2013 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications

Improving OCBP-based Scheduling for Mixed-Criticality Sporadic Task Systems

Chuancai Gu[1], Nan Guan[1,2], Qingxu Deng[1] and Wang Yi[1,2]

[1]Northeastern University, China
[2]Uppsala University, Sweden

Abstract—**Scheduling mixed-criticality systems is a challenging problem. Recently a number of new techniques are developed to schedule such systems, among which an approach called OCBP has shown interesting properties and drawn considerable attentions. OCBP explores the job-level priority order in a very flexible manner to drastically improve the system schedulability. However, the job priority exploration in OCBP involves nontrivial overheads. In this work, we propose a new algorithm LPA (Lazy Priority Adjustment) based on the OCBP approach, which improves the state-of-the-art OCBP-based scheduling algorithm PLRS in both schedulability and run-time efficiency. Firstly, while the time-complexity of PLRS' online priority management is quadratic, our new algorithm LPA has linear time-complexity at run-time. Secondly, we present an approach to calculate tighter upper bounds of the busy period size, and thereby can greatly reduce the run-time space requirement. Thirdly, the tighter busy period size bounds also improve the schedulability in terms of acceptance ratio. Experiments with synthetic workloads show improvements of LPA in all the above three aspects.**

I. Introduction

Modern embedded systems usually integrate multiple functions on a shared platform to meet various constraints on system size, cost and power. These different functions may not be equally critical to the overall system performance, and are usually subject to more or less rigorous certifications/validations. Such mixed-criticality systems bring significant challenges in the design of embedded real-time systems.

In 2007, Vestal [16] first formalized the mixed-criticality scheduling problem, which turned out to be significantly more difficult than the traditional real-time scheduling problem. Traditional scheduling techniques like EDF and Criticality-Monotonic scheduling may lead to poor system schedulability. In recent years, this challenging problem has drawn considerable attentions in the real-time scheduling research community. A number of heuristic algorithms and analysis techniques are developed to achieve better schedulability performance in terms of quantitative performance guarantees and/or acceptance ratio.

Among those, the OCBP (Own Criticality Based Priority) algorithm family has shown interesting properties. First, it is applicable to a wide range of settings (both finite job collections and sporadic task systems; with or without execution time monitoring support). Second, it has the best known quantitative performance guarantee in terms of speedup factor for sporadic task systems. The main idea of OCBP is to explore the job-level priority order in a very flexible manner to optimize system schedulability. This can be viewed somehow as combining the benefits of EDF which makes scheduling decisions based on job-level priorities and fixed-priority scheduling with OPA (optimal priority assignment) [1] which explores task-level priority orders to improve system schedulability. Indeed, OCBP is optimal, in terms of speedup factors, in scheduling a finite collection of mixed-criticality jobs among all job-level priority based algorithms. Although OCBP is originally designed for the mixed-criticality scheduling problem, it actually represents a powerful approach which may be applied to a much wider range of scheduling problems where traditional EDF and fixed-priority scheduling techniques do not perform well.

However, applying OCBP to schedule sporadic task systems may cause nontrivial run-time efficiency problems. Due to the infinite and non-deterministic natural of jobs released by a sporadic task, it is infeasible to provide a complete offline priority assignment plan to all the jobs released at run-time. To solve this problem, Li and Baruah [3] proposed an OCBP-based algorithm (called LB in this paper), which from time to time recomputes the priority assignment of future jobs. LB generalizes OCBP's speedup factor 1.618 in scheduling a finite collection of mixed-criticality jobs to mixed-criticality sporadic tasks. However, the online priority assignment re-computation in LB has pseudo-polynomial time-complexity (polynomial regarding the maximal busy period size), which is too high for realistic run-time schedulers. Later, Guan et al. [9] proposed an algorithm, called PLRS, to improve the run-time efficiency of LB. The time-complexity of the online priority management in PLRS is quadratic (regarding the number of tasks). Indeed, a quadratic-time scheduler is much more convincing than the pseudo-polynomial version, but one may still worry if this is efficient enough in practice, since in run-time schedulers quadratic-complexity operations are still generally considered to be a bit too expensive and should be avoided as much as possible. Moreover, both LB and PLRS need to store the priority assignment plan for future jobs that may be released in the longest busy period, which is of pseudo-polynomial complexity and in practice usually very large.

In this work, we propose a new OCBP-based scheduling algorithm, LPA (Lazy Priority Adjustment), which improves the state-of-the-art OCBP-based algorithm PLRS in both run-time efficiency and schedulability:

- **Improving online time efficiency.** LPA's run-time pri-

ority management has linear time-complexity, which is more efficient than PLRS, whose run-time priority management has quadratic time-complexity. Simulations with synthetic task systems show that LPA is not only superior in theoretical complexity bounds, but also can significantly improve the the run-time overhead in practice comparing with PLRS.
- **Improving online space efficiency.** We propose a tighter bound of the maximal busy period size for mixed-criticality sporadic task systems. Since the run-time space efficiency of OCBP-based algorithms is proportionate to the maximal busy period size, LPA can significantly reduce the run-time storage requirement comparing with PLRS.
- **Improving schedulability.** LPA improves the schedulability of PLRS, in the sense that it can successfully schedule a considerable amount of task systems that are non-schedulable by PLRS. The schedulability improvement is also due to our tighter maximal busy period size bound.

II. Related Work

As far as we know, Vestal [16] firstly identified and formalized the mixed-criticality scheduling problem and proposed a fixed-priority scheduling algorithm based on "Audsley approach" [1] for such systems on a preemptive uni-processor platform. This algorithm was then improved by integrating EDF to schedule tasks assigned to the same priority level [4]. Recently, Baruah et al. [5] studied the response time analysis of fixed-priority scheduling algorithm and disclosed the effect of different level of system support to the schedulability. These works are in the framework of traditional real-time scheduling techniques like EDF and fixed-priority scheduling, which may lead to poor performance for mixed-criticality systems.

Baruah et al. [3] conducted a series of fundamental studies of the mixed-criticality scheduling problem, starting from a simple setting of a finite collection of mixed-criticality jobs with known release times. It is proven that the problem is highly intractable even with this very simple setting [3]. As an efficient heuristic, they proposed the OCBP (Own Criticality Based Priority) [3] algorithm, which has the optimal speedup factor 1.618 among all fixed-job-priority algorithms for this model [2]. As introduced in last section, the OCBP approach is extended to sporadic task systems, but with a pseudo-polynomial complexity at run-time [12], and later is improved to quadratic complexity [9]. One major contribution of our work is to further improve the OCBP approach for linear complexity at run-time.

Another line of work uses the EDF-VD (EDF with Virtual Deadlines) approach to schedule mixed-criticality periodic/sporadic task systems. For the special case where all tasks' relative deadlines equal to their periods, EDF-VD can achieve a better speedup factor than OCBP, but in general OCBP still has the best known speedup factor. Ekberg and Yi [8] extended the demand-bound function [6] analysis techniques for mixed-criticality sporadic tasks, and improved the EDF-based schedulability by the similar idea with EDF-VD.

De Niz et al. [7] developed the Zero-Slack algorithm to schedule mixed-criticality sporadic task system. The key idea is to keep the high-criticality jobs from the interference of low-criticality jobs by adjusting jobs' priorities dynamically. This approach has been extended to handle non-preemptable shared resources platforms [11] and distributed platforms where the mixed-criticality workload needs to be allocated to different execution units [10].

For the multiprocessor setting, Mollison et al. [14] encapsulated tasks with different criticality levels into containers, and used a criticality monotonic priority assignment scheme to schedule these containers on multicore systems. Pathan [15] extended the traditional global multiprocessor scheduling techniques to mixed-criticality systems. Li and Baruah [13] presented the fp-EDF algorithm for mixed-criticality implicit-deadline sporadic task systems based on the EDF-VD approach.

III. System Model and Definitions

In this section we formally define the mixed-criticality (MC) sporadic task model on a preemptive single processor system. Similar with the traditional sporadic task model, we define an MC sporadic system as a finite set of independent MC sporadic tasks, each of which generates a potentially infinite sequence of MC jobs.

A. MC Tasks and MC Jobs

Each MC task is characterized by an 4-tuple $\tau_i = (T_i, D_i, C_i, \zeta_i)$, where

- $T_i \in R^+$ is the minimal inter-arrival separation (period).
- $D_i \in R^+$ is the relative deadline.
- $C_i : N^+ \to R^+$ is the WCET function, specifying the WCET of the task at each criticality level: $C_i(\ell)$ denotes the WCET of task τ_i at criticality level ℓ.
- $\zeta_i \in \{1, 2, \ldots, L\}$ is the criticality level of task τ_i, where a greater value indicates a higher criticality and L is the total number of criticality levels in the system.

Note that relative deadlines can be arbitrary positive real numbers without any restrictions regarding the task period, i.e., D_i can be larger than, smaller than or equal to T_i.

The jth job generated by task τ_i is denoted by J_i^j, which is characterized by its release time $r_i^j \in R^+$. Further, $d_i^j = r_i^j + D_i$ is the absolute deadline of J_i^j and we use $f_i^j : r_i^j < f_i^j \le d_i^j$ to denote the finish time of J_i^j. Note that all jobs of τ_i have the same WCET function C_i and criticality level ζ_i.

The LPA scheduling algorithm of this paper is based on job-level priority, i.e., each job has a fixed-priority. Following the convention in real-time scheduling literature, we use a smaller priority value to represent a higher priority.

B. MC Sporadic Task System Run-time Behavior

The semantic of the MC job is as follows. An MC job J_i^j is released by task τ_i at time instant r_i^j, and needs to execute for γ_i^j time units. The exact value of γ_i^j is not revealed until J_i^j signals that it has completed execution. The values of γ_i^j measured from a given run defines the kind of behavior

exhibited by the MC system during that run. We say that the MC system exhibits $\lambda-criticality$ behavior, where

$$\lambda = \max_{\forall \gamma_i^j} \left\{ \ell | \min \left\{ \ell | \gamma_i^j \leq C_i(\ell) \right\} \right\}. \quad (1)$$

In particular, if any job J_i^j has executed for $C_i(L)$ without signaling completion, we define the behavior of such run as erroneous. In the following of this paper, we assume that the system will not exhibit erroneous behavior.

Equation (1) implies that for each run-time job J_i^j, a longer execution time γ_i^j leads to a potentially higher-criticality behavior of it. Further, the jobs with the highest-criticality behavior determine the criticality of the system behavior.

C. MC Schedulability

The schedulability of an MC system depends on the certifications on each system behavior level. Based on this principle, we give the definition of MC-schedulability under a scheduling algorithm $\mathcal{A}$ as below:

Definition III.1 (MC-schedulability)**.** Given a scheduling algorithm $\mathcal{A}$, an MC task system π is MC-schedulable by $\mathcal{A}$ iff the following implication holds for each λ-criticality system behavior:

$$\forall \tau_i : \zeta_i \geq \lambda \Rightarrow \forall J_i^j : J_i^j \text{ has finished by } d_i^j.$$

If the system exhibits λ-behavior, all the jobs with criticality lower than λ do not need to meet their deadlines. The system designers guarantee the temporal correctness of an MC task τ_i only in the assumption that the criticality of the system behavior does not exceed ζ_i, and it is not required for the jobs generated by τ_i to meet their deadlines in higher-criticality system behaviors.

IV. The New Algorithm LPA

The common reason of PLRS and LB's efficiency problem is that they perform heavy priority re-computations for the whole MC task set at each time instant when a preemption occurs. Some priority assignment plans may be recomputed for many times before the scheduler actually uses them (some of them may be even unused until the current busy period is over), which leads to a lot of redundant computation that are actually irrelevant to the actual scheduling decisions.

In this paper, we propose a new algorithm LPA (Lazy Priorities Adjustment). As suggested by its name, LPA adjusts priorities as lazy as possible, in order to as much as possible avoid redundant priority re-computations at preemption points. When a preemption occurs, LPA does not immediately perform the heavy priority adjustment for all the tasks. Instead, it only lets each task to record this event. Based on the prior preemption records, LPA performs a lightweight priority adjustment for the jobs of task τ_i when a new job J_i^c is released. In this way, LPA avoids the heavy but unnecessary online priority re-computations, but only performs lightweight adjustment when a job is released.

Furthermore, we also propose a new approach to calculate a tighter upper bound of the maximal size of busy periods for an MC sporadic task. By this tighter bound we not only reduce the online space overhead but also further improve the schedulability, in terms of acceptance ratio, comparing with PLRS and LB.

A. Offline Priority Assignment

Similar with LB and PLRS, LPA uses the notion of busy period to solve the problem of priority assignment for infinitely many jobs released by an MC sporadic task system: at any time, one only needs to consider the priority assignment for finitely many jobs that may be released in a busy period. This is because by work-conserving scheduling algorithms the processor will be idle if and only if there are no released jobs without completing execution or being terminated (e.g., system criticality upgrading causes all of the lower-criticality jobs to be terminated), and there must be an idle interval between any two successive busy periods. Therefore, no job released before a busy period can affect the scheduling of jobs released in that busy period.

The first step of the offline priority assignment is to compute the maximal size of busy periods for an MC sporadic task system. Li and Baruah [12] introduced a method to compute an upper bound of the maximal busy period size. In this paper, we will derive a tighter upper bound of the maximal busy period size, which will be discussed in subsection IV-E.

We use I to denote the set of jobs that can be released in a busy period, and $n_i = \left\lceil \frac{\Gamma}{T_i} \right\rceil$ (Γ denotes the upper bound of busy period size) to denote the number of task τ_i's jobs in I. The total number of jobs in I is denoted by $n = \sum_{\tau_i \in \pi} n_i$.

LPA's offline algorithm assigns priorities to all the jobs in I based on the OCBP principle. We use a variable δ_i to denote the number of task τ_i's jobs that have not been assigned a priority yet. Initially, $\delta_i = n_i$. At each step, the algorithm checks each task's largest-index job to find a job $J_k^{\delta_k}$ which can be assigned the lowest priority among all the jobs that have not been assigned a priority, by examining whether the following condition is satisfied:

$$\sum_{\tau_j \in \pi} (C_j(\zeta_k) \cdot \delta_j) \leq T_k \cdot (\delta_k - 1) + D_k \quad (2)$$

The LHS of the inequality represents the total workload of all the jobs that will get priority higher than or equal to $J_k^{\delta_k}$ under the ζ_k-criticality system behavior, and the RHS is the minimal distance between the absolute deadline of $J_k^{\delta_k}$ and the beginning of the current busy period. Generally, the algorithm may find more than one eligible jobs in each step, and we can select any of them to assign the current lowest priority. After assigning the lowest priority to $J_k^{\delta_k}$, we reduce δ_k to $\delta_k - 1$ and repeat the former procedure until no jobs left without being assigned a priority or at some iteration no job is eligible to get the lowest priority. If each job in I has been assigned a priority successfully, we say that the offline algorithm of LPA succeeds, otherwise, it is a failure. We use a table Λ to record the results of the offline priority assignment algorithm.

According to the offline priority assignment algorithm, we have the follow lemma:

Table I
AN EXAMPLE TASK SET

$Task$	T_i	D_i	ζ_i	$C_i(1)$	$C_i(2)$
τ_1	10	10	1(LOW)	1	1
τ_2	20	20	2(HIGH)	1	2
τ_3	30	30	1(LOW)	15	15
τ_4	50	50	2(HIGH)	15	25

Lemma IV.1. *For any two jobs J_i^m and J_i^n of task τ_i in the same busy period, if $n_i > m > n \geq 0$ then $\Lambda_i(m) > \Lambda_i(n)$.*

Example IV.2. Consider the MC task set in Table I. We assume $n_1 = 5, n_2 = 3, n_3 = 2$ and $n_4 = 1$[1]. At the first step, we check the candidate jobs $\left\{J_1^5, J_2^3, J_3^2, J_4^1\right\}$ for the lowest priority. We first check job J_1^5. Since $\zeta_1 = 1$, the LHS of Condition (2) equals:

$$C_1(1) \times \delta_1 + C_2(1) \times \delta_2 + C_3(1) \times \delta_3 + C_4(1) \times \delta_4 = 53.$$

On the other hand, the RHS of Condition (2) is:

$$T_1 \times (\delta_1 - 1) + D_1 = 50 < 53.$$

Since Condition (2) is false for J_1^5, we go to the next candidate J_2^3. Since $\zeta_2 = 2$, the LHS equals: $\sum_{\forall \tau_i} C_i(2) \cdot \delta_i = 66$, but the RHS is 60. So $J_2^3$is not eligible for the lowest priority either, and we go to the next candidate J_3^2. For J_3^2, $LHS = 53$ and $RHS = 60$, so Condition (2) is true and it can be assigned the lowest priority 11 at that step. Then we set $\delta_3 = \delta_3 - 1 = 1$ and go into the next iteration. Following the steps above, we can construct the priority assignment for all the jobs that can be released in a busy period as below:

Λ_1	1	2	4	8	9
Λ_2	3	6	10		
Λ_3	5	11			
Λ_4	7				

Note that the priority assignment obtained in the offline phase of LPA cannot be directly used for online priority assignment. Simply following this plan to assign priorities to jobs that are actually released at run-time does not guarantee the system schedulability. This information is just a reference for actual online priority assignment. The run-time scheduler of LPA will adjust the priority assignment based on current system state information.

B. Run-time Scheduling

We cannot directly use the offline constructed priority assignment plan to assign priorities to jobs released by the system at run-time. This is because that our offline priority assignment assumes that all tasks release their first jobs at the beginning of the current busy period simultaneously and the successive jobs J_i^{c+1} are released exactly at $r_i^c + T_i$, but in sporadic task systems a job may be released at any time instant, as long as the inter-release separation constraint is satisfied. Now we introduce our run-time scheduling of LPA to solve this problem.

[1]The potential number of jobs that can be released in same busy period is actually larger than our assumption. Here we only consider a small subset of them for demonstration.

LPA is a job-level fixed-priority preemptive scheduling algorithm. The system criticality ℓ is assigned to the lowest value 1 when the system starts execution and at any time instant when the processor becomes idle. The system criticality ℓ will be increased to $\ell + 1$ when any job J_i^c executes for more than $C_i(\ell)$ time units without signaling completion. At any time, LPA always only assigns the processor to the highest priority unfinished job among the ones whose criticality levels are equal to or higher than current system criticality. It implies that all the tasks satisfying $\zeta_i \leq \ell$ will be discarded immediately when system criticality upgrades to $\ell + 1$.

For each MC task τ_i, we use a variable idx_i to indicate the sequence number of the coming job J_i^c in the current busy period. We do not directly use $\Lambda_i(c-1)$ for its priority (the index of $\Lambda_i()$ starts from 0, so the cth element in Λ_i is $\Lambda_i(c-1)$). Instead, we use an adjusted value $\Lambda_i(c - \alpha_i) : 0 \leq \alpha_i \leq c$. The key goal of our new algorithm LPA is to efficiently calculate a proper offset variable α_i for each released job J_i^c to assign its run-time priority $prt(J_i^c)$ in the current busy period.

When system starts a new busy period, LPA resets idx_i and α_i to 1 for each task τ_i, i.e., in the current busy period, the first coming job J_s^1 gets the first value $\Lambda_s(0)$ of Λ_s for its priority and $\alpha_i = 1$ will be used to calculate future jobs' priorities until some run-time preemption occurs. Unlike LB and PLRS, LPA does not adjust priorities for all the future jobs at every preemption point. Instead, we define a variable δ_i for each task τ_i to record the maximal priority value of the jobs which are preempted during $(r_i^{c-1}, r_i^c]$, and only do *lightweight* adjustment when a future job is released. This is the key for LPA to have *linear* run-time complexity.

To calculate the proper α_i for each job of task τ_i, LPA also maintains an auxiliary set $\Omega_i = \{(x_1, y_1), (x_2, y_2), \cdots\}$. For each tuple (x_j, y_j) in Ω_i, x_j records a previously used α_i' and y_j records the preemption value δ_i' when this tuple was inserted. The pseudo codes of the online priority assignment routine **AdjustPrt** is shown in Figure 1.

When a job J_k^c is released, LPA executes the priority adjustment routine **AdjustPrt** to check whether all the previously released jobs are finished. If yes, a new busy period starts, and LPA resets the scheduling variables to the initial values. Otherwise, if the criticality of the released job is not lower than current system criticality, LPA will execute **AdjustPrt** to assign it an appropriate priority and add it to the ready queue.

AdjustPrt first uses α' to record the current offset variable α_k, then assigns J_k^c the priority $P_k = \Lambda_k(idx_k - \alpha_k)$ and compares P_k with the maximal between δ_k and current running job's priority value. If P_k is higher (the value is smaller), **AdjustPrt** increases α_k to idx_k and assigns $\Lambda_k(0)$ to P_k. Then the routine checks the preemption condition by comparing P_k with the currently running job's priority. If P_k is higher, each task's preemption record variable δ_i will be updated as shown at line 13 of the **AdjustPrt** algorithm.

The **Modify** function is used to update the auxiliary set

1: **if** there are no unterminated jobs except J_k^c **then**
2: **for** each $\tau_i \in \pi$ **do**
3: $(idx_i, \alpha_i, \delta_i, \Omega_i) \leftarrow (1, 1, 0, \emptyset)$
4: **end for**
5: $prt(J_k^c) \leftarrow \Lambda_k(0);\ idx_k \leftarrow idx_k + 1$
6: **else**
7: $P_{cur} \leftarrow priority\ of\ the\ current\ job$
8: $\alpha' \leftarrow \alpha_k;\ P_k \leftarrow \Lambda_k(idx_k - \alpha')$
9: **if** $P_k < \max\{\delta_k, P_{cur}\}$ **then**
10: $\alpha_k \leftarrow idx_k;\ P_k \leftarrow \Lambda_k(0)$
11: **end if**
12: **if** $P_k < P_{cur}$ **then**
13: update $\delta_i \leftarrow \max\{\delta_i, P_{cur}\}$ for $\forall \tau_i$
14: **end if**
15: Modify($\Omega_k, \alpha', \alpha_k, \delta_k$) {check and update Ω_k}
16: $prt(J_k^c) \leftarrow P_k;\ (idx_k, \delta_k) \leftarrow (idx_k + 1, 0)$
17: **end if**

Figure 1. The online priority assignment routine AdjustPrt.

Modify($\Omega_k, \alpha', \alpha_k, \delta_k$)
1: **if** $\alpha' < \alpha_k$ **then**
2: **for** each (x_i, y_i) in Ω_k satisfies $y_i \le \delta_k$ **do**
3: remove (x_i, y_i) from Ω_k
4: $\alpha' = \min\{\alpha', x_i\}$
5: **end for**
6: add (α', δ_k) to Ω_k
7: **end if**
8: **for** (x_i, y_i) in Ω_k satisfies $y_i \le \Lambda_k(idx_k + 1 - \alpha_i)$ **do**
9: $\alpha_k = \min\{\alpha_k, x_i\}$
10: **end for**
11: remove any tuple (x_i, y_i) satisfying $x_i \ge \alpha_k$ from Ω_k

Figure 2. The Modify function invoked in AdjustPrt.

Ω_k and offset variable α_k. It comprises two parts which are detailed in Figure 2. In the first part, starting from line 1 and ending at line 7, if the previous offset value α' is less than the current value α_k, i.e., α_k is updated just now, the previous value α' needs to be stored in Ω_k for reusing in future. From line 2 to line 5, the **Modify** function removes all of the tuples in which the recorded preemption δ_i is not greater than current preemption value δ_k, and meanwhile, updates α' to the minimum recorded offset value of the removed tuples. Then the **Modify** function checks whether the current offset variable α_k needs to be replaced by some previous offset value in Ω_k following line 8 to line 10, then removes all of the tuples whose recorded offset value α_i is no less than the current offset value α_k.

Finally, **AdjustPrt** assigns P_k as J_k^c's run-time priority and updates idx_k and δ_k to $idx_k + 1$ and 0 respectively.

Any MC sporadic task set that succeeds with the offline priority assignment of LPA is guaranteed to be MC-schedulable by the run-time scheduling of LPA, the proof of which is shown in the appendix.

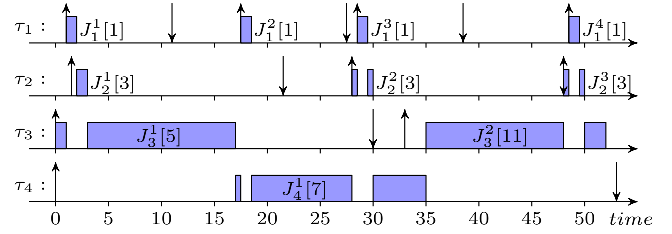

Figure 3. An example of scheduling sequence during a busy period

C. A Running Example

We use the MC task system in Table I and the priority plan computed in Example IV.2 to illustrate how $AdjustPrt(J_i^c)$ works in a busy period BI, as shown in Figure 3.

Without loss of generality we assume that the new busy period starts at time 0, at which time point τ_3 and τ_4 release their first job simultaneously[2] and all of the scheduling parameters are reset to the initial values, i.e., $\forall \tau_i \in \pi$: $idx_i = 1, \alpha' = \alpha_i = 1, \delta_i = 0, \Omega_i = \emptyset$. According to **AdjustPrt**, J_3^1 gets the priority $\Lambda_3(0) = 5$, which is higher than $prt(J_4) = \Lambda_4(0) = 7$, thus J_3^1 starts execution. The following released jobs J_1^1 and J_2^1 get priorities $\Lambda_1(0) = 1$ and $\Lambda_2(0) = 3$ respectively. For the jobs released so far, **AdjustPrt** does not modify the auxiliary set Ω_i and offset variable α_i, because $\alpha' = \alpha_i$.

When J_1^2, the second job of task τ_1, is released at time 17.5, the *current* offset variable $\alpha' = \alpha_1 : 1$ and $AdjustPrt(J_1^2)$ tries to set its priority to $\Lambda_1(idx_1 - \alpha') = 2$, which is higher than the running job J_4^1's priority 7, so this function adjusts current offset variable $\alpha_1 = idx_1 : 2$ and reassigns $P_1 = \Lambda_1(0) : 1 < prt(J_4^1) : 7$, then updates the preemption record parameter δ_i to $\max\{\delta_i, 7\}$ for each task τ_i. Since $\alpha_1 = 2 > 1 = \alpha'$, the **Modify** function adds the tuple $(\alpha' : 1, \delta_1 : 7)$ to set Ω_1.

Similar operations are conducted when J_2^2 is released at time 28. At the beginning of $AdjustPrt(J_2^2)$, $\alpha' = \alpha_2 : 1$, $prt(J_4^1) = 7$, $\Omega_2 = \{\}$, $\Lambda_2(idx_2 - \alpha_2) = 6 < \delta_2 = 7$, so the routine sets $\alpha_2 = idx_2 : 2$ and $P_2 = \Lambda_2(0) : 3$ at line 10, then updates the preemption records for each task. Since $\alpha_2 > \alpha'$, the **Modify** function executes line 2 to line 6 to add $(\alpha' : 1, \delta_2 : 7)$ to Ω_2. However, since $\Lambda_2(idx_2 : 2 + 1 - x_1 : 1) = 10 > y_1 : 7$, this function resets $\alpha_2 = x_1 : 1$ and remove $(1, 7)$ from Ω_2, i.e., $\Omega_2 \leftarrow \emptyset$, following line 8 to line 11 in Figure 2.

D. Run-time Complexity

In this section, we discuss the run-time complexity of **AdjustPrt**. It is obvious that there are only two non-nested loops (line 2 to line 4, and line 13) in Figure 1, and each loop only iterates for N (the number of tasks) times. In the

[2]In this example, we assume the scheduler firstly receives the releasing event of J_3^1, and, of course, it may treat J_4^1 as the first job during the current busy period. However, it does not impact the priority assignment results in this time instant.

following we will show that the complexity of the **Modify** function is also $O(N)$, and thereby the overall complexity of **AdjustPrt** is $O(N)$.

Lemma IV.3. *At any time the number of elements in each* $\Omega_i : \tau_i \in \pi$ *is no more than* N.

Proof: Following the run-time scheduling policy of LPA, once a new element (x, y) is inserted to Ω_i at r_i^c, there must exist a job J_k^p preempted at time $t \in (r_i^{c-1}, r_i^c]$ during the current busy period, and no job with priority lower than $prt(J_k^p)$ was preempted during $(r_i^{c-1}, r_i^c]$.

Then we prove by contradiction. Suppose at a time instant t^1 in the current busy period, Ω_i gets the $(N+1)^{st}$ element (x_{N+1}, y_{N+1}), and the related job is denoted by J^1. Since the total number of tasks is N, there must exist a previous inserted element (x_m, y_m) whose relevant job J^2 was generated by the same task as J^1 at a time instant t^2 during the current busy period. Since the element (x_m, y_m) is still contained in Ω_i at t^1, by the definition of **Modify** function in Figure 2, we know that no job with priority lower than y_m was preempted during the time interval $[t^2, t^1]$. Therefore, each job J that has executed during $[t^2, t^1]$ satisfies $prt(J) < prt(J^2)$, thus, $prt(J^1) < prt(J^2)$. By Lemma VI.5, J^2 cannot be active at the release time r^1 of J^1, i.e., J^2 must have finished its execution during (t^2, t^1). However, when J^2 finishes execution, the current busy period will be over unless another active job with a lower priority executes. It contradicts with $prt(J) < prt(J^2)$. So the lemma is proved. ■

Theorem IV.4. *The run-time priority management of LPA is of complexity* $O(N)$.

Proof: By Lemma IV.3, we know that at any time the maximal number of elements in each Ω_i is N (the number of tasks in the system). Therefore the run-time complexity of the three Ω_i-size based loops (line 2 to line 5, line 8 to line 10, and line 11 in Figure 2) is $O(N)$. Combining the complexity analysis of the other parts of **AdjustPrt**, the overall time complexity of the online priority management of LPA is $O(N)$. ■

E. Bounding the Busy Period Size

In [12], Li and Baruah introduced an upper bound of the maximal busy period size based on the load metric for dual-criticality systems. Their bound is safe, but grossly over-approximate. Since the OCBP approach (all of LB, PLRS and LPA) needs to store the offline constructed priority assignment, a more precise bound is important to reduce the run-time space overhead. Actually, a more precise busy period size bound not only improves the run-time space efficiency, but also improves the system schedulability as will be discussed later in this subsection.

In this paper, we propose an efficient algorithm to compute a tighter upper bound of the maximal busy period size for systems with an arbitrary number of criticality levels.

Definition IV.5. The *criticality*-ℓ workload bound Γ_ℓ of an MC task system π denotes an upper bound of the total workload of the tasks whose criticality levels are equal or lower than ℓ, in any busy period where the system criticality level do not exceed ℓ. Note that Γ_L is equal to the busy period size upper bound of task system π. In particular, we define $\Gamma_0 = 0$.

Theorem IV.6. *Given an MC sporadic task system* π *and the criticality-(*ℓ*-1) workload bound* $\Gamma_{\ell-1}$*, we have*

$$\text{Let } \phi_\ell = \frac{\Gamma_{\ell-1} + \sum_{\zeta_i \geq \ell} C_i(\ell)}{1 - \sum_{\zeta_i \geq \ell} \frac{C_i(\ell)}{T_i}} \quad (3)$$

$$\Gamma_\ell = \Gamma_{\ell-1} + \sum_{\zeta_i = \ell} C_i(\ell) \cdot \left(1 + \left\lfloor \frac{\phi_\ell}{T_i} \right\rfloor\right) \quad (4)$$

Proof: Consider an arbitrary busy period BI, starting at t_s, in which the system criticality level upgrades to ℓ at some time instant t^* and keep this level until BI is over at t_e. We use W_i to denote the workload of task τ_i in BI, and divide the total workload into $W^{l-} = \sum_{\forall \zeta_i < \ell} W_i$, $W^\ell = \sum_{\forall \zeta_i = \ell} W_i$ and $W^{\ell+} = \sum_{\forall \zeta_i > \ell} W_i$. Because $W^{\ell-}$ can only be accumulated from $[t_s, t^*)$ which must be in some busy period in which the system criticality level keep from going beyond $\ell - 1$, it is clear that

$$W^{\ell-} \leq \Gamma_{\ell-1}. \quad (5)$$

Then, we prove by contradictions. We use bl to denote the length of BI, and it is clear that $bl = W_m^{\ell-} + W_m^\ell + W_m^{\ell+}$. By (5), we have that

$$\begin{aligned}
\sum_{\forall \zeta_i = \ell} C_i(\ell) \cdot (1 + \left\lfloor \frac{bl}{T_i} \right\rfloor) &\geq W_m^\ell \\
&> \sum_{\forall \zeta_i = \ell} C_i(\ell) \cdot \left(1 + \left\lfloor \frac{\phi}{T_i} \right\rfloor\right)
\end{aligned}$$

$$\begin{aligned}
\Rightarrow \quad bl \quad &> \quad \phi \\
\Leftrightarrow \quad bl \quad &> \quad \frac{\Gamma_{\ell-1} + \sum_{\forall \zeta_i \geq \ell} C_i(\ell)}{1 - \sum_{\forall \zeta_i \geq \ell} \frac{C_i(\ell)}{T_i}} \\
\Leftrightarrow \quad bl \quad &> \quad \Gamma_{\ell-1} + \sum_{\forall \zeta_{i \geq \ell}} C_i(\ell) \cdot \left(1 + \frac{bl}{T_i}\right) \\
&\geq \quad \Gamma_{\ell-1} + \sum_{\forall \zeta_{i \geq \ell}} C_i(\ell) \cdot \left(1 + \left\lfloor \frac{bl}{T_i} \right\rfloor\right) \\
&\geq \quad W^{\ell-} + W^\ell + W^{\ell+} \\
\Leftrightarrow \quad & bl > bl
\end{aligned}$$

The contradiction proves the theorem. ■

By Theorem IV.6, we give a recursive algorithm to bound the longest busy period through bounding Γ_L as shown in Figure 4.

Comparing with the bound computed in [12], we can also reduce the number of jobs for each task τ_i during same busy

ComputeGamma(ℓ)

1: **if** $\ell = 0$ **then**
2: **return** 0
3: **end if**
4: $\Gamma \leftarrow ComputeGamma(\ell - 1)$
5: $\gamma \leftarrow \dfrac{\Gamma + \sum_{\zeta_i \geq \ell} C_i(\ell)}{1 - \sum_{\zeta_i \geq \ell} \dfrac{C_i(\ell)}{T_i}}$
6: **return** $\Gamma_{\ell-1} + \sum_{\zeta_i = \ell} C_i(\ell) \cdot \left(1 + \left\lfloor \dfrac{\gamma_\ell}{T_i} \right\rfloor\right)$

Figure 4. Compute $Criticality\text{-}\ell$ work load.

period by parameter ϕ_ℓ in (3), as follow

$$N_i = \left\lceil \frac{\phi_{\zeta_i}}{T_i} \right\rceil. \tag{6}$$

It is easy to see our tighter busy period size bound can improve the system space efficiency at run-time. However, it is a bit surprising that this also leads to improved system schedulability. We use the following example to illustrate this. Consider the MC task system comprises two MC tasks :

$Task$	T_i	D_i	ζ_i	$C_i(1)$	$C_i(2)$
τ_1	15	15	2	8	14
τ_2	80	80	1	9	9

According to the approach in LB [12], we can compute the upper bound of busy period length $bl = 3309$. And in such a larger interval, it contains 221 high-criticality jobs from τ_1 and 42 low-criticality jobs from τ_2. Since the utilization of high-criticality task τ_1 is very high (93.33%), jobs of τ_1 cannot afford too many low-criticality jobs to have higher priorities. So at the beginning of offline priority assignment routine, it gives low-priorities to low-criticality jobs. However, a larger number of higher-priority high-criticality jobs will cause the low-criticality jobs to miss their deadlines as well. In our experiment, when the system remains 215 high-criticality jobs of τ_1 and 24 low-criticality jobs of τ_2, we have:

$$C_1(2) \cdot 215 + C_2(2) \cdot 24 = 3226 > T_1 \cdot 214 + D_1 = 3225 \tag{7}$$

$$C_1(1) \cdot 215 + C_2(1) \cdot 24 = 1936 > T_2 \cdot 23 + D_2 = 1920. \tag{8}$$

So both J_1^{215} and J_2^{24} cannot be assigned the current lowest priority, which causes the schedulability test failure. In this case, we can find that $d_2^{24} = 1920$ and τ_1 can release $\lceil \frac{1920}{15} \rceil =$ 128 jobs at most, but the test condition (7) counts 215 jobs from τ_1 which includes too much workload overestimation.

However, with our improved method, we can compute a much tighter upper bound $bl_{HI} = 345$, and it only comprises 23 high criticality jobs from τ_1 and 5 low criticality jobs from τ_2. It reduces the overestimated workload significantly and can successfully complete the offline priority assignment to this task set (therefore, this task set is MC-schedulable by LPA).

V. Experimental Evaluation

In this section we evaluate the performance improvement of LPA over the state-of-the-art OCBP-based algorithm PLRS, in online time efficiency, online space efficiency as well as schedulability. Our experiments use dual-criticality implicit deadline sporadic task model on a uni-processor platform.

We follows the approach in [8] to generate random MC task sets. A task set is generated by starting with an empty task set $\pi = \emptyset$, to which random tasks are successively added. The generation of a random task is controlled by four parameters: the probability P_{HI} of being of high-criticality, the maximal ratio R_{HI} between high- and low-criticality execution time of each high-criticality task, the maximal low-criticality execution time C_{LO}^{max} and the maximal period T^{max}. Each new task τ_i is generated as follows:

- $\zeta_i = HI$ with probability P_{HI}, otherwise $\zeta_i = LO$.
- $C_i(LO)$ is drawn from the uniform distribution over $\{1, 2, \cdots, C_{LO}^{max}\}$.
- $C_i(HI)$ is drawn from the uniform distribution over $\{C_i(LO), C_i(LO) + 1, \cdots, R_{HI} \cdot C_i(LO)\}$ if $\zeta_i = HI$. Otherwise, $C_i(HI) = C_i(LO)$.
- T_i is drawn from the uniform distribution over $\{C_i(\zeta_i), C_i(\zeta_i) + 1, \cdots, T^{max}\}$.
- $D_i = T_i$ since deadlines are implicit.

We define the *average utilization* of a dual-criticality task set π as

$$U_{avg}(\pi) = \frac{U_{LO}(\pi) + U_{HI}(\pi)}{2}.$$

Each task set is generated with a target average utilization U_{min}^* with a acceptable range of errors: $U_{min}^* = U^* - 0.005$ and $U_{max}^* = U^* + 0.005$.

As long as $U_{avg}(\pi) < U_{min}^*$, we generate more tasks and add them to π. If a task is added such that $U_{avg}(\pi) > U_{max}^*$, we discard the whole task set and start with a new empty task set. If a task is added such that $U_{min}^* \leq U_{avg}(\pi) \leq U_{max}^*$, the task set is finished, unless all tasks in π have the same criticality level or $U_{LO}(\pi), U_{HI}(\pi) > 0.99$, in which case the task set is instead discarded. The random task sets for each experiment are generated with parameters $P_{HI} = 0.5$, $R_{HI} = 2$, $C_{LO}^{max} = 10$ and $T^{max} = 100$.

We first evaluate the online scheduling time overheads by simulating the scheduling procedures. For each random task set, the simulator generated 1,000 randomly released sporadic jobs to count the time overhead. We compare the time overhead of LPA and PLRS with the following two metrics:

- **Total overhead** denotes the sum of measured time overhead of these 1,000 jobs in the simulation.
- **Maximal overhead** denotes the maximal measured time overhead among these 1,000 jobs in the same simulation.

Figure 5 compares the online time overheads of LPA and PLRS. Each point in the figure includes at least 5,000 random task sets. The $x\text{-}axis$ is the average utilization and the $y\text{-}axis$ is the ratio between the total and maximal time overhead of LPA and PLRS. For example, considering the point (0.81, 0.29) on the curve labeled *Maximal overhead*, the $x\text{-}value$ 0.81 means the target utilization of the randomly generated task sets is 81%. The $y\text{-}value$ 0.29 means that on average the maximal online time overhead of LPA is 29% of the maximal

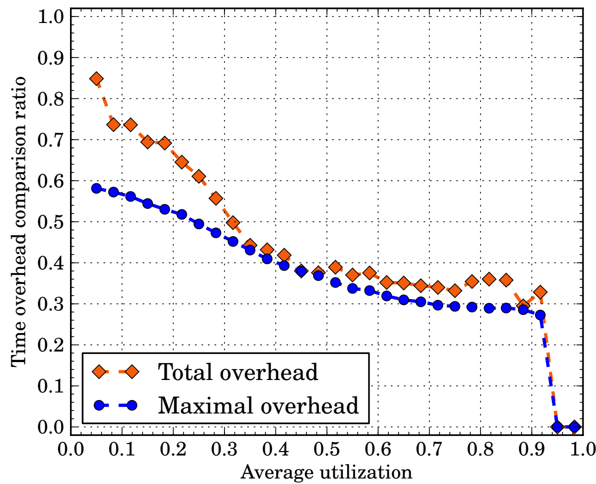

Figure 5. $P_{HI} = 0.5$, $R_{HI} = 2$, $C_{LO}^{max} = 10$ and $T^{max} = 100$

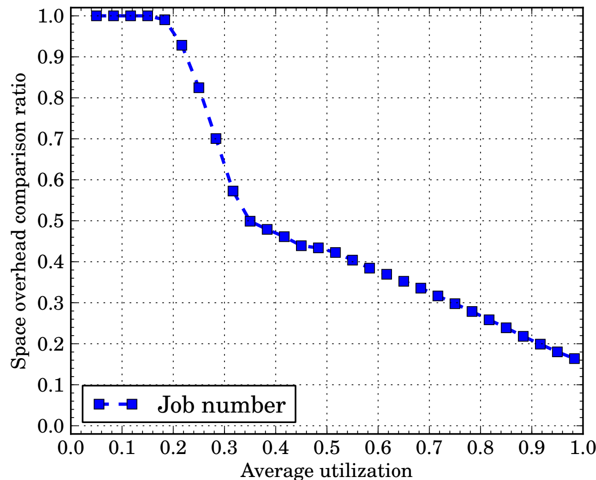

Figure 6. $P_{HI} = 0.5$, $R_{HI} = 2$, $C_{LO}^{max} = 10$ and $T^{max} = 100$

online time overhead of PLRS. We can see that LPA can significantly improve the online time efficiency in both the total time overhead and maximal time overhead over PLRS, especially for task sets with larger utilization.

We evaluate the space overheads by counting the number of jobs that may be released in a busy period, i.e., the size of the priority list Λ. Each point in Figure 6 includes at least 5,000 random task sets. The x-$axis$ is the average utilization and the y-$axis$ is the ratio between the number of jobs that may be released in a busy period under LPA and under PLRS. We can see that LPA can significantly improve the online space efficiency over PLRS as LPA uses a much shorter priority list Λ at run-time. The advantage of LPA is greater for task sets with larger utilization.

In the last experiment, we compare the acceptance ratio of our improved algorithm LPA and PLRS. The simulation result is shown in Figure 7. The x-$axis$ is the average utilization and the y-$axis$ is the acceptance ratio, i.e., the portion of schedulable task sets out of all the random task sets generated

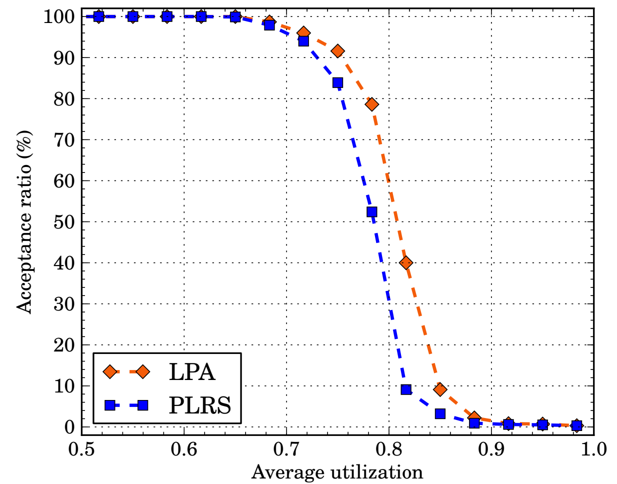

Figure 7. $P_{HI} = 0.5$, $R_{HI} = 2$, $C_{LO}^{max} = 10$ and $T^{max} = 100$

at this utilization range. Each point in this figure is based on 10,000 random task sets. From the figure we can see that our algorithm LPA improves the schedulability over PLRS. This is because our tighter maximal busy period size bound can reduce the pessimism in the workload calculation by excluding many jobs that are not possible to execute before the deadline of the analyzed job. The detailed explanation of this point is provided in Section IV-E.

VI. Conclusion

In this paper we propose an OCBP-based scheduling algorithm LPA, to schedule mixed-criticality sporadic task systems. Comparing with the previous OCBP-based algorithms, it can improve the online time efficiency, online space efficiency, as well as schedulability. The central idea of LPA is to make online priority adjustment as lazy as possible, in order to avoid redundant priority adjustments that are not relevant to the actual scheduling decisions. Experiments with synthetic workloads show the performance improvement of our new algorithm in online time efficiency, online space efficiency and schedulability.

Acknowledgement

Supported in part by "China Fundamental Research Funds for the Central Universities" under grant No. N100204001 and N110804003; and "China Research Fund for the Doctoral Program of Higher Education" under grant No. 20110042110021.

References

[1] N. C. Audsley, *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times.* Citeseer, 1991.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1140–1152, 2012.

[3] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010, pp. 13–22.

[4] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, 2008, pp. 147–155.

[5] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *the 32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011, pp. 34–43.
[6] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *the 11th IEEE Real-Time Systems Symposium (RTSS)*, 1990, pp. 182–190.
[7] D. de Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009, pp. 291–300.
[8] P. Ekberg and W. Yi, "Outstanding paper award: Bounding and shaping the demand of mixed-criticality sporadic tasks," in *the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 135–144.
[9] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems," in *the 32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011, pp. 13–23.
[10] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Mixed-criticality task synchronization in zero-slack scheduling," in *the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011, pp. 47–56.
[11] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno, "Resource allocation in distributed mixed-criticality cyber-physical systems," in *the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2010, pp. 169–178.
[12] H. Li and S. Baruah, "An algorithm for scheduling certifiable mixed-criticality sporadic task systems," in *the 31st IEEE Real-Time Systems Symposium (RTSS)*, 2010, pp. 183–192.
[13] ——, "Outstanding paper award: Global mixed-criticality scheduling on multiprocessors," in *the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 166–175.
[14] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *the 10th IEEE International Conference on Computer and Information Technology (CIT)*, 2010, pp. 1864–1871.
[15] R. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 309–320.
[16] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *the 28th IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 239–243.

APPENDIX: CORRECTNESS OF LPA

We will prove that any MC task set that succeeds with the offline priority assignment of LPA is MC-schedulable by the online scheduling of LPA.

Definition VI.1 (Busy Set)**.** During a certain run-time busy period, suppose there exists a job $J_i^\alpha : \alpha \in N$ getting $\Lambda_i(0)$ as its priority. The busy set S_α is the set of τ_i's jobs J_i^c satisfying $c \geq \alpha$.

Definition VI.2 (Problem Window)**.** Given a run-time job J_k^c with a run-time priority $\Lambda_k(c-\alpha)$, the problem window of J_k^c is the time interval $(r_k^\alpha, f_k^c]$, where r_k^α denotes the release time of a job J_k^α that gets $\Lambda_k(0)$ as its priority in the current busy period.

Note that for each offset value θ used as α_i in the current busy period, it can only be activated when a certain job J_i^θ is released and assigned the priority $\Lambda_i(0)$.

Definition VI.3 (Bound of Interrupt Count)**.** Given a run-time job J_k^c and a task τ_i, the bound of interrupt count, denoted by $BIC(J_k^c, \tau_i)$, is the maximal number of jobs of τ_i whose priority is equal to or higher than J_k^c in J_k^c's problem window.

Definition VI.4 (Active Job)**.** A job that has been released but has not been finished execution or terminated due to system criticality upgrading.

Lemma VI.5. *Suppose an arbitrary job $J_i^c \in S_\alpha$ is released at time instant r_i^c, and finished at time instant f_i^c, i.e., J_i^c is active during (r_i^c, f_i^c). For each job $J_i^h : h > c$, which is released when J_i^c is active, satisfies $J_i^h \in S_\alpha$.*

Proof: We prove by contradiction. Assume job J_i^h is the first job released in (r_i^c, f_i^c) satisfying

$$prt(J_i^h) < \Lambda_i(h-\alpha).$$

Suppose job J_i^h is released at time instant r_i^h. At r_i^h, by the definition of **AdjustPrt**, it is obvious that

$$\delta_i > P_k \geq \Lambda_i(h-\alpha) > \Lambda_i(c-\alpha) = prt(J_i^c).$$

Therefore, there must exist a job J_l of priority δ_i which is preempted in $(r_i^c, r_i^h] \subset (r_i^c, f_i^c)$, i.e., such a job J_l with priority $\delta_i > prt(J_i^c)$ must have executed for a while during $(r_i^c, r_i^h]$. Moreover,J_i^c is active in $(r_i^c, r_i^h]$.

However, a job with lower priority cannot preempt a higher-priority job, so J_i cannot execute in (r_i^c, f_i^c), in which J_i^c is active.

Therefore, this contradiction proves the lemma. ■

Lemma VI.6. *For an arbitrary run-time job J_k^c which gets priority $\Lambda_k(c-\alpha)$ following the rule of **AdjustPrt**, it must be true that no jobs with priority lower than $prt(J_k^c)$ executes during the problem window of J_k^c.*

Proof: If $c = \alpha$, then J_k^c's problem window is $(r_k^c, f_k^c]$ and it is clear that no jobs with priority lower than $prt(J_k^c)$ could execute during it. We prove this lemma under the condition of $c > \alpha$ by contradiction.

Assume J^* is the lowest-priority job executing in J_k^c's problem window $(r_k^\alpha, f_k^c]$, which satisfies $prt(J^*) > prt(J_k^c)$. Since $prt(J^*) > prt(J_k^c)$, J^* cannot execute in time interval $(r_k^c, f_k^c]$, so we only analyze the case that J^* executes in $(r_k^\alpha, r_k^c]$. Once J^* starts execution, it must be the only active job in the system. If no higher priority job releases and preempts it, J^* will finish or be terminated at some instant $t^* \in (r_k^\alpha, r_k^c]$, at which point a lower-priority job starts executing or the current busy period is over before J_k^c's arrival. Both of these two cases contradict with our assumptions. Therefore J^* must be preempted at some time instant $t^p \in (r_k^\alpha, r_k^c]$ and then keep active until f_k^c. We use $J_k^n : \alpha < n \leq c$ to denote the first released job of τ_i during $[t^p, r_k^c]$. As discussed above, we have $P_k < prt(J^*) = \delta'$ at r_k^n, thus α_k is set to $n > \alpha$ and a tuple (α', δ') must be added to Ω_k following the **Modify** function. Furthermore, since J^* is the lowest-priority job during $(r_k^\alpha, r_k^c]$, we have know $\delta_k < \delta'$ during $(r_k^\alpha, r_k^c]$.

If $\alpha' \neq \alpha$, since all the tuple (x, y) satisfying $y \leq \delta'$ will be removed by the **Modify** function at J_k^n's release time, and Ω_k does not include such a tuple containing the offset record α, thus α_k cannot be reset to α after r_k^n during the current busy period. It contradicts with the assumption that some future job J_k^c gets the priority $\Lambda_k(c-\alpha')$. And if $\alpha' = \alpha$, due to $\forall idx < c : \Lambda_k(idx + 1 - \alpha) \leq \Lambda_k(c-\alpha) < \delta'$, α cannot be reused before J_k^c's release time. And it contradicts with the assumption as well. ■

Lemma VI.7. *Given an arbitrary run-time job $J_i^c \in S_\alpha$, for each τ_i's job $J_i^h : h > c$ which belongs to the same busy period as J_i^c, if no jobs with priority lower than $\Lambda_i(h-\alpha)$ executes during the time interval $(r_i^\alpha, r_i^h]$, then $J_i^h \in S_\alpha$, i.e., $prt(J_i^h) \geq \Lambda_i(h-\alpha)$.*

Proof: We prove this lemma by contradiction. Assume $J_i^h : h > c$ is the first job of τ_i satisfying $prt(J_i^h) < \Lambda_i(h-\alpha)$, i.e., $\alpha_i^h > \alpha$. By the definition of the **Modify** function in Figure 2, if no future jobs get priority $\Lambda_i(0)$, the value of α_i is monotonically decreasing, so $prt(J_i^c) = \Lambda_i(c-\alpha_i) \geq \Lambda_i(c-\alpha)$. Since $prt(J_i^h) < \Lambda_i(h-\alpha)$, it must satisfy $prt(J_i^h) = \Lambda_i(0)$, which is true only if $P_k < \max\{\delta_i, P_{cur}\}$.

If $\delta_i > P_{cur}$, there must exist a preemption during (r_i^c, r_i^h) and the preempted job's priority is $\delta_i > P_k$. Otherwise, the current executing job's priority is lower than P_k. Both conditions imply that some job with priority lower than P_k has executed during (r_i^c, r_i^h).

It contradicts with the assumption that no jobs with priority lower than $\Lambda_i(h-\alpha)$ executes during the time interval $(r_i^\alpha, r_i^h]$. ■

Lemma VI.8. *Suppose a job J_k^c is released at time instant r_k^c and gets run-time priority $\Lambda_k(c-\alpha_k) : 1 \leq \alpha_k \leq c$ following the LPA policy. Then $\forall \tau_i \in \pi$ it satisfies*

$$BIC(J_k^c, \tau_i) \leq \| \{x | \Lambda_i(x) \leq prt(J_k^c)\} \| . \tag{9}$$

where $\|s\|$denotes the number of elements in set s.

Proof: Following the priority assignment policy of **AdjustPrt**, there must be a earlier job $J_k^{\alpha_k}$ getting $\Lambda_k(0)$ as its priority. And following Lemma VI.7, each J_i^j that executes in $(r_k^{\alpha_k}, d_k^c]$ satisfies

$$prt(J_i^j) \leq prt(J_k^c). \tag{10}$$

To count the number of higher-priority jobs in J_k^c's problem window, we use x_i to indicate the index of the first entry of Λ_i, which satisfies that $\Lambda_i(x_i) > prt(J_k^c) = \Lambda_k(c-\alpha_k)$, i.e., $x_i = \min\{x|\Lambda_i(x) > prt(J_k^c)\}$. Especially, $\Lambda(x) = +\infty$ for $x \geq N_i$. By Lemma IV.1, it is clear that

$$\| \{x|\Lambda_i(x) \leq prt(J_k^c)\} \| = x_i. \tag{11}$$

We divide the MC task set π into two subsets π_{active} and π_{silent}, according to whether some active jobs of the task exists at time instant $r_k^{\alpha_k}$. For each MC task τ_i,if there exists some active job J_i at $r_k^{\alpha_k}$, then $\tau_i \in \pi_{active}$, otherwise, $\tau_i \in \pi_{silent}$. We distinguish two cases as follows.

1) Consider π_{active}:

For each task $\tau_i \in \pi_{active}$, without loss of generality, we use J_i^a to indicate the least-index active job (the earliest released one) of τ_i at time instant r_k^o. Suppose that J_i^a is released at time instant r_i^a and $J_i^a \in S_{ai}$. It is straightforward that $a \geq ai \geq 1$, so $\Lambda_i(a + x_i - ai) \geq \Lambda_i(x_i)$.

If $prt(J_i^a) > prt(J_k^c)$, since J_k^c has a higher priority than J_i^a, J_i^a will not execute in J_k^c's problem window $(r_k^{\alpha_k}, f_k^c]$, following Lemma VI.6. So J_i^a must keep active during $(r_k^{\alpha_k}, f_k^c]$ and all of task τ_i's jobs, which are released in $(r_k^{\alpha_k}, f_k^c]$, belong to S_{ai}. Therefore, for each job J_i^j, such that $j \geq a$ and $r_i^j \in (r_k^{\alpha_k}, f_k^c]$, it satisfies $prt(J_i^j) \geq \Lambda_i(j - ai) \geq \Lambda_i(a - ai) = prt(J_i^a) > prt(J_k^c)$, i.e., no jobs of τ_i executes in J_k^c's problem window, thus

$$BIC(J_k^c, \tau_i) = 0 \leq x_i. \tag{12}$$

If $prt(J_i^a) < prt(J_k^c)$, from Lemma VI.5, we can conclude that at time instant $r_k^{\alpha_k}$, all of τ_i's active jobs belong to S_{ai} and $\delta_i \leq prt(J_i^a)$. By Lemma VI.6, no jobs with lower priority than $prt(J_k^c)$ executes and is preempted during $(r_k^o, f_k^c]$[3]. Thus, it satisfies that $\delta_i \leq \max\{prt(J_i^a), prt(J_k^c)\} = prt(J_k^c) < \Lambda_i(x_i)$ in the interval $(r_k^{\alpha_k}, f_k^c]$. Therefore, following Lemma VI.7, for each of τ_i's job $J_i^h : h \geq x_i + ai > c$, it must belong to S_{ai}, i.e., $prt(J_i^h) \geq \Lambda_i(x_i) > prt(J_i^c)$. Thus

$$BIC(J_k^c, \tau_i) \leq (x_i + ai) - a \leq x_i \tag{13}$$

2) Consider π_{silent}:

For each task $\tau_i \in \pi_{silent}$, without loss of generality, we use $J_i^s \in S_{si}$ to indicate the τ_i's first job released in $(r_k^{\alpha_k}, f_k^c]$. As above, we also use the definition of x_i. It is obvious that $s \geq si \geq 1$ and $\Lambda_i(s + x_i - si) \geq \Lambda_i(x_i)$. And by Lemma VI.6, during the time interval $(r_i^s, f_k^c] \subset (r_k^{\alpha_k}, f_k^c]$, it satisfies $\delta_i \leq prt(J_k^c) < \Lambda_i(x_i)$. Therefore, following Lemma VI.7, for each of τ_i's job $J_i^h : h \geq x_i + si > c$, it must belong to S_{si}. In addition $J_i^s \in S_{si}$, therefore, for each of τ_i's job releasing in (r_k^c, t_f), it satisfies that if $h \geq x_i + si$ then $prt(J_i^h) \geq \Lambda_i(x_i) > prt(J_i^c)$. Thus

$$BIC(J_k^c, \tau_i) \leq x_i + si - s \leq x_i - 1. \tag{14}$$

By (12), (13), (14) and (11), we have that $\forall \tau_i \in \pi$ it satisfies

$$BIC(J_k^c, \tau_i) \leq x_i - 1 = \| \{x|\Lambda_i(x) \leq prt(J_k^c)\} \| .$$

Therefore, the lemma is proved. ■

Theorem VI.9. *Any MC task system π that succeeds with the offline calculation algorithm of LPA is MC-schedulable by LPA' run-time scheduling.*

Proof: We prove this theorem by contradiction. Assume job $J_k^{m+\alpha-1}$ is the first job missed its deadline at $d_k^{m+\alpha-1}$ and gets priority $\Lambda_k(m-1)$. By Lemma VI.6 and Lemma VI.8, we know that during $J_k^{m+\alpha-1}$'s problem window $(r_k^\alpha, d_k^{m+\alpha-1}]$, it satisfies

$$\begin{aligned}\sum_{\tau_i \in \pi} C_i(\zeta_k) \times BIC(J_k^c, \tau_i) &\leq \sum_{\tau_i \in \pi} C_i(\zeta_k) \times (x_i - 1) \\ &\leq \sum_{\forall i,j:\Lambda_i(j) \leq \Lambda_k(m)} C_i(\zeta_k) \\ &\leq T_k \times (m-1) + D_k \\ &\leq d_k^m - r_k^\alpha\end{aligned}$$

It is contradict with (2). ■

[3]Note that there may be more than one jobs from different tasks release at same time instant t, and the scheduler will handle these events respectively with arbitrary order. To simplify the analysis, we treat the earlier handled job as earlier released one and vice versa.