*, vol. 52, no. 2, pp. 337–352, 2022.

[11] C. Cicconetti, M. Conti, and A. Passarella, "Faas execution models for edge applications," *arXiv preprint arXiv:2111.06595*, 2021.

[12] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI)*, 2019, pp. 225–236.

[13] D. Bermbach, J. Bader, J. Hasenburg, T. Pfandzelter, and L. Thamsen, "Auctionwhisk: Using an auction-inspired approach for function placement in serverless fog platforms," *Software: Practice and Experience*, vol. 52, no. 5, pp. 1143–1169, 2022.

[14] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in *Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 41–50.

[15] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proceedings of the 41st IEEE Conference on Computer Communications (INFOCOM)*, 2022, pp. 1069–1078.

[16] Openwhisk future architecture. [Online]. Available: https://cwiki.apache.org/confluence/display/OPENWHISK/OpenWhisk+future+architecture

[17] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: running latency sensitive serverless computations at the edge," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2021, pp. 239–251.

[18] C. Zhang, H. Tan, G. Li, Z. Han, S. H.-C. Jiang, and X.-Y. Li, "Online file caching in latency-sensitive systems with delayed hits and bypassing," in *Proceedings of the 41st IEEE Conference on Computer Communications (INFOCOM)*, 2022, pp. 1059–1068.

[19] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 724–739.

[20] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *Proceedings of the 2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 20–26.

[21] H. Tan, Z. Han, X.-Y. Li, and F. C. Lau, "Online job dispatching and scheduling in edge-clouds," in *Proceedings of the 36st IEEE Conference on Computer Communications (INFOCOM)*, 2017, pp. 1–9.

[22] Efficient serverless function scheduling at the network edge. [Online]. Available: https://1drv.ms/b/s!AvUGoCG3JpiugcZMFDFUFkSvUUDKzQ?e=sdH7Kc

[23] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020, pp. 30–44.

[24] A. Fuerst and 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 (ASPLOS)*, 2021, pp. 386–400.