QoS-Constrained Service Selection for Networked Microservices

ZHIJUN DING\(^{1,2}\), (Senior Member, IEEE), SHENG WANG\(^{1,2}\), AND MEIQIN PAN\(^{3}\)

\(^{1}\)Department of Computer Science and Technology, Tongji University, Shanghai 201804, China
\(^{2}\)Key Laboratory of Embedded System and Service Computing, Ministry of Education, Tongji University, Shanghai 201804, China
\(^{3}\)School of Business and Management, Shanghai International Studies University, Shanghai 200083, China

Corresponding author: Meiqin Pan (panmqin@sina.com)

This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1704100, in part by the National Natural Science Foundation of China under Grant 61672381, and in part by the Fundamental Research Funds for the Central Universities under Grant 22120180508.

ABSTRACT

In the microservice system, microservices are generally deployed as many microservice instances connected by a network system, and there are several composite services (CSs) in the system, each of which consists of several tasks that need to be executed on instances and has some Quality of Service (QoS) constraints to represent the user requirement for QoS. At runtime, it is necessary to select the microservice instance for each task under constraints and form an execution path of the CS. However, traditional service selection methods focus on the QoS optimization of a single CS, while ignoring sharing and competition of instances. To address this problem, this paper presents a Microservice Service Selection algorithm (MSS) based on list scheduling. Firstly, a workflow model is employed to describe the CS and to analyze the processing speed of instances, the transferring speed of the network and the degree of task concurrency, to calculate the sub-deadline of each task. Then, according to the sub-deadline and other information, the scheduling urgency of each task is calculated and updated in real-time. Finally, two service selection strategies based on the sub-deadline and urgency are proposed to complete the microservice instances selection and constitute the service path. Experiments have been carried out using the standard workflow examples of real-world applications. The results show that the proposed selection strategies can effectively improve the performance of service selection in network systems.

INDEX TERMS

Distributed system, microservice, scheduling, service selection.

I. INTRODUCTION

To solve the problems of maintaining an application in the traditional monolithic system [1], microservices (MS) have been proposed and adopted by many commercial companies, such as Netflix and Amazon [2]. In the microservice architecture, an application is divided into a series of small services that can be independently developed, deployed and run. And then the different microservices are aggregated to form a CS. In general, each type of microservices will deploy several instances, which may be distributed over several servers or data centers and connected by the network, to share the workload. Although there is no functional difference, the QoS these instances provide will vary due to different configurations. Therefore, the system needs to select an instance meeting the QoS requirement from candidate instances at run time, to form a service path to complete the execution.

In related literature, how to obtain the optimal QoS for a single CS is a major problem to target. For example, several studies try to get global optimal solutions by integer programming [15] and other search-based algorithms [16], [17]. But these methods assume the QoS of each service is stable, ignoring the fact that it may fluctuate due to various reasons, e.g., the execution time of a subtask in CS is influenced by the workload of the subtask and other factors [19]. In order to adapt to the dynamic QoS, some recent works describe QoS as discrete random variables [18] or other models [19] and propose the corresponding service selection algorithms. However, in a real system, it is necessary to serve multiple CSs at the same time, and these CSs may share service instances. In this case, the fluctuation of the QoS mainly comes from the sharing of instances and the consequent queue time. When the system serves multiple
CSs, it must schedule the execution of these requests on the limited instances. At this point, the sharing of instances between CSs will result in that CSs interfere with each other. For example, when two CSs call the same instance at the same time, one CS may be forced to wait because the instance is executing another one, failing to achieve the expected QoS. The QoS change caused by the competition of CSs is ignored by traditional method. What’s more, there are data transmissions over the network across tasks, which also have an impact on the overall QoS [22].

Due to the different characteristics, demands and QoS requirements of different CSs [25], there is some room for optimization during executing tasks. There is such an optimization possibility - under the principle of satisfying QoS, appropriately delay the execution of a certain task, giving the service instance to tasks that need to be executed immediately, making these tasks meet the deadline. Therefore, in order to solve the competition problem, a feasible strategy is to use QoS (such as the deadline) as constraints, and make as many requests as possible satisfy the users’ requirements by adjusting the execution order and service selection to achieve global optimization, instead of requiring each CS to have the optimal QoS. To this end, the key is 1) how to find which task needs to be executed immediately and 2) how to select instance for each task for the global optimization.

Thus, we use a workflow to describe the CS, considering the deadline as the main QoS metric, and introduce the list scheduling method of workflow scheduling, analyzing the impact of the execution order, the processing speed and the network transfer time, to select service instances for CSs in the microservice system.

This paper aims to propose a deadline-constrained service selection algorithm for microservices in network systems, which makes as many requests be processed on time as possible and reduces the overall delay time, in the fixed service instances. Firstly, we take into consideration the processing speed of instances, the transferring time of networks and the degree of task concurrency to calculate the sub-deadline of tasks. Then, in order to find the task to be executed immediately, we calculate the execution urgency of each task as the priority and sort tasks based on it. Finally, calculate the QoS that each service instance can provide, then use two proposed service selection strategies to select the appropriate instance for each task based on the above information such as the sub-deadline and the urgency of the task, to achieve the global optimization. The main contributions of this paper are as follows:

1) This paper formalizes the service selection problem in the microservice system, and for the competition of CSs, proposes a service selection algorithm based on list scheduling, optimizing the overall on-time completion rate.

2) Considering that current service selection algorithms ignore the impact of the task execution order and most existing scheduling algorithms only use the static information to sort tasks, this paper proposes a formula for calculating the urgency of tasks as a basis for the task ordering and combines it with service selection strategies to improve the performance.

3) Because the result of each service selection will change workload and the QoS fluctuation of instances, the strategy of updating dynamically the urgency and execution order of tasks is proposed. In addition, idle time gap search strategy and two service selection strategies, including service instance selection and abandoning strategy, are added into the algorithm to achieve the global optimization.

The rest of this paper is organized as follows. Section II discusses the related work. Section III describes the model and the problem of service selection for microservices. In Section IV, the detail of the algorithm is presented. Relevant experiments, results and analysis will be presented in Section V. Finally, Section VI concludes the entire work.

II. RELATED WORK

A. SERVICE SELECTION

Since the service-oriented framework was proposed, how to choose the appropriate service from similar services has been an important research topic. If the service providers cannot meet the promised QoS, they may be penalized [25], and thus to improve users’ satisfaction, QoS metrics, such as response time, have been the main considerations in service selection. A basic solution is to use the greedy strategy to select the service with best QoS for each task to achieve the optimization of entire CS [15]. However, due to the execution logic of the CS [15], the sales promotion of service providers [23] or the user’s preference for QoS [16], the greedy strategy doesn’t get global optimal solutions. Therefore, based on the analysis of the structure of the CS, the integer programming [15] and other search strategies [17] has been widely applied.

In actual scenarios, the QoS provided by the service is not stable and may fluctuate due to the network or other reasons. Some services may even fail so that some tasks must be re-executed, delaying the response time of the CS [26]. Therefore, Hwang et al. [18] considered the QoS fluctuation of service, described the QoS as discrete random variables with probability mass functions, and proposed a service selection algorithm based on it. Similarly, Zhang et al. [19] considered the impact of the number of resources held by the service instance and the size of the task on the QoS and selected the service path for the CS based on the prediction for the performance. Wu et al. [24] used Vickrey-Clarke-Groves auction to obtain the minimum social cost with quality constraints in the dynamic pricing market. Although the above methods all consider the performance change of service instances at runtime, they only optimized the QoS of a single CS. When the service selection for multiple CSs is required, the traditional service selection method must be repeatedly applied to each CS, which doesn’t guarantee the global optimization. Moreover, the “first come first serve” (FCFS) strategy used by traditional methods also ignores the performance improvement brought by task ordering.
B. WORKFLOW SCHEDULING

There are some strategies in workflow scheduling for solving the competition caused by the sharing of instances. Among workflow scheduling algorithms [5]–[7], list scheduling is widely used because of its lower time complexity and high-quality solution [3], compared with the time-consuming meta-heuristic algorithms [27]. HEFT (Heterogeneous Earliest-Finish-Time) and CPOP (Critical-Path-On-a-Processor) [5] are two classic list scheduling methods, and their common idea of calculating the task’s upward rank is followed by many list scheduling algorithms. On the basis of HEFT, Bittencourt et al. [6] and Arabnejad and Barbosa [7] predicted the impact of current processor selection on subsequent scheduling, further improving the makespan.

In deadline-constrained workflow scheduling, how to make use of the deadline constraint in scheduling is an important issue. Abrishami et al. [11], [12] proposed the concept of Partial Critical Path (PCP) and assigned the total deadline to each PCP. Based on whether the sub-deadline of a PCP is met, each PCP is allocated to a certain processor. Wu et al. [3] used the concept of the upward rank in the HEFT to define the probabilistic upward rank, and assigned sub-deadline to each task according to the proportion of the probabilistic upward rank.

The problem most similar to ours is the multi-workflow scheduling problem with deadline constraints. Malawski et al. [20] considered the problem of scheduling the workflow ensembles. The authors analyzed the deadline and budget as constraints, and designed several algorithms aiming at benefiting from completing workflows on time. However, they assume that there is a difference between workflows in priority, which is actually unfair. Bochenina et al. [21] studied the scheduling of multi-workflow with soft deadlines in limited resources and proposed three algorithms to optimize fairness and meet soft deadlines. But the constraint of soft deadlines on the makespan is weak, so that the situation may happen that a certain workflow exceeds its deadline. What’s more, the proposed algorithms schedule all workflows without a trade-off, which makes all workflows violate constraints when the deadline constraint is rigid.

In list scheduling, the heuristic information for task ordering is an important factor which affects the performance of the algorithm. However, most of the above algorithms use the static information as the basis for task ordering (e.g., the upward rank in [3], [5] and the critical path in [11], [12]), while ignoring the effect of the deadline, and the sub-deadlines of tasks are only used in the processor selection phase. Besides, the static task ordering cannot find the task in need of executing soon. To take into consideration the impact of the deadline and the dynamic information at runtime, we propose the urgency of tasks as a basis for the task ordering and the urgency will be updated in real-time. In addition, the new abandoning strategy is added to solve the timeout tasks, to ensure fairness and make more workflows meet constraints in multi-workflows scheduling with deadline constraints.

III. PROBLEM DEFINITION

A. MODEL OF THE COMPOSITE SERVICE

We describe a CS as a workflow that is represented by a Directed Acyclic Graph (DAG). Therefore, every CS can be described by a unique workflow, which can be defined by a two-tuple \( W = (V, E) \), as shown in the \( CS_1 \) of Fig. 1, where \( V = \{ t_i \} \) is the set of vertices, and each vertex represents a task in the CS; \( E = \{ e_{i,j} \} \) is the set of edges, and the \( e_{i,j} \) indicates the dependency between the two tasks. Therefore, the set \( parent(t_i) \) can be defined as the set of predecessor tasks of the task \( t_i \), meaning the task \( t_i \) can be executed only after all tasks in the \( parent(t_i) \) are completed. Similarly, the set \( child(t_i) \) is defined as the set of successor tasks of the task \( t_i \). In particular, the tasks without predecessor tasks or successor tasks are respectively defined as the entry task \( t_{entry} \) or the exit task \( t_{exit} \). In addition, \( w_{i} \) is the attribute of the vertex \( t_i \) indicating its computation workload, and \( data_{i,j} \) is the attribute of the edge \( e_{i,j} \) indicating the transfer amount of data from \( t_i \) to \( t_j \).

There are several microservices and no functional overlap between different types, and each type of microservices will deploy several service instances. Therefore, all microservice instances are represented by a set \( Ins = \{ I_{i,j} \} \), where \( I_{i,j} \) represents the \( j \)-th instance of the \( i \)-th type of microservices, as shown in Fig. 1, and its processing speed can be represented by \( speed_{i,j} \). When task \( t_c \) is assigned to instance \( I_{i,j} \), the execution time is defined as:

\[
ET(t_c, I_{i,j}) = w_{i}\text{/}speed_{i,j}
\]

(1)

In this paper, we assume all instances are connected by a high-performance network and there is no difference in transmission speed and no competition. We use the variable \( B \) to represent the transfer speed between instances, and the data transfer time from task \( t_i \) to task \( t_j \) is defined as:

\[
TT(e_{i,j}) = data_{i,j}\text{/}B
\]

(2)
**B. PROBLEM DEFINITION**

In this paper, we adopt the micro-batch strategy, in which requests entering the system in an algorithm period will be collected and then be processed as a unit. Let the set \( R = \{ r_1 | 1, \ldots, n \} \) represents the requests received by the system within an algorithm period, and there is a variable \( \text{deadline}_i \) for each request to indicate its deadline. As shown in Fig. 1, one or more requests will correspond to a certain CS described by a workflow, which is a one-to-many relation. For example, the requests \( r_1 \) and \( r_2 \) correspond to \( CS_1 \) and \( r_1 \) corresponds to \( CS_N \) and the two CSs are described respectively by two workflows. Here we define a set of tasks \( Task = \{ t_i \} \), including all tasks belonging to \( R \). The service selection needs to select an instance from the set \( Ins \) for each task in the set \( Task \), determining the execution order and the execution duration of tasks to get a service selection scheme \( M \), which is defined as follows:

\[
M = \{ m_j | m_j = \{ x_{i,j,k}, PST_i \} \} \tag{3}
\]

where \( m_j \) is the mapping from tasks to instances, each element \( x_{i,j,k} \) indicates the task \( t_i \) is assigned to the instance \( I_{j,k} \); and \( PST_i \) means the planned start time of the task \( t_i \). By the value of \( PST_i \), we can determine the execution order of each task, and the instance \( I_{j,k} \) will process the task \( t_i \) if \( x_{i,j,k} = 1 \). After the task \( t_i \) is assigned to the instance \( I_{j,k} \), the actual finish time \( AFT(t_i) \) can be determined. If all tasks belonging to the same request are scheduled, the makespan of this request can be calculated by:

\[
\text{makespan}_i = \max_{t_i \in r_i} \{AFT(t_i)\} \tag{4}
\]

According to the makespan of each request, the number of requests completed on time can be obtained by:

\[
\text{num}(M) = \sum_{i=1}^{n} z_i, \quad z_i = \begin{cases} 1, & \text{makespan}_i \leq \text{deadline}_i \\ 0, & \text{otherwise} \end{cases} \tag{5}
\]

Thus, the service selection problem of the microservice system can be defined as, finding a scheme \( M \) that maximizes the \( \text{num}(M) \).

\[
\max \text{num}(M) \tag{6}
\]

\[
s.t. \sum_{I_{j,k} \in \text{Ins}} x_{i,j,k} = 1, \quad \forall t_i \in Task \tag{7}
\]

\[
\max_{t_i \in \text{Task}} \frac{PFT_p}{PST_i}, \quad \forall t_i \in Task \tag{8}
\]

\[
\min_{I_{j,k} \in \text{Ins}} x_{i,j,k} \cdot x_{i,j,k} \cdot \gamma_{1,2} \geq 0, \quad \forall t_{i1}, t_{i2} \in Task \tag{9}
\]

\[
x_{i,j,k} \in [0, 1], \quad \forall t_i \in Task, \quad I_{j,k} \in \text{Ins} \tag{10}
\]

\[
PST_i \in [y | y > 0], \quad \forall t_i \in Task \tag{11}
\]

where

\[
PFT_i = PST_i + \sum_{I_{j,k} \in \text{Ins}} x_{i,j,k} \cdot ET(t_i, I_{j,k}) \tag{12}
\]

\[
\gamma_{1,2} = \begin{cases} \frac{PST_{t_2} - PFT_{t_1}}{PST_{t_1} - PFT_{t_2}} & \text{if PST_{t_1} < PST_{t_2}} \\
\frac{PST_{t_1} - PFT_{t_2}}{PST_{t_1} - PFT_{t_1}} & \text{if PST_{t_2} < PST_{t_1}} \end{cases} \tag{13}
\]

The \( x_{i,j,k} \) and \( PST_i \) are decision variables, which are (10) and (11). The \( PFT_i \) is the planned finish time of the task \( t_i \), which can be calculated by (12). The (7) means each task will be assigned to the only one instance, and the (8) forces each task to be processed only after its all predecessor tasks finished. The two constraints guarantee every task will be executed and ensure the dependencies between tasks.

In (9) and (13), if the task \( t_{i1} \) and \( t_{i2} \) are assigned to the same instance \( I_{j,k} \), that is the product of \( x_{i1,j,k} \) and \( x_{i2,j,k} \) is 1, the two tasks need to be executed one by one, and their execution durations must not overlap each other. For example, if the task \( t_{i1} \) begins to execute first (\( PST_{t_{i1}} < PST_{t_{i2}} \)), it must finish before the task \( t_{i2} \), so \( PST_{t_{i2}} - PFT_{t_{i1}} \) must be equal or greater than 0, as shown in (13). If there exists the task violating the condition, the left side of (9) will be less than 0.

**IV. SERVICE SELECTION ALGORITHM FOR MICROSERVICES**

In this section, we will introduce the Microservice Service Selection (MSS) algorithm proposed in this paper. The algorithm is based on the list scheduling which consists of two phases: the task ordering phase to determine the execution order of tasks and the processor selection phase to select a suitable processor [5], [7], [10]. Following its idea, we improve the traditional list scheduling by modifying or adding some strategies which aim at the problems of the deadline constraint and the service instances competition between CSs, to form a two-phase’s algorithm including the task ordering phase and the service selection phase:

1) Task ordering strategy, including task urgency calculation and idle time gaps search. The former improves the traditional priority calculation by making use of the deadline and the dynamic information at runtime, and the latter follows the search strategy in list scheduling and adjusts the task execution order in the service selection phase.

2) Service selection strategy, including service instance selection and abandoning strategy. The original \( EFT \) (Earliest Finish Time)-based selection method is modified to service instance selection which takes into consideration the deadline and the workload of instances. And the new abandoning strategy is added to solve the timeout tasks for global optimization.

As described in Section III-B, the algorithm needs to select service instances for multiple requests in an algorithm period, and each of them corresponds to a CS. But the list scheduling can’t process the multiple workflows of these CSs directly. So the merging strategy is used to merge their workflows into a single workflow before list scheduling is executed [13].

**A. DEADLINE DISTRIBUTION**

In this paper, we consider its latest finish time as the sub-deadline of a task. In [12], the latest finish time is defined as the latest time when the task must be completed so that its
workflow can complete on time, that is:

\[
LFT(t_{exit}) = \text{deadline} \\
LFT(t_i) = \min_{t_c \in \text{child}(t_i)} \{ LFT(t_c) - \text{min} ET(t_c) - TT(e_{i,c}) \}
\]  
(14)

The latest finish time of the exit task is defined as the deadline for the request. Then, traverse all tasks from the exit task to the entry task and calculate the latest finish time of each task by minimum execution time, \text{min} ET(t_c), on the fastest instance.

However, this formula can’t be used directly in this paper. On one hand, it’s impossible to ensure the task can be always executed on the fastest instance due to limited instances, so the choice of \text{min} ET(t_c) is unreasonable. On the other hand, it lacks the consideration of the potential queuing time caused by the instances competition. Therefore, we modify this formula aiming at the two problems: we firstly choose a value from three alternatives, which is the maximum, the minimum and the average speed, to calculate the \text{ET}(t_c). Then, a coefficient \text{Con} is added to \text{ET}(t_c) to express the degree of concurrency, which is related to two variables: the number of tasks that compete with this task for instances (\text{taskNum}(t_c)) and the number of microservice instances that can execute this task (\text{insNum}(t_c)). The higher the value of the former, the longer the task may wait in the queue. On the contrary, the more the number of instances is, the shorter the potential waiting time is. Based on the two variables, the value of \text{Con} can be one of the following four values:

1) 1  
2) \text{taskNum}(t_c)  
3) 1 / \text{insNum}(t_c)  
4) \text{taskNum}(t_c) / \text{insNum}(t_c)

The modified formula of the latest finish time is as follows:

\[
LFT(t_{exit}) = \text{deadline} \\
LFT(t_i) = \min_{t_c \in \text{child}(t_i)} \{ LFT(t_c) - \text{ET}(t_c) \cdot \text{Con} - TT(e_{i,c}) \}
\]  
(15)

The values of \text{ET}(t_c) and \text{Con} will be determined by the experiments in Section V.

**B. URGENCY CALCULATION**

The traditional task ordering is based on the upward rank [5] of each task, which only reflects the task dependency but ignores the deadline constraint. What’s more, the delay of the predecessor tasks will be accumulated to affect the start time of the successors. Even if the upward rank and the sub-deadline of two tasks are the same, the task with a later start time should be executed first. This dynamic information is available only at runtime. Therefore, we define the urgency of the task using the sub-deadline and the runtime information and update them in real-time as the basis for the task ordering. The formula is as follows:

\[
\text{urgency}(t_i) = \frac{LFT(t_i) - XFT(t_i)}{\text{hop}(t_i)}
\]  
(16)

where \text{hop}(t_i) is the number of tasks to be scheduled on the path from the current task to the exit task. Taking the CSN in Fig. 1 for example, \text{hop}(t_6) is 4 because the longest path from \text{t}_6 to the exit task \text{t}_9 is \{ \text{t}_7, \text{t}_2, \text{t}_8, \text{t}_9 \}, on which the number of tasks is 4. If there are several exit tasks or paths, the value takes the largest one. The more the number of such tasks, the higher the risk is that the execution of the following tasks is affected by various factors. \text{XFT}(t_i) is the expected finish time, which is as the following formulas show:

\[
\text{EST}(t_i, I_{j,k}) = \max \{ \text{avail}(I_{j,k}), \max_{t_p \in \text{parent}(t_i)} (\text{AFT}(t_p) + \text{TT}(e_{p,i})) \}
\]

where \text{AFT}(t_p) is the actual finish time of the task \text{t}_p. The \text{avail}(I_{j,k}) means the available time when the instance completes the assigned tasks and is ready to process the next task. The value of \text{EST}(t_i, I_{j,k}) takes the maximum of the \text{avail}(I_{j,k}) and the time at which the predecessor tasks are all completed and the data transmission is done. The minimum of all values of the earliest finish time \text{EFT}(t_i, I_{j,k}) on all instances is the expected finish time \text{XFT}(t_i)

Above all, the meaning of \text{urgency} is: the closer the expected finish time is to the sub-deadline of the task and the more the number of tasks to be executed later, the smaller the value of the urgency and the higher the degree of urgency is.

**C. SERVICE SELECTION**

In this phase, the algorithm will select the instance for tasks in order of urgency. In every service instance selection, the \text{EFT}(t_i, I_{j,k}) of instances will be calculated according to (17), as the QoS of instances. Based on the sub-deadline assigned to the task, determine whether the task can be completed on time, and then select the appropriate instance to execute it, or abandon the timeout task and its CS, recycling instances to execute other tasks, that is the abandoning strategy. The required formula is as follows:

\[
\text{Laxity}(t_i, I_{j,k}) = LFT(t_i) - \text{EFT}(t_i, I_{j,k})
\]  
(18)

When \text{Laxity}(t_i, I_{j,k}) is positive, it means task \text{t}_i can be completed on time on \text{I}_{j,k}, whereas when the value is negative, the task is considered to be a timeout task. Based on it, we calculate the values of \text{Laxity}(t_i, I_{j,k}), \text{EFT}(t_i, I_{j,k}) and \text{ET}(t_i, I_{j,k}) for all instances that can execute this task and select an instance by the following strategies:

1) All values of \text{Laxity}(t_i, I_{j,k}) are negative, indicating this task cannot be completed on time at any instance. It is considered that the CS including this task will time out, so we move all tasks of this CS into the abandoned queue, waiting for other CSs to complete the selection.
2) All values of $Laxity(t_i, I_{j,k})$ are negative but the task is the last one of the unassigned tasks of this type. That means there will be no task competing for instances. In this case, the instance with the smallest $EFT(t_i, I_{j,k})$ will be selected for this task.

3) Some values of $Laxity(t_i, I_{j,k})$ are positive, indicating the task can be completed on some service instances. In this case, we select the instance with the smallest $ET(t_i, I_{j,k})$ among instances with positive values. At the same time, a degree of load balancing is needed in order to avoid tasks being accumulating on the instance with the fastest speed. That is, if the instance with the smallest $ET(t_i, I_{j,k})$ is $I_{j,k1}$, but the $EST(t_i, I_{j,k1})$ in $I_{j,k1}$ is greater than the $EFT(t_i, I_{j,k2})$ on another instance $I_{j,k2}$, $I_{j,k2}$ will be selected as the execution instance of the task. When there are several instances meeting this condition, the one with the smallest $EFT(t_i, I_{j,k2})$ will be selected.

Furthermore, there may be some idle time gaps between the execution durations of two tasks on the same instance. Making full use of these gaps can improve the resource utilization and shorten the execution time of some CSs. To this end, we propose the idle time gap search, i.e., when calculating $EST(t_i, I_{j,k})$ a search will be done for gaps that meet the following requirements:

$$\max_{t_p \in \text{parents}(t_i)} \{\text{AFT}(t_p) + \text{TT}(e_{p,i}), \text{AFT}(t_{prep})\} + ET(t_i, I_{j,k}) \leq AST(t_{next})$$  \hspace{1cm} (19)

For all predecessor tasks of task $t_i$, the maximum of the sums of the actual finish time and the data transfer time indicates the ready time when $t_i$ is ready to be executed. In addition, each idle time gap exists between two assigned tasks, which are called $t_{prep}$ and $t_{next}$. The maximum of the ready time of $t_i$ and the $\text{AFT}(t_{prep})$ represents the earliest start time of $t_i$ after $t_{prep}$ finishes. And, the value of the earliest start time plus the execution time must not exceed the actual start time of the task $t_{next}$. If there exist such gaps that meet this condition, $\text{avail}(I_{j,k})$ in formula (17) will be equal to the minimum of $\text{AFT}(t_{prep})$ of these gaps.

Every time an instance is selected, the QoS provided by instances will be recalculated to update the urgency of the unassigned tasks. Since $\text{LFT}(t_i)$ of each task is fixed, what needs to calculate is only $\text{XFT}(t_i)$. Since the task ordering will change, it is not necessary to sort all tasks. What the task ordering needs to do is selecting the task with the smallest value of urgency as the next task to select a service instance. After all tasks are assigned or abandoned, the algorithm will select the instance for those abandoned tasks. For these tasks, the algorithm selects the instance with the smallest $\text{EFT}(t_i, I_{j,k})$ for them to optimize the makespan of the CSs including these tasks. Finally, the pseudo code of the algorithm is as shown in Algorithm 1.

D. AN EXAMPLE

To help understand how the algorithm works, we take a simple example where two workflows are scheduled in Fig. 2.

Algorithm 1 MSS Algorithm

```
Input: the set of requests $R$, the set of microservice instances $\text{Ins}$
Output: the scheme of service selection $M$
1: Initialize the selection queue $\text{SQ}$ and the abandoned queue $\text{AQ}$
2: merge workflows of all requests in $R$ into a single one, insert all tasks into the queue $\text{SQ}$
3: calculate the $\text{LFT}$, $\text{XFT}$ and $\text{urgency}$ for each task in $\text{SQ}$
4: WHILE $\text{SQ}$ is not empty DO
5: find task $t_i$ with the smallest $\text{urgency}$ whose predecessor tasks are all completed in $\text{SQ}$
6: FOR each instance $I_{j,k}$ that can execute task $t_i$ DO
7: calculate $\text{EFT}(t_i, I_{j,k}), \text{ET}(t_i, I_{j,k})$ and $\text{Laxity}(t_i, I_{j,k})$
8: END FOR
9: IF all values of $\text{Laxity}$ are negative THEN
10: IF task $t_i$ is the last one of this type THEN
11: $\text{selected} \leftarrow$ microservice instance $I_{j,k1}$ with the smallest $\text{EFT}(t_i, I_{j,k})$
12: ELSE
13: move all tasks related to $t_i$ from $\text{SQ}$ and $M$ into $\text{AQ}$
14: END IF
15: ELSE
16: $\text{selected} \leftarrow$ the microservice instance $I_{j,k1}$ with the smallest $\text{EFT}(t_i, I_{j,k})$
17: IF there exist instances $I_{j,k2}$ where $\text{EFT}(t_i, I_{j,k2}) < \text{EST}(t_i, I_{j,k1})$ THEN
18: $\text{selected} \leftarrow I_{j,k2}$ with the smallest $\text{EFT}(t_i, I_{j,k2})$
19: END IF
20: END IF
21: remove $t_i$ from $\text{SQ}$ and add $\{x_{t_i,j,k}, \text{PST}_t\}$ to $M$
22: update the values of $\text{XFT}$ and $\text{urgency}$ of tasks in $\text{SQ}$
23: END WHILE
24: WHILE $\text{AQ}$ is not empty DO
25: find task $t_i$ with the smallest $\text{urgency}$ and its predecessor tasks are all completed in $\text{AQ}$
26: FOR each instance $I_{j,k}$ that can execute task $t_i$ DO
27: calculate $\text{EST}(t_i, I_{j,k}), \text{EFT}(t_i, I_{j,k})$
28: END FOR
29: $\text{selected} \leftarrow$ the microservice instance $I_{j,k}$ with the smallest $\text{EFT}(t_i, I_{j,k})$
30: remove $t_i$ from $\text{AQ}$ and add $\{x_{t_i,j,k}, \text{PST}_t\}$ to $M$
31: END WHILE
```

There are 3 types of tasks, each of which is represented by one color, and the numbers beside tasks and edges denote the workload of each task and the data transfer amount between tasks, e.g., the computation workload of $t_1$ is 18. The assumed microservice instances and their speeds are shown in Table 1, and there is a functional match between tasks and instances,
e.g., the Thinca tasks, $t_4$ and $t_9$ can only be processed by $ms_{3,1}$. The meanings of the workload and the speed will be described in detail in Section V. The values of variables and the selected instance in each step are listed in Table 2. Finally, the deadlines of the two workflows are 175 and 155 (in seconds) respectively.

Firstly, the workflows are merged into a single one by adding the entry and exit tasks, whose computation workload and transfer amount are zeros. Then the sub-deadline of each task, $LFT$, is calculated, as shown in Table 2. In each step, the $XFT$ and urgency are updated and the task with the smallest urgency is scheduled. If the values of urgency of these two tasks are equal, like $t_2$ and $t_3$, the task is selected randomly.

In step 1, $t_5$ is selected and there are two alternative instances for it because the Laxity of the two instances are both positive. The instance for $t_5$ is selected randomly because the speeds of $ms_{3,1}$ and $ms_{3,2}$ are both 1.0. Step 2 and 3 are similar and tasks in two workflows are scheduled in turn.

In step 4, the alternative instance is only $ms_{2,1}$ because the Laxity of the other instance is negative. And in step 5, both instances for $t_2$ have the negative Laxity, thus there is no alternative and the $t_2$ is judged to have timed out, and other tasks belonging to the same workflow are moved into the abandoned queue, including $t_1$. These tasks will be scheduled after the other workload finishes and the detail are not shown.

### V. EVALUATION

In this section, we use some real workflow applications to conduct some experiments to verify the performance of the algorithm. In each experiment, the number of requests completed on time, represented by $num(M)$, the total delay time of all requests, denoted by $tdl(M)$, and the execution time will be measured.

#### A. EXPERIMENT SETTINGS

Because the real test environment is difficult to build, we use the opensource tool provided by [3] for simulation experiments which runs on a PC with i5 2.6GHz CPU, 8GB memory and JDK 1.8.0. The tool provides a general framework for modeling and simulation of workflow scheduling and allows us to customize the scheduling algorithm and the different workflow benchmarks.

Bharathi [14] provided several benchmark workflows, which are described by DAX files, for testing and evaluating workflow systems. The computation workload of a task is defined by the execution time (in seconds) of the task on a standard computing service. Since each task in the workflow has a different function and there is the execution logic described by the dependency of tasks, we can think of the workflow as a CS. Therefore, we select four types of workflows for experiments, that is LIGO, MONTAGE, CYBERSHAKE, GENOME. The workload of each task and the transfer amount of data are shown in detail in [14].

Because we adopt the micro-batch strategy, we assume that all requests entering the system in a scheduling period will be scheduled and completed. And every period is independent, i.e., the execution of algorithms in the next period won’t change the scheme of the previous period. What’s more, we assume the workload is stable, and the number of requests is fixed.

1) COMPARED ALGORITHMS

Two existing algorithms are implemented to compare the performance with our algorithm: CWSA(Cloud-based Workflow Scheduling Algorithm) [8] and LACO(L-Ant Colony Optimization) [3]. The former is designed to schedule several workflows submitted by multi-tenant. It calculates the schedule utilization to search the schedule gap and move tasks into the found gap, to maximize the weight, which denotes the makespan and the number of workflows within the deadline. The LACO is the meta-heuristic algorithm for deadline-constrained workflow scheduling. It is composed of the ant colony optimization (ACO) and the heuristic algorithm, where the ACO is used to determine the execution order of tasks and then the heuristic algorithm schedules each task to a certain instance. To apply the LACO to the multi-workflow scheduling, all requests are sorted in ascending order of their deadlines, and the workflow with the earliest deadline is scheduled by the LACO one by one, optimizing the makespan.
TABLE 2. The values of variables and the selected instance in each step.

| Step | $t_i$ | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ |
|------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1    | LFT(t) | 88.17 | 169.75| 169.75| 175   | 78.75 | 90.64 | 149.75| 152.75| 155   |
|      | XFT(t) | 0     | 0     | 0     | 175   | 0     | 0     | 0     | 0     | 0     |
|      | urgency(t) | 29.39 | 84.88 | 84.88 | 175   | 26.25 | 30.21 | 74.88 | 76.38 | 155   |
|      | alternatives | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] |
| 2    | XFT(t) | 0     | 0     | 0     | 175   | 0     | 18    | 0     | 0     | 0     |
|      | urgency(t) | 29.39 | 84.88 | 84.88 | 175   | 30.21 | 65.88 | 76.38 | 155   |
|      | alternatives | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] |
| 3    | XFT(t) | 36    | 36    | 0     | 175   | 30.21 | 65.88 | 67.38 | 155   |
|      | urgency(t) | 66.88 | 66.88 | 175   | 30.21 | 65.88 | 67.38 | 155   |
|      | alternatives | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] |
| 4    | XFT(t) | 36    | 36    | 0     | 175   | 18    | 18    | 0     | 0     |
|      | urgency(t) | 66.88 | 66.88 | 175   | 65.88 | 67.38 | 155   |
|      | alternatives | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] |
| 5    | XFT(t) | 36    | 36    | 0     | 175   | 18    | 0     | 0     |
|      | urgency(t) | 66.88 | 66.88 | 175   | 67.38 | 155   |
|      | alternatives | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] |
| 6    | XFT(t) | 18    | 0     | 0     |
|      | urgency(t) | 67.38 | 155   |
|      | alternatives | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] |
| 7    | XFT(t) | 0     | 0     |
|      | urgency(t) | 155   |
|      | alternatives | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] | [ms$_{1,i}$, ms$_{2,j}$] |
| Selected instance | ms$_{1,i}$ | ms$_{2,j}$ | ms$_{1,i}$ | ms$_{2,j}$ | ms$_{1,i}$ | ms$_{2,j}$ | ms$_{1,i}$ | ms$_{2,j}$ | ms$_{1,i}$ |

In addition, a Greedy algorithm is designed as a baseline for comparison, and then we add our task ordering and service selection strategies to it, to research the performance improvement brought by these strategies. This greedy algorithm selects services for multiple CSs at the principle of FCFS (first come, first served), and selects the service instance with the earliest finish time for each task like the processor selection strategy in HEFT.

2) METRICS
To evaluate the performance of algorithms, we define the following metrics:

The number of requests completed on time ($num(M)$). Because the instance resources are limited, it is impossible to make every request to satisfy its deadline. Thus, the primary objective is to make more requests complete before their deadlines.

Tardiness ($td(M)$). It represents the total delay time of all requests. If the request cannot be finished before its deadline, the resulting delay time will affect user experience. Thus, it would be better to reduce total delay time. The tardiness is the sum of the temporal difference of the makespan exceeding the deadline of each request, and the difference is 0 if the request finishes on time:

$$td(M) = \sum_{i=1}^{n} \max(\text{makespan}_i - \text{deadline}_i, 0) \quad (20)$$

Mean execution time. It is the execution time taken by each algorithm that generates the scheme $M$, which is used to measure its efficiency and time complexity.

3) PARAMETER SETTINGS
In order to set a reasonable deadline for each CS, we calculate the length of the critical path of every CS and multiply it by a deadline factor. The critical path can be regarded as the execution time of the CS without queuing time. We set a factor to control the strictness of the deadline. The larger the factor, the more relaxed the deadline constraint. The range of the factor is [1, 5].

We decide in advance the types of tasks that each type of instance can execute. In addition, in order to simulate the competition between different CSs, some types of tasks from different CSs will be selected randomly to share the same type of instance. Take MONTAGE as an example. It includes 9 types of tasks, so we set 9 types of microservice instances and make its mConcatFit task and ZipPSA task and ZipSeis task of CYBERSHAKE share the same type of service instances.

To decide the number of instances and the speed of each instance, we analyze the amount of the workload of tasks. For example, tasks such as mProjectPP and mDiffFit have a larger number and each task has a large amount of computation, so we increase the number of instances of these types. The setting of the speed refers to that in [3], which defines the speed as a multiple of the speed of standard service. For example, when the value of the speed is 2.0, the execution time of a certain task is half of the original value in benchmark workflows. The value of the speed is selected randomly from a closed interval [1, 5] at intervals of 0.5. Then, we adjust the number and the speed of instances to ensure that the processing capacity of instances is sufficient to complete all CSs on time when the factor is 5.
TABLE 3. The number of requests completed on time with different combinations of variables.

|                  | maximum speed | minimum speed | average speed | Total |
|------------------|---------------|---------------|---------------|-------|
| 1                | 355           | 343           | 357           | 1055  |
| *taskNum         | 328           | 341           | 339           | 1008  |
| 1 / insNum       | 359           | 359           | 360           | 1078  |
| *taskNum / insNum| 338           | 337           | 338           | 1013  |
| Total            | 1380          | 1380          | 1394          |       |

To decide the value of two variables in Formula (15), we use the setting above to conduct simulation experiments. The extra setting is:

1) The MSS is performed with different values of speed and \( Con \cdot ET(t_c) \) takes a value from the maximum speed, the minimum speed or the average speed, while \( Con \) is chosen from four values described in Section IV-A to form 12 groups.

2) In each group, the MSS is performed for each type of the benchmark workflow, where the value of the factor is selected from \([1, 5]\) at intervals of 0.2.

Count the number of requests completed on time under different combinations of \( ET(t_c) \) and \( Con \). The result is shown in Table 3.

The result shows, when \( Con \) takes \( 1 / insNum(t_c) \), the \( num(M) \) is 1078 being larger than 1055, 1008, 1013 of the other three alternatives, whereas when the speed is the average speed, the total number is 1394, which is greater than 1380, 1380 of the other two values. And in the combination of \( 1 / insNum(t_c) \) and the average speed the number of CSs completed on time is largest, which is 360.

So the final form of Formula (15) is:

\[
LFT(t_{exit}) = \text{deadline} \\
LFT(t_i) = \min_{t_c \in \text{children}(t_i)} \{LFT(t_c) - \text{avgET}(t_c)/\text{insNum} - TT(e_{i,c})\} 
\]

(21)

B. RESULTS

We make each type of CS be called several times so that the system needs to execute multiple requests in an algorithm cycle. With different combinations of these CSs and different proportions of request times, several experiments are performed, taking the \( num(M) \), the \( td(M) \) (in seconds) and the runtime as metrics. And then the performance of algorithms with different system workloads is evaluated by adjusting the size of workflow and the number of requests.

1) DIFFERENT COMBINATIONS OF CS

We combine the four CS in pairs, each of which is invoked 10 times and consists of 50 tasks, including the following five combinations:

1) MONTAGE and CYBERSHAKE, competing for 2 types of instances;
2) MONTAGE and LIGO, competing for 1 type of instances;
3) MONTAGE and GENOME, competing for 2 types of instances;
4) CYBERSHAKE and GENOME, competing for 1 type of instances;
5) CYBERSHAKE and LIGO, no competition

In each combination, the two CSs will compete for instances, that are combination 1-4. In addition, the different requests of the same CS will compete with each other. Thus, in combination 5, although there is no competition between CYBERSHAKE and LIGO, the competition still exists between CYBERSHAKE and GENOME, LIGO and LIGO.

The results are depicted in Fig. 3-Fig. 7, and the \( num(M) \) and normalized \( td(M) \) with the different combinations are assembled in Fig. 8. In addition, the execution time of the four algorithms is depicted in Fig. 9. We observe that the ratio of on-time completion of these algorithms is gradually increasing, and the delay time is gradually shortened as the factor increases. That’s because the deadline constraints are gradually loose and the algorithms have more alternatives to select instances.
Fig. 8 shows that CWSA is superior to the baseline algorithm Greedy and MSS outperforms Greedy and CWSA in all cases in terms of both $num(M)$ and $td(M)$. Compared with LACO, MSS has a better performance in $num(M)$, except combination 2. For combination 2, the $num(M)$ of MSS and LACO are 185 and 193 respectively and LACO is 4.1% better than MSS. However, MSS has a better overall performance as shown in Fig. 8(a). On average, in terms of $num(M)$, MSS is 34.81% better than Greedy, 20.94% than CWSA and 4.28% than LACO.

Fig. 8(b) shows the normalized delay time of algorithms and it indicates that MSS outperforms Greedy and CWSA but is inferior to LACO. The $td(M)$ of LACO is 22.28% averagely less than MSS. In terms of the two metrics, $num(M)$ and $td(M)$, MSS and LACO have their own advantages and disadvantages. In general, the performance of MSS is...
considered to be close or slightly inferior to LACO. However, the execution time of LACO is much longer than MSS. As Fig. 9 shows, the execution time of LACO is two orders of magnitude higher than that of MSS. In conclusion, MSS can get a near-optimal solution with low time complexity, and its performance is acceptable. On average, MSS achieves a 31.91% improvement over Greedy and 27.83% improvement over CWSA, respectively.

In addition, in the experiment shown in Fig. 5, when factor increases from 1.2 to 1.4, the \( \text{num}(M) \) increases but the \( \text{td}(M) \) also increases. This is because the overall delay time of the system is affected by two factors after using our service selection strategy. One is that more requests are completed on time to reduce the delay time. Another is that requests abandoned by the selection strategy are put off, increasing the delay time. The final delay time is the combined effect of these two factors. This is also the reason why MSS has a better performance than LACO in terms of \( \text{num}(M) \) but is inferior in \( \text{td}(M) \).

Since the Greedy algorithm is based on the strategy of FCFS, the service selection is performed independently for each request, and it lacks in the analysis of the QoS requirements of all requests. When a CS that requires a long execution time and deadline is firstly called by users, Greedy will do service selection for it. If a request that needs to be executed immediately to complete on time enters the system at this time, it will be overtime because it must wait for the former request. And CWSA sorts requests according to the deadline priority and searches the schedule gap to execute the request, trying to minimize their makespan and maximize the number of requests executed within the deadline by moving tasks among different gaps. By using these schedule gaps, CWSA is able to utilize instances more efficiently, and thus it presents a better performance over Greedy, especially in combination 2 and 5.
Although CWSA has the strategy to process multiple requests, it lacks a task ordering strategy. In CWSA, tasks are scheduled one after one by the depth-first search, whose efficiency is inferior to the strategy with the heuristic information. The meta-heuristic algorithm LACO shows a better performance than the two algorithms above. By sorting all requests in ascending order of deadline, the request with the smallest deadline is scheduled firstly, so that the system with limited resources can process more requests. And the meta-heuristic can get a better solution with a shorter makespan thanks to their global searching ability so that the num\(\{M\}\) is larger than CWSA. Because its original objective is minimizing the makespan of each request, LACO can reduce the overall delay time (td\(\{M\}\)), but it may result in some requests are unable to meet their deadlines. Thus, LACO can obtain the smallest td\(\{M\}\) but is inferior to MSS in terms of num\(\{M\}\). However, CWSA's execution time is much longer than others because of the characteristic of meta-heuristic [27].

Unlike other algorithms, MSS analyzes the heuristic information of all tasks of all requests by the merging strategy and the urgency calculating, finding the task needing to be executed immediately and adjusting their execution order. Furthermore, the abandoning strategy is used to remove the timeout CSs, the instances are freed to execute other CSs, making more CSs finish on time.

In order to analyze the impact of the strategies used in MSS, we add the strategies to Greedy one by one, including the task ordering strategy and the service selection strategy. We continue to use the above settings, adding up the num\(\{M\}\) and the td\(\{M\}\) of each experiment and Table 4 is obtained.

2) DIFFERENT RATIO OF NUMBER OF REQUESTS

In the previous experiments, each type of CS is called at the same frequency, which is 10 times in an algorithm period. To verify the performance of MSS in the case that each CS is called at different frequencies, we adjust the ratio of request times of MONTAGE and GENOME. What’s more, the deadlines and execution time of the two composite services are quite different, making easy to analyze the preference of strategies for CSs. The ratio of the number of requests is shown in Table 5.

Because for each type of composite service, the capacity provided by instances can only meet the requirement of

### Table 4. The impact of different strategies on performance.

| No | Strategies                  | Combination 1 | Combination 2 | Combination 3 | Combination 4 | Combination 5 |
|----|------------------------------|---------------|---------------|---------------|---------------|---------------|
|    | num\(\{M\}\) | td\(\{M\}\) (s) | num\(\{M\}\) | td\(\{M\}\) (s) | num\(\{M\}\) | td\(\{M\}\) (s) | num\(\{M\}\) | td\(\{M\}\) (s) |
| 1  | Greedy                       | 163           | 14299         | 136           | 129944        | 138           | 2650825       | 146           | 2676411        | 158           | 177060        |
| 2  | Greedy + idle gaps search    | 170           | 106779        | 168           | 81748         | 140           | 2601771       | 150           | 2661781        | 180           | 140039        |
| 3  | Greedy-S                     | 172           | 192239        | 125           | 145547        | 157           | 5674805       | 168           | 5088425        | 143           | 196569        |
| 4  | Greedy-S + idle gaps search | 182           | 102977        | 180           | 87129         | 159           | 2940685       | 177           | 2994887        | 197           | 134989        |
| 5  | MSS                          | 187           | 95272         | 185           | 77650         | 207           | 1778384       | 218           | 1838713        | 200           | 126869        |
TABLE 5. The ratio and the number of requests of MONTAGE and GENOME.

| No | Ratio (MONTAGE/GENOME) | Number of requests (MONTAGE/GENOME) |
|----|------------------------|-------------------------------------|
| 1  | 3:2                    | 15/10                               |
| 2  | 1:2                    | 5/10                                |
| 3  | 2:1                    | 10/5                                |
| 4  | 2:3                    | 10/15                               |

FIGURE 10. Performance comparison with the different ratio. (a) The number of requests completed on time. (b) Tardiness.

completing 10 requests of a CS on time when factor = 5, we set two numbers for each composite service:
1) the capacity is enough, and the number of requests is 5;
2) the capacity is insufficient, and the number is 15.

By the two different numbers of requests, we can observe the performance of algorithms under different workloads.

The num(M) and td(M) are assembled in Fig. 10 and the curves do not be shown in detail. The trend of the four experiments is similar to the combination 3 in the previous experiments where the ratio of the number of requests is 1:1. MSS and LACO have a better performance than the other two algorithms, and they have their own advantages and disadvantages, as discussed in the previous experiment.

Calculate the improvement rate of MSS against other algorithms by adding up the metrics, and obtain the following Table 6. As we can see from Table 6, the performance of MSS is about 50% higher than that of Greedy and about 40% compared with CWSA in the first metric. However, the improvement drops to 24.43% and 19.85% when the ratio is 10/5, as shown in Fig. 10, it even becomes negative compared with LACO. This is because the number of calls to GENOME is 5, which is much less than the capacity of instances, and when the deadline factor is loose, Greedy can satisfy all requests of

TABLE 6. The improvement of MSS against other algorithms.

| Number of requests | Greedy  | CWSA   | LACO  |
|--------------------|---------|--------|-------|
| 15 / 10            | 51.82%  | 44.44% | 8.90% |
| 5 / 10             | 47.58%  | 42.97% | 7.02% |
| 10 / 10            | 50.72%  | 44.44% | 7.22% |
| 10 / 5             | 24.43%  | 19.85% | -1.81%|
| 10 / 15            | 60.14%  | 53.47% | 18.18%|
| 15 / 10            | 33.22%  | 28.78% | -22.00%|
| 5 / 10             | 32.02%  | 29.23% | -23.08%|
| 10 / 10            | 33.02%  | 29.40% | -22.39%|
| 10 / 5             | 41.19%  | 32.94% | -21.98%|
| 10 / 15            | 25.18%  | 22.68% | -18.03%|

FIGURE 11. Performance comparison with different numbers of requests. (a) The number of requests completed on time. (b) Tardiness. (c) The execution time of algorithms.
3) DIFFERENT WORKLOADS

To evaluate the performance of algorithms with different system load, we repeat experiments of combination 4 and adjust the workload by the following ways: 1) the size of workflow is fixed at 50, the number of requests increases from 10 to 50; 2) the number of requests is fixed at 10, and the size of workflow, which means the number of tasks in each workflow, increases from 50 to 400. The results are shown in Fig. 11 and Fig. 12, respectively.

As the system workload increases, the performance comparisons among algorithms in Fig. 11 and Fig. 12 present similar results:

1) CWSA is superior to the baseline Greedy in both metrics;

2) LACO and MSS have better performance over CWSA and Greedy;

3) The \( \text{num}(M) \) of MSS is higher than that of LACO but is inferior to LACO in terms of \( \text{td}(M) \).

Fig. 11(c) and Fig. 12(c) show that the runtime of LACO is much longer than other algorithms. When the number of requests is larger than 30 and the size of workflow is over 100, LACO requires over 10 seconds to schedule the requests. In a real-time system, the overhead of scheduling algorithm has a great impact on system performance. MSS can obtain the near-optimal solution with less time and low time complexity, which proves our proposed strategies to be efficient.

VI. CONCLUSION

Aiming at the service selection problem in the microservice system, this paper proposes a service selection algorithm MSS based on the list scheduling and introduces the strategies in workflow scheduling to solve the competition of multiple CSs. On one hand, before selecting service, we calculate the urgency calculation of tasks to arrange the execution order for tasks in CSs. On the other hand, when selecting service instances for each task, estimate the QoS provided by instances and abandon the timeout tasks and their CS based on the sub-deadlines of tasks. At the same time, the algorithm continuously updates the urgency of tasks and searches the idle time gaps to adjust the execution order of tasks. Through experiments on the benchmark workflows, we found that MSS is better than the comparison algorithms in terms of the on-time completion rate and the delay time to verify the effectiveness of selection strategies.

This paper focuses on the service selection for microservices with deadline-constrained on the limit resources, and it only considers the QoS related to the execution time. More QoS factors will be in consideration for future research. Moreover, when the microservices are deployed in the cloud environment, the algorithm needs to adapt to the characteristics of the cloud environment, such as elasticity, allowing the microservices to scale on-demand automatically, which will be one of the future research directions.

REFERENCES

[1] C. Esposito, A. Castiglione, and K.-K.-R. Choo, “Challenges in delivering software in the cloud as microservices,” IEEE Cloud Comput., vol. 3, no. 5, pp. 10–14, Sep. 2016.

[2] G. Mazlami, J. Cito, and P. Leitner, “Extraction of microservices from monolithic software architectures,” in Proc. IEEE Int. Conf. Web Services (ICWS), Jun. 2017, pp. 524–531.

[3] Q. Wu, F. Ishikawa, Q. Zhu, Y. Xia, and J. Wen, “Deadline-constrained cost optimization approaches for workflow scheduling in clouds,” IEEE Trans. Parallel Distrib. Syst., vol. 28, no. 12, pp. 3401–3412, Dec. 2017.

[4] Y. Niu, F. Liu, and Z. Li, “Load balancing across microservices,” in Proc. IEEE Conf. Comput. Commun. (INFOCOM), Apr. 2018, pp. 198–206.

[5] H. Topcuoglu, S. Hariri, and M.-Y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” IEEE Trans. Parallel Distrib. Syst., vol. 13, no. 3, pp. 260–274, Mar. 2002.

[6] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, “DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm,” in Proc. 18th Euromicro Conf. Parallel, Distrib. Netw.-Based Process., Feb. 2010, pp. 27–34.
[7] H. Arabnejad and J. G. Barbosa, “List scheduling algorithm for heterogeneous systems by an optimistic cost table,” IEEE Trans. Parallel Distrib. Syst., vol. 25, no. 3, pp. 682–694, Mar. 2014.

[8] B. P. Kimal and M. Maier, “Workflow scheduling in multi-tenant cloud computing environments,” IEEE Trans. Parallel Distrib. Syst., vol. 28, no. 1, pp. 290–304, Jan. 2017.

[9] M. A. Rodriguez and R. Buyya, “Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds,” IEEE Trans. Cloud Comput., vol. 2, no. 2, pp. 222–235, Apr. 2014.

[10] Y. Xu, K. Li, L. He, L. Zhang, and K. Li, “A hybrid chemical reaction optimization scheme for task scheduling on heterogeneous computing systems,” IEEE Trans. Parallel Distrib. Syst., vol. 26, no. 12, pp. 3208–3222, Dec. 2015.

[11] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema, “Cost-driven scheduling of grid workflows using partial critical paths,” IEEE Trans. Parallel Distrib. Syst., vol. 23, no. 8, pp. 1400–1414, Aug. 2012.

[12] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema, “Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds,” Future Generation Comput. Syst., vol. 29, no. 1, pp. 158–169, Jan. 2013.

[13] L. F. Bittencourt and E. R. M. Madeira, “Towards the scheduling of multiple workflows on computational grids,” J. Grid Comput., vol. 8, no. 3, pp. 419–441, Sep. 2010.

[14] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, “Characterization of scientific workflows,” in Proc. 3rd Workshop Workflows Support Large-Scale Sci., Nov. 2008, pp. 1–10.

[15] L. Zeng, B. Benattallah, A. H. H. Ng, M. Dumas, J. Kalagnanam, and H. Chang, “QoS-aware middleware for Web services composition,” IEEE Trans. Softw. Eng., vol. 30, no. 5, pp. 311–327, May 2004.

[16] Y. Liu, A. H. Ng, and L. Z. Zeng, “QoS computation and policing in dynamic web service selection,” in Proc. 13th Int. World Wide Web Conf. Alternate track papers posters (WWW Alt), 2004, pp. 66–73.

[17] Z. Ding, J. Liu, Y. Sun, C. Jiang, and M. Zhou, “A transaction and QoS-aware service selection approach based on genetic algorithm,” IEEE Trans. Syst., Man, Cybern., Syst., vol. 45, no. 7, pp. 1035–1046, Jul. 2015.

[18] S.-Y. Hwang, C.-C. Hsu, and C.-H. Lee, “Service selection for Web services with probabilistic QoS,” IEEE Trans. Serv. Comput., vol. 8, no. 3, pp. 467–480, May 2015.

[19] H. Zhang, N. Yang, Z. Xu, B. Tang, and H. Ma, “Microservice based video cloud platform with performance-aware service path selection,” in Proc. IEEE Int. Conf. Web Services (ICWS), Jul. 2018, pp. 306–309.

[20] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, “Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds,” Future Gener. Comput. Syst., vol. 48, pp. 1–18, Jul. 2015.

[21] K. Bochenina, N. Butakov, and A. Boukhansovsky, “Static scheduling of multiple workflows with soft deadlines in non-dedicated heterogeneous environments,” Future Gener. Comput. Syst., vol. 55, pp. 51–61, Feb. 2016.

[22] A. Klein, F. Ishikawa, and S. Honiden, “Towards network-aware service composition in the cloud,” in Proc. 21st Int. Conf. World Wide Web, 2012, pp. 959–968.

[23] S. Deng, H. Wu, D. Hu, and J. Leon Zhao, “Service selection for composition with QoS correlations,” IEEE Trans. Serv. Comput., vol. 9, no. 2, pp. 291–303, Mar. 2016.

[24] Q. Wu, M. Zhou, Q. Zhu, and Y. Xia, “VCG auction-based dynamic pricing for multigranularity service composition,” IEEE Trans. Autom. Sci. Eng., vol. 15, no. 2, pp. 796–805, Apr. 2018.

[25] M. H. Ghahramani, M. Zhou, and C. T. Hon, “Toward cloud computing QoS architecture: Analysis of cloud systems and cloud services,” IEEE/CAN J. Autom. Sinica, vol. 4, no. 1, pp. 6–18, Jan. 2017.

[26] Y. Du, J. Gai, and M. Zhou, “A Web service substitution method based on service cluster nets,” Enterprise Inf. Syst., vol. 11, no. 10, pp. 1535–1551, Nov. 2017.

[27] Y. Wang, H. Liu, W. Zheng, Y. Xia, Y. Li, P. Chen, K. Guo, and H. Xie, “Multi-objective workflow scheduling with deep-Q-network-based multi-agent reinforcement learning,” IEEE Access, vol. 7, pp. 39974–39982, 2019.