Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems

Nan Guan[1,2], Pontus Ekberg[1], Martin Stigge[1] and Wang Yi[1,2]
[1]Uppsala University, Sweden
[2]Northeastern University, China

***Abstract*—An increasing trend in embedded system design is to integrate components with different levels of criticality into a shared hardware platform for better cost and power efficiency. Such mixed-criticality systems are subject to certifications at different levels of rigorousness, for validating the correctness of different subsystems on various confidence levels. The real-time scheduling of certifiable mixed-criticality systems has been recognized to be a challenging problem, where using traditional scheduling techniques may result in unacceptable resource waste. In this paper we present an algorithm called PLRS to schedule certifiable mixed-criticality sporadic tasks systems. PLRS uses fixed-job-priority scheduling, and assigns job priorities by exploring and balancing the asymmetric effects between the workload on different criticality levels. Comparing with the state-of-the-art algorithm by Li and Baruah for such systems, which we refer to as LB, PLRS is both more effective and more efficient: (i) The schedulability test of PLRS not only theoretically dominates, but also on average significantly outperforms LB's. (ii) The run-time complexity of PLRS is polynomial (quadratic in the number of tasks), which is much more efficient than the pseudo-polynomial run-time complexity of LB.**

I. Introduction

A major trend in modern real-time embedded systems is to integrate different functionalities into a single shared computing platform to meet rapidly increasing cost, power and thermal constraints. Typically, these different functionalities are not equally critical to the overall system performance. For example, in the control system of an unmanned aerial vehicle executing surveillance missions, it is more important to guarantee the correctness for the flight-critical functionalities such that the vehicle does not crash, than for the mission-critical functionalities like capturing images.

The functionalities with different criticalities in the system are usually subject to more or less rigorous forms of analysis depending on their overall criticality. For example [4], in order to get permission for an unmanned aerial vehicle to operate over civilian airspace, it is mandatory that its flight-critical functionalities be certified by authorities like US Federal Aviation Authority or European Aviation Safety Agency. The certification by such authorities is extremely rigorous: the system is examined under exceedingly pessimistic assumptions, which are very unlikely to occur in reality. However, these authorities are not interested in anything else except the safety of the vehicle. It is not important for them whether surveillance missions like capturing images are executed in time or not. On the other hand, the whole system, including both the flight-critical and mission-critical functionalities, must be validated by the manufacturers or other qualification agencies, who usually use a less rigorous standard than the aviation authorities.

The design of such *certifiable mixed-criticality* real-time systems has been recognized to be a very important but challenging problem in the emerging discipline of Cyber-Physical Systems [1], [4]. Roughly speaking, in such systems the "importance" and "urgency" of the workload are decoupled, and need to be carefully balanced in the scheduling. Neither the "importance" (i.e., criticality) nor the "urgency" (i.e., deadline) on its own can be used as a good scheduling criterion. Indeed, the problem of optimally scheduling such mixed-criticality systems is highly intractable even with very simple system models [2].

Baruah et. al. [4] proposed an effective algorithm, called OCBP (Own Criticality Based Priority), to schedule a simple version of such systems, which consists of a finite number of non-recurrent jobs. The strength of OCBP is to use more global knowledge of the system to better explore the asymmetric effects between different criticality levels. Such a global knowledge is much more effective than simple criteria like deadlines or criticalities, and OCBP provides significantly better performance than other strategies like EDF and Criticality Monotonic (higher criticality jobs have higher priorities). Indeed, OCBP is optimal, in terms of speedup factor [9], among all the fixed-job-priority algorithms for the finite non-recurrent job set model [2].

However, real-time tasks are typically recurrently executing, and are usually modeled as *sporadic* tasks. Recently, Li and Baruah [12] proposed an algorithm, which we refer to as LB, to extend OCBP to sporadic tasks. The main idea of LB is to *at run-time recompute* the priority assignment for future jobs from time to time according to the system state (Section II will introduce LB in detail). Although LB brings very interesting ideas on how to apply the OCBP priority assignment principle to sporadic tasks, it still has serious limitations in both effectiveness and efficiency:

- The performance of LB is unsatisfactory as it relies on very pessimistic schedulability tests based on load bound conditions.
- The run-time overhead of LB is large as it needs on-line *pseudo-polynomial* priority assignment recomputation.

In this paper, we present a new algorithm PLRS (Priority List Reuse Scheduling) to schedule certifiable mixed-criticality sporadic task systems, which overcomes both the above limitations of LB:

- The schedulability test of PLRS not only theoretically dominates, but also on average significantly outperforms LB's. This is analogous to the well-known relation between the response time analysis and utilization bounds: PLRS's schedulability test still maintains LB's load bound, but can accept many task systems that are denied by LB's load bound.
- The run-time complexity of PLRS is *polynomial* (quadratic in the number of tasks), which is much more efficient than the *pseudo-polynomial* run-time complexity of LB. In practise, PLRS's run-time overhead can be several orders of magnitude smaller than LB's.

The key for PLRS to overcome both of LB's limitations is to understand and utilize the "critical instant" of the scheduling in a more abstract way. Although the system behavior can not be represented by a single critical instant (that's why LB needs to perform the heavy run-time priority assignment recomputation and relies on the pessimistic load bounds), we still can abstract the workload characterization for *a set of* critical instants by a particular scenario. Therefore, on one hand we can off-line analyze the system with this particular scenario for a much better analysis precision; on the other hand we can at run-time schedule the system according to (with some lightweight adjustments) the priority assignment generated off-line under this scenario, which results in much more efficient run-time scheduling.

II. Related Work

The mixed-criticality scheduling problem was first identified and formalized by Vestal in [18], where he proposed a fixed-task-priority algorithm to schedule such systems. Dorin et. al. [8] formally proved that the algorithm in [18] is optimal in the scope of fixed-task-priority preemptive algorithms. However, as pointed out by Baruah and Vestal [5], the algorithm in [18] is by no means optimal if we are not restricted to fixed-task-priority preemptive algorithms, and is actually incomparable with the EDF algorithm.

Recognizing the ineffectiveness of applying traditional scheduling techniques to mixed-criticality systems, Baruah et. al. conducted a series of fundamental works on a simpler model consisting of a finite number of jobs with fixed release times. First they showed that deciding the feasibility of such job sets is strongly NP-hard even if all the jobs are released at the same time [2]. Then in [4] they proposed an effective heuristic algorithm OCBP, which guarantees to successfully schedule any feasible job set with two criticality levels on a 1.618 times faster machine. One can also use the insight from [4] to derive a load bound for OCBP, and the bound is refined in [13]. The results in [4] were further extended to arbitrary number of criticality levels [3].

The state-of-the-art technique of scheduling mixed-criticality *sporadic* task systems is the LB algorithm proposed by Li and Baruah [12]. LB adopts the effective OCBP principle, therefore, it is more flexible than fixed-task-priority algorithms or EDF. Applying OCBP to sporadic tasks is not a trivial extension due to at least two problems: (i) Since a sporadic task system will generate infinitely many jobs, the off-line priority computation procedure of OCBP will not terminate. (ii) OCBP requires the release time of each job to be known. However, in sporadic task systems the release time of each job is not known beforehand. LB solved the first problem by only computing the priorities for the jobs that can be released in one busy interval. LB solved the second problem by the following approach: Before the system starts running, LB computes a priority assignment for all jobs that can be released in a busy interval, assuming an as-early-as-possible job release pattern. Then jobs are scheduled according to this priority assignment, until some time point when the job releases deviate from the assumed pattern. Under these circumstances, LB will recompute a new priority assignment, and use it to schedule jobs until the next time some job's released does not exactly follow the expectation.

Although LB brings very interesting ideas on how to apply the OCBP priority assignment principle to sporadic tasks, it still has serious limitations in the following two aspects: (i) The performance of LB is unsatisfactory as it relies on very pessimistic schedulability test conditions. (ii) The run-time overhead of LB is large as it needs pseudo-polynomial run-time computation.

Our new algorithm PLRS will address both of these two problems. PLRS can be analyzed by a much more precise scheduliability test, and thereby provides significantly better performance. The run-time scheduling of PLRS is of polynomial complexity, and (the safe bound of) its run-time overhead could be several orders of magnitude smaller than LB's.

A. Other Related Works

De Niz et al. [7] considered a different aspect of mixed-criticality systems regarding effective scheduling of mixed-criticality tasks that may overrun. Nevertheless, [7] provides interesting ideas on how to dynamically adjust a task/job priority to protect the high criticality tasks from the interference of low criticality tasks, while still as much as possible maintain a "good" priority order from the urgency point of view. This approach has been later extended to handle systems with non-preemptable shared resources [10] and distributed/parallel systems where the mixed-criticality workload needs to be allocated to different execution units [11]. Pellizzoni et. al. [16] proposed a reservations-based approach to ensure strong isolation among subsystems of different criticalities. Petters et. al. [17] also considered the use of temporal isolation of subsystems for mixed-criticality systems, and addressed many practical issues in building such systems in reality. The drawback of the resource/temporal isolation approach is that it relies on severely over-provisioning computing resources, which may result in significant cost and energy waste. Mollison et. al. [15] adopt the criticality monotonic priority assignment for mixed-criticality scheduling on multi-core platforms. The higher-criticality tasks run with high priorities, and in the common case where they use only a small fraction of their execution time budgets, the lower-criticality tasks can execute in the remaining slack time.

III. Problem Model

We consider the scheduling of Mixed-Criticality (MC) sporadic task systems on a preemptive single processor. As in traditional real-time systems, a MC sporadic task generates a potentially infinite sequence of MC jobs. We start with the definition of MC jobs.

A. MC Jobs

Each MC job is characterized by a 4-tuple: $J_i = \langle a_i, d_i, \ell_i, c_i \rangle$, where

- $a_i \in \mathbb{R}_+$ is the release time.
- $d_i \in \mathbb{R}_+$ is the (absolute) deadline.
- $\ell_i \in [1, 2, \cdots, L]$ is the criticality of the job, where L is the number of criticality levels in the system.
- $c_i \in \mathbb{R}_+^L$ is a vector. The ℓ^{th} element in the vector, denoted by $c_i(\ell)$, specifies the worst-case execution time (WCET) estimate of job J_i at criticality level ℓ.

We follow the convention in real-time scheduling literatures that a *smaller* priority value represents a *higher* priority. We use a *larger* criticality value to represent a *higher* criticality.

Further, we adopt the following assumptions about the execution time of a MC job J_i, as in the original work of OCBP [4]:

- $\forall \ell^a > \ell^b : c_i(\ell^a) \geq c_i(\ell^b)$. This corresponds to the fact that the execution time estimation on a higher criticality level is more conservative.
- $\forall \ell^a > \ell_i : c_i(\ell^a) = c_i(\ell_i)$. No job is allowed to execute for more than its WCET at its own specified criticality.

The semantics of the MC job model is as follows: Job J_i is released at time a_i, has a deadline at d_i, and needs to execute for some amount of time γ_i. However, the value of γ_i is not known beforehand, but only becomes revealed by actually executing the job until it signals that it has completed execution. Job J_i is said to have exhibited a λ-criticality behavior, where

$$\lambda = \min\{\ell | \gamma_i \leq c_i(\ell)\}.$$

If it does not signal completion upon having executed for $c_i(L)$ (L is the highest criticality level), its behavior is erroneous, denoted by $L + 1$.

B. MC Tasks

Each MC sporadic task is characterized by a 4-tuple: $\tau_k = \langle D_k, T_k, \ell_k, C_k \rangle$, where

- $D_k \in \mathbb{R}_+$ is the relative deadline.
- $T_k \in \mathbb{R}_+$ is the minimal release separation (period).
- $\ell_k \in [1, 2, \cdots, L]$ is the criticality level of the task.
- $C_k \in \mathbb{R}_+^L$ is a vector. The ℓ^{th} element in the vector, denoted by $C_k(\ell)$, specifies the worst-case execution time (WCET) estimate of task τ_k at criticality level ℓ.

Note that there is no constraint on the relation between the relative deadline and period of a task: D_k can be larger than, smaller than or equal to T_k.

A MC task system τ consists of N independent MC tasks. Each MC task τ_k potentially releases an infinite sequence of MC jobs, with successive jobs being released at least T_k time apart. We use $J \in \tau_k$ to denote job J is released by task τ_k.

C. MC-Schedulablity

The MC task system is subjected to certifications on each criticality level. The system is temporally correct, i.e., schedulable, if and only if it passes all the certifications.

We say that the system behavior is of criticality-λ, if the highest criticality level of any job's behavior in the system is λ. If any job in the system exhibits erroneous behavior, the system's behavior is erroneous. We define the MC-schedulability under a scheduling algorithm $\mathcal{A}$ as follows:

Definition III.1 (MC-schedulability)**.** *Under a given scheduling algorithm $\mathcal{A}$, a job J_i is MC-schedulable if and only if for criticality-λ system behavior the following implication holds:*

$$\lambda \leq \ell_i \Rightarrow J_i \text{ has finished by } d_i$$

A task τ_i is MC-schedulable if and only if all the jobs released by τ_i are MC-schedulable. An MC task system τ is MC-schedulable if and only if all the tasks in τ are MC-schedulable.

By the above definition we can see that if the system exhibits a behavior with criticality higher than job J_i's criticality ℓ_i, then J_i does not need to meet its deadline for the scheduling to be considered successful. This is because no certification authority will require that J_i meets its deadline in this situation: for the authorities certifying the system at a criticality level higher than ℓ_i, meeting J_i's deadline is not required; for authorities at a criticality level lower than or equal to ℓ_i, the system behavior is not within their assumption.

IV. The New Algorithm PLRS

As introduced in Section II, both of LB's limitations are due to the run-time priority recomputation. The reason why LB has to repeatedly perform the recomputation is that the priority assignment obtained assuming the as-early-as-possible job release pattern does not guarantee the system schedulability if some jobs are released later than expected. The crucial observation behind our new algorithm PLRS is that, although the as-early-as-possible job release pattern itself is not a concrete worst-case system behavior, the information contained in this pattern can actually represent the worst-case system behavior in an abstract way. By correctly extracting and utilizing such information, we only need to perform the priority computation once off-line. The results of this computation can be used (with some lightweight calculations) at run-time to assign job priorities. In this way, PLRS avoids the heavy on-line priority assignment recomputations, and solves both the performance and run-time overhead limitations of LB.

In the following we first introduce PLRS's off-line computation, then introduce how the results of the off-line computation are used in PLRS's run-time scheduling. Later in Section V we will prove that the system's schedulability is completely determined on the off-line computation, and in Section VI we will show that PLRS's run-time scheduling is of polynomial complexity.

A. Off-line Computation

As pointed out in [12], although a sporadic task system will potentially release an infinite number of jobs, at any time we only need to consider the jobs that can be released in the current busy interval. This is because before the system goes into the next busy interval, there must be a time point at which the processor becomes idle and the system is reset to the same state as in the beginning of the previous busy interval. Therefore, the jobs released in the next interval can be scheduled by the same principle as in the previous one. For the same reason, the off-line computation of PLRS only needs to consider a set of jobs (denoted by I) that can be released in one busy interval. We can derive a pseudo-polynomial upper bound[1] on the number of jobs from each task in I [12]. In the following, we use n_k to denote this bound for each task τ_k.

The first step of PLRS's off-line computation is to compute a priority order for all the jobs in I. Since all the jobs from the same task are identical, we can always assign priorities to jobs from the same task in the way that later jobs never have higher priorities. Among the jobs from the same task, we thus already have a reasonable priority order, and we only need to consider the relative priority orders between jobs from different tasks.

The priority assignment is computed based on the OCBP principle, which is essentially the same as the run-time priority recomputation in LB: Each task τ_k is related to a number δ_k which denotes the number of τ_k's jobs that have not been assigned a priority. Initially, $\delta_k = n_k$. The algorithm first determines which task's largest-index job can be assigned the lowest priority. Task τ_k's largest-index job $J_k^{\delta_k}$ is eligible to be assigned the lowest priority if it satisfies the condition:

$$\sum_{\tau_j \in \tau} (\delta_j \times C_j(\ell_k)) \leq (\delta_k - 1) \times T_i + D_i. \tag{1}$$

The LHS of the condition represents the total workload of all the remaining jobs in I if the system's behavior is of criticality level ℓ_k, and the RHS is the minimal distance between the absolute deadline of $J_k^{\delta_k}$ and the beginning of the busy interval. So if the LHS does not exceed the RHS, we can guarantee that $J_k^{\delta_k}$ is MC-schedulable if all other jobs have higher priorities. In general there could be more than one task whose largest-index job is eligible to be assigned the lowest priority, and in this case we can arbitrarily choose one of them. After deciding the lowest priority job $J_k^{\delta_k}$, we set $\delta_k \leftarrow \delta_k - 1$ to exclude that job from the consideration in future steps.

We then repeat the above procedure until all the jobs are assigned a priority each, or at some point no job is eligible to be assigned the lowest priority. If the algorithm terminates with the first case, we say that the off-line computation algorithm succeeds, otherwise, it is a failure. Note that the priority assignment itself is not meant to provide any schedulability guarantee, i.e., even if the off-line computation algorithm succeeds, the task system may still be not MC-schedulable if at run-time the jobs are scheduled strictly following this priority assignment.

TABLE I
AN EXAMPLE TASK SYSTEM.

Task	T_i	D_i	ℓ_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

Example IV.1. *Consider the MC task system in Table I. We assume[2] that initially $\delta_1 = 6$, $\delta_2 = 3$, $\delta_3 = 2$ and $\delta_4 = 1$. The following is a possible result by PLRS's off-line computation for this example:*

high											*low*
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
J_1^1	J_1^2	J_2^1	J_1^3	J_3^1	J_2^2	J_4^1	J_1^4	J_1^5	J_2^3	J_3^2	J_1^6

We choose J_4^1 to illustrate the usage of Condition (1). At the step of assigning the lowest priority to J_4^1, the remaining jobs are $\{J_1^1, J_1^2, J_2^1, J_1^3, J_3^1, J_2^2, J_4^1\}$, i.e., $\delta_1 = 3$, $\delta_2 = 2$, $\delta_3 = 1$ and $\delta_4 = 1$. Since J_4^1's criticality level is 2, so the LHS of Condition (1) equals:

$$\delta_1 \times C_1(2) + \delta_2 \times C_2(2) + \delta_3 \times C_3(2) + \delta_4 \times C_4(2) = 47.$$

On the other hand, the RHS of Condition (1) is:

$$(\delta_4 - 1) \times T_4 + D_4 = 50.$$

So Condition (1) is true for J_4^1 and it can be assigned the lowest priority at that step.

By now we have obtained a priority order for the jobs that can be released in a busy interval. However, this priority assignment will not be directly used in the on-line scheduling of PLRS. Instead, we will derive an individual *priority list* Λ_k for each task τ_k, by collecting the priorities assigned to the jobs of task τ_k in an ordered list. We use $\Lambda_k(x)$ to denote the x^{th} priority value in the individual priority list.

Example IV.2. *Corresponding to the resulting priority order in Example IV.1, the priority list Λ_k for each task is as follows. For example, we have $\Lambda_1(4) = 8$ since the fourth priority value in Λ_1 is 8.*

Λ_1	*1*	*2*	*4*	*8*	*9*	*12*
Λ_2	*3*	*6*	*10*			
Λ_3	*5*	*11*				
Λ_4	*7*					

B. Run-Time Scheduling

PLRS is fixed-job-priority preemptive scheduling. PLRS will calculate priority $\mathsf{prt}(J)$ for each job J. For each task τ_k, PLRS at run time maintains a plan for the priorities of its

[1]The bound presented in [12] is for MC task systems with two criticality levels, however, it can be easily extended to systems with arbitrary number of criticality levels by induction.

[2]The number of jobs that can be released in a busy interval is actually larger according to the bound in [12]. Here we only consider a subset of these jobs for illustration.

future jobs that can be released in the current busy interval. We use Ψ_k to denote τ_k's *priority plan*, which records a set of indices directing to the priority values in Λ_k. According to the priority plan, the first future job will get the priority in Λ_k identified by the smallest index stored in Ψ_k, and the next future job will get the priority identified by the second smallest index and so on.

We can use a pair (α, β) to abstractly represent several consecutive indices in Ψ_k, where α is the first one and β the last one of these consecutive indices. So we can represent Ψ_k by a set of such pairs $\{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \cdots\}$. We use $|(\alpha_m, \beta_m)| = \beta_m - \alpha_m + 1$ to denote the number of indices represented by this pair. For example, the priority plan recording indices $\{1, 2, 5, 6, 7, 11\}$ is represented by $\{(1, 2), (5, 7), (11, 11)\}$, and $|(1, 2)| = 2$, $|(5, 7)| = 3$ and $|(11, 11)| = 1$. Note that, this abstract representation of Ψ_k is the key for PLRS to have *polynomial* run-time complexity.

When a job J_i of task τ_i is released, PLRS executes the priority management routine $\mathsf{PrtMng}(J_i)$ to first adjust the priority plans according to the system state, and then assign the released job a priority. In the following, we will in detail introduce the working principle of $\mathsf{PrtMng}(J_i)$, and later in Section VI we discuss its computational complexity.

$\mathsf{PrtMng}(J_i)$ first checks whether the processor is currently idle. If yes, a new busy interval starts, and each task will reset its priority plan to the initial state, in which the coming jobs will be assigned priorities simply following the priority lists $\Lambda_1, \ldots, \Lambda_N$. If currently a job J_{cur} is running, then $\mathsf{PrtMng}(J_i)$ compares $\mathsf{prt}(J_{cur})$ with the planned priority of the released job J_i according to the current plan Ψ_i. If J_i's planned priority is higher, $\mathsf{PrtMng}(J_i)$ adjusts each task's priority plan. Finally, J_i gets its priority according to the new priority plan after the adjustment, and J_i's information is removed from the plan.

The priority plan adjustment is the key step of PLRS. Intuitively, for each task τ_k, the adjustment will find a "borderline" in its priority plan Ψ_k according to the priority of the currently running job J_{cur}. Then the indices directing to priorities higher than $\mathsf{prt}(J_{cur})$ will be promoted (become smaller) as much as possible, while the other ones remain unchanged.

The algorithm in Figure 1 shows the pseudo-code of $\mathsf{PrtMng}(J_i)$ with six operations on the priority plan. We use $\Psi_i = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \cdots\}$ to denote τ_i's priority plan before an operation, and Ψ'_i after an operation, to explain the functionality of each operation as follows:

- $\mathsf{Reset}(\Psi_i)$ resets Ψ_i to its initial state:

$$\Psi'_i \leftarrow \{(1, n_i)\}. \tag{2}$$

  where n_i is the maximal number of jobs τ_i can release in a busy interval.
- $\mathsf{GetFirst}(\Psi_i)$ returns $\Lambda_i(\alpha_1)$, where Λ_i is the priority list of τ_i and α_1 is the first index in the first pair of Ψ_i.
- $\mathsf{Locate}(\Lambda_i, p_{cur})$ returns the largest index x with an entry strictly less than p_{cur}, i.e., $\max\{x \mid \Lambda_i(x) < p_{cur}\}$
- $\mathsf{Split}(\Psi_k, \mu)$ divides the pair (α_m, β_m) satisfying $\alpha_m \leq$

1: **if** the processor is currently idle **then**
2:   **for** each τ_k **do**
3:     $\mathsf{Reset}(\Psi_k)$
4:   **end for**
5: **else**
6:   $p_{cur} \leftarrow$ the currently running job's priority
7:   $p_{rls} \leftarrow \mathsf{GetFirst}(\Psi_i)$
8:   **if** $p_{rls} < p_{cur}$ **then**
9:     **for** each τ_k **do**
10:       $\mu \leftarrow \mathsf{Locate}(\Lambda_k, p_{cur})$
11:       $\mathsf{Split}(\Psi_k, \mu)$
12:       $\mathsf{Merge}(\Psi_k, \mu)$
13:     **end for**
14:   **end if**
15: **end if**
16: $\mathsf{prt}(J_i) \leftarrow \mathsf{GetFirst}(\Psi_i)$
17: $\mathsf{RmvFirst}(\Psi_i)$

Fig. 1. The priority management routine $\mathsf{PrtMng}(J_i)$.

  $\mu < \beta_m$, if there is one, into two pairs:

$$\Psi'_i \leftarrow \{(\alpha_1, \beta_1), \cdots, \underbrace{(\alpha_m, \mu), (\mu + 1, \beta_m)}_{\text{the orginial } (\alpha_m, \beta_m)}, \cdots\}. \tag{3}$$

- $\mathsf{Merge}(\Psi_k, \mu)$ merges the pairs $(\alpha_1, \beta_1), \cdots, (\alpha_m, \beta_m)$ into one pair, where (α_m, β_m) is the last pair satisfying $\beta_m \leq \mu$.

$$\Psi'_i \leftarrow \{(\alpha, \beta), (\alpha_{m+1}, \beta_{m+1}), \cdots\} \tag{4}$$

  and the resulting new pair (α, β) is:

$$(\alpha, \beta) \leftarrow (1, \ \textstyle\sum_{x=1}^{m} |(\alpha_x, \beta_x)|) .$$

- $\mathsf{RmvFirst}(\Psi_i)$ removes the first index represented in Ψ_i:

$$\Psi'_i \leftarrow \begin{cases} \{(\alpha_1 + 1, \beta_1), (\alpha_2, \beta_2), \cdots\} & \alpha_1 < \beta_1 \\ \{(\alpha_2, \beta_2), \cdots\} & \alpha_1 = \beta_1 \end{cases} \tag{5}$$

.

Example IV.3. *We use the task system in Example IV.1 and the priority lists in Example IV.2 to illustrate how* $\mathsf{PrtMng}(J_i)$ *works. We assume that all the tasks release the first job at time* 0*, and the initial priority plans of* τ_1 *and* τ_2 *are shown in Figure 3-(a). Figure 3-(b) shows their priority plans at time* 36*, just before the release of* J_2^2*. Comparing with Figure 3-(a), both* Ψ_1 *and* Ψ_2 *have shrunk for one index due to the release of* J_1^1 *and* J_1^2*. When* J_2^2 *is released, by* $\mathsf{GetFirst}(\Psi_2)$ *we know its planned priority is* 6*, which is higher than the currently running job* J_3^2*'s priority* 11*, so* $\mathsf{PrtMng}(J_i)$ *executes line 9 to line 13 to adjust each task's priority plan. For* τ_1*, the adjustment first uses* $\mathsf{Locate}(\Lambda_1, 11)$ *to find the borderline* 5*, which is the largest index in* Λ_1 *directing to a priority higher than the currently running job's priority* 11*.* $\mathsf{Split}(\Psi_1, 5)$ *splits the pair* $(2, 6)$*, which is crossed by the borderline* 5*, into two pairs* $(2, 5)$ *and* $(6, 6)$*, and* $\mathsf{Merge}(\Psi_1, 5)$ *merges all the pairs before the borderline (in this example only one pair), and pushes them to smaller indices as much as possible. So the pair* $(2, 5)$ *becomes* $(1, 4)$*. For* τ_2*, first* $\mathsf{Locate}(\Lambda_2, 11)$ *finds*

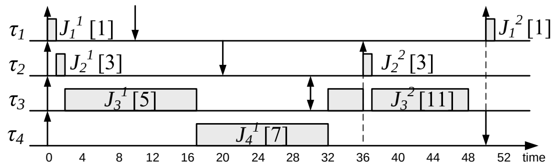
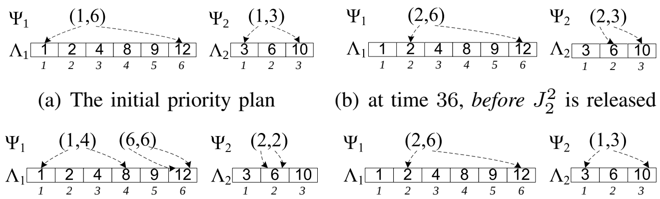


Fig. 2. An example to illustrate $\mathsf{PrtMng}(J_i)$. The number in brace after each job name denotes the priority this job obtained at release.



(a) The initial priority plan (b) at time 36, *before* J_2^2 is released

(c) at time 36, *after* J_2^2 is released (d) at time 50, *after* J_1^2 is released

Fig. 3. Illustration of how the priority plans of τ_1 and τ_2 changes over time.

the borderline 3*, and then* $\mathsf{Merge}(\Psi_2, 3)$ *pushes the pair* $(2, 3)$ *to small indices, i.e., change* $(2, 3)$ *to* $(1, 2)$*. Finally,* J_2^2 *gets its priority* 3 *according to the new priority plan, and the first index represented in* Ψ_2 *is removed to delete the priority plan for the released job* J_2^2*. At time* 50*,* J_1^2 *is released at idle time, so each task will reset its priority plan to the initial state. The released job* J_1^2 *gets the priority* 1 *according to the initial priority plan, after which the first index represented in* Ψ_1 *is removed to exclude the priority plan for* J_1^2*.*

Finally, we address a subtle technical issue: a higher priority job J_h may be released at the same time as a lower priority job J_l finished its work. In this case, we construct the scheduler so that J_l temporally does not signal completion, but will be preempted by J_h, and wait until the earliest time instant when J_l is scheduled to execute again and signal its completion. By this construction, we exclude the possibility that a higher priority job starts execution right after a low priority J_l signals completion. In other words, right after a job signalled completion, the processor must be running a job with priority lower (including the idle job). By such a construction, we have the following property, which will be useful in the proof of PLRS's schedulability and run-time complexity in later sections:

Lemma IV.4. *Suppose a job* J_a *signalled completion at time* t_a*, and later at time* t_b *another job* J_b *with* $\mathsf{prt}(J_b) < \mathsf{prt}(J_a)$ *is preempted. Then there must be a job* J_c *with* $\mathsf{prt}(J_c) > \mathsf{prt}(J_a)$ *which is preempted at some time point* $t_c \in (t_a, t_b)$*.*

Proof: We prove by contradiction. Assume there is no such job J_c with priority $\mathsf{prt}(J_c) > \mathsf{prt}(J_a)$ which is preempted at any $t_c \in (t_a, t_b)$, i.e., every job with priority lower than J_a executing in (t_a, t_b) can execute to completion without interruption.

Therefore, right after J_a signalled completion, the processor started to run a job J_1 with lower priority (recall that we construct the run-time scheduler of PLRS in the way that right after a job J_l signalled completion, the processor must be running a job with priority lower than J_l, as discussed at the end of the Section IV-B), and by our assumption J_1 will execute to completion without interruption. For the same reason, right after J_1 signalled completion, another job J_2 with priority lower than J_1 will execute to completion. The procedure repeats until some job J_x signalled completion and J_b starts execution at t_b. It follows that all these jobs have lower and lower priorities, so we have

$$\mathsf{prt}(J_a) < \mathsf{prt}(J_1) < \mathsf{prt}(J_2) < \cdots < \mathsf{prt}(J_x) < \mathsf{prt}(J_b).$$

This contradicts the assumption $\mathsf{prt}(J_b) < \mathsf{prt}(J_a)$. ■

V. Schedulability of PLRS

In this section we will show that the schedulability of PLRS is determined by off-line computation, i.e., any task set τ that succeeds with PLRS's off-line computation is MC-schedulable by PLRS's run-time scheduling algorithm.

To simplify the presentation, from now on we will view an idle processor as executing an "idle" job $J_\perp$ with the lowest priority $+\infty$. Any job released by the task system has higher priority than $J_\perp$ and thereby preempts $J_\perp$, which corresponds to the fact that a released job will immediately execute if the processor is currently idle.

In the run-time scheduling of PLRS, each job actually has been planned a priority before it is released (with the priority plans Ψ). However, the planned priority may change from time to time until the job is released. To capture a job's priority that is planned by PLRS until it is released, we introduce the concept of *expected priority* $\mathsf{epp}(J, t)$. Intuitively, $\mathsf{epp}(J, t)$ represents the priority which J will eventually get if all the jobs from the same task strictly follow the priority plan at time t. Consider the example in Figures 2 and 3. We have $\mathsf{epp}(J_1^6, 40) = 12$, since if the unreleased jobs $J_1^2 \cdots J_1^6$ strictly follow the priority plan at time 40, which is the same as Figure 3-(c), then J_1^6 will get the priority 12. Below is the formal definition of $\mathsf{epp}(J, t)$, in which (6) describes how to parse the information in Ψ_i to obtain the corresponding priority value.

Definition V.1 (Expected Priority)**.** *Given a job* J *and a time instant* t strictly before *its release. Suppose* J *is the* x^{th} *job ever released by* τ_i *and* τ_i *has released* y *jobs by* t*. Let* $\Psi_i = \{(\alpha_1, \beta_2), (\alpha_2, \beta_2), \cdots\}$ *be* τ_i*'s priority plan at* t *(after priority adjustment if there is any). Then* J*'s expected priority* $\mathsf{epp}(J, t)$ *at time* t *is defined as follows:*

$$\mathsf{epp}(J, t) = \Lambda_i(x - y - \sum_{z=1}^{m-1} |(\alpha_z, \beta_z)| + \alpha_m - 1) \tag{6}$$

where (α_m, β_m) *is the pair in* Ψ_i *satisfying:*

$$\sum_{z=1}^{m-1} |(\alpha_z, \beta_z)| < x - y \leq \sum_{z=1}^{m} |(\alpha_z, \beta_z)|. \tag{7}$$

Note that $x > y$ since t is strictly before J's release time, otherwise $\mathsf{epp}(J, t)$ is not defined. Further, there always exists a pair satisfying (7), since the job set used to construct the priority assignment in PLRS's off-line computation is large enough to cover all the jobs that will be released in a busy interval. In other words, the system is always reset to the initial state before all the indices in Ψ_i are consumed. The expected priorities also follow the convention that a *smaller* value represents a *higher* priority.

Further, we use t^- to denote a time instant that is before, but arbitrarily close to t, and thereby we can use $\mathsf{epp}(J, t^-)$ to denote the concept of J's planned priority at time t just before all the priority adjustments at t.

Lemma V.2. *If job J is released at time r, we have the following properties:*

1) $\forall t < r : \mathsf{epp}(J, t) \geq \mathsf{prt}(J)$
2) $\forall t < r$: *if* $\mathsf{epp}(J, t^-) > \mathsf{epp}(J, t)$*, then there must be some job J_{cur} (possibly the idle job) with* $\mathsf{prt}(J_{cur}) > \mathsf{epp}(J, t^-)$ *being preempted at t.*

Proof: These properties follow directly from the definition of $\mathsf{epp}(J, t)$ and the construction of PLRS's run-time scheduling. The first property: When a job J is released at r, its priority is assigned by its up-to-date expected priority after the priority adjustment at t. Also the priority adjustment never causes the expected priority of a job to become lower (increase in value): Both Reset and Merge only "move" a pair to smaller indices. Therefore we know at any time the expected priority for a job is not higher than its priority. The second property: From PLRS's run-time rules, we know that a job's expected priority only changes when the priority adjustment is triggered. For this, there must be some job released whose expected priority is higher than the currently running job. After the priority adjustment, this released job will get a priority no lower than the expected priority before, because of the first property. This must cause a preemption to the currently running job. ■

Now we will use the expected priority concept and its properties to prove PLRS's schedulability: Any task set for which PLRS's off-line computation succeeds is guaranteed to be MC-schedulable by PLRS's run-time scheduling. The overall proof strategy is by contradiction: We assume PLRS's off-line computation is successful for a task set τ, but τ is not MC-scheduable by PLRS's run-time scheduling. We let a job J_i of task τ_i be the first job that is not MC-schedulable, i.e., the system behavior is no higher than J_i's criticality level ℓ_i before J_i's deadline d_i, and J_i has not signalled completion by d_i. We will show that this contradicts the assumption that the off-line computation of PLRS was successful, by which the proof is established.

In the remaining part of this section, J_i (a job of task τ_i) denotes the first job that is not MC-schedulable. The proof will focus on the workload that occurs in a particular time interval ending with J_i's deadline d_i:

Definition V.3 (Problem Window)**.** *The* problem window *is the time interval $(t_0, d_i]$, where t_0 is the* latest *time point before d_i at which some job with priority lower than* $\mathsf{prt}(J_i)$ *(possibly the idle job $J_\perp$) is preempted.*

Lemma V.4. *Any job J_k that executes in the problem window $(t_0, d_i]$ satisfies both of the following conditions:*

1) *J_k is released no earlier than t_0*
2) $\mathsf{epp}(J_k, t_0) \leq \mathsf{epp}(J_i, t_0)$

Proof: We prove the first claim by contradiction. We let J_p be the job preempted at t_0, and by the definition of t_0 we know $\mathsf{prt}(J_p) > \mathsf{prt}(J_i)$. Suppose J_k is a job released before t_0, which executes in $(t_0, d_i]$. Since at t_0 it is J_p, but not J_k, being preempted, we know $\mathsf{prt}(J_k) > \mathsf{prt}(J_p)$. So we can conclude that $\mathsf{prt}(J_k) > \mathsf{prt}(J_i)$. Therefore, there must be some time point $t_1 \in (t_0, d_i]$ at which the processor starts to execute the jobs whose priorities are higher than $\mathsf{prt}(J_i)$ (otherwise J_i would be able to finish its work before deadline). So by Lemma IV.4 we know at t_1 some job with priority lower than $\mathsf{prt}(J_i)$ is preempted. This contradicts with that t_0 is the latest time point before d_i at which some job (possibly the idle job $J_\perp$) with priority lower than $\mathsf{prt}(J_i)$ is preempted.

We prove the second claim also by contradiction. Assume J_k is a job executing in $(t_0, d_i]$, which satisfies

$$\mathsf{epp}(J_k, t_0) > \mathsf{epp}(J_i, t_0). \tag{8}$$

By the first property of Lemma V.2, we know $\mathsf{prt}(J_k) \leq \mathsf{epp}(J_k, t_0)$. Then we distinguish the following two cases:

1) $\mathsf{prt}(J_k) < \mathsf{epp}(J_k, t_0)$
2) $\mathsf{prt}(J_k) = \mathsf{epp}(J_k, t_0)$

Consider case 1). Since $\mathsf{prt}(J_k) < \mathsf{epp}(J_k, t_0)$, we know there must be some time point $t_1 > t_0$ at which J_k's expected priority for the first time becomes higher than $\mathsf{epp}(J_k, t_0)$, i.e., t_1 satisfies:

$$\mathsf{epp}(J_k, t_1^-) = \mathsf{epp}(J_k, t_0) \tag{9}$$

and

$$\mathsf{epp}(J_k, t_1) < \mathsf{epp}(J_k, t_0).$$

Then by the second property of Lemma V.2 we know it must be true that at t_1 some job J_l with priority satisfying

$$\mathsf{prt}(J_l) > \mathsf{epp}(J_k, t_1^-) \tag{10}$$

is preempted. By combining (8), (9) and (10) we have

$$\mathsf{prt}(J_l) > \mathsf{epp}(J_i, t_0).$$

By the first property of Lemma V.2 we also know $\mathsf{epp}(J_i, t_0) \geq \mathsf{prt}(J_i)$, so we have $\mathsf{prt}(J_l) > \mathsf{prt}(J_i)$, i.e., at t_1, a time point strictly later than t_0, a job with priority lower than $\mathsf{prt}(J_i)$ is preempted, which contradicts with the definition that t_0 is the latest time point before d_i at which a job with priority lower than $\mathsf{prt}(J_i)$ is preempted.

Now consider case 2). By $\mathsf{prt}(J_k) = \mathsf{epp}(J_k, t_0)$ and (8) we have $\mathsf{prt}(J_k) > \mathsf{epp}(J_i, t_0)$. By the first property of Lemma V.2 we also have $\mathsf{epp}(J_i, t_0) \geq \mathsf{prt}(J_i)$, so we know $\mathsf{prt}(J_k) > \mathsf{prt}(J_i)$.

Since J_k's priority is lower than J_i's, there must exist some time point $t_1 \in (t_0, d_i]$ at which the processor starts to execute jobs whose priorities are no lower than J_i's priority $\mathsf{prt}(J_i)$. So by Lemma IV.4 we know at t_1 some job with priority lower than $\mathsf{prt}(J_i)$ is preempted. This contradicts with that t_0 is the latest time point before d_i at which some job (possibly the idle job $J_\perp$) with priority lower than $\mathsf{prt}(J_i)$ is preempted.

In summary, the assumption leads to a contradiction in both cases, so the second claim is proved. ■

Now we are ready to establish the main theorem for PLRS's schedulability.

Theorem V.5. *Any MC task system τ that succeeds with the off-line calculation algorithm of* PLRS *is MC-schedulable by* PLRS*'s run-time scheduling.*

Proof: We prove by contradiction, and use the same notation as above: Let J_i be the first job that is not MC-schedulable, and $(t_0, d_i]$ be the problem window.

By Lemma V.4 we know all the jobs that can execute in $(t_0, d_i]$ are released no earlier than t_0, and have expected priorities no lower than J_i after the adjustment at t_0. We use I_1 to denote the set of these jobs.

Assume J_i is the x^{th} job of τ_i in I_1. Since τ succeeds with PLRS's off-line computation, Condition (1) holds for τ_i at each step in the off-line computation, and in particular, the following holds:

$$\sum_{\tau_j \in \tau} \delta_j^x \times C_j(\ell_i) \leq (x-1) \times T_i + D_i \tag{11}$$

where δ_j^x denotes the number of τ_j's jobs that had not been assigned yet when assigning the priority of the x^{th} job of τ_i during the off-line computation.

By Lemma V.4 we know all the jobs in I_1 have not been released before t_0, and each J_k of these jobs satisfies $\mathsf{epp}(J_k, t_0) \leq \mathsf{epp}(J_i, t_0)$. Therefore, the number of jobs in I_1 for any task τ_j is at most δ_j^x (otherwise some of τ_j's jobs in I_1 will end up with expected priorities lower than J_i's). So we have the following:

$$\sum_{J_j \in I_1} c_j(\ell_i) \leq \sum_{\tau_j \in \tau} \delta_j^x \times C_j(\ell_i). \tag{12}$$

Since J_i is the x^{th} job in I_1, we know

$$(x-1) \times T_i + D_i \leq d_i - t_0. \tag{13}$$

By (11), (12) and (13) we have

$$\sum_{J_j \in I_1} c_j(\ell_i) \leq d_i - t_0. \tag{14}$$

On the other hand, we know that before d_i each job J_j executes for at most $c_j(\ell_i)$, since the system behavior is no higher than ℓ_i before d_i. Therefore we know the total workload of the jobs that executed in $(t_0, d_i]$ (the ones in I_1) is no larger than $\sum_{J_j \in I_1} c_j(\ell_i)$. And since at least one of these jobs (J_i) has not finished yet by d_i, we have

$$\sum_{J_j \in I_1} c_j(\ell_i) > d_i - t_0$$

which contradicts with (14). ■

A. Comparing with LB

We start with introducing the *load* concept in the context of MC task systems. In traditional (non MC) real-time systems, the *load* is the maximum over all time intervals, of the cumulative execution requirement by the whole task system over the interval, normalized by the interval length [14]. Informally, the load represents a lower bound on the portion of processing capacity required by this task system to meet all deadlines.

Analogous to this concept, we can define the *load* for a MC system *on each criticality level*.

Definition V.6. *The* criticality-ℓ load *of a MC task system τ is defined by*

$$\mathsf{Ld}_\ell(\tau) = \max_{0 \leq t_1 \leq t_2} \left\{ \sum_{\forall J_i : \ell_i \geq \ell \wedge t_1 \leq a_i \wedge d_i \leq t_2} c_i(\ell)/(t_2 - t_1) \right\}.$$

For any criticality level ℓ, $\mathsf{Ld}_\ell(\tau)$ can be computed using well-known techniques [6] for determining the loads of traditional (i.e., non MC) sporadic task systems. Intuitively, $\mathsf{Ld}_\ell(\tau)$ represents a lower bound on the portion of processing capacity required by this task system with which it can meet all deadlines *only subjecting to the certification on criticality level ℓ*. Clearly to correctly execute a MC task system, a necessary condition is that the required portion of processing capacity on each level should not exceed 1.

In [12], LB is presented in the context of systems with two criticality levels, however, it can be easily extended to handle MC task systems with any number of criticality levels, and one can use the knowledge from [3] to get the following MC-schedulability test condition:

$$\forall \ell \in [1, L] : \mathsf{Ld}_\ell(\tau) \leq \mathsf{LoadBound}(L) \tag{15}$$

where Ld_ℓ is the *load* of criticality level ℓ, and $\mathsf{LoadBound}(L)$ is a function with respect to the total number of criticality levels L of the system, which is recursively calculated as follows:

$$\mathsf{LoadBound}(1) = 1$$
$$\mathsf{LoadBound}(L) = \frac{2}{1 + \sqrt{4(1/\mathsf{LoadBound}(L-1))^2 + 1}}.$$

For the special case of $L = 2$, a more precise load condition is available [13], [12]:

$$(\mathsf{Ld}_2(\tau))^2 + \mathsf{Ld}_1(\tau) \leq 1. \tag{16}$$

The off-line computation algorithm of PLRS is essentially the same as the run-time priority recomputation of LB, so we know that any task set that satisfies the load bound test of LB, can succeed with PLRS's off-line computation, and thereby is MC-schedulable by PLRS. So we know

Corollary V.7. PLRS*'s schedulability test in Theorem V.5 dominates the* LB*'s load bound test.*

We have also conducted experiments with randomly generated MC task sets to compare the acceptance ratio[3] of PLRS and LB. Our experiments show that PLRS indeed exhibits a significantly better performance than LB, especially for systems with more criticality levels.

VI. Run-Time Complexity

In this section, we discuss the run-time complexity of PLRS. In particular, we analyze the run-time priority management algorithm in Figure 1, which involves several operations on the task priority plans Ψ_i. We will show that the number of elements in each task priority plan Ψ_i is bounded by $N+1$, where N is the number of tasks in the system. We will use this to show that all these operations are of complexity $\mathsf{O}(N)$. Since PLRS's run-time priority management operates on each task's priority plan, the overall complexity of PLRS's run-time priority management will therefore be $\mathsf{O}(N^2)$.

Lemma VI.1. *The operations* Reset*,* GetFirst*,* Locate*,* Split*,* Merge*,* RmvFirst *can all be implemented with linear complexity regarding the number of pairs in* Ψ_i*.*

Proof: The proofs for Reset, GetFirst, Split, Merge and RmvFirst are straightforward. Assuming an implementation of Ψ_i as a linked list, these operations either operate only the first pair or only need to linearly traverse the whole list.

To implement Locate, we can either do a binary search in Λ_i resulting in polynomial complexity for this operation, or even use an off-line pre-computed look-up table for deriving the desired index in constant time. ∎

In order to show that there are at most $N+1$ elements in Ψ_i, we introduce the *causer job* concept. Each pair $(\alpha, \beta) \in \Psi_i$ is assigned such a causer job, denoted by $\mathsf{CJ}(\alpha, \beta)$. Once assigned, the causer job of a pair (α, β) does not change. The key idea is to show that at any time, all pairs will have different causer jobs, but the number of causer jobs is bounded.

A causer job is assigned to a pair (α, β) when it is created, which only happens in Reset, Split and Merge operations. The causer job of a newly created pair is assigned according to the following rules:

- When Reset resets Ψ_i to its initial state which only contains one pair $(1, n_k)$, we set its causer job by:
$$\mathsf{CJ}(1, n_k) \leftarrow \text{the idle job } J_\perp.$$
- When Split splits a pair (α_m, β_m) into two pairs (α_m, μ) and $(\mu+1, \beta_m)$ at time t, we set:
$$\mathsf{CJ}(\mu+1, \beta_m) \leftarrow \mathsf{CJ}(\alpha_m, \beta_m). \tag{17}$$
- When Merge merges $(\alpha_1, \beta_1), \ldots, (\alpha_m, \beta_m)$ into one pair (α, β) at time t, we set:
$$\mathsf{CJ}(\alpha, \beta) \leftarrow \text{the job that was executing at } t^-. \tag{18}$$

Note that in the Split operation we do not need to assign the causer job to the first resutling pair (α_m, μ) since later in the Merge operation we will do this for either this pair or the new pair merging this pair with the ones before it.

We have the following properties for the causer jobs:

Lemma VI.2. *For any pair* (α, β) *in* Ψ_k *we have:*
$$\Lambda_k(\beta) \leq \mathsf{prt}(\mathsf{CJ}(\alpha, \beta)). \tag{19}$$

Proof: We prove by induction on the number of times for which Reset, Split or Merge have been applied to Ψ_k.

The base case considers the initial state of Ψ_k, in which there is only one pair, whose causer job is set to $J_\perp$. The lemma trivially holds in that case.

For the inductive step we show that the condition still holds after a Reset operation or a priority adjustment at an arbitrary time point t.

Reset: Same argument as the base case.

priority adjustment: Recall that the priority adjustment first splits a pair (α_m, β_m) into two pairs (α_m, μ) and $(\mu+1, \beta_m)$ (if needed), and assigns a causer job to the second resulting pair. Then it merges several pairs $(\alpha_1, \beta_1), \ldots, (\alpha_x, \beta_x)$ before the borderline (including the first resulting pair of Split if there is) into a new pair $(\alpha, \beta) = (1, \sum_{j=1}^{x} |(\alpha_j, \beta_j)|)$, and assigns a causer job to it. We first consider the second resulting pair of the Split operation. Condition (19) still holds for this pair, since it inherits the causer job of the original pair. Then we consider the resulting pair (α, β) of merging several pairs $(\alpha_1, \beta_1), \ldots, (\alpha_x, \beta_x)$ before the borderline. First, since the pairs $(\alpha_1, \beta_1), \cdots, (\alpha_x, \beta_x)$ do not overlap with each other, we know $\beta_x \geq \sum_{j=1}^{x} |(\alpha_j, \beta_j)| = \beta$. We also know $\beta_x \leq \mu$ by the definition of Merge, so we have $\beta \leq \mu$, i.e., $\Lambda_k(\beta) \leq \Lambda_k(\mu)$. By the definition of Locate we also know $\Lambda_k(\mu) < \mathsf{prt}(J)$, where J is the the job executing at t^-. So we have $\Lambda_k(\beta) \leq \mathsf{prt}(J)$, and by (18) we finally have $\Lambda_k(\beta) \leq \mathsf{prt}(\mathsf{CJ}(\alpha, \beta))$.

∎

Lemma VI.3. *Suppose a job* J_{cur} *is preempted at time* t*. After the priority adjustments at* t*, for each task* τ_k *we have*
$$\forall (\alpha, \beta) \in \Psi_k : \mathsf{prt}(\mathsf{CJ}(\alpha, \beta)) \geq \mathsf{prt}(J_{cur}). \tag{20}$$

Proof: We will prove by induction on the number of preemptions since system start.

The base case is when the system starts and the idle job $J_\perp$ is preempted. In this case, there is only one pair in Ψ_k, and its causer job is $J_\perp$, so the lemma holds for the base case.

The inductive step is to show if the condition holds *before* a preemption, it will still hold *after* a preemption. We use Ψ'_k to denote the state of Ψ_k after the priority adjustment at t.

The preemption at t happens because some job J_i is released at t, and finally gets a priority higher than the J_{cur}. So we know the priority adjustment (reset) was invoked at t: If the priority adjustment (or Reset) was *not* invoked, then $\mathsf{epp}(J_i, t^-) \geq \mathsf{prt}(J_{cur})$ must be true (see line 8 in Figure 1),

[3] The acceptance ratio of a scheduling algorithm (schedulability test) is the ratio between the number of task sets that are deemed to be MC-schedulable by the algorithm (test), and the total number of the task sets in the experiment.

and without the priority adjustment (or Reset) J_i's final priority won't promote, and will not preempt $\mathsf{prt}(J_{cur})$. So we can conclude that at t, either Reset or the priority adjustment (Locate, Split and Merge) is performed.

Reset: Same argument as the base case.

Priority adjustment: If the priority adjustment is performed at t, then there can be two types of pairs in Ψ_k': the pairs that already exist in Ψ_k, and the pairs that are newly created during the adjustment.

We first consider the pairs that already exist in Ψ_k. Each such pair (α, β) is unchanged because it satisfied $\mu < \alpha$ with μ being the index returned by Locate. From the definition of Locate we thus know that $\mathsf{prt}(J_{cur}) \leq \Lambda_k(\alpha) \leq \Lambda_k(\beta)$. Further, we have from Lemma VI.2 that $\Lambda_k(\beta) \leq \mathsf{prt}(\mathsf{CJ}(\alpha, \beta))$ and can conclude $\mathsf{prt}(\mathsf{CJ}(\alpha, \beta)) \geq \mathsf{prt}(J_{cur})$.

Second, we consider the pairs that are newly created in Split or Merge, focusing on Split first. Suppose a pair (α_m, β_m) is split into two pairs, and a causer job is assigned to the second resulting pair $(\mu+1, \beta_m)$. By Lemma VI.2 we know $\Lambda_k(\beta_m) \leq \mathsf{prt}(\mathsf{CJ}(\alpha_m, \beta_m))$, and by (17) we have

$$\Lambda_k(\beta_m) \leq \mathsf{prt}(\mathsf{CJ}(\mu+1, \beta_m)). \tag{21}$$

By the definition of Split, the split pair (α_m, β_m) satisfies $\mu < \beta_m$, which implies $\Lambda_k(\mu+1) \leq \Lambda_k(\beta_m)$. By combining this and (21) we have

$$\Lambda_k(\mu+1) \leq \mathsf{prt}(\mathsf{CJ}(\mu+1, \beta_m)). \tag{22}$$

By the definition of Locate we know μ is the maximal index of Λ_k satisfying $\Lambda_k(\mu) < \mathsf{prt}(J_{cur})$, so we know $\Lambda_k(\mu+1) \geq \mathsf{prt}(J_{cur})$. By this and (22), we have $\mathsf{prt}(J_{cur}) \leq \mathsf{prt}(\mathsf{CJ}(\mu+1, \beta_m))$, so the lemma also holds for the second resulting pair $(\mu+1, \beta_m)$.

Finally we consider the pair newly created in Merge. By the causer assignment rule in Merge (18), we know its causer job is set to be J_{cur}, so the lemma still holds for this new pair.

■

Lemma VI.4. *If some element of Ψ_k is split at time t_s, then at t_s after the priority adjustment operations, all causer jobs in Ψ_k are active (started execution but not yet finished) jobs. The idle job is considered to be always active.*

Proof: By definition, it's clear that a job may become a causer job only after it has started execution. So we only need to prove that a causer job has not been finished.

We prove by contradiction. Suppose a pair of Ψ_k is split at time t_s, and let J_s be the job executing at t_s^-, i.e., the job that was preempted at t_s. We assume a causer job J_f has finished at some time point $t_f < t_s$.

A job can only become a causer job when it is preempted, i.e., before it is finished. So if this job is *not* a causer job at a time point t after it is finished, it can not become a causer job after t. Therefore, since J_f has finished at t_f and it is still a causer job at t_s, we know J_f has became a causer job before t_f, and has been continuously being a causer job in $[t_f, t_s]$.

Now we know that at time t_f, job J_f signalled completion, and later at time t_s a job J_s is preempted which has a higher priority than causer job J_f because of Lemma VI.3. Thus, we know from Lemma IV.4 that some job J_l with

$$\mathsf{prt}(J_l) > \mathsf{prt}(J_f) \tag{23}$$

is preempted at some time point $t_l \in (t_f, t_s)$.

On the other hand, by Lemma VI.3 we know that after J_l is preempted at $t_l \in (t_f, t_s)$, all the causer jobs of Ψ_k have priority lower than $\mathsf{prt}(J_l)$. Since J_f is continuously being a causer job in $[t_f, t_s]$, and particularly, J_f is a causer job at t_l, we have $\mathsf{prt}(J_f) > \mathsf{prt}(J_l)$, which contradicts with (23). ■

Lemma VI.5. *At any time a priority plan Ψ_k contains at most $N+1$ pairs.*

Proof: The size of Ψ_k can grow only when the Split operation is executed, and by Lemma VI.4 we know that after the splitting, all the causer jobs of Ψ_k are active jobs (including the idle job). Since at any time each task has at most one active job[4], the number of active jobs in the system at any time is at most $N+1$ (from N tasks plus the idle job). Therefore we know the number of causer jobs related to the pairs in Ψ_k is at most $N+1$.

Next we prove that no two pairs in Ψ_k share the same causer job. According to the causer job assignment rules, there are only two opportunities to introduce a new causer job: (1) the Reset operation and (2) the Merge operation. After the Reset operation there is only one pair in Ψ_k, so this will clearly not lead to any causer job sharing. In the following we focus on the Merge operation. We prove by contradiction, assuming that at some time point t several pairs are merged into (α, β) and it gets causer job J, which is the same as the one of another pair (α', β') in Ψ_k. In this case it must be true that $\beta < \beta'$ since all the pairs with smaller indices than β have been merged into (α, β). By Lemma VI.2 we also know that $\Lambda_k(\beta') \leq \mathsf{prt}(J)$ (note that J is the job executing at t^-). So due to the existence of (α', β'), we know (α, β) is *not* the last pair whose largest index directing to the priority equal to or higher than the preempted job J, which contradicts the definition of the Merge operation.

By now we have shown the number of causer jobs related to the pairs in Ψ_k is at most $N+1$, and each pair in Ψ_k has a distinguished causer job, so the number of pairs in Ψ_k is bounded by $N+1$.

■

By Lemma VI.1 and VI.5 we know the operation on each priority plan is of complexity $\mathsf{O}(N)$, and since there are N

[4]For sporadic tasks with constrained deadlines this is clearly true. For sporadic tasks with arbitrary deadlines this is also true. The intuition is that, in PLRS a job will never get a higher priority than an earlier job released by the same task before this earlier job is finished. The formal proof for this claim is omitted due to the space limit.

priority plans in the system, we can conclude the main result of this section:

Theorem VI.6. *The run-time priority management of* PLRS *is of complexity* $\mathsf{O}(N^2)$.

A. Comparing with LB

Now we can see that the computational complexity of PLRS's run-time scheduling is significantly superior to LB. What about the comparison of their overheads in practise? Indeed, the number of jobs involved in LB's run-time recomputation is typically very large, especially for the systems with higher workload and/or more criticality levels. The run-time overhead of PLRS can be of several orders of magnitude smaller than LB for common task systems.

One may expect that the average-case overhead of LB is not as bad as its worst-case bound. However, in the certification on high criticality levels, we need a *safe* upper bound on the run-time overhead. Therefore, even if in many cases the average-case run-time overhead of LB is not very expensive, we still have to adopt its pseudo-polynomial worst-case bound in the certification (on high criticality levels), which would be unacceptable in many realistic systems.

VII. Conclusion and Future Work

In this paper we present an algorithm PLRS to schedule certifiable mixed-criticality sporadic task systems on a preemptive uniprocessor machine. To better balance the asymmetric interference between different criticality levels, PLRS employs the flexible priority assignment principle OCBP, which has been proven very effective for the simple model of a finite set of jobs with known release times. Applying the OCBP principle to sporadic tasks is a difficult problem since a sporadic task will potentially generate a infinite number of jobs, and the release time of each job is not known a priori. The previous algorithm LB solved this problem by on-line recomputing the future job priority assignment, which results in both poor real-time performance and pseudo-polynomially large run-time overhead. Our new algorithm PLRS addressed both of these two problems. First, PLRS not only theoretically dominates, but also on average significantly outperforms LB in terms of acceptance ratios. Second, the run-time complexity of PLRS is *polynomial* (quadratic in the number of tasks), which is much more efficient than the pseudo-polynomial run-time scheduling in LB. In practise, PLRS's run-time overhead can be several orders of magnitude smaller than LB's.

We consider the certifiable mixed-criticality scheduling problem to be highly relevant in the design of future real-time embedded systems and cyber-physical systems, especially when the system is deployed on *multi-core* platforms. On multi-cores, the gap between the safe estimation and the typical measurement of a program's execution time can be huge due to the non-deterministic resource contention. As future work, we plan to extend PLRS to global multiprocessor scheduling. Our preliminary work indicates that such an extension is not trivial: directly applying PLRS to multiprocessor setting would cause deadline miss. The reason is similar to the key challenge in the traditional multiprocessor scheduling problem (of non-MC task systems), that the synchronous task release pattern is not necessarily the worst-case scenario. Since the (abstract) critical-instant in PLRS is also based on the synchronous task release pattern, the same problem raises in applying PLRS to multiprocessor scheduling. Another potential direction of our future work is to study the scheduling of certifiable mixed-criticality task systems with inter-task dependencies and shared resources.

References

[1] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Stanfill, D. Stuart, and R. Urzi. Mcar white paper: A research agenda for mixed-criticality systems. In *CPS Week 2009 Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling Real-Time Mixed-Criticality Jobs. In *the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2010.

[3] S. Baruah, H. Li, and L. Stougie. Mixed-criticality scheduling: Improved resource-augmentation results. In *the ISCA 25th International Conference on Computers and Their Applications (CATA)*, 2010.

[4] 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.

[5] 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.

[6] G. Buttazzo. Hard real-time computing systems: Predictable scheduling algorithms and applications, second edition. 2005.

[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.

[8] F. Dorin, P. Richard, M. Richard, and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems.*, 46:305–331, December 2010.

[9] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *Journal of ACM*, 2000.

[10] K. Lakshmanan, D. de Niz, and R. Rajkumar. Resource allocation in distributed mixed-criticality cyber-physical systems. In *the 30th International Conference on Distributed Computing Systems (ICDCS)*, 2010.

[11] 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.

[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.

[13] H. Li and S. Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *the 10th ACM international conference on Embedded software (EMSOFT)*, 2010.

[14] J. W. S. Liu. *Real-time systems*. Prentice Hall, 2000.

[15] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. *the 7th IEEE International Conferences on Embedded Software and Systems (ICESS)*, 2010.

[16] R. Pellizzoni, P. Meredith, M. Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed criticality in soc-based realtime embedded systems. In *the 9th ACM IEEE International Conference on Embedded software (EMSOFT)*, 2009.

[17] S. Petters, M. Lawitzky, R. Heffernan, and K. Elphinstone. Towards real multi-criticality scheduling. In *the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2009.

[18] 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.