Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 6663199, 11 pages
https://doi.org/10.1155/2021/6663199

Research Article

A Smart Semipartitioned Real-Time Scheduling Strategy for Mixed-Criticality Systems in 6G-Based Edge Computing

Wenle Wang,[1] **Chengying Mao**,[2] **Shuai Zhao**,[3] **Yuanlong Cao**,[1] **Yugen Yi**,[1] **Shaolong Chen**,[1] **and Qinghua Liu**[1]

[1]*School of Software, Jiangxi Normal University, Nanchang, 330022 Jiangxi, China*
[2]*School of Software and IoT Engineering, Jiangxi University of Finance and Economics, Nanchang, 330013 Jiangxi, China*
[3]*Department of Computer Science, University of York, YO10 5GH, UK*

Correspondence should be addressed to Chengying Mao; maochy@yeah.net

Received 30 October 2020; Revised 24 February 2021; Accepted 6 March 2021; Published 23 March 2021

Academic Editor: Longzhe Han

With the rapid growth of 6G communication and smart sensor technology, the Internet of Things (IoT) has attracted much attention now. In the 6G-based IoT applications on the multiprocessor platform, the partitioned scheduling has been widely applied. However, these partitioned scheduling approaches could cause system resource waste and uneven workload among processors. In this paper, a smart semipartitioned scheduling strategy (SSPS) was proposed for mixed-criticality systems (MCS) in 6G-based edge computing. Besides tasks' acceptance rate and weighted schedulability, QoS is considered in SSPS to improve the service quality of the system. The SSPS allocates tasks into each processor, and some tasks can migrate to other processors as soon as possible. By comparing with the several existing algorithms, the experimental results show that the SSPS achieves the best in the schedulability and QoS of the system.

1. Introduction

Nowadays, with the 6G wireless communication networks and various smart sensors widely applied, the IoT applications grow rapidly in a wide range of areas, including industrial robot, driverless car, and edge computing [1–5]. Especially, edge computing, a new application paradigm, is growing popular with 6G technology's development. For these systems in 6G-based edge computing, various sensors and mobile devices with different importance or criticality levels are integrated into a single computation platform for less space and energy. Criticality is designed for assurance needed against system failure [6]. Generally, criticality can be divided into several levels, such as low and high criticality. For example, in automotive systems, the tasks brought by steering and braking sensor are safety-related high, while the tasks of multimedia players, used for infotainment, are low-criticality (*LO-criticality*) tasks. These systems that have components with more than one distinct criticality level are mixed-criticality systems (MCS) [7], which are the special kind of 6G-based application with multiple criticality. The task scheduling is a fundamental issue in MCS, to reconcile the conflicting requirements for resource usage. The proper task is scheduled so that all high-criticality (*HI-criticality*) tasks' execution is guaranteed which is a major challenge of MCS. Because the reliability verification and the mixed criticality exist simultaneously in MCS, the traditional real-time scheduling algorithms cannot be directly adopted [8, 9].

With the prompt of computing requirements, the platforms of MCS are migrating from a single processor to multiprocessor hardware. MCS scheduling on multiprocessors can be mainly divided into global scheduling [10–12] and partitioned scheduling [13–17]. Fully global scheduling, because of task migration globally, has an overhead of the context switching and associated caches, while purely partitioned scheduling, in which some processors are too busy and others are too idle because of forbidden migration, causes waste of system resources [18]. Following this, the researchers mixed the above two scheduling methods and proposed the semipartitioned scheduling strategy [19–21].

The semipartitioned algorithms apply two-phase allocations for the different system criticality modes. During a phase of criticality mode update, the executing low-criticality (*LO-criticality*) tasks (jobs) will be aborted and new ones can be executed on a different processor, and thus, these jobs' deadlines are met to achieve better schedulability of system.

However, in most existing MCS semipartitioned scheduling algorithms, when the system criticality mode switches into *HI-criticality* from *LO-criticality*, the *LO-criticality* tasks are directly discarded to ensure *HI-criticality* tasks' completion [22–24], which seem too negative. Firstly, *LO-criticality* levels are not noncritical, and dropping the executing *LO-criticality* tasks may damage the system's acceptance rate. During the scheduling process, the processors could be so idle that they can be assigned to perform *LO-criticality* tasks and thereby improving system utilization and task acceptance [25, 26].

Furthermore, acceptance rate and utilization rate are the main schedulability concerning parameters in MCS, to ensure the *HI-criticality* tasks' completion. However, for tasks with the identical criticality, they can have different influences to MCS in actual applications, where some tasks are more significant or have higher quality of service (QoS) to a certain extent. To describe the QoS property of task, a notion, for example, value [27, 28], is usually used. The higher the value, the better the quality brought by task. And calculative value brought by finished tasks is recorded as TV to respect the whole tasks' QoS under scheduling algorithm [29, 30].

1.1. Organization. The paper's structure is listed as follows: the related work is described in Section 2. Section 3 describes the paper's overall framework. Section 4 defines the proposed MCS model and notation in detail. We analyze the schedulability of tasks in MCS in Section 5. Section 6 designs the task priority assignment. The detail of the proposed scheduling algorithm SSPS is introduced in Section 7. The simulation experiment setup and results are presented in Section 8. Finally, in Section 9, we summarize the conclusion and future work.

2. Related Work

In recent years, with the 6G network's development, related IoT applications are widely applied and researched [1–3]. Especially, 6G-based edge computing is growing popular with 6G wireless communication developing rapidly [4, 5]. And the scheduling issue of these applications on multiprocessors, such as systems with multiple criticalities, has become prominent [6–12]. We review the related work as follows.

The review of the literature [13–17] shows that the partitioned scheduling approach can achieve better schedulability than the global scheduling approach. By partitioned scheduling method, the task sets are firstly allocated to each processor, and then, they are executed according to the single-processor scheduling algorithm. The optimal partitioning of task sets on multiprocessors is a NP-hard problem, and the researchers mainly use heuristic partitioned algorithms to obtain suboptimal solutions. For the MCS on the identical multiprocessor platform, a fixed partitioned scheduling algorithm was firstly proposed in [14], and the impact of different task set sorting as well as a heuristic division on the system performance has been investigated. It showed that decreasing criticality (DC) can gain better schedulability than decreasing utilization (DU). In implicit-deadline sporadic MCS, a partitioned scheduling algorithm MC-PARTITION based on DC was proposed, which can get a better speedup bound. Since the task criticality level may change, the tasks are fixed via the Best-Fit Decreasing (BFD) of continuous criticality and utilization and improved resource utilization [15]. However, the pure partitioned scheduling algorithm may reduce the utilization of the entire system because of the migration forbidden between processors [16, 17].

These above reasons lead to the emergence of a semipartitioned scheduling strategy. In this scheduling, most tasks are assigned to the fixed processor and some tasks can be scheduled to different processors globally [18–20]. And for the MCS, a series of semipartitioned scheduling algorithms have been proposed. Santy et al. designed a heuristic scheduling strategy, combining reserved, semipartitioned, and periodic conversion, which reduces the migration overhead and obtain better performance [21].

In the original Vestal model, *LO-criticality* jobs sometimes are treated the same as noncritical jobs that will be not guaranteed in *HI-criticality* system mode, which ensures the completion requirements of *HI-criticality* tasks. Nevertheless, from the engineering perspective, *LO-criticality* task is not an NO-criticality task; it cannot be dropped easily [22–26]. Su and Zhu firstly focus on the *LO-criticality* tasks dropped in mixed critical scheduling and discuss the feasibility of restarting *LO-criticality* tasks from a multimodal perspective [22]. Burns and Baruah constructed an elastic mixed critical task model, which enables more frequent execution of *LO-criticality* tasks set through elastic processing [23]. And Baruah et al. [24] introduced an additional less pessimistic WCET for *LO-criticality* jobs to guarantee service regardless of the executions of *HI-criticality* jobs. The works in [25] follow the MC-Fluid framework to address the corresponding scheduler to handle *LO-criticality* service, having a good speedup factor.

Some researchers agree that real-time task has importance or quality, which should be treated as a factor to improve the quality of service (QoS) of system or application [27, 28]. In these papers above, a notion, namely, value, is given to respect the quality of a task, as a basis of the scheduling algorithm. Moreover, the value density (value of a time unit) and urgency of a task are considered comprehensively into dynamic scheduling algorithm and improved the real-time application performance [29, 30].

3. Overall Framework

We consider the scheduling on mixed-criticality systems (MCS) under a multiprocessor platform, in 6G-based edge computing environment. Firstly, the schedulability analysis based on response time is used to obtain schedulable tasks. Then, these tasks are sorted by priority assigned by criticality, value, and deadline. The tasks are divided to processor by first fit (FF) in the priority of descending order. The smart semipartitioned scheduling strategy (SSPS) is proposed, in

which some tasks can be migrated to other processors as needed. During the scheduling, the slack time collection is working to execute more task's job. The overall framework of the SSPS is shown in Figure 1.

3.1. Our Contributions. For the mixed-criticality systems (MCS) in 6G-based edge computing of homogeneous multiprocessors, the timing and service quality of the system tasks are taken into consideration, and a smart semipartitioned scheduling strategy (SSPS) is proposed in the paper. Besides, when the system mode switches from *LO-criticality* to *HI-criticality*, a mechanism that facilitates *LO-criticality* tasks (jobs) is designed in SSPS, to improve both the schedulability and the QoS.

4. System Model and Notation

4.1. System Model and Notation. Here, a mixed-criticality system (MCS) $S = (T, P)$ is defined as below, a task set T comprised of n independent and periodic tasks $\tau_1, \tau_2, \cdots, \tau_n$ and n processor set P with m identical processors $p_1, p_2, \cdots, p_m$.

Meanwhile, the dual MCS model is adopted in this paper, which runs in either a *HI-criticality* mode or a *LO-criticality* mode.

Definition 1. MCS tasks. The task of MC model can be characterized by a 5-tuple of parameters: $\tau_i = (\zeta_i, C_i, D_i, T_i, V_i)$, where

(1) $\zeta_i \in \{LO, HI\}$ denotes the criticality of task τ_i, where $LO < HI$. A task with *HI-criticality* is subject to be certified, whereas a *LO-criticality* task does not need to be certified

(2) $C_i(l)$ denotes the task τ_i's worst-case execution time (WCET) in criticality mode l, where $l \in \{LO, HI\}$. $C_i(HI)$ and $C_i(LO)$ denote the WCET of task τ_i at *HI-criticality* mode and *LO-criticality* mode, respectively. It meets the constraint $C_i(LO) < C_i(HI)$

(3) $D_i \in R^+$ is the relative deadline of task τ_i

(4) $T_i \in R^+$ is the period of task τ_i

(5) $V_i(l)$ specifies the value of task τ_i in criticality mode l, where $l \in \{LO, HI\}$. $V_i(LO)$ and $V_i(HI)$ respect the value of task τ_i at *LO-criticality* mode and *HI-criticality* mode, respectively, and it meets $V_i(LO) < V_i(HI)$

Each task τ_i in MCS can give rise to potentially infinite sequence of jobs.

Definition 2. MCS jobs. Each job J_i^j released by task τ_i can be described by a 4-tuple of parameters: $J_i^j = (a_i^j, e_i^j, d_i^j, f_i^j)$, where

(1) $a_i^j \in R^+$ is the release time of job J_i^j

(2) $e_i^j \in R^+$ is the estimated execution time of J_i^j, with the constraint $e_i^j \le C_i$

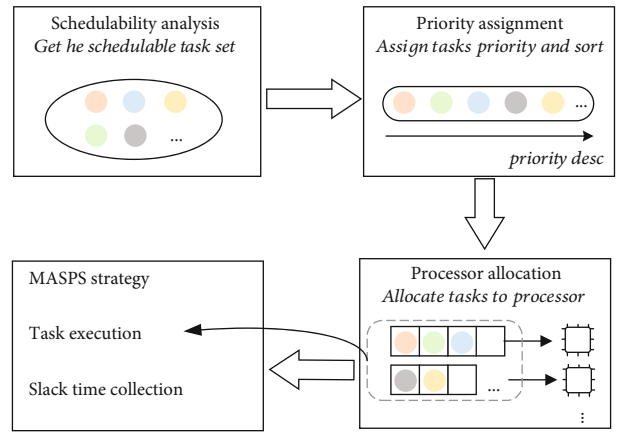

Figure 1: The overall framework of SSPS.

(3) $d_i^j \in R^+$ denotes the absolute deadline of J_i^j, satisfying $d_i^j = a_i^j + D_i$

(4) $f_i^j \in R^+$ denotes the finish time of job J_i^j. If J_i^j succeed, it should be the constraint $f_i^j \le d_i^j$

The system starts in the *LO-criticality* mode and remains in this mode as long as all jobs finished their execution.

If any job does not complete its execution within its *LO-criticality* execution time $C_i(LO)$, the system criticality mode will arise and *HI-criticality* tasks are executed with $C_i(HI)$.

4.2. Assumptions of the Model. In the MCS model, the *LO-criticality* does not mean noncriticality, and these tasks should be executed to the extent possible.

Assumption 3. If MCS switches into *HI-criticality* mode from *LO-criticality* mode, some of the *LO-criticality* tasks (and jobs) will be not dropped directly and allowed to be scheduled later.

Assumption 4. Tasks are independent of each other; they only share a processor, but not any other resource, such as bandwidth or memory.

5. Schedulability Analysis

In this section, we will investigate the schedulability by analyzing the response time of the job.

For job J_i^j, released by task τ_i, its response time R_i^j is denoted by $R_i^j = f_i^j - a_i^j$. Task τ_i's response time is denoted as R_i, which equals to the value of the maximum response time of all jobs released by τ_i.

We test the schedulability of the job J_i^j by comparing the response time R_i^j with the task τ_i's deadline D_i. If $R_i^j \le D_i$, J_i^j can be scheduled; otherwise, it cannot be scheduled.

The job J_i^j's response time includes two parts: its estimated execution time e_i^j and interference time I_i caused by higher priority tasks.

When a task τ_i is allocated to a processor, assuming a job J_i^j, released by τ_i at a_i^j, J_i^j waits other higher priority jobs' completion until b_i^j, its finish time is f_i^j, and its deadline is d_i^j. The execution constraint is illustrated by Figure 2.

Suppose t_{raise} is the time when the system mode is raised from *LO-criticality* to *HI-criticality*. When $t_{\text{raise}} \in [a_i^j, f_i^j)$, discussion about R_i^j is as follows:

(1) If $t_{\text{raise}} \in [a_i^j, b_i^j)$, it means that the system has been raised to *HI-criticality* mode before the job J_i^j starts

(a) The system initials at *LO-criticality* mode during the interval $[a_i^j, t_{\text{raise}})$. And the job J_i^j executes in *LO-criticality* mode; the interference time can be calculated as

$$I_{i1} = \sum_{\tau_i \in \text{hp}_{(i)}} \left\lceil \frac{t_{\text{raise}}}{T_j} \right\rceil \times C_j(LO), \quad (1)$$

where $\text{hp}_{(i)}$ indicates the tasks with higher priority than task τ_i.

(b) In the interval $[t_{\text{raise}}, b_i^j]$, the system is raised to *HI-criticality* mode. Then, the job J_i^j executes at *HI-criticality* mode; the interference time is defined as

$$I_{i2} = \sum_{\tau_i \in \left(\text{hp}_{(i)} \cup \text{hc}_{(i)}\right)} \left\lceil \frac{f_i^j - t_{\text{raise}}}{T_j} \right\rceil \times C_j(HI), \quad (2)$$

where $\text{hc}_{(i)}$ indicates the tasks with higher criticality than task τ_i.

(2) When $t_{\text{raise}} \in [b_i^j, f_i^j)$, it means that the system critical mode is improved during the execution of the job J_i^j

(a) In the interval $[b_i^j, t_{\text{raise}})$, the job J_i^j executes in *LO-criticality* mode; the interference time equals to I_{i1} of Equation (1)

(b) In the interval $[t_{\text{raise}}, b_i^j)$, the job J_i^j executes in *HI-criticality* mode; the interference time equals to I_{i2} of Equation (2)

In summary, the response time of J_i^j can be expressed as

$$R_i^j = e_i^j + I_{i1} + I_{i2}. \quad (3)$$

Based on Equation (3), the task τ_i's response time R_i, which can be determined by the jobs released by τ_i, selects the maximum response time of its job and satisfies $R_i = \max_{J_i^j} R_i^j$.

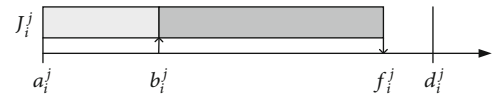

Figure 2: Execution diagram of J_i^j.

According to the discussion above, the pseudocode of the schedulability analysis algorithm can be described as Algorithm 1. In this algorithm, the inputs include two pieces of information: the undivided task set T and unallocated processors P. The output is the partitioned task queue of the processors. At first, each processor is not allocated any task (lines 1–3). Then, sort the tasks T according to its priority in order (line 4). After this, the algorithm allocates the tasks to processors (lines 5-15). At last, return the partitioned result (line 16). For each task τ_i in the queue (line 5, the outer for), starting from the first, the algorithm tries to allocate τ_i to a processor p_j to execute (line 6, the inner for). If τ_i can finish, insert it to the processor p_j's ready queue and allocate it to the next task (lines 7-10).

In Algorithm 1, it contains two-layer loops, where the outer-layer loop (lines 1, 5) can be evaluated in constant time $O(n)$ and the inner-layer loop (line 7)'s complexity is also constant level $O(m)$. The step of calculate R_i in line 6, between the outer-layer loop (line 5) and the inner-layer loop (line7), in which complexity is $O(n)$. Consequently, Algorithm 1's run time complexity is $O(n * m)$.

6. Priority Assignment of Task

In general, the priority of a task is the basis of schedule. This section mainly considers the criticality level and the value of task and proposes the priority assignment strategy.

6.1. Criticality and Value. According to Definition 1, *HI-criticality* task τ_i's value V_i is related to the criticality ζ_i, satisfying $V_i(HI) > V_i(LO)$.

In the existing strategies, the *LO-criticality* level tasks will be dropped out when the system criticality mode upgrades. Total value (TV) of the system can be expressed as $\text{TV} = \sum_{\tau_i \in T} V_i(LO) + \sum_{\tau_i \in T_{HI}} V_i(HI)$, where $T_{HI} \subset T$ is the *HI-criticality* task set, $\sum_{\tau_i \in T} V_i(LO)$ is the system total value in *LO-criticality* system mode, and $\sum_{\tau_i \in T_{HI}} V_i(HI)$ is the system total value in HI-criticality mode.

To compare the two values in *HI-criticality* mode and *LO-criticality* separately, $V_i(HI)$ and $V_i(LO)$, let $V_i(HI) = \text{CF} \times V_i(LO)$, where CF is criticality factor of task, which satisfies $\text{CF} > 1$. ΔTV indicates the total value difference between in *HI-criticality* mode and in *LO-criticality* mode.

$$\begin{aligned} \Delta\text{TV} &= \sum_{\tau_i \in T_{HI}} V_i(HI) - \sum_{\tau_i \in T} V_i(LO) \\ &= \sum_{\tau_i \in T_{HI}} \text{CF} \times V_i(LO) - \sum_{\tau_i \in T} V_i(LO) \\ &= \sum_{\tau_i \in T_{HI}} (\text{CF} - 1) \times V_i(LO) - \sum_{\tau_i \in T_{LO}} V_i(LO). \end{aligned} \quad (4)$$

Inputs:
Task set to be partitioned $T = \tau_1,...,\tau_n$;
Processors set to be allocated $P = p_1, \cdots, p_m$.
Outputs:
PT = $Que\_Ready\_p_1,...,Que\_Ready\_p_m$.
/*where $Que\_Ready\_p_i$ is the tasks ready queue on p_i.*/
1: **for** each p_j in P **do**
2: Set $Que\_Ready\_p_j = \{\}$;
3: **end for**
4: Sorted T according to *priority* in descending
5: **for** each τ_i in T**do**
6: Calculate R_i by Eq. (3);
7: **for** each p_j in P by descending order **do**
8: **if** $R_i \le D_i$ **then**
9: Add τ_i into $Que\_Ready\_p_j$;
10: **break**;
11: **end if**
12: **end for**
13: **end for**
14: **return** PT;

Algorithm 1: Schedulablity analysis.

In Equation (4), $\sum_{\tau_i \in T_{HI}} (CF - 1) \times V_i(LO)$ indicates the value difference of *HI-criticality* tasks in *HI-criticality* mode and in *LO-criticality* mode, and $\sum_{\tau_i \in T_{LO}} V_i(LO)$ represents the values obtained by the *LO-criticality* tasks.

If $\Delta TV > 0$, it means that the TV increases as system criticality mode upgrades. In other words, the value difference of *HI-criticality* tasks in different criticality modes is larger than the *LO-criticality* tasks' values at this time.

If $\Delta TV \le 0$, it means the TV does not rise when the system criticality mode switches from *LO-criticality* to *HI-criticality*.

6.2. Assignment of Task's Priority. In MCS, the task's priority should reflect its attributes, including criticality level, value, and deadline. When constructing the priority assignment function Pr_i, we consider the importance of these factors.

(1) All *HI-criticality* tasks should be executed firstly

(2) Next, tasks with high value are prioritized in the same criticality level

In different system criticality modes, for a task τ_i, its priority is recorded as Pr_i^l:

$$Pr_i^l = e^{\frac{C_i(l)}{d_i}} \times \ln [V_i(l)], \quad (5)$$

where $C_i(l)$ and $V_i(l)$ vary as the system critical mode l changes. And l satisfies $l \in \{LO, HI\}$.

Theorem 5. *For the task τ_i, it satisfies $Pr_i^{HI} > Pr_i^{LO}$.*

Proof. In dual MCS with *HI-criticality* and *LO-criticality*, there are $C_i(HI) > C_i(LO)$ and $V_i(HI) > V_i(LO)$. Therefore, $Pr_i^{HI} > Pr_i^{LO}$.

Lemma 6. *When the system is in HI-criticality mode, for each HI-criticality taskτ_i and LO-criticality taskτ_j, it satisfies $Pr_i^{HI} > Pr_i^{LO}$.*

7. Scheduling Algorithm

For the MCS under a homogeneous multiprocessor platform, we propose a smart semipartitioned scheduling strategy (SSPS).

7.1. Smart Semipartitioned Scheduling Strategy (SSPS). SSPS includes both the processes of partitioned scheduling and global scheduling; the details are as follows:

(1) Task order. All the tasks of T are sorted in a descending order according to their priorities calculated by Equation (5)

(2) Processors allocation. Each sorted task is allocated to processors by First-Fit Decreasing (FFD) method

(3) Schedulability test. The task subset's schedulability is tested by Algorithm 1

(4) Task execution. This process includes executing jobs released by the task and collecting the processors' idle time for *LO-criticality* tasks' execution

(a) Tasks allocated in each processor execute by priority, and these tasks do not migrate. During this process, the slack times of each processor are collected and stored in the queue Que_Slack

(b) When the system criticality mode upgrades, all unfinished *LO-criticality* jobs are sorted and managed globally and then assign their execution times in Que_Slack. At the high mode, we allow the execution of *LO-criticality* tasks but do not allow them to preempt *HI-criticality* tasks; i.e., the *HI-criticality* tasks will not incur any interference from the ones with a *LO-criticality* level

Here, the queue Que_Slack is used to store feasible slack fragment sf. Each sf is represented in the form of (q, d), where q is the length of time and d is the end time of sf. The algorithm of slack time collection is shown as Algorithm 2.

In Algorithm 2, the inputs include two pieces of information, including the given executing job J_{exe}^j and slack fragment sf_i of processor p_i. The output of the algorithm is the idle time queue Que_Slack. At first, the condition of collect processor slack time is that the job J_{exe}^j can finish until its deadline d_{exe}^j (line 1). And if the input sf_i is null, insert into queue Que_Slack (lines 2-6). Otherwise, discuss the values of J_{exe}^j executing time $e_{exe}^j = q$ and the sf_i's length q, the former should be not larger than the latter to ensure J_{exe}^j's execution. If the two are equal, then remove the sf_i from Que_Slack (lines 9-10), and if e_{exe}^j is less than q, q's remaining time after completing J_{exe}^j is update to Que_Slack (lines 11-14). At last, return Que_Slack (line 17).

Inputs:
Executing job J^j_{exe};
Slack fragment sf_i.
Output:
The queue for collect idle time *Que_Slack*.
1: **if** J^j_{exe} finish at $t_0(t_0<d^j_{exe})$ **then**
2: **if** sf_i = null **then**
3: $q = (C_{exe} - e^j_{exe})$;
4: $d = d^j_{exe}$;
5: $sf_i \Leftarrow (q, d)$;
6: Insert sf_i into *Que_Slack*;
7: **else**
8: Get q,d from sf_i;
9: **if** $e^j_{exe} = q$ **then**
10: Remove sf_i from *Que_Slack*;
11: **else if** $e^j_{exe} < q$ **then**
12: Set $q = q - e^j_{exe}$;
13: Update sf_i to *Que_Slack*;
14: **end if**
15: **end if**
16: **end if**
17: **return** *Que_Slack*;

Algorithm 2: Slack time collection.

Example 1. A task set including 5 tasks is shown in Table 1 and is divided into two homogeneous processors.

The system is in *LO-criticality* mode at the initial time, and if the task set is presorted using the DU method, the sequence of system tasks is τ_1, τ_3, τ_4, τ_2, and τ_5. In accordance with the FFD strategy, tasks τ_1 and τ_3 are divided into processor p_1, while tasks τ_2, τ_4, and τ_5 are divided into processor p_2. In this case, two processors meet the conditions. In *LO-criticality* system mode, the resource utilization of p_1 is 83.3%, and the corresponding resource utilization of p_2 is 78.3%. When the system mode upgrades to a *HI-criticality* mode, all *LO-criticality* tasks τ_3, τ_4, and τ_5 are discarded. In this case, processor p_1 obtains a resource utilization of 62.5% and the resource utilization of processor p_2 is 50%.

The existing scheduling algorithms, such as MC-PARTITION, drop out all *LO-criticality* tasks directly in *HI-criticality* system mode, even if there is some idle time on the processor at some moment. In order to improve the acceptance rate of the task and the utilization of the processor, the cutoff period and value attribute of the task can be considered comprehensively, and the *LO-criticality* task is dispatched globally by using the idle time of the processor when the system mode switches from *LO-criticality* into *HI-criticality*. A hyperperiod execution (here, 24 time units) for the task set of Example 1 is shown in Figure 3, in which the task set is presorted according to the priority function of Equation (5), with the same result as the DU method.

After the system critical mode upgrades, we perform global scheduling for *LO-criticality* tasks τ_3, τ_4, and τ_5 on the premise of ensuring *HI-criticality* tasks τ_1 and τ_2 and allocate idle time on processors p_1 and p_2 dynamically. This method can achieve high acceptance ratio and improve the utilizations of both processors to 91.7% (11/12).

Table 1: A task set of 5 tasks.

	$C_i(LO)$	$C_i(HI)$	ζ_i	D_i	$V_i(LO)$	$V_i(HI)$
τ_1	4	5	HI	8	20	50
τ_2	2	4	HI	8	20	40
τ_3	2	2	LO	6	35	35
τ_4	4	5	LO	3	20	20
τ_5	4	5	LO	5	10	10

7.2. Analysis of SSPS. In the smart semipartitioned scheduling strategy (SSPS), the task's schedulability is analyzed by Algorithm 1; tasks allocated by first fit (FF) are assigned priority by Equation (4). The queue *Que¯Slack* is used to collect slack time by Algorithm 2, and the *LO-criticality* jobs are collected to the queue *Que¯Low*. The system selects the segment in *Que¯Slack* to execute the job in *Que¯Low*. Meanwhile, the queue *Que¯Ready* is used to store prepared tasks. And the pseudocode of SSPS is shown in Algorithm 3 as follows.

In Algorithm 3 mentioned above, the inputs include seven parameters: the task set T, the processor set P, the queue for LO-criticality jobs, the queue storing processor's idle time, the queue for ready tasks, the initial system mode ($Sys¯Mode = LO$), the total successful jobs' number N_{ST} and the total job number N_T. The output of Algorithm 3 is the acceptance ratio which equals N_{ST}/N_T. At first, analyze all tasks of T and allocate to a ready queue $Que¯Ready¯p_m$ of each processor p_m (line 1). Then, for the queue $Que¯Ready¯p_m$, get the task J^m_{exe} and compare its response time R^m_{exe} to $C_{exe}(LO)$ (lines 2-4). If J^m_{exe} cannot finish its completion ($R^m_{exe} > C_{exe}(LO)$), then discuss if J^m_{exe} is a *HI-criticality* job and $Sys¯Mode = LO$; the system mode switches to *HO-criticality* from *LO-criticality* according the MCS definition; abort *LO-criticality* tasks in ready queue of each process, insert *LO-criticality* jobs into the queue *Que¯Low*, and execute the J^m_{exe} (lines 5–10); if J^m_{exe} is a *LO-criticality* job, then abort it and continue (lines 11-14). Otherwise, J^m_{exe} can finish, execute it, and collect the slack time of its completion (lines 15-19).

During the scheduling in *HO-criticality* system mode, the processor of executing *LO-criticality* uses idle time of each processor (lines 20-30). For each slack fragment sf_s in queue *Que¯Slack*, the top job J^m_{exe} of *Que¯Low* is chosen to execute. It is necessary to compare e^m_{exe} to q_s, the length of sf_s. If the former is not less than the letter, then complete J^m_{exe} and collect the slack time of its completion (lines 22-25).

Algorithm 3 calls Algorithm 1 (line 1) that contains a two-layer loop. And its another two-layer loop in algorithm, where the outer-layer loop is in lines 3-31 of $O(m)$, and the inner-layer loop (lines 21-29) can be evaluated in constant time of $O(n)$. Therefore, the time complexity of the Algorithm 3 is in the pseudopolynomial order.

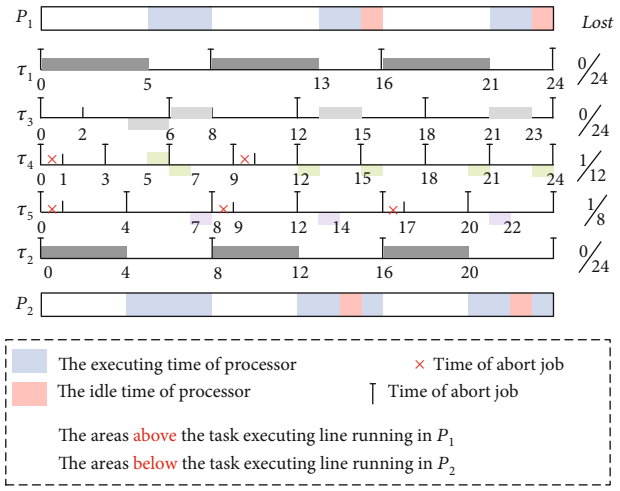

Figure 3: Execution of the task set in Example 1.

For the SSPS algorithm, it is necessary to discuss the system value V_i.

(1) In *LO-criticality* system mode, the system total value of the T is $\sum_{\tau_i \in T} V_i(LO)$.

(2) In *HI-criticality* system mode, the total value contains two parts: the value obtained by the *HI-criticality* task $\sum_{\tau_i \in T_{HI}} V_i(HI)$, where $T_{HI} \subset T$, and the value obtained by the finished *LO-criticality* task set, named as $T_{LO}{}'$, $\sum_{\tau_i \in T_{LO}{}'} V_i(LO)$.

The total value $\mathrm{TV}_{\mathrm{SSPS}}$ can be described as

$$\mathrm{TV}_{\mathrm{SSPS}} = \sum_{\tau_i \in T} V_i(LO) + \sum_{\tau_i \in T_{HI}} V_i(HI) + \sum_{\tau_i \in T_{LO}{}'} V_i(LO). \quad (6)$$

According to Equation (6) about the TV of the classic strategies, $\mathrm{TV}_{\mathrm{SSPS}} \geq \mathrm{TV}$ obviously.

To discuss the $\mathrm{TV}_{\mathrm{SSPS}}$ changed with system criticality mode, let $\Delta\mathrm{TVV}_{\mathrm{SSPS}}$ indicate the total value difference between in *HI-criticality* mode and in *LO-criticality* mode. If $\Delta\mathrm{TV}_{\mathrm{SSPS}} > 0$, it indicates that the $\mathrm{TV}_{\mathrm{SSPS}}$ in *HI-criticality* mode is bigger; conversely, it means that the $\mathrm{TV}_{\mathrm{SSPS}}$ in *HI-criticality* mode is smaller.

$$\begin{aligned}\Delta\mathrm{TV}_{\mathrm{SSPS}} &= \sum_{\tau_i \in T_{HI}} V_i(HI) + \sum_{\tau_i \in T_{LO}{}'} V_i(LO) - \sum_{\tau_i \in T} V_i(LO) \\ &= \sum_{\tau_i \in T_{HI}} CF \times V_i(LO) + \sum_{\tau_i \in T_{LO}{}'} V_i(LO) - \sum_{\tau_i \in T} V_i(LO) \\ &= \sum_{\tau_i \in T_{HI}} (CF - 1) \times V_i(LO) - \sum_{\tau_i \in \left(T - T_{HI} - T_{LO}{}'\right)} V_i(LO),\end{aligned} \quad (7)$$

where $\mathrm{CF} = V_i(HI)/V_i(LO)$ is the criticality factor of task, and $\mathrm{CF} > 1$.

In Equation (7), $\sum_{\tau_i \in T_{HI}} (\mathrm{CF} - 1) \times V_i(LO)$ is the obtained value of *HI-criticality* task difference in *HI-criticality* mode and in *LO-criticality* mode. $\sum_{\tau_i \in (T - T_{HI} - T_{LO}{}')} V_i(LO)$ is the value obtained by the dropped *LO-criticality* tasks.

Inputs:
$T = \tau_1, \tau_2, \cdots, \tau_n$;
$P = p_1, p_2, \cdots, p_m$;
Que_Low, $Que\_Slack = null$;
$Que\_Ready = T$;
$Sys\_Mode = LO$;
$N_{ST}, N_T = 0$.
Output:
Acceptance Ratio.
1: PT= schedulability analysis(Que_Ready, P)
/*$PT = \{Que\_Ready\_p_1, \cdots, Que\_Ready\_p_m\}$, where $Que\_Ready\_p_1$ is the tasks allocated to p_1 */
2: **if** $PT \neq null$ **then**
3: **for** each $Que\_Ready\_p_m$ in PT **do**
4: Get the top element J^m_{exe} out of $Que\_Ready\_p_m$;
5: N_T + +;
6: $R^m_{exe} > C_{exe}(LO)$ **then**
7: **if** J^m_{exe} is *HI-criticality* and $Sys\_Mode = LO$ **then**
8: Abort all Jobs of *LO-criticality* in PT;
9: Insert *LO-criticality* jobs into Que_Low as Pr_{p_LO} in descending order;
10: $Sys\_Mode = HI$;
11: Execute J^m_{exe} in *HI-criticality*;
12: N_{ST} + +;
13: **else**
14: Abort J^m_{exe};
15: **continue**;
16 **end if**
17: **else**
18: Execute J^m_{exe};
19: N_{ST} + +;
20: Que_Slack= slack time collection(J^m_{exe}, *null*);
21: **continue**;
22: **end if**
23: **if** $Sys\_Mode = HI$ **then**
24: **for** each sf_s from Que_Slack **do**
25: Get the top element J^j_{exe} out of Que_Low;
26: **if** $e^j_{exe} \leq q_s$ **then**
27: Execute J^j_{exe};
28: N_{ST} + +, N_T + +;
29: Que_Slack= slack time collection(J^j_{exe}, sf_s);
30: **else**
31: **continue**;
32: **end if**
33: **end for**
34: **end if**
35: **end for**
36: return N_{ST}/N_T;
37: **end if**

Algorithm 3: SSPS.

8. Simulations and Analysis

8.1. Simulation Experiments. The simulation experiments are performed to test the scheduling algorithm SSPS. All

experiments were run on a PC with a 3.40 GHz 4 identical processor and 8 GB memory. In the simulation, we compared SSPS to the existing partition scheduling algorithms, DC-RM [13] and MC-PARTITION [14], which are the classic and representative algorithms in the research community of MCS partition scheduling. Based on these two algorithms, there are lots of derived algorithms for other real-time application scenarios [16–19]. The task set parameters of the experiments were randomly generated as follows:

(1) The utilization of each task U_i^{LO} was in the range [0.025, 0.975] and was generated by the Uunifast-Discard algorithm

(2) The proportion of high-critical tasks to task set T HTP, which directly affects the execution of *LO-criticality* tasks and then affects the performance of our SSPS strategy, was 0.5 by default. And the criticality factor $\mathrm{CF} = C_i(HI)/C_i(LO)$ was set to 1.5, according to Definition 1 where $\mathrm{CF} > 1$

(3) Each task τ_i's *LO-criticality* execution time $C_i(LO)$ is randomly generated in the range from 1 to 10 in accordance with uniform distribution. And, τ_i's execution time of *HI-criticality* $C_i(HI)$ satisfies $C_i(HI) = \mathrm{CF} \times C_i(LO)$

(4) The task τ_i's relative deadline D_i satisfies: $D_i = C_i(LO)/U_i^{LO}$, and τ_i's period T_i is $T_i = D_i$

(5) The task τ_i's value of *LO-criticality* $V_i(LO)$ was generated randomly between 10 and 50, and its *HI-criticality* $V_i(HI)$ was set to $V_i(HI) = 5 \times \mathrm{CF} \times V_i(LO)$

The performance indicators include acceptance ratio (AR), weighted schedulability (WS), and total values (TV).

(1) $\mathrm{AR} = N_{ST}/N_T$, where N_{ST} is the number of successful tasks and N_T is the number of system taskset. The AR shows the proportion of successful tasks to total task set

(2) $\mathrm{WS} = (\sum_i b_i \times U_i^{LO})/\sum_i U_i^{LO}$, where $b_i \in 0, 1$. The WS indicates the total utilization of each task

(3) $\mathrm{TV} = \sum_{\tau_i \in ST} V_i$, where ST is the successful tasks. The TV represents the QoS of whole successful tasks

In order to measure the average number of job migrations, 100 trials of simulations with different tasks are conducted in the experiment.

8.2. Acceptance Ratio Analysis. The AR changes as tasks' different LO-critical utilization U_i^{LO} scheduled by MC-PARTITION, DC-RM, and SSPS are demonstrated in Figure 4. It can be seen that all algorithms' AR decrease significantly as the U_i^{LO} grows from 0.3 to 1.0. And the DC-RM algorithm has the least AR and the SSPS algorithm obtains the best AR. When the U_i^{LO} is below 0.5, SSPS is close to MC-PARTITION and DC-RM in AR. But as U_i^{LO} becomes larger than 0.6, the SSPS begins to outperform the other two algorithms, because in *HI-criticality* mode, SSPS executes the *LO-criticality* tasks selectively, improving the AR of the whole system, while the other two algorithms discard *LO-criticality* tasks directly, which leads to a sharp descending in AR.

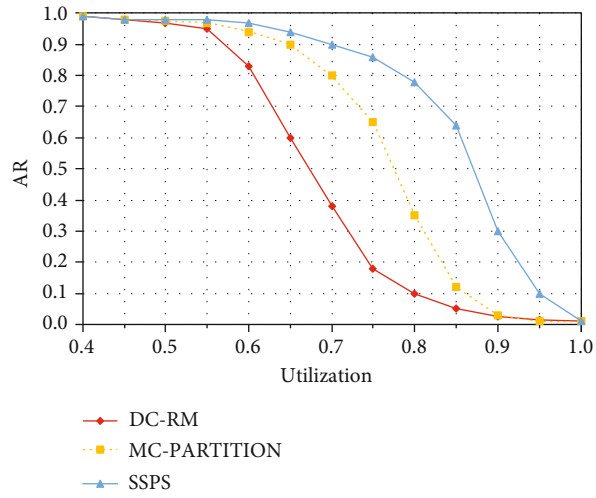

Figure 4: Results of AR with the change of U_i^{LO}.

It is illustrated that the AR varies in different high-critical task proportion *HTP* in Figure 5, when U_i^{LO} is set to 0.6 and the HTP grows from 0.3 to 1.0. As shown by the result, with the HTP increases, the additional *HI-criticality* tasks require more executing time; thus, all algorithms's AR continues to decline. When the HTP is less than 0.5, all algorithms' AR is close because the competition of tasks' execution is not intense in *LO-criticality* system mode. As the HTP grows larger than 0.6, the intense competition among tasks reduces the AR, in which some task cannot finish its execution and the system mode arises to *HI-criticality*.

Once the system mode upgrades to *HI-criticality*, the task's C_i becomes large. Compared to the MC-PARTITION and DC-RM, the SSPS algorithm achieves a more stable and higher AR, because the former two algorithms drop the *LO-criticality* tasks directly. When the system mode switches from *LO-criticality* to *HI-criticality*, the SSPS algorithm executes some *LO-criticality* jobs in an idle time of the processor and gradually decreases as HTP increases.

8.3. Schedulability Analysis. It is shown that the weighted schedulability WS is declining with U_i^{LO} arguments, where HTP is set to 0.5 (see Figure 6). Compared to MC-PARTITION and DC-RM algorithms, the SSPS gets higher and more stable in WS through schedule the *LO-criticality* task in *HI-criticality* system mode. In the beginning, the WS obtained by all algorithms is falling steadily. When HTP becomes larger than 0.6, MC-PARTITION and DC-RM accelerate degradation in WS due to the more execution time required by increased *HI-criticality* tasks and the discarded of *LO-criticality* tasks directly in *HI-criticality* system mode.

That weighted schedulability WS results change as HTP grows is plotted in Figure 7, where $U_i^{LO} = 0.5$. The HTP is

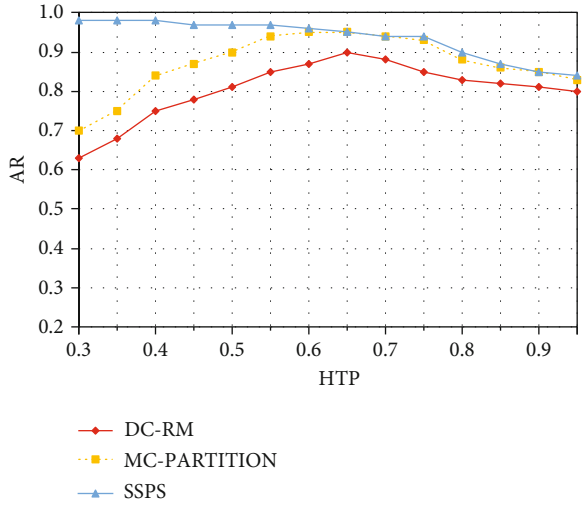

Figure 5: Results of AR as HTP changes.

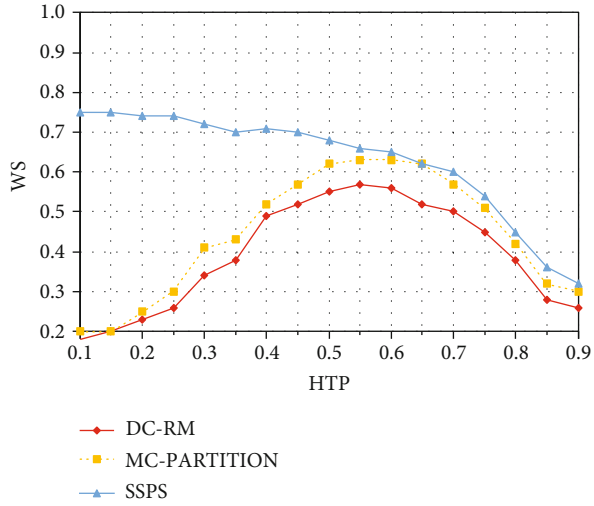

Figure 7: Results of WS with the change of HTP.

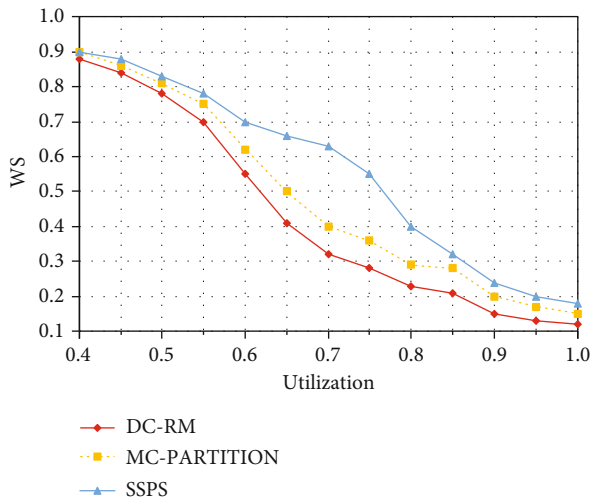

Figure 6: Results of WS with the change of U_i^{LO}.

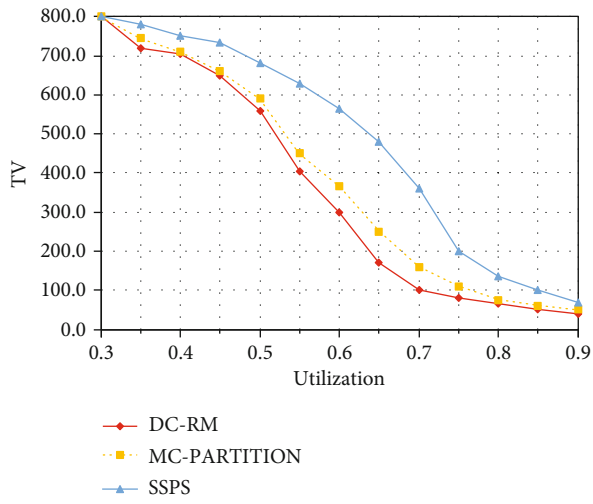

Figure 8: Results of TV with the change of HTP.

represented on the horizontal axis, and the vertical axis is WS. We can see that WS is gradually declining as HTP arguments. Because with the number of *HI-criticality* tasks increasing, the system resources they need are increased. When the HTP is below 0.3, SSPS is almost identical to MC-PARTITION and DC-RM algorithms that get a steady decline in WS. With the HTP becoming larger and the system criticality level arising, the execution time of the *HI-criticality* task becomes longer, which can intensify competition among tasks and reduces the system schedulability. The SSPS can obtain better WS than the other two methods, because SSPS in *HI-criticality* mode can take advantage of slack time produced by *HI-criticality* task, to execute the selected *LO-criticality* task globally.

8.4. Total Value Analysis. The simulation results for total value TV changed with LO-critical system utilization U_i^{LO}'s growth are shown in Figure 8, where the horizontal axis is U_i^{LO} and the vertical axis is TV. As shown in Figure 8, all algorithms' TV decrease as the U_i^{LO} increases from 0.3 to 0.9. Compared to the other two algorithms, the MAPPS has a significant advantage over the other two in TV, which gradually decreases as the U_i^{LO} grows. Because only the SSPS chooses the task with high urgency and high value, thereby obtaining better TV and improving the performance of the system.

Figure 9 plots the TV with the change of HTP. It shows that the total value TV presents the fluctuation of first up then down as HTP grows from 0.1 to 0.9. In the beginning, the growing number of *HI-criticality* tasks can take a larger value. But with HTP increasing, the *HI-criticality* task's longer executing time reduces the WS of the system, as shown in Figure 9, which brings the TV decreasing. And SSPS can obtain the best TV due to its choice of high urgency and high-value tasks.

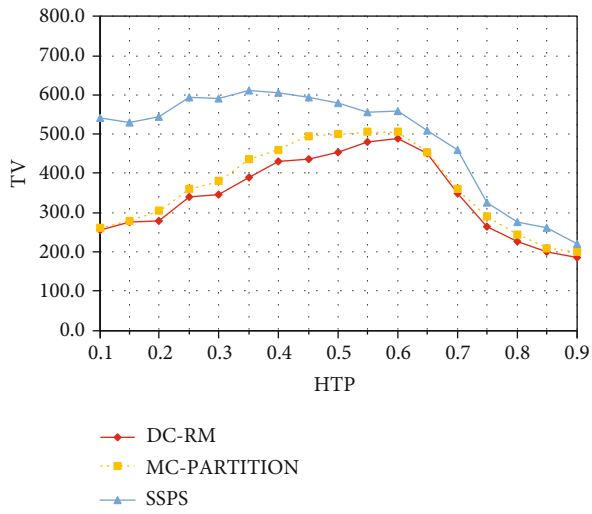

Figure 9: Results of TV with the change of HTP.

9. Conclusions and Future Works

In recent years, with the increasing popularity of 6G wireless communication technology, mixed-criticality systems (MCS) in 6G-based edge computing have been grown quickly in application scenarios. Meanwhile, with the multiprocessors' development widely applied, including homogeneous, the relative MCS scheduling technique is necessary to research. In this paper, a smart semipartitioned scheduling algorithm (SSPS) was designed on MCS in the homogeneous multiprocessors. Firstly, we analyze the task's schedulability based on the response time and allocate the processors. Then, a task's priority assignment function with multiple attributes, including criticality, urgency, and the total value, is constructed. Besides, a scheduling algorithm titled by SSPS has been proposed with the schedulability analysis algorithm and the priority assignment above. In the SSPS, we allocate the tasks in *LO-criticality* mode, while in *HI-criticality* mode, the SSPS not only finish the *HI-criticality* tasks but also choose the *LO-criticality* tasks to execute under the utilization of the processor's slack time globally. The experimental results illustrate that the SSPS could achieve the best performance among the existing algorithms.

However, there are still some limitations of the SSPS algorithm. In practical 6G-based edge computing applications, the task real-time scheduling is often related to the sharing of limited resources. With the heterogeneous multiprocessors' development, heterogeneous will be more the case for the 6G-based real-time applications. We will explore the scheduling and resource sharing issues of edge computing of heterogeneous multiprocessors based on the SSPS algorithm. Besides, in other complex real-time applications, like parallel industry systems and smart industrial networks [31–33], it needs to consider several factors in data transmission and task scheduling; we are also planning to investigate these issues. Moreover, we notice that modern IoT devices are increasingly being equipped with multiple network interfaces; our future work will consider to apply the proposed SSPS algorithm to optimize the promising multipath parallel data transmission methods [34, 35] for the multihomed IoT environment.

Data Availability

The data, including task's properties and performance indicators in the experiments, used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research is funded by the National Natural Science Foundation of China (NSFC) under Grant Nos. 61762040, 61962026, and 62041702; the Natural Science Foundation of Jiangxi Province under Grant No. 20192ACBL21031; the Provincial Key Research and Development Program of Jiangxi Province (No. 20181ACE50029); the Science and Technology Research Project of Jiangxi Provincial Department of Education (Nos. GJJ170234 and GJJ160781); and the doctoral research project of Jiangxi Normal University (No. 12020361).

References

[1] F. Song, M. Zhu, Y. Zhou, I. You, and H. Zhang, "Smart collaborative tracking for ubiquitous power IoT in edge-cloud interplay domain," *IEEE Internet of Things*, vol. 7, no. 7, pp. 6046–6055, 2020.

[2] H. A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, "Dynamic task offloading and scheduling for low-latency IoT services in multi-access edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 668–682, 2019.

[3] D. Wu, X. Nie, E. Asmare et al., "Towards distributed SDN: mobility management and flow scheduling in software defined urban IoT," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1400–1418, 2020.

[4] S. Hu and G. Li, "Dynamic request scheduling optimization in mobile edge computing for IoT applications," *IEEE Internet of Things*, vol. 7, no. 2, pp. 1426–1437, 2020.

[5] S. Pandiyan, T. S. Lawrence, V. Sathiyamoorthi, M. Ramasamy, Q. Xia, and Y. Guo, "A performance-aware dynamic scheduling algorithm for cloud-based IoT applications," *Computer Communications*, vol. 160, pp. 512–520, 2020.

[6] S. Baruah, "Schedulability analysis for a general model of mixed-criticality recurrent real-time tasks," in *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016)*, pp. 25–34, Porto, Portugal, December 2016.

[7] A. Burns and R. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys*, vol. 50, no. 6, pp. 1–37, 2017.

[8] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the 28th IEEE International Real-Time Systems*

Symposium (RTSS 2007), pp. 239–243, Tucson, Arizona, USA, December 2007.

[9] S. Baruah, V. Bonifaci, G. D'Angelo et al., "Scheduling real-time mixed-criticality jobs," *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1140–1152, 2012.

[10] H. Li and S. Baruah, "Global mixed-criticality scheduling on multi-processors," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pp. 166–175, Pisa, Italy, July 2012.

[11] B. B. Brandenburg and M. Gul, "Global scheduling not required: simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016)*, pp. 99–110, Porto, Portugal, December 2016.

[12] K. Yang and J. Anderson, "On the soft real-time optimality of global EDF on uniform multiprocessors," in *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS 2017)*, pp. 319–330, Paris, France, December 2017.

[13] O. R. Kelly, H. Aydin, and B. Zhao, "On partitioned scheduling of fixed-priority mixed-criticality task sets," in *Proceedings of the 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 1051–1059, Changsha, China, November 2011.

[14] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, 2014.

[15] D. Niz and L. T. X. Phan, "Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms," in *Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS 2014)*, pp. 111–122, Berlin, Germany, April 2014.

[16] S.-W. Cheng, J.-J. Chen, J. Reineke, and T.-W. Kuo, "Memory bank partitioning for fixed-priority tasks in a multi-core system," in *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS 2017)*, pp. 209–219, Paris, France, December 2017.

[17] G. Chen, N. Guan, D. Liu et al., "Utilization-based scheduling of flexible mixed-criticality real-time tasks," *IEEE Transactions on Computers*, vol. 67, no. 4, pp. 543–558, 2018.

[18] S. Senobary and M. Naghibzadeh, "Semi-partitioned scheduling for fixed-priority real-time tasks based on intelligent rate monotonic algorithm," *International Journal of Grid and Utility Computing*, vol. 6, no. 3/4, pp. 184–191, 2015.

[19] J. Anderson, J. Erickson, U. Devi, and B. Casses, "Optimal semi-partitioned scheduling in soft real-time systems," *Journal of Signal Processing Systems*, vol. 84, no. 1, pp. 3–23, 2016.

[20] E. Cannella and T. P. Stefanov, "Energy efficient semi-partitioned scheduling for embedded multiprocessor streaming systems," *Design Automation for Embedded Systems*, vol. 20, no. 3, pp. 239–266, 2016.

[21] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pp. 155–165, Pisa, Italy, July 2012.

[22] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Proceedings of the 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE 2013)*, pp. 147–152, Grenoble, France, March 2013.

[23] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Proceedings of the Workshop on Mixed-Criticality Systems (WMC 2013)*, pp. 1–6, Vancouver, Canada, December 2013.

[24] S. Baruah, A. Burns, and Z. Guo, "Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*, pp. 131–138, Toulouse, France, July 2016.

[25] S. Baruah, A. Easwaran, and Z. Guo, "MC-Fluid: simplified and optimally quantified," in *Proceedings of the 36th IEEE Real-Time Systems Symposium (RTSS 2015)*, pp. 327–337, San Antonio, TX, USA, December 2015.

[26] J. Lee, S. Ramanathan, K. Phan, A. Easwaran, I. Shin, and I. Lee, "MC-Fluid: multi-core fluid-based mixed-criticality scheduling," *IEEE Transactions on Computers*, vol. 67, no. 4, pp. 469–483, 2018.

[27] H. Chen, "A real-time tasks scheduling method based on dynamic priority," *Journal of Circuits, Systems and Computers*, vol. 23, no. 2, article 1450029, 2014.

[28] S. Asyaban and M. Kargahi, "An exact schedulability test for fixed-priority preemptive mixed-criticality real-time systems," *Real-time Systems*, vol. 54, no. 1, pp. 32–90, 2018.

[29] S. Zhao, P. Dziurzanski, and L. S. Indrusiak, "Value-driven manufacturing planning using cloud-based evolutionary optimisation," 2019, http://arxiv.org/abs/1912.01562.

[30] Z. Jiang, S. Zhao, D. Pan et al., "Re-thinking mixed-criticality architecture for automotive industry," in *IEEE International Conference on Computer Design (ICCD 2020)*, Hartford, CT, USA, October 2020.

[31] F. Song, Y.-T. Zhou, Y. Wang, T.-M. Zhao, I. You, and H.-K. Zhang, "Smart collaborative distribution for privacy enhancement in moving target defense," *Information Sciences*, vol. 479, pp. 593–606, 2019.

[32] F. Song, Y. Zhou, L. Chang, and H. Zhang, "Modeling space-terrestrial integrated networks with smart collaborative theory," *IEEE Network*, vol. 33, no. 1, pp. 51–57, 2019.

[33] H. Chen, H. Jin, and S. Wu, "Minimizing inter-server communications by exploiting self-similarity in online social networks," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 4, pp. 1116–1130, 2016.

[34] Y. Cao, L. Zeng, Q. Liu, G. Lei, M. Huang, and H. Wang, "Receiver-assisted partial-reliable multimedia multipathing over multi-homed wireless networks," *IEEE Access*, vol. 7, pp. 177675–177689, 2019.

[35] Y. Cao, M. Collotta, S. Xu, L. Huang, X. Tao, and Z. Zhou, "Towards adaptive multipath managing: a lightweight path management mechanism to aid multihomed mobile computing devices," *Applied Sciences*, vol. 10, no. 1, pp. 1–18, 2020.