Recursive Partitioned Scheduling for Real-Time Gang Tasks

Seongtae Lee
Dept. of Computer Science & Engineering
Sungkyunkwan University
Republic of Korea
yuns0509@skku.edu

Nan Guan
Dept. of Computer Science
City University of Hong Kong
Hong Kong SAR
nanguan@cityu.edu.hk

Jinkyu Lee*
Dept. of Computer Science & Engineering
Yonsei University
Republic of Korea
jinkyu.lee@yonsei.ac.kr

Abstract—**The development of parallel computing architectures has created a growing need for scheduling real-time gang tasks, in which a specified number of threads per task must be executed simultaneously under timing constraints. However, existing approaches struggle to handle a fundamental challenge—the heterogeneity in the number of threads across gang tasks. To address the challenge, this paper proposes a novel scheduling framework, called *Recursive Partitioned Scheduling* (RPS), in which each partition can be recursively divided into subpartitions whose assigned processor sets are disjoint and collectively equal to that of the parent, forming a tree-like hierarchical structure. RPS provides a flexible interface that allows each task to be assigned to an appropriate level in the hierarchy based on the number of threads it requires. To fully exploit RPS, we adopt fixed-priority scheduling and address two key issues. First, we develop a tight schedulability analysis, which not only utilizes the well-known exact schedulability analysis results for uniprocessor scheduling but also leverages the relationship between intra- and inter-partition interference. Second, based on the insights from the analysis, we design an effective partition generation and task assignment algorithm specialized for RPS, and further enhance it through task priority reassignment. Simulation results demonstrate that our approach significantly outperforms existing approaches in terms of schedulability.**

I. Introduction

Due to advances in parallel computing architectures [1], [2], gang scheduling has been widely studied in the real-time systems area, in which the pre-defined number of threads of a gang task should be executed on different processors at the same time. A fundamental challenge in guaranteeing the timeliness of gang tasks arises from the heterogeneity of tasks in terms of the number of threads to be executed in parallel, as the problem of scheduling gang tasks with the same number of threads reduces to scheduling non-parallel tasks (each occupying only one processor) on multiprocessors. To address the challenge, there are two primary directions that leverage different processor-task mapping modes: (i) the *global* scheduling in which the threads of a gang task can be executed on any set of processors at run-time [3]–[9], and (ii) the *stationary* scheduling in which a task should be executed on a designated set of processors whose number is the same as that of threads of the task [10]. Recently, a third approach, known as *Strictly Partitioned Scheduling* (SPS), has gained attention due to its empirically superior schedulability performance [11], where gang tasks and processors are statically divided into disjoint sets, allowing each partition to be scheduled independently.[1]

Although SPS is conceptually simple, it has opened a new avenue for gang scheduling. From the perspective of decomposition, it provides a framework that allows each partition to be scheduled using global or stationary scheduling or, alternatively, to execute tasks exclusively (therefore reducing the scheduling problem to uniprocessor scheduling and utilizing its exact schedulability analysis). However, SPS has inherent limitations in addressing the fundamental challenge of gang scheduling—the heterogeneity of tasks with respect to the number of threads, as follows.

L1. If there exists even a single task whose number of threads is equal to (or close to) the total number of processors, partitioning becomes infeasible. In such cases, the scheduling problem reverts to global or stationary scheduling, negating the advantages of SPS.

L2. Even when partitioning is possible, significant variation in the number of threads among tasks within a partition can lead to poor schedulability performance. That is, if tasks are scheduled exclusively within the partition, underutilization is inevitable; alternatively, applying global or stationary scheduling within a partition inherits the pessimism in calculating interference.

To address the limitations of SPS, we propose a novel scheduling framework, called *Recursive Partitioned Scheduling* (RPS), for real-time gang tasks. While SPS allows only a single level of disjoint partitions, RPS adopts a tree-like hierarchical structure with partitions organized across multiple levels. Specifically, each partition in RPS can be recursively subdivided into disjoint partitions at the next level. A task is then assigned to one of the partitions located at any level in this hierarchy, based on the number of threads it requires. RPS is a generalization of SPS, as it becomes equivalent to SPS

*Jinkyu Lee (corresponding author) initiated this work while affiliated with Sungkyunkwan University.

[1]For non-parallel tasks (each occupying only one processor), the concepts of strictly partitioned and stationary scheduling in gang scheduling can be interpreted as clustered scheduling [12] and partitioned scheduling, respectively.

when task assignments are restricted to only the first-depth partitions.

Although RPS provides a flexible interface that addresses the limitations L1 and L2, realizing its full potential requires careful management of direct/indirect interference among tasks assigned to same/different partitions. Therefore, the effectiveness of RPS heavily depends on two key components: (i) the proper selection of a prioritization policy in conjunction with a processor-task mapping mode, and (ii) the development of partition generation and task assignment algorithms that collectively minimize such interference. For (i), we adopt fixed-priority exclusive-execution scheduling for RPS (RPS-FP-EE). Since interference only occurs from higher-priority tasks to lower-priority ones under FP, combining FP with exclusive execution, where only one task can be executed in each partition, has the potential to minimize or even eliminate cascading interference across partitions. For (ii), to be specialized for RPS-FP-EE, we identify the following questions for design requirements.

R1. In RPS-FP-EE, what is the relationship between intra-partition and inter-partition interference, and how can we develop a schedulability analysis that accurately models and effectively leverages this relationship?

R2. Based on the insights from R1, how can we design an effective partition generation and task assignment algorithm under a given fixed-priority ordering?

R3. Extending R2, how can we further enhance the partitioning and assignment process by actively incorporating priority reassignment to efficiently exploit the schedulability analysis framework developed in R1?

To address R1, we classify higher-priority tasks that influence the execution of a target task into two categories: `DHP` and `IHP` (Directly/Indirectly interfering Higher-Priority tasks). This classification enables a structured understanding of the relationship between intra-partition and inter-partition interference. Building upon the notions of `DHP` and `IHP`, we propose a method for identifying the largest subset of higher-priority tasks to which tight interference analysis can be applied, leveraging the exact interference bounds from uniprocessor FP scheduling. To this end, we formally derive the necessary conditions that such a subset must satisfy with respect to `DHP` and `IHP` relationships. By applying tight interference analysis to the identified subset, we develop a schedulability test for RPS-FP-EE that accurately and interpretably captures cascading interference across partitions.

For R2, we design a partitioning and task assignment strategy that realizes the full potential of RPS under FP-EE. Specifically, we propose a hierarchical algorithm that incrementally constructs partition trees (with their root partitions). Each task is first assigned to an existing schedulable partition whenever possible. If no such partition exists, the algorithm either creates a new partition tree or subdivides an existing partition to accommodate the task. This process preserves schedulability at every step by applying the developed tight schedulability analysis. The algorithm also accounts for both task parallelism (the number of threads it requires) and intra-/inter-partition interference, guiding partitioning and task-to-partition mapping decisions.

To address R3, we investigate how task priorities influence the schedulability under the answers of R1 and R2. We identify specific tasks whose priorities significantly affect interference bounds, and mitigate analysis pessimism by promoting their priorities in the process of performing the proposed partitioning & task assignment framework. This enhancement is integrated through systematic modifications, ensuring both compatibility with the original framework and correctness of schedulability.

We evaluate the schedulability performance of our partitioning & task assignment algorithm (i.e., the answer of R2) and its improved version with priority reassignment (i.e., the answer of R3), both of which leverage the proposed schedulability analysis for RPS-FP-EE (i.e., the answer of R1). Our approach demonstrates superior schedulability performance compared to state-of-the-art methods, particularly when task parallelism heterogeneity is significant.

This paper makes the following key contributions.

- Development of a novel concept of recursive partitioned scheduling (RPS) and strategies for its effective use.
- Design of a schedulability analysis tailored for RPS-FP-EE,
- Proposal of an effective partitioning and task assignment method based on the developed analysis,
- Enhancement of RPS-FP-EE performance through priority reassignment, and
- Demonstration of the effectiveness of the proposed RPS-FP-EE via extensive simulation studies.

II. System Model

We consider a system of m identical processors $\Pi = \{P_x\}_{x=1}^{m}$, executing sporadic rigid gang tasks $\tau = \{\tau_i\}_{i=1}^{n}$, as widely used (e.g., [6], [8]–[11]). Each task τ_i invokes a series of jobs with the minimum inter-arrival time (or period) T_i, the worst case execution time C_i, the relative deadline D_i, and the parallelism m_i. Once a job of τ_i is released at t, it should be performed for at most C_i time units no later than its absolute deadline at $t + D_i$; also, whenever the job of τ_i is performed, it requires m_i threads to be executed simultaneously on different m_i processors. We target a set of implicit-deadline (i.e., $D_i = T_i$) or constrained-deadline (i.e., $D_i \leq T_i$) tasks. We call a job *active* at t, if it has remaining execution at t. The response time of τ_i (denoted by R_i) is an upper bound of the time interval length between the release and finishing time of all jobs invoked by τ_i; therefore, a job of τ_i released at t finishes its execution by $t + R_i$. Although the concept of RPS can be applied to non-preemptive scheduling, this paper focuses on the preemptive setting, meaning that every job is preemptable without any preemption cost. Let $|X|$ denote the number of elements in X.

A task set τ is said to be *schedulable* by a scheduling algorithm on m processors, if there is no single job deadline miss for every legal job sequence generated by τ when it is scheduled by the scheduling algorithm on m processors.

III. Recursive Partitioned Scheduling (RPS)

In this section, we present our design of RPS and provide a roadmap for achieving its full potential.

A. Designing RPS

RPS consists of two key components: (i) constructing partitions and (ii) mapping tasks to partitions. In the partition construction aspect, while strict partitioned scheduling (SPS) divides the entire processor set Π into disjoint partitions, recursive partitioned scheduling (RPS) introduces a hierarchical structure by recursively subdividing each partition into subpartitions, enabling further division at each depth level. In RPS, the outermost disjoint partitions—referred to as partition trees and corresponding to the disjoint partitions in SPS—are formally defined as follows.

Definition 1: Under RPS, the entire processor set Π is disjointly assigned to a set of *partition trees* $\mathcal{TR} = \{\mathcal{TR}^r\}$. A set of processors assigned to a partition tree $\mathcal{TR}^r$ is denoted by $\mathcal{A}(\mathcal{TR}^r)$. By definition, $\mathcal{A}(\mathcal{TR}^r) \cap \mathcal{A}(\mathcal{TR}^p) = \emptyset$ holds for $r \neq p$, and $\bigcup_{\forall r} \mathcal{A}(\mathcal{TR}^r) = \Pi$ holds.

Then, each partition tree can be divided in a recursive manner, formally as follows.

Definition 2: Under RPS, a partition tree $\mathcal{TR}^r$ with an assigned processor set $\mathcal{A}(\mathcal{TR}^r)$ has a root node (i.e., root partition) whose assigned processors are identical to those of the partition tree itself. Each parent partition in $\mathcal{TR}^r$ can be recursively subdivided into a set of child partitions whose assigned processor sets are disjoint and collectively equal to that of the parent.

Each partition in $\mathcal{TR}^r$ is denoted by ρ_v^r, where v is the partition index. For simplicity, we use ρ_v when the partition tree index r is not required for clarity.

From the task mapping perspective, we design RPS so that each task is mapped to leaf partition(s) as follows.

Definition 3 (Task-partition mapping): Under RPS, each task τ_i is assigned to exactly one partition tree. Within the assigned partition tree, a task is mapped to at least one (possibly multiple) leaf nodes (i.e., leaf partitions); let $m_i(\rho_v)$ denote the number of processors allocated to τ_i within the given leaf partition ρ_v (See Example 2 in this section and Example 4 in the next section). Also, let $\mathcal{L}(\mathcal{TR}^r)$ denote the set of leaf partitions in $\mathcal{TR}^r$. The total number of processors assigned to a set of leaf partitions to which each task τ_i is mapped must be no less than m_i; otherwise, τ_i is not eligible for execution.

Under RPS, most (if not all) task scheduling strategies can be applied independently within each partition tree; this includes a combination of (i) scheduling modes such as global, stationary, or exclusive-execution scheduling and (ii) prioritization schemes such as FP and EDF. An important requirement is that a prioritization order must be defined among jobs that share the same partition.

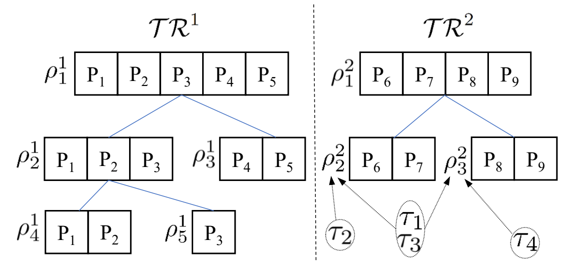


Fig. 1: Illustration of hierarchical structure and task-partition mapping in RPS

We assume each task τ_i utilizes at least one processor from each of its assigned partitions whenever it is executed. If this condition is not met, it implies that at least one of the assigned partitions is redundant and should not have been assigned to τ_i. For notations, let $\mathcal{L}(\tau_i)$ denote a set of leaf partitions that τ_i belongs to. Also, let $\mathcal{A}(\rho_v)$, $|\mathcal{A}(\rho_v)|$, and $\tau(\rho_v)$ denote a set of processors assigned to ρ_v, the number of processors assigned to ρ_v, and a set of tasks that belong to ρ_v, respectively.

Example 1: Fig. 1 illustrates an example of an RPS structure, consisting of two partition trees $\mathcal{TR}^1$ and $\mathcal{TR}^2$, with $\mathcal{A}(\mathcal{TR}^1)$=$\{P_1, P_2, P_3, P_4, P_5\}$ and $\mathcal{A}(\mathcal{TR}^2)$=$\{P_6, P_7, P_8\}$. Within $\mathcal{TR}^1$, there are five partitions ρ_1^1 to ρ_5^1, where ρ_1^1 is the root partition, and ρ_3^1, ρ_4^1, and ρ_5^1 are leaf partitions. Likewise, $\mathcal{TR}^2$ has three partitions: the root partition ρ_1^2, and the leaf partitions ρ_2^2 and ρ_3^2. Note that if a partition tree consists of only a root partition, the root partition also serves as a leaf partition.

Example 2: Here is an example of a task-to-partition mapping. There are a set of tasks τ_1 with m_1=4, τ_2 with m_2=2, τ_3 with m_3=3, and τ_4 with m_4=2, mapped to $\mathcal{TR}^2$ in Fig. 1 as follows. $\mathcal{L}(\tau_1)$=$\{\rho_2^2, \rho_3^2\}$ with $m_1(\rho_2^2)$=2, and $m_1(\rho_3^2)$=2, $\mathcal{L}(\tau_2)$=$\{\rho_2^2\}$ with $m_2(\rho_2^2)$=2, $\mathcal{L}(\tau_3)$=$\{\rho_2^2, \rho_3^2\}$ with $m_3(\rho_2^2)$=2, and $m_3(\rho_3^2)$=1, and $\mathcal{L}(\tau_4)$=$\{\rho_3^2\}$ with $m_4(\rho_3^2)$=2.

We design RPS so that tasks are assigned only to leaf nodes. However, if all child nodes of an intermediate node ρ_v are among the nodes to which a task τ_i is assigned, then τ_i can be regarded as being assigned to the intermediate node ρ_v. This allows the task assignment to be equivalently expressed as if tasks could be assigned to intermediate nodes. For example, if RPS were designed to allow tasks to be assigned to intermediate nodes, τ_1 and τ_3 in Fig. 1 could be regarded as being assigned to $\{\rho_1^2\}$ instead of $\{\rho_2^2, \rho_3^2\}$. Note that when a task is assigned to $\{\rho_3^1, \rho_4^1\}$, it is not equivalent to being assigned to $\{\rho_1^1\}$.

Lemma 1: RPS has the following properties.

- RPS is a generalization of SPS.
- The execution of a task within a given partition tree does not affect the execution of tasks in other partition trees, ensuring inter-tree isolation.

Proof: The SPS is equivalent to RPS when every partition tree contains only a root partition, without any further subdivision. Inter-tree isolation is ensured by the non-overlapping processor sets between partition trees (by Definition 1) and the unique assignment of each task to a single tree (by Definition 3). ■

B. Leveraging RPS

RPS provides a flexible interface that allows each task to be assigned to an appropriate level in the hierarchy based on the number of threads it requires. This results in the following pros and cons.

- *Pros*: By recursively dividing a parent partition into child partitions, RPS ensures that there is no direct interference between tasks assigned only to different child partitions, offering significant potential for improving schedulability.
- *Cons*: Since each task can be assigned to multiple partitions, indirect interference can occur. For example, consider τ_k belongs to ρ_1^r and ρ_2^r, and τ_i belongs to ρ_2^r and ρ_3^r in $\mathcal{TR}^r$. Then, $\tau_h (\neq \tau_k)$ belonging to ρ_1^r may affect the execution of $\tau_g (\neq \tau_i)$ belonging to ρ_3^r, even though τ_h and τ_g do not share any partition; this is because of the existence of τ_k and τ_i that shares the same partition ρ_2^r.

To leverage the advantage while mitigating the disadvantage for RPS, we adopt FP as a prioritization policy and EE (exclusive execution) as a scheduling mode, i.e., RPS-FP-EE.

- *Prioritization*: Under FP, interference occurs only in one direction—from higher-priority to lower-priority tasks. This property can be effectively exploited in the partitioning and task assignment to minimize cascading direct/indirect interference.
- *Scheduling mode*: Under EE, only one task can be executed in each partition. Due to its simplicity, EE not only reduces run-time scheduling overhead, but also provides a foundation for simplifying and tightening the schedulability analysis to be developed.

In the following sections, we aim to maximize the schedulability performance of RPS-FP-EE through the following steps.

- We develop a schedulability analysis that tightly and interpretably captures cascading interference across partitions under a given fixed-priority assignment (Section IV).
- Building on this analysis, we propose a partitioning and task-to-partition mapping strategy that considers cascading interference under the fixed-priority order (Section V).
- We further extend our approach by incorporating priority reassignment across partitions to more effectively mitigate cascading interference (Section VI).

IV. Schedulability Analysis for RPS-FP-EE

Under RPS, a task τ_i cannot affect the execution of another task τ_k, if they belong to different partition trees (by Lemma 1). Therefore, targeting a task set τ within a partition tree executed on m processors, this section analyzes the properties of RPS-FP-EE and then develops a schedulability analysis for RPS-FP-EE. Throughout the remainder of this paper, we assume a fixed-priority assignment where tasks with smaller indices have higher priority without loss of generality; i.e., τ_1 has the highest priority and τ_n the lowest.

A. Properties of RPS-FP-EE

Under RPS-FP-EE, while it is straightforward that every task whose priority is lower than τ_k cannot affect the execution of τ_k due to the prioritization of FP, there are two types of higher-priority tasks whose execution affects the execution of τ_k either *directly* or *indirectly*.

Definition 4: $\texttt{DHP}(\tau_k)$ and $\texttt{IHP}(\tau_k)$ are defined as follows.

- $\texttt{DHP}(\tau_k)$ (Directly interfering Higher-Priority task set): a set of higher-priority tasks of τ_k that cannot be executed with τ_k at the same time, which is formally defined as a set of $\tau_i \in \tau$ such that (i) τ_i has a higher priority than τ_k (i.e., $i<k$) and (ii) τ_i is assigned to at least one of the same leaf partitions as τ_k is assigned, i.e., $\texttt{DHP}(\tau_k) = \bigcup_{\rho_v \in \mathcal{L}(\tau_k)} \{\tau_i \in \tau(\rho_v) | i<k\}$.
- $\texttt{IHP}(\tau_k)$: (Indirectly interfering Higher-Priority task set): a set of higher-priority tasks not in $\texttt{DHP}(\tau_k)$, each of which belongs to $\texttt{DHP}$ of $\texttt{DHP}$ of τ_k, $\texttt{DHP}$ of $\texttt{DHP}$ of $\texttt{DHP}$ of τ_k, or so on, which is formally defined as a set of $\tau_i \in \tau \backslash \texttt{DHP}(\tau_k)$ such that there exists a $\texttt{DHP}$ sequence of tasks $\tau_{i_1}, \tau_{i_2}, ..., \tau_{i_q}$ such that (i) $\tau_{i_1}=\tau_i$, (ii) $\tau_{i_q}=\tau_k$, and (iii) $\tau_{i_p} \in \texttt{DHP}(\tau_{i_{p+1}})$ holds for every $p = 1, 2, ...q-1$.

Example 3: Consider a set of tasks $\{\tau_1, \tau_2, \tau_3\}$, assigned to $\mathcal{TR}^1$ in Fig. 1 as follows: $\mathcal{L}(\tau_1) = \{\rho_4^1, \rho_5^1\}$, $\mathcal{L}(\tau_2) = \{\rho_3^1, \rho_4^1, \rho_5^1\}$ and $\mathcal{L}(\tau_3) = \{\rho_3^1\}$. Then, $\texttt{DHP}(\tau_3)=\{\tau_2\}$ and $\texttt{DHP}(\tau_2)=\{\tau_1\}$ hold. On the other hand, $\texttt{IHP}(\tau_3)=\{\tau_1\}$ holds, since (i) τ_1 has a higher priority than τ_3, (ii) τ_1 does not belong to $\texttt{DHP}(\tau_3)$, and (iii) the $\texttt{DHP}$ task sequence exists (i.e., $\tau_2 \in \texttt{DHP}(\tau_3)$ & $\tau_1 \in \texttt{DHP}(\tau_2)$). Therefore, although τ_1 does not directly interfere τ_3, it can affect the execution of τ_3 by directly interfering τ_2 that directly interferes τ_3.

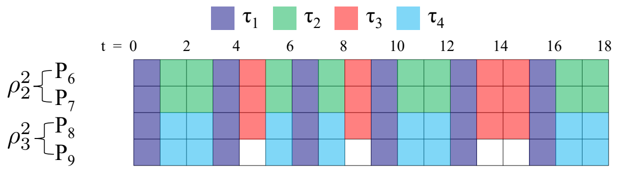
Lemma 2: Under RPS-FP-EE, the following properties hold.

- $\texttt{DHP}(\tau_k) \cap \texttt{IHP}(\tau_k) = \emptyset$.
- The execution of τ_k is indepedent of any task that does not belong to $\texttt{DHP}(\tau_k) \cup \texttt{IHP}(\tau_k)$.

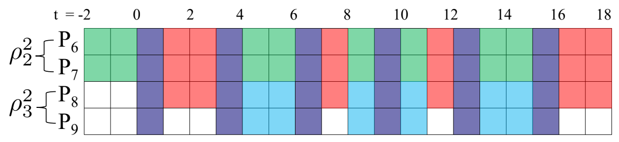
Proof: The properties hold immediately by the definition of $\texttt{DHP}(\tau_k)$ and $\texttt{IHP}(\tau_k)$. ■

Using the above definitions and properties, we establish the following design principles for the schedulability analysis:

- Leveraging the relationship between $\texttt{DHP}$ and $\texttt{IHP}$ to construct an interference-aware schedulability analysis,
- Exploiting the tightness of the existing exact uniprocessor FP schedulability analysis as much as possible, and
- Formalizing these relationships to guide partitioning and task-to-partition assignment in a manner that improves overall schedulability.



(a) Schedule with the synchronous job release



(b) Schedule with the non-synchronous job release

Fig. 2: An example of schedules under RPS-FP-EE

In the next subsections, we first investigate which existing results can be applicable to RPS-FP-EE. Based on the investigation, we derive a tight schedulability analysis for RPS-FP-EE by utilizing the exact schedulability analysis results for uniprocessor scheduling while leveraging the relationship between DHP and IHP. Before proceeding, we define the validity of a task-to-partition assignment, as follows.

Definition 5: A task-to-partition assignment from τ to $\mathcal{TR}^r$ is said to be *valid*, if the following two conditions hold for every $\tau_i \in \tau$: (i) the total number of processors assigned to τ_i across all its partitions must be equal to m_i (i.e., $\sum_{\rho_v \in \mathcal{L}(\tau_i)} m_i(\rho_v) = m_i$), and (ii) the number of processors assigned to τ_i in each partition must not exceed the size of that partition (i.e., $m_i(\rho_v) \leq |\mathcal{A}(\rho_v)|$ holds for every $\rho_v \in \mathcal{L}(\tau_i)$).

B. Existing Results: To Be and Not To Be Utilized

Consider a set of tasks that share the same leaf partition(s). Then, RPS-FP-EE is equivalent to FP uniprocessor scheduling designed for non-parallel tasks, which makes it possible to utilize its exact schedulability analysis [13] as follows.

Lemma 3 (from [13]): Suppose a set of tasks τ is scheduled under RPS-FP-EE within a partition tree $\mathcal{TR}^r$, where the task-to-partition assignment from τ to $\mathcal{TR}^r$ is valid. Consider that all tasks share the same set of leaf partitions $\mathcal{L}(\tau_k)$. Then, τ is schedulable, if there exists $R_k \leq D_k$ that satisfies Eq. (1) for every $\tau_k \in \tau$.

$$R_k = C_k + \sum_{\tau_i \in \texttt{DHP}(\tau_k)} I_i^{\texttt{NO-CI}}(R_k), \tag{1}$$

where $I_i^{\texttt{NO-CI}}(\ell)$ (interference of τ_i in an interval of length ℓ without any carry-in[2] job) can be calculated by

$$I_i^{\texttt{NO-CI}}(\ell) = \left\lceil \frac{\ell}{T_i} \right\rceil \cdot C_i. \tag{2}$$

Since every task is executed on the same partition(s), $\texttt{DHP}(\tau_k)$ is a set of all tasks whose priority is higher than τ_k, i.e., $\texttt{DHP}(\tau_k) = \{\tau_i \in \tau | i < k\}$.

[2]A job is said to be *carry-in* for an interval, if it is released before the interval but has remaining execution at the beginning of the interval.

TABLE I: An example of a task set assigned to a partition tree

Task	T_i	C_i	D_i	m_i	$\mathcal{L}(\tau_i)$	$m_i(\rho_v)$
τ_1	3	1	3	4	$\{\rho_2^2, \rho_3^2\}$	$m_1(\rho_2^2) = 2, m_1(\rho_3^2) = 2$
τ_2	5	2	5	2	$\{\rho_2^2\}$	$m_2(\rho_2^2) = 2$
τ_3	9	2	9	3	$\{\rho_2^2, \rho_3^2\}$	$m_3(\rho_2^2) = 2, m_3(\rho_3^2) = 1$
τ_4	18	8	18	2	$\{\rho_3^2\}$	$m_4(\rho_3^2) = 2$

The lemma holds by the well-known response time analysis for FP uniprocessor scheduling in [13], where R_k in Eq. (1) is obtained by a fixed-point iteration: starting with $R_k = C_k$ on the RHS (Right-Hand-Side) of Eq. (1), the value of R_k is iteratively updated on the LHS (Left-Hand-Side) until convergence. The second term of the RHS of Eq. (1) does not include the contribution of any carry-in job of τ_i in the target interval of length ℓ. This is because of a well-known result for FP uniprocessor scheduling: a critical instant of τ_k (in which the response time of τ_k is maximized) happens when every task whose priority is higher than τ_k is released at the same time of the job of interest of τ_k [14]. However, as shown in the following example, the result does not generally hold for RPS-FP-EE.

Example 4: Consider the following four tasks $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ from Example 2, allocated in a partition tree $\mathcal{TR}^2$ with four processors $\{P_6, P_7, P_8, P_9\}$, shown in Fig. 1 and Table I, scheduled by RPS-FP-EE.

Since τ_4 is executed on ρ_3^2, τ_2 executed on ρ_2^2 does not belong to $\texttt{DHP}(\tau_4)$. If we calculate R_4 based on Lemma 3, we find $R_4 = 18$ such that $C_4 + \lceil \frac{18}{T_1} \rceil \cdot C_1 + \lceil \frac{18}{T_3} \rceil \cdot C_3 = 8 + 6 \cdot 1 + 2 \cdot 2 = 18$ holds in Eq. (1). Note that it is easily checked that the highest-priority tasks τ_1 and τ_2 are schedulable. Fig. 2(a) illustrates the schedule of $[0, 18)$ where the first job of every task is released at $t = 0$ and the following jobs are released periodically, which seems to support the above result where τ_4 is schedulable (i.e., $R_4 = 18 \leq D_4 = 18$).

However, τ_4 is actually not schedulable (i.e., $R_4 = 18$ is wrong), as shown in Fig. 2(b) in which the jobs of τ_2 and τ_3 are periodically released from $t = -2$ while the jobs of τ_1 and τ_4 are periodically released from $t = 0$. In $[0, 18)$, the job of τ_4 executes only for 6 units instead of $C_4 = 8$ units, resulting in a job deadline miss.

Why is Lemma 3 wrong in the example? This is due to the existence of τ_2 that belongs to $\texttt{IHP}(\tau_4)$; although τ_2 cannot directly prevent τ_4 from executing by competing with at least one partition commonly assigned to both τ_2 and τ_4, τ_2 is capable of changing the execution pattern of τ_3 that directly interferes with τ_4. Therefore, we cannot guarantee that the critical instant for τ_4 occurs when a job of τ_3 is released at the same time instant, which was demonstrated in the job deadline miss of τ_4 in Fig. 2(b).

One simple approach to address the above problem is to apply the largest possible interference (i.e., $I_i^{\texttt{CI}}(\ell)$ in Eq. (3)) from every task $\tau_i \in \texttt{DHP}(\tau_k)$ regardless of the critical instant, yielding the following lemma, which was explicitly/implicitly proved/used in many schedulability tests [10], [15], [16] (so we omit the proof).

Lemma 4 (from [10], [15], [16]): Suppose a set of tasks τ is scheduled under RPS-FP-EE within a partition tree $\mathcal{TR}^r$, where the task-to-partition assignment from τ to $\mathcal{TR}^r$ is valid. Then, τ is schedulable, if there exists $R_k \leq D_k$ that satisfies Eq. (3) for every $\tau_k \in \tau$. Note that we calculate R_k from the highest-priority task τ_1 to the lowest-priority task τ_n, sequentially.

$$R_k = C_k + \sum_{\tau_i \in \mathtt{DHP}(\tau_k)} I_i^{\mathtt{CI}}(R_k), \tag{3}$$
$$\text{where } I_i^{\mathtt{CI}}(\ell) = \left\lceil \frac{\ell + R_i - C_i}{T_i} \right\rceil \cdot C_i$$

C. Development of Tighter Schedulability Analysis

Although Lemma 4 is correct, we can easily find that the lemma is pessimistic. For example, if we focus on the task set in Example 4, we know that judging the schedulability of τ_3 in the presence of τ_1 and τ_2 is exactly the same as uniprocessor scheduling. Therefore, it is correct to apply Lemma 3 for the schedulability of τ_3. Then, by adjusting the (T_i, C_i, D_i) parameters of τ_1, τ_2 and/or τ_3 without changing other parameters, we can easily find the case where τ_3 is deemed schedulable by Lemma 3 (implying τ_3 is actually schedulable) but not deemed schedulable by Lemma 4.

To develop a tighter schedulability analysis, we need to systematically identify situations where we can apply the interference term used in Lemma 3 (i.e., $I_i^{\mathtt{NO\text{-}CI}}(\ell)$) instead of that used in Lemma 4 (i.e., $I_i^{\mathtt{CI}}(\ell)$). To this end, we find a set of higher-priority tasks which allow applying the former for the schedulability of τ_k (denoted by $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$), which are defined and exemplified as follows.

Definition 6: Let τ' denote the set of tasks $\tau_j \in \mathtt{DHP}(\tau_k) \cup \{\tau_k\}$ that satisfies (i) $\mathtt{IHP}(\tau_j){=}\emptyset$ and (ii) $\mathtt{DHP}(\tau_j){\subseteq}\mathtt{DHP}(\tau_k)$. Then, $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$ is defined using τ' as follows:

$$\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k) = \bigcup_{\tau_j \in \tau'} \mathtt{DHP}(\tau_j) \cup \{\tau_j\} \setminus \{\tau_k\}. \tag{4}$$

Example 5: Consider the schedulability of five tasks assigned to $\mathcal{TR}^1$ in Fig. 1: τ_1 to $\{\rho_5^1\}$, τ_2 to $\{\rho_3^1, \rho_4^1, \rho_5^1\}$, τ_3 to $\{\rho_4^1, \rho_5^1\}$, τ_4 to $\{\rho_3^1\}$, τ_5 to $\{\rho_3^1, \rho_4^1, \rho_5^1\}$, and τ_6 to $\{\rho_4^1, \rho_5^1\}$. Our target is the schedulability of τ_6, i.e, τ_6 is τ_k in Definition 6. $\mathtt{DHP}(\tau_6) = \{\tau_1, \tau_2, \tau_3, \tau_5\}$, which belong to at least one of ρ_4^1 and ρ_5^1. Then, among $\mathtt{DHP}(\tau_6) \cup \{\tau_6\}$, only τ_1, τ_2 and τ_3 satisfy both (i) and (ii) in Definition 6, because τ_5 and τ_6 violate (ii) and (i), respectively. Therefore, $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_6) = \bigcup_{\tau_j \in \{\tau_1, \tau_2, \tau_3\}} \mathtt{DHP}(\tau_j) \cup \{\tau_j\} \setminus \{\tau_6\} = \{\tau_1, \tau_2, \tau_3\}$.

Note that one may wonder the definition of $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$ in Eq. (4) is too complex and can be replaced by $\tau' \setminus \{\tau_k\}$. Although such a simplified definition is safe in terms of timing guarantees to be developed in Lemma 6 and Theorem 1, it is only a subset of the original definition in Eq. (4), yielding lower schedulability performance.

For the schedulability of τ_k, we claim that it is possible to apply the interference term used in Lemma 3 (i.e., $I_i^{\mathtt{NO\text{-}CI}}(\ell)$) to all tasks in $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$, to be explained step by step.

Lemma 5: Suppose a set of tasks τ is scheduled under RPS-FP-EE within a partition tree $\mathcal{TR}^r$, where the task-to-partition assignment from τ to $\mathcal{TR}^r$ is valid. The execution of every task $\tau_i \in \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k \in \tau)$ is not affected by that of any task that does not belong to $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$.

Proof: Negating the lemma, suppose that $\tau_i \in \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$ is affected by $\tau_g \notin \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$. This means that τ_g belongs to either $\mathtt{DHP}(\tau_i)$ or $\mathtt{IHP}(\tau_i)$, which also implies there exists a $\mathtt{DHP}$ sequence from τ_g to τ_i. For τ_i, there exists τ_j that satisfies Definition 6 such that $\tau_i \in \mathtt{DHP}(\tau_j)$ or $\tau_j = \tau_i$ according to Eq. (4). In case of $\tau_i \in \mathtt{DHP}(\tau_j)$, $\tau_g \in \mathtt{DHP}(\tau_j)$ contradicts Eq. (4), yielding $\tau_g \notin \mathtt{DHP}(\tau_j)$. Since $\tau_i \in \mathtt{DHP}(\tau_j)$ holds and there exists a $\mathtt{DHP}$ sequence from τ_g to τ_i, a sequence from τ_g to τ_j exists yielding $\tau_g \in \mathtt{IHP}(\tau_j)$ (because of $\tau_g \notin \mathtt{DHP}(\tau_j)$), which contradicts the condition of τ_j in Definition 6. The case of $\tau_j = \tau_i$ can be proved similarly. ■

Lemma 6: Suppose a set of tasks τ is scheduled under RPS-FP-EE within a partition tree $\mathcal{TR}^r$, where the task-to-partition assignment from τ to $\mathcal{TR}^r$ is valid. Then, Eq. (5) is an upper bound of the duration in which tasks in $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$ are executed on at least one of τ_k' leaf partitions (i.e., $\mathcal{L}(\tau_k)$) in any interval of length ℓ.

$$\sum_{\tau_i \in \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)} I_i^{\mathtt{NO\text{-}CI}}(\ell) \tag{5}$$

Proof: We construct a task set corresponding to $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$, denoted by $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)'$ as follows. The (T_i', C_i', D_i') parameters of $\tau_i' \in \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)'$ are the same as the (T_i, C_i, D_i) parameters of the corresponding $\tau_i \in \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$. On the other hand $\mathcal{L}(\tau_i')$ is set to $\bigcup_{\tau_j \in \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)} \mathcal{L}(\tau_j) \cup \mathcal{L}(\tau_k)$ and m_i' is set to $|\bigcup_{\tau_j \in \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)} \mathcal{L}(\tau_j) \cup \mathcal{L}(\tau_k)|$. Since RPS-FP-EE is preemptive and work-conserving, the following holds for a pair of a given interval of length ℓ and a given job release pattern: the duration in which jobs of tasks in $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$ are executed on at least one of the partitions in $\mathcal{L}(\tau_k)$ is upper-bounded by the duration for $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)'$. Since every $\tau_i' \in \mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)'$ uses the same partition set, the calculation of the duration in which jobs of tasks in $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)'$ are executed on at least one of the partitions in $\mathcal{L}(\tau_k)$ is equivalent to the uniprocessor scheduling case. Along with Lemma 5, the duration is upper-bounded by Eq. (5) for a pair of any interval of length ℓ and any job release pattern. Therefore, the lemma holds. ■

Using Lemmas 5 and 6, we can develop a tighter schedulability analysis by applying the interference term used in Lemma 3 (i.e., $I_i^{\mathtt{NO\text{-}CI}}(\ell)$) to tasks in $\mathtt{DHP}^{\mathtt{NO\text{-}CI}}(\tau_k)$.

Theorem 1: Suppose a set of tasks τ is scheduled under RPS-FP-EE within a partition tree $\mathcal{TR}^r$, where the task-to-partition assignment from τ to $\mathcal{TR}^r$ is valid. Then, τ is schedulable, if there exists $R_k \leq D_k$ that satisfies Eq. (6) for every $\tau_k \in \tau$. Note that we calculate R_k from τ_1 to τ_n, sequentially.

$$R_k = C_k + \sum_{\texttt{DHP}(\tau_k)} I^*_{i\to k}(R_k), \text{ where} \tag{6}$$

$$I^*_{i\to k}(\ell) = \begin{cases} I_i^{\texttt{NO-CI}}(\ell) = \left\lceil \frac{\ell}{T_i} \right\rceil \cdot C_i, & \text{if } \tau_i \in \texttt{DHP}^{\texttt{NO-CI}}(\tau_k), \\ I_i^{\texttt{CI}}(\ell) = \left\lceil \frac{\ell + R_i - C_i}{T_i} \right\rceil \cdot C_i, & \text{otherwise.} \end{cases} \tag{7}$$

Proof: By Lemma 5, the execution of every task $\tau_i \in \texttt{DHP}^{\texttt{NO-CI}}(\tau_k)$ is not affected by any other tasks, and by Lemma 6, the duration in which jobs of tasks in $\texttt{DHP}^{\texttt{NO-CI}}(\tau_k)$ in an interval of length ℓ prevent τ_k from executing is at most Eq. (5). Also, by [10], [15], [16], the duration in which jobs of each single task τ_i are executed in an interval of length ℓ is upper-bounded by $I_i^{\texttt{CI}}(\ell) = \left\lceil \frac{\ell + R_i - C_i}{T_i} \right\rceil \cdot C_i$.

Therefore, if a job of τ_k released at t cannot finish its execution until $t + R_k$, where R_k satisfies Eq. (6), at least one of Lemmas 5 and 6 contradicts. ■

Time-Complexity. To test Theorem 1, we need to know $\texttt{DHP}(\tau_k)$, $\texttt{IHP}(\tau_k)$ and $\texttt{DHP}^{\texttt{NO-CI}}(\tau_k)$ for every task. To find $\texttt{DHP}(\tau_k)$, we need to count the number of tasks occupying at least one of its partitions $\mathcal{L}(\tau_k)$. Considering the smallest number of processors for each partition is one, the time-complexity of finding $\texttt{DHP}(\tau_k)$ is $O(n \cdot m_k)$. To find $\texttt{IHP}(\tau_k)$, we put a queue initialized with $\texttt{DHP}(\tau_k)$. For each task τ_i to be dequeued, if $\tau_j \in \texttt{DHP}(\tau_i)$ is not in the queue and not in $\texttt{IHP}(\tau_k)$, then enqueue τ_j. Also, if $\tau_i \notin \texttt{DHP}(\tau_k)$, add τ_i to $\texttt{IHP}(\tau_k)$. Since dequeueing occurs at most n times, and for each dequeued task τ_i, we look at $\texttt{DHP}(\tau_i)$, the time-complexity to find $\texttt{IHP}(\tau_k)$ is $O(n^2)$. To find $\texttt{DHP}^{\texttt{NO-CI}}(\tau_k)$, i) for each $\tau_i \in \texttt{DHP}(\tau_k)$, ii) check whether $\texttt{DHP}(\tau_i) \subseteq \texttt{DHP}(\tau_k)$ and $\texttt{IHP}(\tau_i) = \emptyset$. If it is true, add every task $\tau_j \in \texttt{DHP}(\tau_i)$ to $\texttt{DHP}^{\texttt{NO-CI}}(\tau_k)$. By i) and ii), the time-complexity of finding $\texttt{DHP}^{\texttt{NO-CI}}(\tau_k)$ is $O(n^2)$.

Since we need to find $\texttt{DHP}(\tau_k)$, $\texttt{IHP}(\tau_k)$ and $\texttt{DHP}^{\texttt{NO-CI}}(\tau_k)$ for every task, the time-complexity finding them for every task is $O(n^2 \cdot \max(n, \max_{\tau_k \in \tau} m_k))$.

Once we have $\texttt{DHP}(\tau_k)$, $\texttt{IHP}(\tau_k)$ and $\texttt{DHP}^{\texttt{NO-CI}}(\tau_k)$ for every task, the time-complexity for finding R_k that satisfies Eq. (6) is the same as the traditional uniprocessor RTA (equivalent to Lemma 3) using fixed-point iteration techniques, which is $O(n^2 \cdot \max_{\tau_k \in \tau} D_k)$. Therefore, the total time complexity for testing Theorem 1 is $O(n^2 \cdot \max(n, \max_{\tau_k \in \tau} m_k, \max_{\tau_k \in \tau} D_k))$, which is affordable as Theorem 1 is performed offline.

V. Partitioning & Task Assignment for RPS-FP-EE

In this section, we first briefly review the Strictly Partitioned Scheduling (SPS) approach proposed in [11]. Building upon the foundation laid in the previous section, we then develop a Recursive Partitioned Scheduling (RPS) framework applicable to a combination of the prioritization of FP and the scheduling mode of exclusive execution (i.e., FP-EE).

A. Existing Partitioning & Task Assignment for SPS

The core idea of SPS lies in two key decisions:

- (K1) Determining the size of each partition, and
- (K2) Assigning tasks to these partitions.

Prior work on SPS [11] proposes the First-Fit Decreasing Volume (FFDV) heuristic, which leverages the parallelism requirement of each task m_i and is inspired by algorithms for the 2-D strip packing problem [17]. Note that each partition is scheduled using a combination of a scheduling mode (e.g., global, stationary, or exclusive execution) and a prioritization policy (e.g., EDF or FP). The FFDV heuristic first sorts tasks in non-increasing order of their parallelism (volume) m_i. The sorted tasks are then assigned sequentially to a series of partitions $\rho^s = \{\rho_1^s, \rho_2^s, ...\rho_v^s\}$. Each task τ_i is placed in the lowest-indexed existing partition that remains schedulable after the task is added. If no such partition exists, a new partition of size m_i is created and τ_i is assigned to it, provided that sufficient processors are available. The heuristic succeeds if all remaining tasks are successfully assigned; otherwise, it fails.

B. Developing Partitioning & Task Assigning for RPS-FP-EE

RPS extends the key decisions of SPS (i.e., K1 and K2) by introducing two additional key challenges, which are significantly more complex and critical:

- (K3) Determining both the number and size of sub-partitions at each recursive level, and
- (K4) Determining the assignment of tasks to each individual sub-partition.

We propose a framework that addresses K3 and K4 for RPS-FP-EE, in which (i) Algorithm 1 generates a set of disjoint partition trees (with their root partitions) based on the FFDV heuristic, addressing K1 and K2 corresponding to SPS and (ii) Algorithm 2 constructs each partition tree by recursively subdividing its leaf partitions, addressing K3 and K4 specialized for RPS-FP-EE. An important design principle is to initially assign each task to a *single* leaf partition, with the understanding that it may later span two sub-partitions as the leaf partition is recursively subdivided. When a leaf partition ρ_v is subdivided, a task τ_i assigned to ρ_v is reallocated to at least one of its child partitions $\{\rho_{v1}, \rho_{v2}\}$, while also determining $m_i(\rho_{v1})$ and $m_i(\rho_{v2})$, i.e., how many processors τ_i will be used in each sub-partition.

In Algorithm 1, we first sort the task set τ in a non-increasing order of m_i, with ties broken by given task priority, and initialize a set of partition trees $\mathcal{TR}$ and the processor budget (Lines 1–3). Each sorted task τ_i is considered for assignment to the leaf partitions $\rho_v \in \mathcal{L}(\mathcal{TR}^r \in \mathcal{TR})$ examined in order from the lowest to the highest index. The task is assigned to the first leaf partition (with $m_i(\rho_v) \leftarrow m_i$) that satisfies the conditions: the partition's size is no smaller than m_i, and the partition tree $\mathcal{TR}^r$ remains schedulable according to Theorem 1 (Lines 4–10). Note that when the first task is tested, no partition tree has been constructed yet, so the for-loop in Lines 6–10 is skipped, and execution proceeds directly to Line 11.

Algorithm 1 Partition tree generation with task assignment

```
1: Input : $\tau$, $m$
2: $\mathcal{T} \leftarrow \mathrm{sort}(\tau)$ by $m_i \downarrow$ (tie-breaking with given priority)
3: $\mathcal{TR} \leftarrow \emptyset$, *budget* $\leftarrow m$
4: for $\tau_i$ in $\mathcal{T}$ do
5:     *schedulable* $\leftarrow$ Fail
6:     for $\rho_v$ in $\bigcup_{\mathcal{TR}^r \in \mathcal{TR}} \mathcal{L}(\mathcal{TR}^r)$ do
7:         if $|\mathcal{A}(\rho_v)| \geq m_i$ and Theorem 1 holds for $\mathcal{TR}^r$ with $\tau(\rho_v) = \tau(\rho_v) \cup \{\tau_i\}$ then
8:             $\tau(\rho_v) \leftarrow \tau(\rho_v) \cup \{\tau_i\}$, $m_i(\rho_v) \leftarrow m_i$, *schedulable* $\leftarrow$ Pass, and *break*
9:         end if
10:    end for
11:    if *schedulable* $==$ Fail and *budget* $\geq m_i$ then
12:        Construct a new $\mathcal{TR}^r$ (and its root partition $\rho_1^r$) having the next $m_i$ unassigned processors, $\tau(\rho_1^r) \leftarrow \{\tau_i\}$, $m_i(\rho_1^r) \leftarrow m_i$
13:        $\mathcal{TR} \leftarrow \mathcal{TR} \cup \{\mathcal{TR}^r\}$, *budget* $\leftarrow$ *budget* $- m_i$
14:    else if *schedulable* $==$ Fail then
15:        for $\rho_v$ in $\bigcup_{\mathcal{TR}^r \in \mathcal{TR}} \mathcal{L}(\mathcal{TR}^r)$ do
16:            if $|\mathcal{A}(\rho_v)| \geq m_i$ and Algorithm 2 for $\tau(\rho_v) = \tau(\rho_v) \cup \{\tau_i\}$, $\rho_v$ and $\mathcal{TR}^r$ does not return Fail then
17:                Connect the sub-partitions $\rho_{v1}$ and $\rho_{v2}$ generated by Algorithm 2 to $\rho_v$, reassigning the tasks from $\rho_v$ to $\rho_{v1}$ and $\rho_{v2}$, *schedulable* $\leftarrow$ Pass and *break*
18:            end if
19:        end for
20:        if *schedulable* $==$ Fail, then Return *unschedulable*
21:    end if
22: end for
23: Return *schedulable*
```

Algorithm 2 sub-partition generation with task reassignment

```
1: Input : $\tau(\rho_v)$, $\rho_v$, $\mathcal{TR}^r$
2: $\mathcal{T} \leftarrow \mathrm{sort}(\tau(\rho_v))$ by $m_i(\rho_v) \downarrow$ (tie-breaking with given priority)
3: $\rho_{v1}, \rho_{v2} \leftarrow \emptyset$, $\mathcal{T}_{\mathtt{shared}} \leftarrow \emptyset$, $m_{\mathtt{min}} \leftarrow \min_{\tau_i \in \mathcal{T}} m_i(\rho_v)$
4: for $\tau_i$ in $\mathcal{T}$ do
5:     if $m_{\mathtt{min}} + m_i(\rho_v) > |\mathcal{A}(\rho_v)|$, then $\mathcal{T}_{\mathtt{shared}} \leftarrow \mathcal{T}_{\mathtt{shared}} \cup \{\tau_i\}$
6:     else $k \leftarrow i$
7: end for
8: if $\mathcal{T} == \mathcal{T}_{\mathtt{shared}}$ then Return Fail
9: $\mathcal{A}(\rho_{v1}) \leftarrow$ the first $m_k(\rho_v)$ elements of $\mathcal{A}(\rho_v)$, $\tau(\rho_{v1}) \leftarrow \emptyset$
10: $\mathcal{A}(\rho_{v2}) \leftarrow A(\rho_v) \setminus \mathcal{A}(\rho_{v1})$, $\tau(\rho_{v2}) \leftarrow \emptyset$
11: for $\tau_i$ in $\mathcal{T}_{\mathtt{shared}}$ do
12:     $\tau(\rho_{v1}) \leftarrow \tau(\rho_{v1}) \cup \{\tau_i\}$, $\tau(\rho_{v2}) \leftarrow \tau(\rho_{v2}) \cup \{\tau_i\}$
13:     $m_i(\rho_{v1}) \leftarrow |\mathcal{A}(\rho_{v1})|$, $m_i(\rho_{v2}) \leftarrow m_i(\rho_v) - m_i(\rho_{v1})$
14: end for
15: for $\tau_i$ in $\mathcal{T} \setminus \mathcal{T}_{\mathtt{shared}}$ do
16:     *schedulable* $\leftarrow$ Fail
17:     for $\rho_w$ in $\{\rho_{v1}, \rho_{v2}\}$ do
18:         if $|\mathcal{A}(\rho_w)| \geq m_i(\rho_v)$ and Theorem 1 holds for $\mathcal{TR}^r$ connecting $\{\rho_{v1}, \rho_{v2}\}$ to $\rho_v$ and setting $\tau(\rho_w)$ to $\tau(\rho_w) \cup \{\tau_i\}$ then
19:             *schedulable* $\leftarrow$ Pass, $\tau(\rho_w) \leftarrow \tau(\rho_w) \cup \{\tau_i\}$, *break*
20:         end if
21:     end for
22:     if *schedulable*$==$Fail, then Return Fail
23: end for
24: Return the new sub-partitions $\rho_{v1}$ and $\rho_{v2}$
```

If the task cannot be assigned to any existing leaf partition, we check whether a sufficient processor budget is available. If so, a new partition tree $\mathcal{TR}^r$ of size m_i is created, starting with a single root partition, and τ_i is assigned to the root partition with $m_i(\rho_1^r) \leftarrow m_i$. The processor budget is then reduced by m_i accordingly (Lines 11–13). Otherwise, we examine, starting from the lowest-indexed leaf partition, whether the task can be assigned through sub-partitioning (to be explained in Algorithm 2). If successful, the selected leaf partition is sub-partitioned by Algorithm 2, and the resulting sub-partitions are connected to the original leaf partition, replacing the tasks previously assigned to it (Lines 14–19). If τ_i cannot be assigned even after attempting sub-partitioning of all leaf partitions, Algorithm 1 returns *unschedulable* (Line 20). Conversely, if all tasks are successfully assigned, the algorithm returns *schedulable* (Line 23).

Algorithm 2 assigns a new task by dividing a leaf partition ρ_v into two sub-partitions. The tasks originally assigned to the leaf partition are then redistributed to one or both of the resulting sub-partitions, while ensuring the schedulability of the overall partition tree. The procedure consists of five steps:

- (Step 1: Identify shared tasks) Among the tasks assigned to a given leaf partition, identify the shared tasks—those that must appear in both resulting sub-partitions.
- (Step 2: Determine the size of sub-partitions) Among the non-shared tasks, select the one with the largest parallelism $m_i(\rho_v)$ as the size of one of the sub-partitions.
- (Step 3: Assign shared tasks) Insert all shared tasks into both sub-partitions. Note that we do not need to check the schedulability because the shared tasks are already deemed schedulable before sub-partitioning.
- (Step 4: Assign non-shared tasks) Attempt to assign the remaining non-shared tasks to one of the two sub-partitions using a first-fit strategy. Each assignment is accepted only if the resulting partition tree remains schedulable.
- (Step 5: Check completion) If all non-shared tasks are successfully assigned under the schedulability constraint, the sub-partitioning is considered valid.

Algorithm 2 takes as input a leaf partition ρ_v, its assigned tasks $\tau(\rho_v)$, and the partition tree $\mathcal{TR}^r$ containing ρ_v (Line 1), from Algorithm 1, where the addition of a new task renders the partition tree unschedulable without sub-partitioning. Initially, the tasks in $\tau(\rho_v)$ are sorted in non-increasing order of $m_i(\rho_v)$, with ties broken by given task priority. Two sub-partitions, ρ_v^1 and ρ_v^2, are created and initialized to $\emptyset$. A set of shared tasks, $\mathcal{T}_{\mathtt{shared}}$, is also initialized, and $m_{\min}$ is set to the smallest $m_i(\rho_v)$ among all tasks in $\tau(\rho_v)$ (Lines 2–3). Each task τ_i is then examined to determine whether it is a shared task by checking if $m_i(\rho_v) + m_{\min}$ exceeds the total number of processors assigned to ρ_v; if so, τ_i is added to $\mathcal{T}_{\mathtt{shared}}$ (Step 1, Lines 4–7). Note that in Line 8, if all tasks are shared tasks, the algorithm returns Fail (as this case is already checked and failed in Lines 6–10 of Algorithm 1). The size of the first sub-partition ρ_{v1} is determined by the processor requirement of the first non-shared task (i.e., the non-shared task with the largest $m_i(\rho_v)$). The size of the second sub-partition ρ_{v2} is then set to the remaining processors in ρ_v after subtracting the size of the first sub-partition (Step 2, Lines 9–10).

Every shared task τ_i is assigned to both sub-partitions, where τ_i first utilizes all available processors in the first sub-partition (i.e., $m_i(\rho_{v1}) \leftarrow |\mathcal{A}(\rho_{v1})|$), and any remaining processor requirement is fulfilled by the second sub-partition (i.e., $m_i(\rho_{v2}) \leftarrow m_i(\rho_v) - m_i(\rho_{v1})$) (Step 3, Lines 11–14). Each non-shared task is then considered for assignment—first to ρ_v^1, then to ρ_v^2—and is assigned only if the resulting partition tree remains schedulable after the assignment (Step 4, Lines 15–21). If any non-shared task cannot be assigned to either sub-partition without violating schedulability, the algorithm returns `Fail`. Otherwise, if all non-shared tasks are successfully assigned, the algorithm finalizes the two sub-partitions along with their respective task assignments (Step 5, Lines 22 and 24).

Example 6: Consider a scenario where five tasks τ_1 to τ_5, with m_1=2, m_2=5, m_3=2, m_4=3, and m_5=1, construct $\mathcal{TR}^1$ in Fig. 1 (recall tasks with smaller indices have higher priorities); we also assume that there are only five processors P_1 to P_5 in the system. Sorted by m_i, the tasks are tested in the following order: τ_2, τ_4, τ_1, τ_3 and then τ_5. First, τ_2 is assigned ρ_1^1, creating $\mathcal{TR}^1$. Then, suppose that τ_4 is assigned to ρ_1^1 as it satisfies Theorem 1, but τ_1 does not. As a result, by Lines 15–19 of Algorithm 1, Algorithm 2 is triggered for ρ_1^1, where τ_2 becomes a shared task (Lines 4–7), and two sub-partitions are created (ρ_2^1 of size $m_4 = 3$, and ρ_3^1 of size $5-3 = 2$) with τ_2 being assigned to both sub-partitions (Lines 9–14). Then, in Lines 15–23 of Algorithm 2, τ_4 is assigned to ρ_2^1 and τ_1 is assigned to ρ_3^1. Afterwards, suppose that τ_3 is assigned to ρ_2^1, but τ_5 is not, according to Lines 6–10 of Algorithm 1. Similar to ρ_1^1 in the previous step, Algorithm 2 is triggered for ρ_2^1, which is then split into ρ_4^1 of size $m_3 = 2$ and ρ_5^1, with the two shared tasks, τ_2 and τ_4, assigned to both sub-partitions. The non-shared tasks τ_3 and τ_5 are assigned to ρ_4^1 and ρ_5^1, respectively. As a result, $\mathcal{L}(\tau_1) = \{\rho_3^1\}$, $\mathcal{L}(\tau_2) = \{\rho_3^1, \rho_4^1, \rho_5^1\}$, $\mathcal{L}(\tau_3) = \{\rho_4^1\}$, $\mathcal{L}(\tau_4) = \{\rho_4^1, \rho_5^1\}$, and $\mathcal{L}(\tau_5) = \{\rho_5^1\}$.

Theorem 2: Suppose that Algorithm 1 associated with Algorithm 2 returns *schedulable* for a gang task set τ with a given task priority order on m processors. Then, τ is actually schedulable by RPS-FP-EE on m processors.

Proof: Suppose that a deadline miss occurs even though Algorithm 1 returns schedulable. The only scenarios in which Algorithm 1 adds a task to a leaf partition or reallocates a task for sub-partitioning are as follows: (Case 1) Lines 7–8, (Case 2) Lines 12–13, and (Case 3) Lines 16–18 that calls Algorithm 2.

If the deadline miss is caused by Case 1, then Theorem 1 is invalid, which leads to a contradiction. If the deadline miss is caused by Case 2, then it implies that a single task cannot be executed on a root partition, which also leads to a contradiction.

If a deadline miss arises from Case 3, we need to examine Algorithm 2 as follows. First, if the deadline miss occurs for a shared task, then either Line 8 of Algorithm 2 does not return `Fail` (that contradicts Line 8 as it is), or the deadline miss must result from Case 1 or 2 of Algorithm 1 (because shared tasks are already deemed schedulable by Case 1 or 2), which have already been shown to lead to contradiction. Second, if the deadline miss occurs for a non-shared task, then the validity check of Theorem 1 at Line 18 of Algorithm 2 must have failed, which again contradicts.

By contradiction, the theorem holds. ∎

Time-complexity. In Algorithm 2, Line 15 has complexity dependent on $O(n)$, and Line 18 that checks Theorem 1 is tested up to twice (for ρ_{v1} and ρ_{v2}), resulting in a time-complexity of $O(n^3 \cdot \max(n, \max_{\tau_k \in \tau} m_k, \max_{\tau_k \in \tau} D_k))$. Algorithm 1, for every task, checks whether Theorem 1 is satisfied for each leaf partition (Lines 6–10), construct a partition tree (Lines 12–13), and call Algorithm 2 for each leaf partition (Lines 15–19); therefore, the time-complexity is dominated by the last one (calling Algorithm 2). Considering the number of leaf partitions is upper-bounded by m, the time-complexity of Algorithm 1 is $O(n^4 \cdot m \cdot \max(n, \max_{\tau_k \in \tau} m_k, \max_{\tau_k \in \tau} D_k))$, which is affordable as Algorithm 1 is performed offline.

VI. Advanced Partitioning & Task Assignment

The proposed partitioning & task assignment framework presented in the previous section is a general heuristic applicable to a given task priority order. This section enhances the schedulability performance of the framework by reassigning task priorities to adjust the inter-task relationship analyzed in Section IV, based on understanding of an impact of task priorities on schedulability under Algorithm 1 with Algorithm 2.

A. Impact of Task Priorities on Schedulability

According to Theorem 1, the response time of a task τ_k is affected by two of the primary factors: (i) the number of its directly interfering higher-priority tasks, denoted as $\mathtt{DHP}(\tau_k)$, and (ii) whether the interference from these tasks τ_i is computed as $I_i^{\mathtt{CI}}(\ell)$ or $I_i^{\mathtt{NO-CI}}(\ell)$ in Eq. (7) (representing interference with/without carry-in, respectively). Regarding (i), for a given leaf partition ρ_v, the sum of all the number of DHPs of each task within the partition is fixed as $\sum_{x=0}^{|\tau(\rho_v)|-1} x$, which does not change with the task priority ordering.

However, regarding (ii), a shared task τ_i with a higher priority than a task $\tau_k \in \tau(\rho_v)$ can affect how interference is calculated. Specifically, suppose that τ_i is the only shared task assigned to both ρ_v and ρ_w, where there is even higher priority task in ρ_w. Then, due to τ_i, all intermediate-priority tasks $\tau_j \in \rho_v$ with $i < j < k$ impose interference of $I_j^{\mathtt{CI}}(\ell)$ to τ_k (rather than $I_j^{\mathtt{NO-CI}}(\ell)$). Conversely, if the shared task is assigned the highest priority among all tasks in any partition, they will be considered $I_j^{\mathtt{NO-CI}}(\ell)$, as explained in the following examples.

Example 7: Consider τ_3 with $\mathcal{L}(\tau_3) = \{\rho_4^1\}$ under analysis in Example 6. In this situation, $\tau_2 \in \mathtt{DHP}(\tau_3)$ with $\mathcal{L}(\tau_2) = \{\rho_3^1, \rho_4^1, \rho_5^1\}$ contributes interference to τ_3 as $I_2^{\mathtt{CI}}(\ell)$ (not $I_2^{\mathtt{NO-CI}}(\ell)$), due to the presence of a higher priority

Algorithm 3 Advanced partitioning & task assignment

```
• Modification to Algorithm 1
    Added to Line 5: $SP(\tau_i) \leftarrow \infty$
    Added to Line 12: $Depth(\rho_1^r) \leftarrow 0$
• Modification to Algorithm 2
    Added to Line 3: $Depth(\rho_{v1}), Depth(\rho_{v2}) \leftarrow Depth(\rho_v) + 1$
    Added to Line 12: If $SP(\tau_i) \neq \infty$ then $SP(\tau_i) \leftarrow Depth(\rho_v)$
```

task $\tau_1 \notin \texttt{DHP}(\tau_3)$ with $\mathcal{L}(\tau_1) = \{\rho_3^1\}$ that is shared with $\tau_2 \in \texttt{DHP}(\tau_3)$. For the same reason, τ_4 with $\mathcal{L}(\tau_4) = \{\rho_4^1, \rho_5^1\}$ and τ_5 with $\mathcal{L}(\tau_5) = \{\rho_5^1\}$ (both of which do not occupy ρ_3^1) are interfered by their individual higher-priority DHP tasks by $I_i^{\texttt{CI}}(\ell)$ (not $I_i^{\texttt{NO-CI}}(\ell)$). This issue can be mitigated by promoting the priority of shared tasks relative to non-shared tasks, as demonstrated in the following example.

Example 8: Consider the same situation as Example 7. If we promote the priority of shared tasks τ_2 and τ_4 over non-shared tasks, such that the priority order is $\tau_2 \succ \tau_4 \succ \tau_1 \succ \tau_3 \succ \tau_5$, then the following holds. Since the priority of τ_2 becomes higher than that of τ_1, the presence of τ_1 no longer causes the shared task τ_2 to impose interference of $I_2^{\texttt{CI}}(\ell)$ to lower-priority tasks. The same holds for the priority promotion of τ_4 over τ_3. As a result, although the set of DHP tasks for each task changes (which affects schedulability), this priority promotion effectively alleviates the interference amount from $I_i^{\texttt{CI}}(\ell)$ into $I_i^{\texttt{NO-CI}}(\ell)$.

Note that the priority promotion of shared tasks is a heuristic to reduce interference, not guaranteeing optimal schedulability. However, as demonstrated in Section VII, applying priority promotion—described in the following subsection—effectively improves empirical schedulability performance.

B. Leveraging Priority Reassignment for Shared Tasks

According to the analysis in the previous subsection, we propose a simple, yet effective priority reassignment for shared tasks in Algorithm 2. The basic idea is to assign higher priority to shared tasks than to non-shared tasks during the partitioning process in Algorithm 2. Importantly, the relative priority order among non-shared tasks is preserved as originally given.

The key challenge, however, lies in determining the relative priorities among shared tasks—particularly between tasks that become shared due to partitioning at different levels. To address this, we assign higher priority to tasks that are shared in higher-level (i.e., shallower-depth, closer to the root partition) leaf partitions. Among tasks that become shared within the same partition, their original priority order is preserved. To reflect this, Algorithm 3 details the required changes to Algorithms 1 and 2.

We define a *shared priority order* (with $SP(\tau_i)$ returning the shared priority order of τ_i) that takes precedence over the regular task priority. That is, task priorities are compared first by their shared priority order, and ties are broken using the original priority order. In support of this mechanism, Line 5 of Algorithm 1 initially sets the shared priority value of each task τ_i to $SP(\tau_i) \leftarrow \infty$. When a root partition tree is created (Line 12), its depth denoted by $Depth(\rho_1^r)$ is initialized to 0. In Algorithm 2, Line 3 sets the depth of the two newly created sub-partitions ρ_{v1} and ρ_{v2} to the depth of ρ_v plus one. In Line 12, the shared priority of each task τ_i is then set to the depth of ρ_v, unless τ_i already has a finite shared priority value, in which case it inherits its previous value.

Theorem 3: Consider Algorithms 1 and 2 are modified by Algorithm 3. Suppose that the modified version of Algorithm 1 associated with the modified version of Algorithm 2 returns *schedulable* for a gang task set τ with a given task priority order on m processors. Then, τ is actually schedulable by RPS-FP-EE (with task priority order change) on m processors.

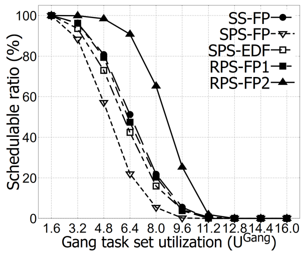
Proof: The proof follows the same structure as that of Theorem 2, and thus it suffices to check whether the schedulability of each shared and non-shared task is guaranteed with the priority promotion of a shared task. From the perspective of a shared task, since the shared task's priority has been promoted relative to when the schedulability was checked at Lines 7–8 of Algorithm 1, the modification cannot violate schedulability. As to a non-shared task, any potential impact from the change in the shared task's priority is captured by the schedulability check in Line 18 of Algorithm 2; therefore, no additional verification is required. ■

VII. Evaluation

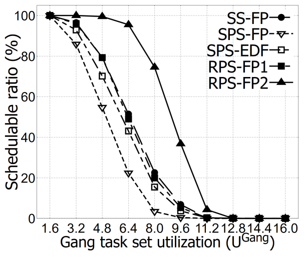
In this section, we evaluate the schedulability performance of the proposed RPS framework.

Task set generation. We generate gang task sets based on [6], [8], [11], [18]. The settings explore all combinations of five parameters: (S1) the number of processors m is chosen from $\{8, 16\}$; (S2) the number of tasks n is selected from $\{m, \frac{3}{2}m, 2m, \frac{5}{2}m\}$; (S3) the task parallelism m_i is uniformly selected from either $[1, m/2]$ (denoted by L) or $[1, m]$ (denoted by H) ; (S4) the deadline model of each task is either implicit-deadlines $D_i = T_i$ (denoted by I) or constrained-deadlines $D_i \leq T_i$ (denoted by C); and (S5) the gang task set utilization $U^{\texttt{Gang}} = \sum(m_i \cdot C_i/T_i)$ is varied from $0.1m$ to $1.0m$ in increments of $0.1m$.

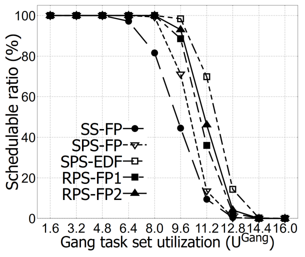
For each task set, given m, n and $U^{\texttt{Gang}}$ in S1, S2 and S5, respectively, m_i and $U_i{=}m_i{\cdot}C_i/T_i$ of each task are determined by S3 and the UUnifast algorithm [19], such that all tasks in each task set satisfy $U_i \leq m_i$. Then, the period T_i for each task is uniformly selected from $[100, 1000]$ms, and the worst-case execution time C_i is set to $U_i \cdot T_i/m_i$. The relative deadline D_i is determined by T_i for I, and uniformly distributed in $[\max(0.8T_i, C_i), T_i]$ for C. For each combination of parameters S1–S5, we generate 1000 task sets, resulting in a total of $2 \times 4 \times 2 \times 2 \times 10 \times 1{,}000 = 320{,}000$ task sets. Each unique configuration of parameters S1–S4 is represented as a tuple, where "*" means all values in the corresponding parameter; for instance, the configuration (16, 40, H, *) corresponds to task sets with m=16, n=40, $m_i \in [1, m]$, and both implicit- and constrained-deadline task models, respectively.



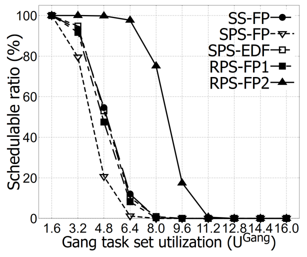
(a) (`16,16,H,C`) task sets



(b) (`16,16,H,I`) task sets



(c) (`16,16,L,I`) task sets



(d) (`16,40,H,I`) task sets

Fig. 3: The ratio of task sets deemed schedulable by each approach, under different task set configurations

Comparison approaches. We compare the ratio of task sets deemed schedulable by the following approaches.

- `SS-FP`: stationary scheduling with FP, tested by its schedulability analysis [10],
- `SPS-FP`: strictly partitioned scheduling with exclusive execution and FP [11], tested by the exact FP uniprocessor schedulability analysis [13],
- `SPS-EDF`: strictly partitioned scheduling with exclusive execution and EDF [11], tested by the exact EDF uniprocessor schedulability analysis [14], [20],
- `RPS-FP1` and `RPS-FP2`: RPS-FP-EE with Algorithms 1 and 2, tested by Theorems 2 and 3, respectively.

For comparison, the Deadline Monotonic (DM) prioritization policy is given to all FP approaches.[3] We do not compare existing global scheduling approaches in [6] and existing strictly partitioned scheduling approaches in which each partition is scheduled by global scheduling [11], as it has been reported in [11] that they exhibit poor performance compared to the above comparison approaches.

TABLE II: The ratio of task sets deemed schedulable by each approach, normalized by `SPS-FP` (%)

(m, n)	SS-FP	SPS-FP	SPS-EDF	RPS-FP1	RPS-FP2
(8, 8)	108.98%	100.00%	113.64%	112.20%	119.30%
(8, 12)	108.28%	100.00%	116.45%	114.45%	126.92%
(8, 16)	105.59%	100.00%	119.02%	114.62%	131.53%
(8, 20)	103.45%	100.00%	120.73%	114.04%	134.63%
(16, 16)	103.64%	100.00%	118.73%	114.11%	132.90%
(16, 24)	99.67%	100.00%	121.54%	114.56%	140.97%
(16, 32)	95.92%	100.00%	122.90%	112.64%	143.93%
(16, 40)	93.63%	100.00%	124.29%	111.63%	146.17%

Overall schedulability performance. Table II presents the overall ratio of schedulable task sets for the combinations of S1 and S2, using `SPS-FP` as a baseline. The results are aggregated over both deadline models (`C` and `I` in S4) and both task parallelism ranges (`L` and `H` in S3). While `SPS-EDF` consistently outperforms `SPS-FP` due to the superiority of EDF over FP on uniprocessor scheduling, `RPS-FP2` shows better schedulability performance than `SPS-EDF` by 4.98%–17.60% even though `RPS-FP2` employs the prioritization policy of FP.

[3]Recall that `RPS-FP2` employs modified task priorities from DM, according to Algorithm 3.

`RPS-FP2` also outperforms `SS-FP` by 9.47%–56.11%, which is known as a state-of-the-art non-SPS scheduling.

Table II reveals a similar trend for both m=8 and m=16: as the number of tasks n increases, the schedulability ratio of `SS-FP` declines, whereas those of `SPS-EDF` and `RPS-FP2` improve. Also, we observe that the schedulability performance trend for different deadline models does not vary much; for example, compare Fig. 3(a) for (16, 16, `H`, `C`) and Fig. 3(b) for (16, 16, `H`, `I`), both of which share the same configuration except the deadline model. Therefore, the remainder of this section presents evaluation results by focusing $m = 16$ and the implicit-deadline deadline model.

Impact of task parallelism heterogeneity. Fig. 3(b) and (c) present the schedulable task sets for the (16, 16, *, `I`) configuration, differing only in the task parallelism ranges `H` and `L`. As shown in Fig. 3(b), the proposed approach `RPS-FP2` outperforms all other approaches by a significant margin of at least 43.1% under (16, 16, `H`, `I`) where task parallelism spans the entire processor range from 1 to m. In this configuration, `SPS-EDF` ranks third in schedulability performance, performing even worse than `SS-FP`. On the other hand, as shown in Fig. 3(c) under limited task parallelism (up to $m/2$), `SPS-EDF` performs best but slightly outperforms `RPS-FP2` by 6.1%, while `RPS-FP2` provides the best performance among all FP-based methods.

This can be attributed to the fundamental design of RPS. While SPS suffers from performance degradation when handling tasks with large parallelism due to its restricted flexibility in forming multiple strict partitions, RPS effectively addresses this limitation through recursive partitioning, thereby successfully resolving the task parallelism heterogeneity problem of gang scheduling, which is the goal of this paper.

Impact of the number of tasks. Fig. 3(b) and (d) show the ratio of schedulable task sets under (16, *, `H`, `I`) that differ only in the number of tasks (16 vs. 40). While the overall trend remains consistent across both cases, the performance gap between `RPS-FP2` and other approaches becomes more pronounced as the number of tasks increases. In both configurations, the second-best performing approach is `SS-FP`, with `RPS-FP2` outperforming it by 43.1% under (16, 16, `H`, `I`) and by 88.6% under (16, 40, `H`, `I`). The results demonstrate that our proposed approach introduces less pessimism for more tasks in utilizing remaining processors and applying schedulability analysis, compared to existing approaches.

Effectiveness of shared-task priority promotion. While `RPS-FP2` consistently outperforms `RPS-FP1` across all task set configurations, we observe the performance gap is particularly influenced by the task parallelism range. For example, when m=16, `RPS-FP2` outperforms `RPS-FP1` by 2.8% and 3.9% under the (16, *, `L`, `C`) and (16, *, `L`, `I`), respectively. The improvement becomes significantly more pronounced in high-parallelism range `H`, where `RPS-FP2` achieves 65.9% and 71.7% improvements over `RPS-FP1` under the (16, *, `H`, `C`) and (16, *, `H`, `I`), respectively. This can be attributed to a key characteristic of RPS-FP-EE, where each task can be interfered not only from directly competing tasks but also from indirectly interfering ones (as described in Section IV). The observed results indicate that the prposed `RPS-FP2` effectively alleviates indirect interference by employing a simple yet powerful mechanism: elevating the priority of shared tasks (with larger m_i).

VIII. Related Work

Gang scheduling, which is a special case of parallel scheduling, is designed and considered for efficient operations for high-performance computers subject to high-throughput [21]–[23]. With the development of parallel processing units (e.g., [1], [2]) and various parallel programming models (e.g, [24], [25]), recent studies consider real-time gang scheduling. In this context, two main scheduling paradigms have been considered: global scheduling (GS), where the threads of a gang task can be executed on any available processors at run-time, and stationary scheduling (SS), where each task is statically assigned to a fixed subset of processors. Regarding preemptive gang scheduling for GS, there have been several studies on the timing guarantee of FP-GS [3]–[6] and EDF-GS [6]–[9], while some studies try to find optimal scheduling algorithm for GS [5], [26]. Regarding non-preemptive gang scheduling for GS, there exist a few studies that try to guarantee schedulability [18], [27]–[30]. When it comes to SS, a study in [10] is the only existing approach to SS-FP that offers timing guarantees by its schedulability test and processor-task mapping algorithm.

Recently, strictly partitioned scheduling (SPS) has been proposed and evaluated in [11], [31]. Despite its simplicity, SPS has shown superior empirical performance in terms of schedulability compared to GS and SS. Building on this, we generalize the SPS framework to allow each partition to recursively accommodate different gang tasks with varying degrees of parallelism m_i efficiently, thereby addressing the core challenge in gang scheduling—the heterogeneity of task parallelism requirements.

IX. Conclusion

This paper presented RPS, a hierarchical scheduling framework that generalizes SPS for real-time gang tasks. To fully realize the potential of RPS, we developed a tight and interpretable schedulability analysis of RPS-FP-EE and designed effective partitioning and task assignment strategies by leveraging the proposed analysis. Our results demonstrated the effectiveness of RPS in improving schedulability. As future work, we plan to explore the application of RPS under other scheduling modes and prioritization policies.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2024-00438248, RS-2024-00460389).

References

[1] D. Kirk, "NVIDIA Cuda software and GPU parallel computing architecture," in *Proceedings of the international symposium on memory management*, 2007, pp. 103–104.

[2] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE micro*, vol. 31, no. 5, pp. 7–17, 2011.

[3] F. A. Silva, E. P. Lopes, E. P. Aude, F. Mendes, J. Silveira, H. Serdeira, M. Martins, and W. Cirne, "Response time analysis of gang scheduling for real time systems," in *Proceedings of the SPECTS 2002-2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. Citeseer, 2002.

[4] J. Goossens and V. Berten, "Gang FTP scheduling of periodic and parallel rigid real-time tasks," *CoRR, abs/1006.2617 URL: http://arxiv.org/abs/1006.2617*, 2010.

[5] V. Berten, P. Courbin, and J. Goossens, "Gang fixed priority scheduling of periodic moldable realtime tasks," in *In 5th Junior Researcher Workshop on Real-Time Computing*, 2011, pp. 9–12.

[6] S. Lee, S. Lee, and J. Lee, "Response time analysis for real-time global gang scheduling," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 92–104.

[7] S. Kato and Y. Ishikawa, "Gang EDF scheduling of parallel task systems," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, December 2009, pp. 459–468.

[8] Z. Dong and C. Liu, "Analysis techniques for supporting hard real-time sporadic gang task systems," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, December 2017, pp. 128–138.

[9] ——, "Analysis techniques for supporting hard real-time sporadic gang task systems," *Real-Time Systems*, vol. 55, no. 4.

[10] N. Ueter, M. Gunzel, G. von der Bruggen, and J.-J. Chen, "Hard real-time stationary GANG-scheduling," in *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)*, 2021, pp. 10:1–10:19.

[11] B. Sun, T. Kloda, and M. Caccamo, "Strict partitioning for sporadic rigid gang tasks," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024, pp. 252–264.

[12] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*. IEEE, 2007, pp. 247–258.

[13] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard real-time scheduling: the deadline-monotonic approach," in *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1991, pp. 133–137.

[14] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[15] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 149–158.

[16] J.-J. Chen, G. Nelissen, and W.-H. Huang, "A unifying response time analysis framework for dynamic self-suspending tasks," in *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 61–71.

[17] E. G. Coffman, Jr, M. R. Garey, D. S. Johnson, and R. E. Tarjan, "Performance bounds for level-oriented two-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 9, no. 4, pp. 808–826, 1980.

[18] S. Lee, N. Guan, and J. Lee, "Design and timing guarantee for non-preemptive gang scheduling," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 132–144.

[19] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[20] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 1990.

[21] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and distributed Computing*, vol. 16, no. 4, pp. 306–318, 1992.

[22] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of International Conference on Distributed Computing Systems*, 1982, pp. 22–30.

[23] M. A. Jette, "Performance characteristics of gang scheduling in multiprogrammed environments," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1997.

[24] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[25] P. Pacheco, *Parallel programming with MPI.* Morgan Kaufmann, 1997.

[26] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information processing letters*, vol. 106, no. 5, pp. 180–187, 2008.

[27] G. Nelissen, J. Marcè i Igual, and M. Nasri, "Response-time analysis for non-preemptive periodic moldable gang tasks," in *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)*, 2022, pp. 12:1–12:22.

[28] Z. Dong and C. Liu, "A utilization-based test for non-preemptive gang tasks on multiprocessors," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS).* IEEE, 2022, pp. 105–117.

[29] B. Sun, T. Kloda, J. Chen, C. Lu, and M. Caccamo, "Schedulability analysis of non-preemptive sporadic gang tasks on hardware accelerators," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 147–160.

[30] ——, "Response time analysis and optimal priority assignment for global non-preemptive fixed-priority rigid gang scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 3, pp. 455–470, 2025.

[31] B. Sun, T. Kloda, C.-G. Wu, and M. Caccamo, "Partitioned scheduling and parallelism assignment for real-time dnn inference tasks on multi-tpu," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2024, pp. 333:1–6.