In this paper we present an algorithm called PLRS scheduling techniques may result in unacceptable resource waste. 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 certifications with different interests and assumptions. 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 analog to the well-known relation be-
The mixed-criticality scheduling problem was first identified and formalized by Vestal in [17], where he proposed a fixed-task-priority algorithm to schedule such systems. Dorin et al. [8] formally proved that the algorithm in [17] is optimal in the scope of fixed-task-priority preemptive algorithms. However, as pointed out by Baruah and Vestal [5], the algorithm in [17] 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 provides a higher scheduling flexibility 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 schedulability 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. [15] proposed a reservations-based approach to ensure strong isolation among subsystems of different criticalities. Petters et al. [16] 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. [14] 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

In this section we define the Mixed-Criticality (MC) sporadic task system studied in this paper. 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: \( J_i = (a_i, d_i, \ell_i, c_i) \), where
- \( a_i \in \mathbb{R}_+ \) is the release time.
- \( d_i \in \mathbb{R}_+ \) is the (absolute) deadline.
- \( \ell_i \in [1, 2, \ldots, L] \) 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 \( \ell^{th} \) element in the vector, denoted by \( c_i(\ell) \), specifies the worst-case execution time (WCET) estimate of job \( J_i \) at criticality level \( \ell \).

In real-time scheduling literatures, a smaller priority value represents a higher priority. We follow this convention for criticality values: a smaller criticality value represents a higher priority. Further, we assume \( \forall \ell < \ell' : c_i(\ell) \geq c_i(\ell') \). This corresponds to the fact that the execution time estimation on a higher criticality level is more conservative.

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 \( \gamma_i \). However, the value of \( \gamma_i \) 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 \( \lambda \)-criticality behavior, where
\[
\lambda = \max\{ \ell | \gamma_i \leq c_i(\ell) \}
\]
If it does not signal completion upon having executed for \( c_i(1) \) (1 is the highest criticality level), its behavior is erroneous.

B. MC Tasks
Each MC sporadic task is characterized by a 4-tuple: \( \tau_k = (D_k, T_k, \ell_k, C_k) \), where
- \( D_k \in \mathbb{R}_+ \) is the relative deadline.
- \( T_k \in \mathbb{R}_+ \) is the minimal release separation (period).
- \( \ell_k \in [1, 2, \ldots, L] \) is the critical level of the task.
- \( C_k \in \mathbb{R}_+^L \) is a vector. The \( \ell^{th} \) element in the vector, denoted by \( C_k(\ell) \), specifies the worst-case execution time (WCET) estimate of task \( \tau_k \) at criticality level \( \ell \).

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 \( \tau \) consists of \( N \) independent MC tasks. Each MC task \( \tau_k \) 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 \( \tau_k \).

C. MC-Schedulability
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-\( \lambda \), if the highest criticality level of any job’s behavior in the system is \( \lambda \). If any job in the system exhibits erroneous behavior, the system’s behavior is erroneous, denoted by 0. We define the MC-schedulability under a scheduling algorithm \( A \) as follows:

**Definition III.1 (MC-schedulability).** Under a given scheduling algorithm \( A \), a job \( J_i \) is MC-schedulable if and only if for criticality-\( \lambda \) system behavior the following implication holds:
\[
\lambda \geq \ell_i \Rightarrow J_i \text{ has finished by } d_i
\]
A task \( \tau_i \) is MC-schedulable if and only if all the jobs released by \( \tau_i \) are MC-schedulable. An MC task system \( \tau \) is MC-schedulable if and only if all the tasks in \( \tau \) 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 \( \ell_i \), 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 \( \ell_i \), meeting \( J_i \)’s deadline is not required; for authorities at a criticality level lower than or equal to \( \ell_i \), 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:

- PLRS’s performance is significantly improved over LB. The schedulability test of PLRS not only theoretically dominates LB’s, but also provides a significantly higher average-case performance in terms of acceptance ratio.
- The run-time scheduling 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 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 $\tau_k$.

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 $\tau_k$ is related to a number $\delta_k$ which denotes the number of $\tau_k$’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 $\tau_k$’s largest index job $J_k^{i_k}$ is eligible to be assigned the lowest priority if it satisfies the condition:

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

(1)

The LHS of the condition represents the total workload of all the higher priority jobs in $I$ if the system’s behavior is of criticality level $\ell_k$, and the RHS is the minimal distance between the absolute deadline of $J_k^{i_k}$ and the beginning of the busy interval. So if the LHS does not exceed RHS, we can guarantee that $J_k^{i_k}$ 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 $J_k^{i_k}$, 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, 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, a job may still be not MC-schedulable if all jobs are scheduled strictly following this priority assignment.

\(^1\)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 much larger according to the bound in \([12]\). Here we only consider a subset of these jobs for illustration.

### 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:

<table>
<thead>
<tr>
<th>Task</th>
<th>$T_i$</th>
<th>$D_i$</th>
<th>$\ell_i$</th>
<th>$C_i(1)$</th>
<th>$C_i(2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>10</td>
<td>10</td>
<td>2 (low)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>20</td>
<td>20</td>
<td>1 (high)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>30</td>
<td>30</td>
<td>2 (low)</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>50</td>
<td>50</td>
<td>1 (high)</td>
<td>25</td>
<td>15</td>
</tr>
</tbody>
</table>

In Table I, we have $\tau_1$ with $\ell_1 = 2$, $\tau_2$ with $\ell_2 = 1$, $\tau_3$ with $\ell_3 = 2$, and $\tau_4$ with $\ell_4 = 1$. Given $\delta_1 = 6$, $\delta_2 = 3$, $\delta_3 = 2$, and $\delta_4 = 1$. We choose $J_1^1$ to illustrate the usage of Condition (1). At the step of assigning the lowest priority to $J_1^1$, the remaining jobs are $I = \{J_2^1, J_2^2, J_2^3, J_3^2, J_3^3, J_4^2\}$, i.e., $\delta_1 = 3$, $\delta_2 = 2$, $\delta_3 = 1$, and $\delta_4 = 1$. Since $J_1^1$’s criticality level is 1, so the LHS of Condition (1) equals:

$$\delta_1 \times C_1(1) + \delta_2 \times C_2(1) + \delta_3 \times C_3(1) + \delta_4 \times C_4(1) = 47$$

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

$$(\delta_1 - 1) \times T_1 + D_4 = 50$$

So Condition (1) is true for $J_1^1$ 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 $\Lambda_k$ for each task $\tau_k$, by collecting the priorities assigned to the jobs of task $\tau_k$ 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 $\Lambda_k$ for each task is as follows. We have $\Lambda_1(4) = 8$ since the fourth priority value in $\Lambda_1$ is 8.

| $\Lambda_1$ | 1 | 2 | 4 | 8 | 9 | 12 |
| $\Lambda_2$ | 3 | 6 | 10 |
| $\Lambda_3$ | 5 | 11 |
| $\Lambda_4$ | 7 |

### B. Run-Time Scheduling

The run-time scheduling of PLRS has two major parts: (i) criticality monitoring/control, and (ii) priority management.

PLRS’s criticality monitoring/control mechanism works in the same way as OCBP and LB: The scheduler keeps track of how long each job has been executing. Whenever some job $J_i$ executes for more than $c_i(\ell)$ without signalling that it has completed execution (which means the system is in criticality level $\ell - 1$), the scheduler discards all the tasks with criticality equal to or lower than $\ell$.
In the following we introduce the novel priority management mechanism of PLRS. PLRS is fixed-job-priority preemptive scheduling. Each job \( J \) will get a priority \( \text{prt}(J) \) when it is released. For each task \( \tau_k \), PLRS at run time maintains a plan for the priorities of its future jobs that can be released in the current busy interval. We use \( \Psi_k \) to denote \( \tau_k \)'s priority plan, which records a set of indices directing to the priority values in \( \Lambda_k \). According to the priority plan, the first future job will get the priority in \( \Lambda_k \) identified by the smallest index stored in \( \Psi_k \), and the next future job will get the priority identified by the second smallest index and so on.

We can use a pair \((\alpha, \beta)\) to abstractly represent several consecutive indices in \( \Psi_k \), where \( \alpha \) is the first one and \( \beta \) the last one of these consecutive indices. So we can represent \( \Psi_k \) by a set of such pairs \( \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots \} \). 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 \( \Psi_k \) is the key for PLRS to have polynomial run-time complexity.

When a job \( J_i \) of task \( \tau_i \) is released, PLRS executes the priority management routine \( \text{PrtMng}(J_i) \) 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 \( \text{PrtMng}(J_i) \), and later in Section VI we discuss its computational complexity.

\( \text{PrtMng}(J_i) \) 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 \( J_{\text{cur}} \) is running, then \( \text{PrtMng}(J_i) \) compares \( \text{prt}(J_{\text{cur}}) \) with the planned priority of the released job \( J_i \) according to the current plan \( \Psi_i \). If \( J_i \)'s planned priority is higher, \( \text{PrtMng}(J_i) \) 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 \( \tau_k \), the adjustment will find a “borderline” in its priority plan \( \Psi_k \) according to the priority of the currently running job \( J_{\text{cur}} \). Then the indices directing to priorities higher than \( \text{prt}(J_{\text{cur}}) \) will be promoted (become smaller) as much as possible, while the others remain unchanged.

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

1. **Reset(\( \Psi_i \))** resets \( \Psi_i \) to its initial state:
   \[
   \Psi_i' \leftarrow \{(1, n_i)\}
   \]
   where \( n_i \) is the maximal number of jobs \( \tau_i \) can release in a busy interval.

2. **GetFirst(\( \Psi_i \))** returns \( \Lambda_i(\alpha_1) \), where \( \Lambda_i \) is the priority list of \( \tau_i \) and \( \alpha_1 \) is the first index in the first pair of \( \Psi_i \).

3. **Locate(\( \Psi_k, \mu \))** returns the largest index \( x \) with an entry strictly less than \( \mu \), i.e., \( \max \{ x \mid \Lambda(x) < \mu \} \).

4. **Split(\( \Psi_k, \mu \))** divides the pair \((\alpha, \beta)\) satisfying \( \alpha \leq \beta \), if there is one, into two pairs:
   \[
   (\alpha, \beta) \leftarrow \{(\alpha', \beta'), (\alpha'' \beta'')\}
   \]
   where \( \alpha' = \alpha, \beta' = \mu + 1, \beta'' = \beta \).

5. **Merge(\( \Psi_k, \mu \))** merges the pairs \((\alpha_1, \beta_1), \ldots, (\alpha_m, \beta_m)\) into one pair, where \( \alpha_m, \beta_m \) is the last pair satisfying \( \beta_m \leq \mu \):
   \[
   \Psi'_1 \leftarrow \{(\alpha, \beta), (\alpha_{m+1}, \beta_{m+1})\}
   \]
   and the resulting new pair \((\alpha, \beta)\) is:
   \[
   (\alpha, \beta) \leftarrow (1, \sum_{x=1}^{\infty} (|\alpha_n, \beta_n|))
   \]

6. **RmvFirst(\( \Psi_i \))** removes the first index represented in \( \Psi_i \):
   \[
   \Psi'_i \leftarrow \{(\alpha_1 + 1, \beta_1), (\alpha_2, \beta_2), \ldots \} \quad \text{if} \quad \alpha_1 < \beta_1
   \]
   \[
   \{\alpha_2, \beta_2, \ldots \} \quad \text{if} \quad \alpha_1 = \beta_1
   \]

**Example IV.3.** We use the task system in Example IV.1 and the priority lists in Example IV.2 to illustrate how \( \text{PrtMng}(J_i) \) works. We assume that all the tasks release the first job at time 0, and the initial priority plans of \( \tau_1 \) and \( \tau_2 \) are shown in Figure 3(a). Figure 3(b) shows their priority plans at time 36, just before the release of \( J_2^3 \). Comparing with Figure 3(a), both \( \Psi_1 \) and \( \Psi_2 \) have shrunk for one index due to the release of \( J_1^4 \) and \( J_2^4 \). When \( J_2^3 \) is released, by \( \text{GetFirst}(\Psi_2) \) we know its planned priority is 6, which is higher than the currently running job \( J_1^4 \)'s priority 11, so \( \text{PrtMng}(J_i) \) executes line 9 to line 13 to adjust each task’s priority plan. For \( \tau_1 \), the adjustment first uses \( \text{GetFirst}(\Lambda_1, 11) \) to find the borderline 5, which is the largest index in \( \Lambda_1 \) directing to a priority higher than the currently running job’s priority 11. \( \text{Split}(\Psi_1, 5) \) splits the pair \((2, 6)\), which is crossed by the borderline 5, into two pairs \((2, 5)\) and \((6, 6)\), and \( \text{Merge}(\Psi_1, 5) \) 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
In this section we will show that the schedulability of PLRS is determined by off-line computation, i.e., any task set $\tau$ 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 $J_\bot$ with the lowest priority $+\infty$. Any job released by the task system has higher priority than $J_\bot$ and thereby preempts $J_\bot$, 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 $\Psi$). 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 $\text{epp}(J,t)$. Intuitively, $\text{epp}(J,t)$ 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 $\text{epp}(J_1,40) = 12$, since if the unreleased jobs $J_2^0 \cdots J_6^0$ strictly follow the priority plan at time 40, which is the same as Figure 3-(c), then $J_1^0$ will get the priority 12. Below is the formal definition of $\text{epp}(J,t)$, in which (6) describes how to parse the information in $\Psi_i$ 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 $\tau_i$ and $\tau_i$ has released $y$ jobs by $t$. Let $\Psi_i = \{(\alpha_1, \beta_2), (\alpha_2, \beta_2), \cdots\}$ be $\tau_i$’s priority plan at $t$ (after priority adjustment if there is any). Then $J$’s expected priority $\text{epp}(J,t)$ at time $t$ is defined as follows:

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

where $(\alpha_m, \beta_m)$ is the pair in $\Psi_i$ satisfying:

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

Note that $x > y$ since $t$ is strictly before $J$’s release time, otherwise $\text{epp}(J,t)$ 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 word, the system is always reset to the initial state before all the indices in $\Psi_i$ 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 $\text{epp}(J,t^-)$ to denote the concept of $J$’s priority value at time $t$ before all priority adjustments at $t$.

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

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

These properties follow directly from the definition of $\text{epp}(J,t)$ 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 $\tau$, but $\tau$ is not MC-schedulable by PLRS’s run-time scheduling. We let a job...
the off-line computation of $J_i$ be the first job that is not MC-schedulable, i.e., the system behavior is no higher than $J_i$’s criticality level $\ell_i$ 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.

The proof will focus on the workload that occurs in a particular time interval:

**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 $\text{prt}(J_i)$ (possibly the idle job $J_{\perp}$) is preempted.*

**Lemma V.4.** Suppose $J_i$ is the first job that is not MC-schedulable. $J_i$’s deadline is $d_i$ and $(t_0, d_i]$ is the problem window. Then any job $J_k$ that executed in $(t_0, d_i]$ satisfies both of the following conditions:

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

**Proof:** We prove the first claim by contradiction. Suppose $J_k$ is a job released before $t_0$, which executes in $(t_0, d_i]$. At $t_0$ a job whose priority is lower than $\text{prt}(J_i)$ is preempted, so we know $J_k$’s priority is also lower than $\text{prt}(J_i)$ (otherwise $J_i$ will not execute, and thereby will not be preempted at $t_0$).

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 no lower than $\text{prt}(J_i)$, which 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 $\text{prt}(J_i)$ is preempted.

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

$$\text{epp}(J_k, t_0) > \text{epp}(J_i, t_0)$$

Note that $J_k$ could be the idle job.

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

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

Consider case 1). Since $\text{prt}(J_k) < \text{epp}(J_k, t_0)$, 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 $\text{epp}(J_k, t_0)$, i.e., $t_1$ satisfies:

$$\text{epp}(J_k, t_1) = \text{epp}(J_k, t_0)$$

and

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

Then by the third property of Lemma V.2 we know it must be true that at $t_1$ some job $J_l$ (possibly the idle job $J_{\perp}$) with priority satisfying

$$\text{prt}(J_l) > \text{epp}(J_k, t_1)$$

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

$$\text{prt}(J_l) > \text{epp}(J_k, t_0)$$

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

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

Since $J_k$’s priority is lower than $J_i$’s, there 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 $\text{prt}(J_i)$, which 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 $\text{prt}(J_i)$ is preempted.

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

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

**Theorem V.5.** Any MC task system $\tau$ 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 latest at $t_0$, 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 $\tau_i$ in $I_1$. Since $\tau_i$ succeeds with PLRS’s off-line computation, Condition (1) holds for $\tau_i$ at each step in the off-line computation, and in particular, the following holds:

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

where $\delta^x_{\tau_j}$ denotes the number of $\tau_j$’s jobs that had not been assigned yet when assigning the priority of the $x$th job of $\tau_i$ 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 $\text{epp}(J_k, t_0) \leq \text{epp}(J_i, t_0)$. Therefore, the number of jobs in $I_1$ for any task $\tau_j$ at most $\delta^x_{\tau_j}$ (otherwise some of $\tau_j$’s jobs in $I_1$ will end up with expected priorities lower than $J_i$’s). So we have the following:

$$\sum_{J_j \in I_1} c_j(\ell_i) \leq \sum_{\tau_j \in \tau} \delta^x_{\tau_j} \times C_j(\ell_i)$$

Since $J_i$ is the $x$th job in $I_1$, we know

$$(x - 1) \times T_i + D_i \leq d_i - t_0$$

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

$$\sum_{J_j \in I_1} c_j(\ell_i) \leq d_i - t_0$$
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 \( \ell_i \) 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 \( \sum_{j \in I_1} c_j(\ell_i) \). And since at least one of these jobs \((J_i)\) has not finished yet by \( d_i \), we have

\[
\sum_{j \in I_1} c_j(\ell_i) > d_i - t_0
\]

which contradicts with (14).

### A. Comparing with LB

The load bound test condition for LB in [12] only works for systems with two criticality levels. However, one can use the knowledge from [3] to derive the load bounds for systems with arbitrary number of criticality levels.

The off-line computation algorithm of PLRS is essentially the same as the run-time re-computation 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.6.** PLRS’s schedulability test in Theorem V.5 dominates 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 (detailed experiment results can be found in the appendix).

### 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 \( \Psi_i \). We will show that the number of elements in each task priority plan \( \Psi_i \) 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 \( O(N) \).

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 \( O(N^2) \).

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

Proof: The proofs for Reset, GetFirst, Split, Merge and RmvFirst are straightforward. Assuming an implementation of \( \Psi_i \) 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 \( \Lambda_i \) 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 the following we will prove that the number of pairs in each priority plan \( \Psi_i \) is bounded by \( N+1 \). We start with a lemma that will be used later.

**Lemma VI.2.** Suppose a job \( J_a \) signalled completion at time \( t_a \), and later at time \( t_b \) another job \( J_b \) with \( \text{prt}(J_b) < \text{prt}(J_a) \) is pre-empted. Then there must be a job \( J_c \) with \( \text{prt}(J_c) > \text{prt}(J_a) \) which is pre-empted 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 \( \text{prt}(J_c) > \text{prt}(J_a) \) which is pre-empted 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\(^4\), 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 \( J_x \) starts execution at \( t_b \). It follows that all these jobs have lower and lower priorities, so we have

\[
\text{prt}(J_a) < \text{prt}(J_1) < \text{prt}(J_2) < \cdots < \text{prt}(J_x) < \text{prt}(J_b).
\]

This contradicts the assumption \( \text{prt}(J_0) < \text{prt}(J_a) \).

In order to show that there are at most \( N+1 \) elements in \( \Psi_i \), we introduce the *causer job* concept. Each pair \((\alpha, \beta) \in \Psi_i \) is assigned such a causer job, denoted by \( \text{CJ}(\alpha, \beta) \). Once assigned, the causer job of a pair \((\alpha, \beta) \) 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 \((\alpha, \beta) \) when it is created, which only happens in Reset, Split and Merge. The causer job of a newly created pair is assigned by the following rules:

- **When Reset** resets \( \Psi_i \) to its initial state which only contains one pair \((1, n_b)\), we set its causer job by:
  \[
  \text{CJ}(1, n_b) \leftarrow \text{the idle job } J_{\perp}
  \]

- **When Split** splits a pair \((\alpha_m, \beta_m)\) into two pairs \((\alpha_m, \mu)\) and \((\mu+1, \beta_m)\) at time \( t \), we set:
  \[
  \text{CJ}(\alpha_m, \mu) \leftarrow \text{the job that was executing at } t^-
  \]

  \[
  \text{CJ}(\mu+1, \beta_m) \leftarrow \text{CJ}(\alpha_m, \beta_m)
  \]

- **When Merge** merges \((\alpha_1, \beta_1), \ldots, (\alpha_m, \beta_m)\) into one pair \((\alpha, \beta)\) at time \( t \), we set:
  \[
  \text{CJ}(\alpha, \beta) \leftarrow \text{the job that was executing at } t^-
  \]

\(^4\)There is a technical subtlety here: a higher priority job \( J_0 \) may be released at the same time as a lower priority job \( J_1 \) finished its work. In this case, we construct the scheduler so that \( J_1 \) temporally does not signal completion, but will be pre-empted by \( J_0 \), and wait until the earliest time instant when \( J_1 \) 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 \( J_1 \) signals completion. Therefore, right after a job \( J_1 \) signalled completion, the processor must be running a job with priority lower than \( J_1 \) (including the idle job) to execute.
We have the following properties for the causer jobs:

Lemma VI.3. For any pair $(\alpha, \beta)$ in $\Psi_k$, we have:

$$\Lambda_k(\beta) \leq \text{prt}(CJ(\alpha, \beta))$$

(18)

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

The base case considers the initial state of $\Psi_k$, 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 is invariant for the operations Reset, Split and Merge.

Reset: Same argument as the base case.

Split: Recall that this operation splits a pair $(\alpha_m, \beta_m)$ into two pairs $(\alpha_m, \mu)$ and $(\mu + 1, \beta_m)$. Suppose it is executed at time $t$, and let $J$ be job executing at $t^-$. By the Locate operation we know $\Lambda_k(\mu) < \text{prt}(J)$, and by (15) we know $\text{prt}(J) = CJ(\alpha_m, \mu)$. So (18) still holds for the first resulting pair $(\alpha_m, \mu)$. It also holds for the other pair $(\mu + 1, \beta_m)$ by induction hypothesis since this pair gets the causer job of the old pair $(\alpha_m, \beta_m)$ assigned.

Merge: Recall that this operation merges several pairs $(\alpha_1, \beta_1), \ldots, (\alpha_m, \beta_m)$ into a new pair $(\alpha, \beta) = \left(1, \sum_{x=1}^{m} |(\alpha_x, \beta_x)| \right)$. First, since the pairs $(\alpha_1, \beta_1), \ldots, (\alpha_m, \beta_m)$ do not overlap with each other, we know $\beta_m \leq \beta_m \geq \sum_{x=1}^{m} |(\alpha_x, \beta_x)| = \beta$. We also know $\beta_m \leq \mu$ 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 $\Lambda_k(\mu) < \text{prt}(J)$, so we have $\Lambda_k(\beta) \leq \text{prt}(J)$. By (17) we finally have $\Lambda_k(\beta) \leq \text{prt}(CJ(\alpha, \beta))$.

Lemma VI.4. Suppose a job $J_{\text{cur}}$ is preempted at time $t$. After the priority adjustments at $t$, for each task $t_k$ we have

$$\forall(\alpha, \beta) \in \Psi_k : \text{prt}(CJ(\alpha, \beta)) \geq \text{prt}(J_{\text{cur}})$$

(19)

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

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 $\Psi_k$, 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 $\Psi_k$ to denote the state of $\Psi_k$ 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 $J_{\text{cur}}$. So we know the priority adjustment (reset) was invoked at $t$: If the priority adjustment (or Reset) was not invoked, then $\text{epp}(J_i, t^-) \geq \text{prt}(J_{\text{cur}})$ must be true (see line 8 in Figure 1), and without the priority adjustment (or Reset) $J_i$’s final priority won’t promote, and will not preempt $\text{prt}(J_{\text{cur}})$. 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 $\Psi_k$: the pairs that already exist in $\Psi_k$, and the pairs that are newly created during the adjustment.

We first consider the pairs that already exist in $\Psi_k$. Each such pair $(\alpha, \beta)$ is preempted because it satisfied $\mu < \alpha$ with $\mu$ being the index returned by Locate. From the definition of Locate we thus know that $\text{prt}(J_{\text{cur}}) \leq \Lambda_k(\alpha) \leq \Lambda_k(\beta)$. Further, we have from Lemma VI.3 that $\Lambda_k(\beta) \leq \text{prt}(CJ(\alpha, \beta))$ and can conclude $\text{prt}(CJ(\alpha, \beta)) \geq \text{prt}(J_{\text{cur}})$.

Second, we consider the pairs that are newly created in Split or Merge, focusing on Split first. Suppose a pair $(\alpha_m, \beta_m)$ is split into two pairs $(\alpha_m, \mu)$ and $(\mu + 1, \beta_m)$. By (15) we know (19) still holds for the first resulting pair $(\alpha_m, \mu)$. We consider now the priority of the causer job of the second resulting pair $(\mu + 1, \beta_m)$. By Lemma VI.3 we know $\Lambda_k(\beta_m) \leq \text{prt}(CJ(\alpha, \beta_m))$, and by (16) we have

$$\Lambda_k(\beta_m) \leq \text{prt}(CJ(\mu + 1, \beta_m))$$

(20)

By the definition of Split, the split pair $(\alpha_m, \beta_m)$ satisfies $\mu < \beta_m$, which implies $\Lambda_k(\mu + 1) \leq \Lambda_k(\beta_m)$. By combining this and (20) we have

$$\Lambda_k(\mu + 1) \leq \text{prt}(CJ(\mu + 1, \beta_m))$$

(21)

By the definition of Locate we know $\mu$ is the maximal index of $\Lambda_k$ satisfying $\Lambda_k(\mu) < \text{prt}(J_{\text{cur}})$, so we know $\Lambda_k(\mu + 1) \geq \text{prt}(J_{\text{cur}})$. By this and (21), we have $\text{prt}(J_{\text{cur}}) \leq \text{prt}(CJ(\mu + 1, \beta_m))$, 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 (17), we know its causer job is set to be $J_{\text{cur}}$, so the lemma still holds for this new pair.

Lemma VI.5. If some element of $\Psi_k$ is split at time $t_s$, then at $t_s$ after the priority adjustment operations, all the causer jobs in $\Psi_k$ are active (released but not yet finished) jobs. The idle job is considered to be always active.

Proof: It’s clear that a job may become a causer job only after it is released. So we only need to prove that a causer job has not been finished.

We prove by contradiction. Suppose a pair of $\Psi_k$ is split at time $t_s$, and let $J_i$ 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 point time $t_f < t_s$.

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 become 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 $t_f$ a 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.4. Thus, we know from Lemma VI.2 that some job $J_i$ with

$$\text{prt}(J_i) > \text{prt}(J_f)$$

(22)
is preempted at some time point \( t_J \in (t_f, t_s) \).

On the other hand, by Lemma VI.4 we know that after \( J_I \) is preempted at \( t_J \in (t_f, t_s) \), all the causer jobs of \( \Psi_k \) have priority lower than \( \text{pr}(J_I) \). Since \( J_f \) is continuously being a causer job in \( [t_f, t_s] \), and particularly, \( J_f \) is a causer job at \( t_J \), we have \( \text{pr}(J_f) > \text{pr}(J_I) \), which contradicts with (22).

**Lemma VI.6.** At any time a priority plan \( \Psi_k \) contains at most \( N + 1 \) pairs.

**Proof:** The size of \( \Psi_k \) can grow only when the Split operation is executed, and by Lemma VI.5 we know that after the splitting, all the causer jobs of \( \Psi_k \) are active jobs (including the idle job). Since at any time each task has at most one active job\(^5\), 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 pairs in \( \Psi_k \) is at most \( N + 1 \).

By Lemma VI.1 and VI.6 we know the operation on each priority plan is of complexity \( O(N) \), and since there are \( N \) priority plans in the system, we can conclude the main result of this section:

**Theorem VI.7.** The run-time priority management of PLRS is of complexity \( O(N^2) \).

### A. Comparing with LB

It is clear that the computational complexity of PLRS’s run-time scheduling is significantly superior to LB. What about the comparison of their overheads in practice? 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

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 this work to the multiprocessor setting, as well as to systems with non-preemptable shared resources and task precedence constraints.

### References


### Appendix

**Experimental Comparison of PLRS and LB**

We first introduce LB’s load bounds in Section A, then present the experimental comparison results in Section B.
A. LB’s Load Bounds

We start with introducing the load concept in the context of MC task systems. In traditional (non MC) real-time systems (see, e.g., [12, page 81]), 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. Informally, the load represents a lower bound on the whole task system over the interval, normalized by the interval length. For MC systems, we introduce the load concept analogously, which is the maximum of the cumulative execution requirement by the whole task system over the interval, normalized by the interval length.

**Definition A.1.** The criticality-\(\ell\)-load of a MC task system \(\tau\) is defined by:

\[
Ld_{\ell}(\tau) = \max_{0 \leq t_1 \leq t_2} \left\{ \sum_{\forall j: t_j \leq t_1, t_j \leq \tau_j, \tau_j \leq t_2} c_i(\ell) / (t_2 - t_1) \right\}
\]

For any criticality level \(\ell\), \(Ld_{\ell}(\tau)\) can be computed using well-known techniques [6] for determining the loads of traditional (i.e., non MC) sporadic task systems. Intuitively, \(Ld_{\ell}(\tau)\) represents a lower bound on the portion of processing capacity required by this task system with which it can meet all deadlines. Formally, the load 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* \(\ell\). 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.

We also define the concept of average load of a MC task system as:

\[
Ld_{avg}(\tau) = \frac{\sum_{\ell=1}^{L} Ld_{\ell}(\tau)}{L}
\]

Clearly \(Ld_{avg}(\tau) \leq 1\) is also a necessary condition for a MC task set to be MC-schedulable. 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:

\[
\max\{Ld_{\ell}(\tau)|\ell \in [1, L]\} \leq \text{LoadBound}(L) \quad (23)
\]

where \(Ld_{\ell}\) is the load of criticality level \(\ell\).

LoadBound\((L)\) is a function with respect to the total number of criticality levels \(L\) of the system, which is recursively calculated as follows:

\[
\begin{align*}
\text{LoadBound}(1) &= 1 \\
\text{LoadBound}(L) &= \frac{2}{1 + \sqrt{4(1/\text{LoadBound}(L-1))^2 + 1}}
\end{align*}
\]

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

\[
(Ld_2(\tau))^2 + Ld_1(\tau) \leq 1. \quad (24)
\]

B. Experimental Comparison

We compare the acceptance ratio of LB’s load bounds presented above, and the schedulability test of PLRS in Theorem V.5 by randomly generated MC task sets. We first introduce our method of generating MC task sets. For each group of experiments, we set a fixed number of criticality levels in the system \(L\), and generate the first task set with \(L\) tasks. Then we iteratively increase the number of tasks by 1 to generate a new task set, until the average load \(Ld_{avg}(\tau)\). Then the above process is repeated, starting with a new task set of \(L\) tasks, until a reasonable sample space has been generated. To generate each task set, we let the possibility for each task to be on each criticality level be uniformly distributed. And we set the following parameter ranges for each task set:

- \([T_{down}, T_{up}]\): The range of a task’s period. For each task, we randomly choose a number in \([T_{down}, T_{up}]\) as its period.
- \([DoT_{down}, DoT_{up}]\): The range of the ratio between a task’s period and its deadline. For each task, we randomly choose a number in \([DoT_{down}, DoT_{up}]\) to multiple this task’s period, to get its relative deadline.
- \([U_{down}, U_{up}]\): The range of a task’s utilization. For each task, we randomly choose a number in \([U_{down}, U_{up}]\) and multiply it with this task’s period, to get its execution time on the lowest criticality level \(L\).
- \([Z_{down}, Z_{up}]\): The range of the ratio between a task’s execution time on two consecutive levels. For each task, we randomly choose a number in \([Z_{down}, Z_{up}]\) and multiply it with its execution time on the second lowest criticality level, to get its execution time on the second lowest criticality level. Then we randomly choose another number in \([Z_{down}, Z_{up}]\) and multiply it with its execution time on the second lowest criticality level, to get its execution time on the third lowest criticality level. We repeat this process until its execution times on all criticality levels have been generated.

The parameter setting of the experiments in Figure 4-(a) is as follows: \(L = 2\), \(T_{down} = 50\), \(T_{up} = 200\), \(DoT_{down} = 0.9\), \(DoT_{up} = 1.1\), \(U_{down} = 0.02\), \(U_{up} = 0.2\), \(Z_{down} = 1\). We set different \(Z_{up}\). For example, the curve LB-[1,2] represents the acceptance ratio of LB’s load bound test with \(Z_{up} = 2\), and the curve \(LB - [1,3]\) represents the one with \(Z_{up} = 3\). The x-axis of the figure is the average load \(Ld_{avg}(\tau)\), and the y-axis is the acceptance ratio.

The setting for Figure 4-(b) changes the number of criticality levels \(L\) to 3, while keeps other setting the same as (a). In Figure 4-(c) the \(Z_{up}\) is fixed, and we tune the number of criticality levels in each group of experiments.

From Figure 4 we can see that the acceptance ratio of PLRS indeed significantly outperform LB, especially for the task systems with more than 2 criticality levels.
Fig. 4. Acceptance ratio comparison between PLRS and LB.