Schedulability Analysis of Non-preemptive Sporadic Gang Tasks on Hardware Accelerators

Binqi Sun, Tomasz Kloda, Jiyang Chen, Cen Lu, Marco Caccamo

▶ **To cite this version:**

Binqi Sun, Tomasz Kloda, Jiyang Chen, Cen Lu, Marco Caccamo. Schedulability Analysis of Non-preemptive Sporadic Gang Tasks on Hardware Accelerators. IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS 2023), May 2023, San Antonio, United States. pp.147-160, 10.1109/RTAS58335.2023.00019 . hal-04803556

HAL Id: hal-04803556

https://laas.hal.science/hal-04803556v1

Submitted on 25 Nov 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Schedulability Analysis of Non-preemptive Sporadic Gang Tasks on Hardware Accelerators

Binqi Sun*, Tomasz Kloda†, Jiyang Chen*, Cen Lu* and Marco Caccamo*
*Technical University of Munich, Germany
†LAAS-CNRS, Université de Toulouse, INSA, Toulouse, France
Email: {binqi.sun, jiyang.chen, cen.lu, mcaccamo}@tum.de, tkloda@laas.fr

***Abstract*—Non-preemptive rigid gang scheduling combines the performance benefits of parallel execution with the low overhead of non-preemptive scheduling and rigid task programming model. This approach appears particularly well-suited for parallel hardware accelerators where the context switch and migration overheads are critical and should be avoided. One of the most notable examples today is Google's Edge Tensor Processing Unit (TPU) used for neural network inference on embedded boards.**

The paper studies sporadic non-preemptive rigid gang scheduling applied to multi-TPU edge AI accelerators. Each gang task spawns a fixed number of threads that must execute simultaneously on distinct processing units. We consider non-preemptive fixed-priority gang (NP-FP-Gang) scheduling and propose the first carry-in limitation for gang task response time analysis. The gang task carry-in limitation differs from conventional sequential tasks due to the intra-task parallelism. We formulate it as a generalized knapsack problem and develop a linear programming relaxation and a dynamic programming approach to solve the problem under different time complexities. The performance of the proposed schedulability analysis is evaluated through randomly generated synthetic task sets and a case study using neural network benchmarks executed on commercial off-the-shelf multi-TPU edge AI accelerators. The evaluation results show that the proposed response time analysis effectively improves the state-of-the-art NP-FP-Gang schedulability test even by 85.7% for the Edge TPU benchmarks in particular.

***Index Terms*—Response Time Analysis, Non-preemptive, Gang Scheduling, Tensor Processing Unit.**

I. Introduction

Gang scheduling appeared as an efficient solution to the problem of job scheduling on highly-parallel embedded architectures [1], [2]. Application threads grouped into a single gang are scheduled concurrently on distinct processing units. Scheduling them at the same time can avoid the overhead of context switching [3] or busy waiting at synchronization points [4] and help in utilizing the inter-thread cache benefit [5]. Due to the prohibitively expensive preemption cost of the parallel hardware accelerators, running the gang tasks non-preemptively is often the only feasible solution. While the non-preemptive rigid gang scheduling has already been addressed in the literature in the context of power scheduling [6], we propose new schedulability tests for fixed-priority non-preemptive gang and show their applicability and potential on the Edge Tensor Processing Units (TPUs) [7].

A. Motivating example: Edge TPU scheduling

Edge TPU is a custom ASIC designed for accelerating neural network inference on edge devices. It can improve the inference time by 30x compared with embedded CPUs [8]. Integrating Edge TPU accelerators into edge devices present several challenges and can incur large memory and scheduling overhead if configured incorrectly. To investigate these issues, we benchmarked eight representative convolutional neural network models of various sizes on commercial off-the-shelf (COTS) Edge TPU hardware. Specific hardware setup and details of the neural network models are reported in Section V-B. In what follows, we outline some key findings related to real-time scheduling on multiple Edge TPUs.

Preemption cost. Edge TPU has small on-chip memory (8MB), thus limiting the size of the neural network model that can be stored in internal SRAM. Switching between different models causes a significant overhead as the model needs to be loaded into Edge TPU's internal cache every time. This can be observed in our experiments by comparing the neural network inference latency with and without parameter loading. Take Inception-v1 [9] as an example. The average inference latency without parameter loading is 6.58 ms. However, it grows to 20.17 ms when including parameter loading time.

Parallel processing. Besides the context switch overhead, running models larger than 8MB on an Edge TPU requires fetching the model parameters from the main memory for every inference, which incurs a high memory transaction latency. One approach to avoid latency is to pipeline a large model with multiple Edge TPUs. The model is divided into multiple segments using the Edge TPU Compiler, and each segment runs

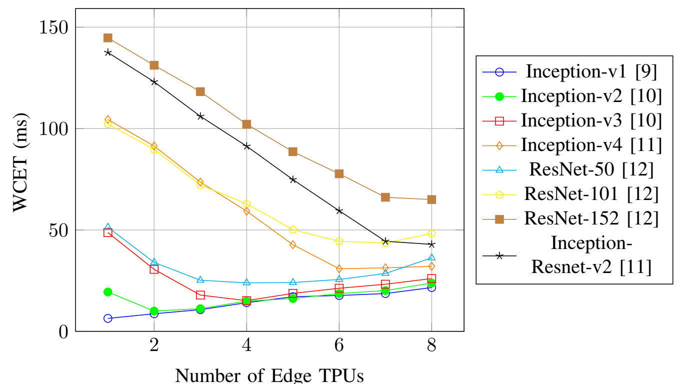


Fig. 1: Edge multi-TPU neural network benchmarks executed on ASUS AI Accelerator CRL-G18U-P3D with 8 Edge TPUs.

on a different Edge TPU. Although a model can be segmented with as many Edge TPUs as one likes, using more accelerators does not necessarily mean better performance. Figure 1 shows the relationship between neural network inference time and the number of Edge TPUs used by the networks. For a large network, *e.g.,* ResNet-152 (57.53MB) [12], having more Edge TPUs can indeed reduce inference time. However, for a small network such as Inception-v1 (5.72MB) [9], the inference time increases with the number of Edge TPUs. This is because pipelining a model requires sending intermediate tensors from one Edge TPU to another and adds I/O latency.

Memory space limitation. It can be beneficial to change the number of segments of a model at run time when multiple models are executed on multiple Edge TPUs. However, a different segmentation of the same model generates a separate model executable file. This will occupy extra disk space and precious memory of the edge devices during run time. Given these trade-offs, we assume only one segmentation is used for each network model. We also assume that the segmentation is given by the system designer. Determining the optimal number of segments is out of the scope of this paper.

Rigid non-preemptive gang. In light of the above-discussed Edge TPU characteristics (high preemption cost, parallel execution, and constrained memory space), we propose to model neural networks running on Edge TPUs as *non-preemptive rigid gang* tasks. The non-preemptive execution avoids the slowdown caused by the model reloading. With pipelining, each neural network task can take more than one accelerator (intra-task parallelism). Such gang task runs simultaneously on a fixed number of distinct processing units in parallel as the model segmentation is fixed. In contrast to traditional multi-threaded scheduling, where the threads can execute independently between the synchronization points, gang scheduling starts all task threads simultaneously. While we believe that new customized scheduling policies can be proposed to efficiently overcome certain disadvantages of non-preemptive gang execution, in this work, we assume the traditional *fixed-priority* policy used in most embedded real-time applications [13] and propose a gang task carry-in limitation technique to reduce the pessimism of the analysis.

B. Contributions

We propose new schedulability tests for non-preemptive rigid gang scheduling. Our contributions are as follows:

- We present the first TPU-Pipelining runtime performance benchmarks on multi-TPU edge AI accelerators.
- We provide a linear-time utilization bound test for any work-conserving non-preemptive rigid gang scheduling.
- We propose two schedulability tests with quadratic and pseudo-polynomial complexity, respectively, for non-preemptive fixed-priority (NP-FP) rigid gang scheduling.
- We demonstrate the effectiveness of our schedulability test on both synthetic data and Edge TPU benchmarks. Our proposed schedulability test can achieve up to 46.5% and 85.7% additional task sets schedulable on synthetic task sets and Edge TPU benchmarks, respectively, compared to the state-of-the-art non-preemptive gang schedulability analysis proposed in [6].

C. Paper organization

The remainder of this paper is organized as follows. Section II gives the system model and notations used in the paper. Section III opens with a sufficient schedulability condition for any work-conserving non-preemptive rigid gang scheduling algorithm. Then a simple utilization-based schedulability test with linear-time complexity is given. Section IV addresses the fixed-priority scheduling algorithms. First, two schedulability conditions are derived, and second, a knapsack problem is formulated to limit the number of interfering carry-in jobs. On the basis of these results, a quadratic-time schedulability test and a pseudo-polynomial response time analysis are derived. Section V contains experimental evaluations of the proposed schedulability tests using synthetic task sets and neural network profiles measured on the Edge TPUs. Section VI covers related work regarding non-preemptive and gang scheduling. Section VII concludes the paper and discusses future research directions.

II. System model and notations

Tasks and processors. We consider a multiprocessor platform comprised of M identical processors. A task set $\tau = \{\tau_1, ..., \tau_n\}$ is comprised of n independent sporadic rigid gang tasks τ_i ($1 \leq i \leq n$) running on M identical processors. Each task $\tau_i \in \tau$ gives rise to an infinite sequence of jobs with consecutive jobs' invocations (arrivals) separated by at least T_i time units (*i.e., sporadic* activation model). We use J_i to denote a job of task τ_i. Job J_i released at time r_i (arrival time) has an absolute deadline $r_i + D_i$ and must complete its execution by that time where $D_i \leq T_i$ (*i.e.,* constrained deadlines). Each job of task τ_i executes simultaneously on $m_i \leq M$ processors for at most C_i time units. Thus, the execution demand of a single job of task τ_i can be represented as an $C_i \times m_i$ rectangle in time and processor space. As a result, a gang task can be characterized by a 4-tuple $\tau_i = (C_i, T_i, D_i, m_i)$. Additionally, we define for each task $\tau_i \in \tau$ its longest starting interval $S_i = D_i - C_i$ such that every τ_i's job released at r_i and starting within time interval $[r_i, r_i + S_i]$ must meet its deadline if executed non-preemptively. Without loss of generality, we assume that all the above parameters are non-negative integers. Moreover, we define the utilization of a task τ_i as $U_i = C_i \cdot m_i / T_i$, and the utilization of the task set τ as $U = \sum_{\tau_i \in \tau} U_i$.

Scheduling policy. In this paper, we consider global *work-conserving* scheduling, which means each job can be scheduled on any processor, and no processor can be left idle as long as at least one active job, whose requested number of processors is no bigger than the current number of idle processors, is not executing. Moreover, we focus on *non-preemptive* scheduling, which means a job cannot be preempted once it starts its execution. We also consider a *fixed-priority* scheduling algorithm where each task has a constant

(static) priority determined offline and the active jobs with: i) the highest priorities at a given time, and ii) the number of required processors less than the number of processors that are currently idle are selected for execution. We assume each task has a unique priority, and the smaller the task index, the higher the task priority (*i.e.,* the priority of τ_i is higher than that of τ_j if $i < j$).

Response time and schedulability. The worst-case response time R_i of task τ_i is the maximum time duration between the release of any job of τ_i and the time the job completes its execution. We say that task τ_i is schedulable if each job of τ_i always meets its deadline (*i.e.,* $R_i \leq D_i$).

III. General utilization bound test

In this section, we give a general schedulability condition and present a linear-time utilization-based schedulability test for any work-conserving non-preemptive gang scheduling algorithm.

A. Sufficient schedulability condition

Let us consider task $\tau_k \in \tau$ and suppose that its job J_k is the first job to miss its deadline. Since our scheduling algorithm is non-preemptive, once a job starts its execution, it will continue without interruption until full completion. If job J_k misses its deadline then it must be unable to start its execution from its release time r_k until its latest starting time at $r_k + S_k$ that could guarantee meeting the deadline. Since our scheduling algorithm is work-conserving, the job cannot start if m_k processors needed for its execution are never idle in the interval $[r_k, r_k+S_k]$. Consequently, other jobs interfering with J_k's execution should continuously occupy at least $M - m_k + 1$ processors. For the sake of notation simplicity, we introduce M_k as:

$$M_k = M - m_k + 1$$

Let us denote by $W_{k\leftarrow i}(\Delta)$ an upper bound on the task τ_i's workload interfering with job J_k's execution within any generic interval of length $\Delta > 0$. Since J_k misses its deadline, the worst-case interference workload in any interval $[r_k, r_k + \Delta]$ is no less than $M_k \cdot \Delta$ for $0 < \Delta \leq S_k$. Job J_k is assumed to be the first τ_k job to miss its deadline, so all previous τ_k jobs meet their deadlines and do not execute within the time interval $[r_k, r_k+S_k]$ as we consider constrained deadlines (*i.e.,* $D_k \leq T_k$). Then, a necessary condition for τ_k to miss its deadline can be given as:

$$\forall \Delta : 0 < \Delta \leq S_k : \sum_{\tau_i \in \tau \setminus \{\tau_k\}} W_{k\leftarrow i}(\Delta) \geq M_k \cdot \Delta$$

By negating the above condition, we obtain the following sufficient condition for τ_k's schedulability.

Lemma III.1. *Given a gang task set τ running on M processors, a task $\tau_k \in \tau$ is schedulable by a work-conserving non-preemptive scheduling algorithm if:*

$$\exists \Delta : 0 < \Delta \leq S_k : \sum_{\tau_i \in \tau \setminus \{\tau_k\}} W_{k\leftarrow i}(\Delta) < M_k \cdot \Delta \qquad (1)$$

By applying this condition for each task $\tau_k \in \tau$, we can get a sufficient schedulability test. Next, we show how to compute the interfering workload $W_{k\leftarrow i}(\Delta)$.

B. Interfering workload computation

We compute an upper bound on the task τ_i's workload $W_{k\leftarrow i}(\Delta)$ interfering with job J_k's execution within any generic interval of length $\Delta > 0$. If a job J_k is active but not executing while a job of task τ_i is executing, we say that τ_i is interfering with the execution of J_k.

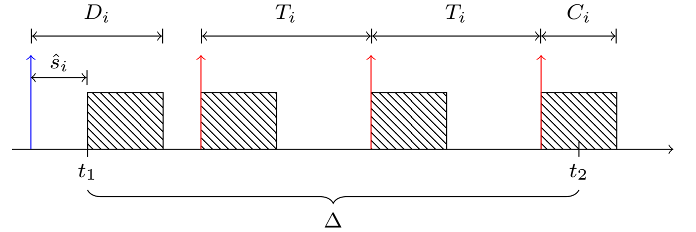


Fig. 2: Task τ_i's worst-case workload in time interval $[t_1, t_2]$ of length $\Delta > 0$.

We first evaluate how many τ_i jobs' execution can fall within the interval of interest $[t_1, t_2]$ with length $\Delta = t_2 - t_1$ where $t_2 > t_1 \geq 0$. The approach follows the same logic as outlined in [14]–[16]. In the worst-case (see Figure 2), the start of the first interfering job J_i of task τ_i is at the beginning of the interval (*i.e.,* t_1) and the job starts its execution C_i time units before its deadline (*i.e.,* as late as possible). Then, all subsequent jobs arrive as soon as possible. Given such an arrival sequence, there are at most $N_i(\Delta)$ job releases of task τ_i within a time interval of length Δ (*e.g.*, the three job releases are marked in red in Figure 2):

$$N_i(\Delta) = \left\lfloor \frac{\Delta + \hat{s}_i}{T_i} \right\rfloor \qquad (2)$$

where $\hat{s}_i$ is the latest starting time of task τ_i, and we consider $\hat{s}_i = S_i$ if not stated otherwise. The execution of the last job of τ_i in the interval Δ might terminate after t_2. The overlapping part of its execution time can be upper bounded as follows:

$$\xi_i(\Delta) = \min(C_i, \Delta + \hat{s}_i - N_i(\Delta) \cdot T_i) \qquad (3)$$

Therefore, the interference time $I_i(\Delta)$ of task τ_i within any interval of length $\Delta > 0$ can be upper bounded by:

$$I_i(\Delta) = \min(\Delta, N_i(\Delta) \cdot C_i + \xi_i(\Delta)) \qquad (4)$$

We compute $W_{k\leftarrow i}(\Delta)$ by bounding the number of processors τ_i can occupy to prevent J_k starting execution. The jobs of task τ_i reserve m_i processors. However, if $m_i > M_k$, only M_k processors are accounted for in the interference workload, since we need to know whether the sum of all busy processors at each time instant is no smaller than $M_k = M - m_k + 1$. Therefore,

$$W_{k\leftarrow i}(\Delta) = m_i^k \cdot I_i(\Delta) \qquad (5)$$

where

$$m_i^k = \min(m_i, M_k)$$

The task τ_i's interfering jobs can be considered in three separate categories:

- *Carry-in*: jobs released before t_1 and having remaining execution at t_1 (*e.g.,* the first job in Figure 2).
- *Body*: jobs with both release time and deadline in the interval $[t_1, t_2]$ (*e.g.,* the second and third job in Figure 2).
- *Carry-out*: jobs released before t_2 and with deadline after t_2 (*e.g.,* the last job in Figure 2).

In Section IV, we will differentiate between task τ_i's workload with carry-in $W^{CI}_{k\leftarrow i}(\Delta)$ (computed the same way as $W_{k\leftarrow i}(\Delta)$ in Equation (5)) and without carry-in $W^{NC}_{k\leftarrow i}(\Delta)$ (computed as $W_{k\leftarrow i}(\Delta)$ in Equation (5) by considering $\hat{s}_i = 0$ in Equations (2) and (3)).

C. Utilization bound

Based on Lemma III.1, we can derive a sufficient utilization bound test of linear-time complexity for an arbitrary work-conserving non-preemptive gang scheduling algorithm.

Theorem III.2. *(UB-NP-Gang). A gang task set τ is schedulable by a work-conserving non-preemptive scheduling algorithm on M processors if for all tasks $\tau_k \in \tau$:*

$$U \; < M_k + U_k \left(2 + \frac{T_k}{S_k}\right) - \frac{1}{S_k} \sum_{\tau_i \in \tau} U_i \left(S_i + T_i\right) \qquad (6)$$

Proof: We prove the theorem by contradiction. Suppose there exists a task $\tau_k \in \tau$ that satisfies Inequality (6) but is not schedulable. Let J_k be the first job missing its deadline. By Lemma III.1 and Equation (5), we have:

$$\sum_{\tau_i \in \tau \setminus \{\tau_k\}} m_i^k \cdot I_i(S_k) \;\geq\; M_k \cdot S_k$$

Since $I_i(S_k) \leq \left\lfloor \dfrac{S_k + S_i}{T_i} \right\rfloor \cdot C_i + C_i \leq \dfrac{C_i}{T_i}\left(S_k + S_i + T_i\right)$, we have:

$$\sum_{\tau_i \in \tau \setminus \{\tau_k\}} m_i \cdot \frac{C_i}{T_i}\left(S_k + S_i + T_i\right) \geq M_k \cdot S_k$$

By extracting τ_k from the summation in the LHS, we have:

$$\begin{aligned} LHS &= \sum_{\tau_i \in \tau} U_i \left(S_k + S_i + T_i\right) - U_k \cdot (2S_k + T_k) \\ &= U \cdot S_k + \sum_{\tau_i \in \tau} U_i \left(S_i + T_i\right) - U_k \cdot (2S_k + T_k) \end{aligned}$$

Apply it back to the above inequality, we have:

$$U \geq M_k + U_k \left(2 + \frac{T_k}{S_k}\right) - \frac{1}{S_k} \sum_{\tau_i \in \tau} U_i \left(S_i + T_i\right)$$

which contradicts our assumption that $\tau_k \in \tau$ satisfies Inequality (6). ■

The above utilization bound test can be done within linear time complexity by first computing $\sum_{\tau_i \in \tau} U_i \left(S_i + T_i\right)$, and then checking Inequality (6) for each $\tau_k \in \tau$.

IV. Schedulability conditions for NP-FP Gang

We derive our schedulability tests for the fixed-priority non-preemptive rigid gang scheduling. Our approach is based on the concept of *problem window*. We use two different approaches for bounding the start of the problem window leading to two distinct schedulability conditions. In both approaches, we assume that a job J_k is not schedulable if released within its problem window. By negating the necessary *unschedulability* conditions, we formulate sufficient schedulability tests. The general idea of our approach is similar to many previous works [17], [18] including non-preemptive scheduling [19]–[21]. The new challenges lie in the reasoning and accounting of the interference of non-preemptive gang jobs that can run on more than one processor.

A. Interfering tasks

We now identify the interfering jobs with regards to their priorities and volumes (*i.e.,* m_i). We consider the job under analysis J_k is interfered by another active job J_i. The interfering job J_i falls into one of the following five categories:

- higher-priority and lower-or-equal-volume $hplev(k)$: Job J_i can always start before J_k as J_i has higher priority and requires fewer or equal processors.
- higher-priority and higher-volume $hphv(k)$: Job J_i requires more processors than J_k so it is possible that the latter starts first. Unfortunately, we must assume the worst-case scenario in which m_i processors become idle at the same time and J_i starts execution before J_k as it has higher priority.
- lower-priority and lower-volume $lplv(k)$: Job J_i can start before J_k if the number of idle processors is sufficient for J_i but not for J_k. Such priority-inversion can occur multiple times [22] (see Figure 3).
- lower-priority and higher-or-equal-volume $lphev(k)$: Job J_i released at or after r_k while J_k is waiting for the processor(s) cannot start before J_k as J_i requires more or equal processors than J_k. If J_i is released before r_k and starts its execution before r_k, it can execute while J_k is active and not executing. Consequently, only one job of task τ_i can interfere with τ_k and we can note that the total number of processors occupied by lower-priority higher-volume jobs cannot be greater than M.
- τ_k previous instance: Job J_k cannot be directly blocked by the previous τ_k jobs since the tasks have constrained deadlines (*i.e.,* $D_k \leq T_k$). Nonetheless, the other interfering jobs released before the J_k's arrival might suffer from blocking from the previous τ_k job [23], [24].

Additionally, we also use $lephev(k)$ to denote task set $lphev(k) \cup \{\tau_k\}$ for ease of presentation.

B. Problem window: job's release time (r_k)

We first define the *problem window* as the time interval between J_k's release time r_k and its starting time $r_k + \Delta$ that can guarantee meeting its deadline, $[r_k, r_k + \Delta]$, where

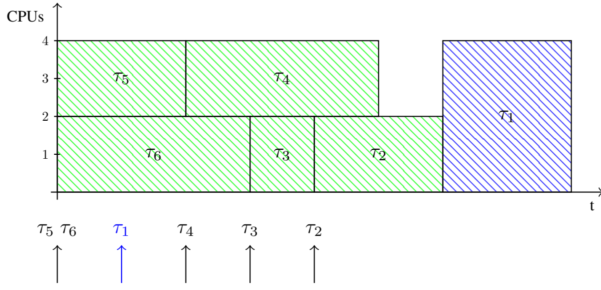


Fig. 3: Lower-priority and lower-volume jobs blocking. Lower-priority tasks (τ_2, τ_3, τ_4, τ_5, τ_6) require 2 processors while task τ_1 requires 4 processors. Arrows show the jobs releases.

$0 \leq \Delta \leq S_k$. We assume that task τ_k is not schedulable and J_k is its first job that misses its deadline. This means that J_k cannot start its execution within time interval $[r_k, r_k + \Delta]$ since m_k processors required by J_k are never idle (*i.e.,* among all M processors, at least M_k must be running other jobs).

We recall that the critical instant (*i.e.,* task arrival leading to the longest task response time) in multiprocessor scheduling might be unknown [25] and postponing the arrival of consecutive jobs of a task can have a detrimental effect on the task set schedulability [26]. Consequently, we assume that the interfering jobs can have carry-ins and we will try to limit the interference of such carry-in jobs. We do not include the previous τ_k job in the interference as it must finish its execution before the start of the problem window r_k considering that J_k is the first job missing its deadline and we consider constrained deadlines.

To reduce the pessimism of our analysis, we limit the interference of $lphev(k)$ jobs as observed in the previous section. The following lemma constrains each $lphev(k)$ to have at most one job interfering with J_k.

Lemma IV.1. *At most one job of each $lphev(k)$ task can execute while a job J_k of task τ_k is active and not executing.*

Proof: Job J_k is assumed to be non-schedulable so it cannot start within the problem window $[r_k, r_k + \Delta]$. This implies that no job with lower priority and higher or equal volume can start within the problem window $[r_k, r_k+\Delta]$. Such jobs can execute within the problem window $[r_k, r_k+\Delta]$ only if they start before r_k. Since tasks have constrained deadlines, only one job of each task can be active at a time. ■

Furthermore, we limit the number of $lphev(k)$ tasks having a job running within the problem window $[r_k, r_k + \Delta]$.

Lemma IV.2. *The number of processors occupied by $lphev(k)$ carry-in jobs is at most M.*

Proof: By Lemma IV.1, job J_i of task $\tau_i \in lphev(k)$ can execute within the problem window $[r_k, r_k+\Delta]$ only if J_i has started its execution before r_k. If J_i is interfering with J_k, it must still be running at r_k. All jobs execute non-preemptively and cannot be preempted or suspended until the end of execution. At each time instant, the number of processors occupied by the executing jobs cannot be greater than the total number of available processors M. Consequently, $lphev(k)$ jobs that started before r_k and that are still running at r_k cannot occupy more than M processors. ■

We can now derive a sufficient schedulability condition for task τ_k by applying Lemma III.1 and the observations from Lemmas IV.1 and IV.2 limiting the interference of $lphev(k)$ carry-in jobs:

Theorem IV.3. *Consider a gang task set τ, a task $\tau_k \in \tau$ is schedulable by a fixed-priority non-preemptive scheduling algorithm on M processors if:*

$$\sum_{\tau_i \in \tau \setminus lephev(k)} W^{CI}_{k \leftarrow i}(\Delta) \; + \sum_{\tau_i \in \tau^B(k)} W^{one}_{k \leftarrow i}(\Delta) \; < \; M_k \cdot \Delta \qquad (7)$$

where:

$W^{CI}_{k \leftarrow i}(\Delta)$ *is workload comprising a carry-in job generated by the jobs of task τ_i within a time interval of length $\Delta > 0$ and can be computed with Equation (5),*

$W^{one}_{k \leftarrow i}(\Delta)$ *is the maximum workload that can be generated within time interval of length $\Delta > 0$ by a single job of task τ_i and can be computed as follows:*

$$W^{one}_{k \leftarrow i}(\Delta) = m_i^k \cdot \min(C_i, \Delta) \qquad (8)$$

$\tau^B(k)$ *is the set of $lphev(k)$ jobs with the maximal workload that fulfills the conditions of Lemma IV.2.*

The interference of $lphev(k)$ jobs (*i.e.,* $\tau^B(k)$) can be computed by solving an instance of the knapsack problem. The entire procedure is described in Section IV-D.

C. Problem window: processors' busy time ($t_0 \leq r_k$)

We now consider a problem window that can start earlier than the release time r_k of the non-schedulable job J_k. Shifting the problem window will permit to get tighter bounds of carry-in interference. In particular, we will limit the interference of $hplev(k)$ carry-in jobs. As in the previous case, we assume that task τ_k is non-schedulable and J_k is the first job that misses its deadline. Job J_k is released at r_k but its release can be pulled backwards without letting it start execution as long as: at least one active $hplev(k)$ job is not executing (such a job will prevent J_k to start) or $hplev(k)$ jobs are running on at least M_k processors (hence J_k does not have enough processors to start). Similar to [19], we define a problem window $[t_0, t_0+\Delta]$, where t_0 is defined as follows.

Definition IV.1. *Let $t_0 < r_k$ denote the earliest time instant that $\forall t \in [t_0, r_k)$ one of the following two conditions holds:*

1) *At least M_k processors are occupied by tasks in $hplev(k)$ and all active tasks in $hplev(k)$ are executing at t;*
2) *at least one active task in $hplev(k)$ is not executing.*

If there does not exist such a t_0, we set $t_0 = r_k$.

Based on Definition IV.1, the following lemma is derived, which is a necessary condition for J_k to miss its deadline.

Lemma IV.4. $\forall t \in [t_0, t_0 + \Delta]$, *at least M_k processors are busy at time instant t.*

Proof: We consider interval $[t_0, r_k)$ and interval $[r_k, t_0 + \Delta]$ separately. For interval $[t_0, r_k)$, let us consider the two conditions in Definition IV.1. For any time instant $t \in [t_0, r_k)$, if the first condition holds, obviously at least M_k processors are busy; if the second condition holds, there must also be at least M_k processors busy as some active jobs of $hplev(k)$ cannot execute under a work-conserving policy. For interval $[r_k, t_0 + \Delta]$, since J_k misses its deadline, it cannot start its execution before $t_0 + \Delta \leq r_k + \Delta$. Therefore, we know at least M_k processors are busy during $[r_k, t_0 + \Delta]$. ■

Furthermore, we investigate which jobs can execute within the problem window $[t_0, t_0 + \Delta]$.

Lemma IV.5. *$\forall t \in [t_0, t_0 + \Delta]$, a job of $lephev(k)$ task can execute at time instant t only if it starts execution before t_0.*

Proof: We prove the lemma by contradiction. Suppose there is a job J_i of task in $lephev(k)$ starting its execution within $[t_0, t_0 + \Delta]$. We consider interval $[t_0, r_k)$ and interval $[r_k, t_0 + \Delta]$ separately. If J_i starts execution at $t \in [t_0, r_k)$, we know that at most $M - m_i$ processors are occupied by $hplev(i)$ jobs, and there are no active $hplev(i)$ jobs waiting for execution. Given that $m_i \geq m_k$, $hplev(k) \subseteq hplev(i)$, it contradicts Definition IV.1. If J_i starts execution at $t \in [r_k, t_0 + \Delta]$, it obviously contradicts the assumption that J_k is the first job missing its deadline. ■

By Lemma IV.4 and IV.5, we know that at least M_k processors must be occupied by the interfering jobs within the entire problem window, as illustrated in Figure 4.

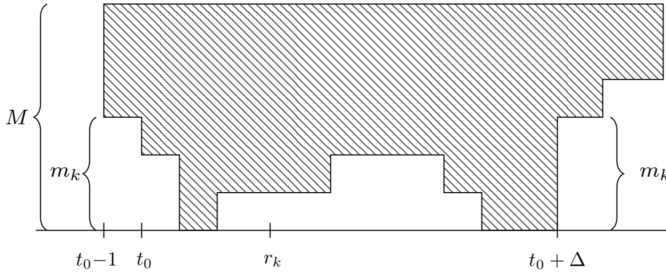


Fig. 4: Problem window $[t_0, t_0 + \Delta]$.

Next, we show how to limit the carry-in interference within problem window $[t_0, t_0+\Delta]$ in Lemmas IV.6 and IV.7. Like in the previous subsection, it is possible to limit the interference of $lphev(k)$ carry-in jobs by applying similar reasoning as before. Moreover, by redefining the problem window, we can also limit the carry-in interference of $hplev(k)$ tasks. On the downside, since we consider time interval that starts before r_k, we need to consider that the previous τ_k's job can run within the problem window (*i.e.,* the sub-interval $[t_0, r_k]$).

Lemma IV.6. *The number of processors occupied by $hplev(k)$ carry-in jobs is at most $M - m_k$.*

Proof: Since t_0 is the earliest instant that some active task in $heplev(k)$ is not executing (the second condition of Definition IV.1), we know that all active $hplev(k)$ jobs must be executing at $t_0{-}1$. Therefore, $hplev(k)$ tasks can have carry-in jobs only if they are executing at $t_0{-}1$. Moreover, since t_0 is the earliest instant such that at least M_k processors are occupied by tasks in $hplev(k)$ (the first condition of Definition IV.1), $hplev(k)$ tasks can only occupy at most $M - m_k$ processors at $t_0 - 1$. Hence, the number of processors occupied by $hplev(k)$ carry-in jobs is at most $M - m_k$. ■

Lemma IV.7. *The number of processors occupied by $hplev(k)$ and $lephev(k)$ carry-in jobs is at most M.*

Proof: By Lemma IV.5 and IV.6, we know that $lephev(k)$ and $hplev(k)$ carry-in jobs exist only if they start executing before t_0. Hence, the number of processors occupied by $lephev(k)$ and $hplev(k)$ carry-in jobs cannot exceed the total available processor number M. ■

Now, we derive a sufficient schedulability condition for task τ_k by negating the necessary condition in Lemma IV.4 and applying the observations from Lemmas IV.5, IV.6 and IV.7 limiting the number of carry-in jobs:

Theorem IV.8. *Consider a gang task set τ, a task $\tau_k \in \tau$ is schedulable by a fixed-priority non-preemptive scheduling algorithm on M processors if:*

$$\sum_{\substack{\tau_i \in hphv(k) \\ \cup \\ lplv(k)}} W^{CI}_{k\leftarrow i}(\Delta) + \sum_{\tau_i \in hplev(k)} W^{NC}_{k\leftarrow i}(\Delta) + \sum_{\tau_i \in \tau^0(k)} W^{diff}_{k\leftarrow i}(\Delta) < M_k \cdot \Delta \tag{9}$$

where:

$W^{CI}_{k\leftarrow i}(\Delta)$ *is the maximum workload including a carry-in job generated by task τ_i within a generic time interval of length Δ and can be computed with Equation (5),*

$W^{NC}_{k\leftarrow i}(\Delta)$ *is the maximum workload without a carry-in job generated by task τ_i within a generic time interval of length Δ and can be computed with Equation (5) considering $\hat{s}_i = 0$ in Equations (2) and (3),*

$W^{diff}_{k\leftarrow i}(\Delta)$ *is defined as follows:*

$$W^{diff}_{k\leftarrow i}(\Delta) = \begin{cases} W^{CI}_{k\leftarrow i}(\Delta) - W^{NC}_{k\leftarrow i}(\Delta) & \text{if } \tau_i \in hplev(k) \\ W^{one}_{k\leftarrow i}(\Delta) & \text{if } \tau_i \in lephev(k) \end{cases} \tag{10}$$

and $W^{one}_{k\leftarrow i}(\Delta)$ is the maximum workload that can be generated within a time interval of length Δ by a single job of task τ_k that can be computed with Equation (8),

$\tau^0(k)$ *is the set of $hplev(k) \cup lephev(k)$ with the maximal workload that fulfils the constraints on carry-in jobs in Lemmas IV.6 and IV.7.*

The carry-in tasks included in set τ^0 satisfying constraints in Lemmas IV.6 and IV.7 are selected by solving an instance of the knapsack problem as described in the next subsection.

D. Knapsack problems

Now we show how to calculate the maximum workload with carry-in limitation in Inequality (7) and (9) by formulating them as knapsack problems. In Inequality (7), our objective is to maximize the total workload by selecting tasks from $lphev(k)$ under the constraint that the number of processors

occupied by the selected tasks cannot exceed the total number of processors M (Lemma IV.2). This is obviously equivalent to the 0-1 knapsack problem, where M processors are interpreted as a knapsack with capacity M, and tasks are interpreted as the items to be packed into the knapsack. In Inequality (9), the problem is slightly different, since we have two types of tasks in the candidate set, *i.e.,* tasks in $hplev(k)$ and tasks in $levhev(k)$. On the one hand, both types of tasks need to share the capacity of M processors (Lemma IV.7), which is the same as the ordinary 0-1 knapsack problems. On the other hand, for tasks in $hplev(k)$, we have an additional constraint that the processors occupied by such tasks cannot exceed a capacity of $M - m_k$ (Lemma IV.6). This can be seen as a generalized version of 0-1 knapsack problem, where an additional capacity constraint needs to be ensured for a special type of items (*i.e.*, $hplev(k)$ tasks in our case).

There have been many solutions proposed for 0-1 knapsack problem in the literature. Next, we generalize two existing approaches to solve our generalized knapsack problem by handling the additional constraint arisen in Inequality (9). The first approach is a linear-time complexity upper bound generalized from the linear programming (LP) relaxation approach proposed in [27]. The second is dynamic programming (DP), which is an exact approach widely used in many knapsack problems [28]. Note that our generalized approaches can also be used to solve the knapsack problem in Inequality (7), since it is a special case of the generalized problem.

For simplicity, in the remaining part of this subsection, we use a generic notation W_i to denote the interference workload $W_{k\leftarrow i}^{one}(S_k)$ in Inequality (7) and the interference workload difference $W_{k\leftarrow i}^{diff}(S_k)$ in Inequality (9). Moreover, we use $\overline{\tau}$ to denote the candidate task set, which is $lphev(k)$ in Inequality (7) and $hplev(k) \cup lephev(k)$ in Inequality (9).

1) LP relaxation upper bound:

We first formulate our optimization problem in Inequality (9) as an integer linear programming (ILP):

$$\text{maximize} \quad \sum_{\tau_i \in \overline{\tau}} W_i \cdot x_i \tag{11}$$

$$\text{subject to:} \quad \sum_{\tau_i \in \overline{\tau}} m_i \cdot x_i \leq M \tag{12}$$

$$\sum_{\tau_i \in hplev(k)} m_i \cdot x_i \leq M - m_k \tag{13}$$

$$x_i \in \{0, 1\} \quad \forall \tau_i \in \overline{\tau} \tag{14}$$

where x_i is a binary decision variable to determine whether W_i is included in the total interference. An LP relaxation can be derived by replacing the integrality constraints (14) with continuous constraints:

$$0 \leq x_i \leq 1 \quad \forall \tau_i \in \overline{\tau} \tag{15}$$

where x_i is a continuous decision variable representing the proportion of W_i included in the total interference.

Then, we obtain the optimal solution of LP relaxation in two steps:

- Step 1. We define the workload density of task τ_i as W_i/m_i and sort the tasks in the candidate task set according to the decreasing order of their workload density:

$$\frac{W_{(1)}}{m_{(1)}} \geq \frac{W_{(2)}}{m_{(2)}} \geq ... \geq \frac{W_{(n)}}{m_{(n)}} \tag{16}$$

  where (i) denotes the index of task with the i-th biggest workload density, and n is the number of candidate tasks.
- Step 2. We select tasks according to the order in (16) consecutively and fill them into the processors until all M processors are full. The proportion $x_{(i)}$ of each task $\tau_{(i)}$ is set as large as possible without violating constraints (12), (13) and (15). Formally, we have:

Lemma IV.9. *The optimal solution $\overline{x}$ of the LP relaxation is*

$$\overline{x}_{(j)} = \frac{\min(m_{(j)}, M'_j)}{m_{(j)}} \qquad \tau_{(j)} \in lephev(k)$$

$$\overline{x}_{(j)} = \frac{\min(m_{(j)}, M'_j, M''_j)}{m_{(j)}} \quad \tau_{(j)} \in hplev(k)$$

where M'_j denotes the remaining number of processors after assigning the first $j-1$ tasks, and M''_j denotes the remaining number of processors that can be occupied by tasks in $hplev(k)$ after assigning the first $j-1$ tasks:

$$M'_j = M - \sum_{1 \leq i \leq j-1} \overline{x}_{(i)} \cdot m_{(i)} \quad 1 \leq j \leq n$$

$$M''_j = M - m_k - \sum_{\substack{1 \leq i \leq j-1 \\ \tau_{(i)} \in hplev(k)}} \overline{x}_{(i)} \cdot m_{(i)} \quad 1 \leq j \leq n$$

Proof: We prove the lemma by contradiction. Let x^* be the optimal solution of the LP relaxation. Observe that any optimal solution x of the LP relaxation must be maximal, in the sense that $\sum m_i \cdot x_i = M$. Suppose $x^*_{(p)} < \overline{x}_{(p)}$ for some $\overline{x}_{(p)} > 0$, then we must have $x^*_{(q)} > \overline{x}_{(q)}$ for at least one $q > p$. Given a sufficiently small $\epsilon > 0$, we could increase the value of $x^*_{(p)}$ by ϵ and decrease that of $x^*_{(q)}$ by $\epsilon \cdot m_{(p)}/m_{(q)}$, thus increasing the value of the objective function by $\epsilon \cdot (W_{(p)} - W_{(q)} \cdot m_{(p)}/m_{(q)})$. Since $W_{(p)}/m_{(p)} > W_{(q)}/m_{(q)}$, we know $\epsilon \cdot (W_{(p)} - W_{(q)} \cdot m_{(p)}/m_{(q)}) > 0$, which contradicts with our assumption that x^* is an optimal solution. In the same way we can prove that $\forall p, x^*_{(p)} > \overline{x}_{(p)}$ is impossible. Hence, there does not exist any solution better than $\overline{x}$. ■

The optimal objective value of LP relaxation follows:

$$\overline{obj} = \sum_{\tau_i \in \overline{\tau}} W_{(i)} \cdot \overline{x}_{(i)}$$

Since $\overline{obj}$ is the optimal objective value of LP relaxation, it is a valid upper bound of the original ILP. Moreover, because of the integrality of $W_{(i)}$ and $m_{(i)}$, a valid upper bound of ILP is thus $\lfloor \overline{obj} \rfloor$.

For clarity, specific procedures to compute UB are given in Algorithm 1. It is clear that the time complexity of line 2-11 in Algorithm 1 is $O(n)$, where n is the number of tasks in $\overline{\tau}$. If we take line 1 into account, it will take $O(n \log n)$

Algorithm 1: LP relexation upper bound

```
Input: Number of processors M, task under analysis τ_k,
       candidate task set τ̄;
Output: Interference workload upper bound UB;
1  Sort tasks in τ̄ according to Formula (16);
2  UB ← 0, M' ← M, M'' ← M − m_k, i ← 1;
3  while M' > 0 do
4      if τ_(i) ∈ hplev(k) then
5          UB ← UB + (min(m_(i), M', M'') / m_(i)) · W_(i);
6          M' ← M' − min(m_(i), M', M'');
7          M'' ← M'' − min(m_(i), M', M'');
8      else
9          UB ← UB + (min(m_(i), M') / m_(i)) · W_(i);
10         M' ← M' − min(m_(i), M');
11     i ← i + 1;
12 return UB;
```

Algorithm 2: Dynamic programming

```
Input: Number of processors M, task under analysis τ_k,
       candidate task set τ̄, in which each task τ̄_i has
       volume m̄_i and interference workload W̄_i;
Output: Maximum interference workload W_{n,M,M−m_k};
1  Initialize W_{i,m,m'} ← 0, ∀i = 0,...,n, m = 0,...,M, m' =
     0,...,M − m_k;
2  for i ← 1 to n do
3      for m ← 1 to M do
4          for m' ← 1 to M − m_k do
5              if τ̄_i ∈ hplev(k) then
6                  if m̄_i > min(m, m') then
7                      W_{i,m,m'} ← W_{i−1,m,m'};
8                  else
9                      W_{i,m,m'} ← max(W_{i−1,m,m'},
                         W_{i−1,m−m̄_i,m'−m̄_i} + W̄_i);
10             else
11                 if m̄_i > m then
12                     W_{i,m,m'} ← W_{i−1,m,m'};
13                 else
14                     W_{i,m,m'} ← max(W_{i−1,m,m'},
                         W_{i−1,m−m̄_i,m'} + W̄_i);
15 return W_{n,M,M−m_k};
```

time complexity for task sorting. However, we can apply the acceleration technique proposed in [29] to make our computation still performed in $O(n)$ time. Algorithm 1 can be used to solve the knapsack problem in Inequality (7) by initializing $M'' = M'$. The complexity remains the same.

2) Dynamic programming:

The general idea of DP is to simplify a complex optimization problem by breaking it down into simpler sub-problems and solving them in a recursive manner [28]. For our generalized knapsack problem, this is achieved by dividing the solution process into n stages, where n is the number of tasks in the candidate set $\overline{\tau}$. In each stage $i = 1, ..., n$, we solve a sub-problem of computing $W_{i,m,m'}$, which is defined as the maximum interference workload of the first i tasks on m processors with the constraint that the number of processors occupied by $hplev(k)$ tasks cannot exceed m'. The sub-problems can be solved in an iterative manner starting from $W_{0,1,1}$ towards $W_{n,M,M-m_k}$, as shown in Algorithm 2.

The time complexity of Algorithm 2 is $O(nM^2)$ for the generalized knapsack problem in Inequality (9). For the problem in Inequality (7), the time complexity reduces to $O(nM)$, since there is no additional constraint on $hplev(k)$ tasks so that we do not need to iterate m' from 1 to $M - m_k$.

E. Schedulability tests

We derive two sufficient schedulability tests based on the conditions obtained in Sections IV-B and IV-C and the knapsack problem formulations discussed in Section IV-D.

The first test `Fixed` consists in checking the schedulability conditions expressed with Formulas (7) and (9) for each task $\tau_k \in \tau$ with $\Delta = S_k$ and solving the knapsack problem using the *LP* relaxation upper bound given in Algorithm 1. The time complexity of the resulting test is $O(n^2)$.

The schedulability test proposed above can be further refined by applying an iterative procedure to restrict the end of the problem interval [14]. The previously proposed test checks the schedulability only in one fixed problem window ending after $\Delta = S_k$. The sufficient schedulability condition given by Formula (1) can hold for any integer $\Delta \in (0, S_k]$ at which or before which job under analysis can start its execution. The goal of the response time analysis is to find an upper bound on the task τ_k's latest starting time $s_k \in (0, S_k]$ such that:

$$\sum_{\tau_i \in \tau \setminus \{\tau_k\}} W_{k \leftarrow i}(s_k) < M_k \cdot s_k \tag{17}$$

We can solve the above inequality by the fixed-point iteration on the value s_k as described in [30], [31]:

$$s_k \leftarrow \left\lfloor \frac{\sum_{\tau_i \in \tau} W_{k \leftarrow i}(s_k)}{M_k} \right\rfloor + 1 \tag{18}$$

When testing the entire task set τ, once a new upper bound on the task τ_k's worst-case response time is obtained as $R_k = s_k + C_k$, we can update the value of $\hat{s}_k \leftarrow s_k$. This information can be exploited to reduce the overestimation in τ_k's workload computation (see Equations (2) and (3)) when testing the next tasks in τ. We continue the test until: i) all tasks are deemed schedulable, or ii) the tasks' worst-case response times remain unaffected. The analysis is outlined in Algorithm 3.

In our second test, `RTA`, we apply the response time analysis for the schedulability conditions described in Formulas (7) and (9). We apply both conditions for each task and consider the shortest worst-case response time upper bound (line 9 in Algorithm 3). To limit the number of carry-in jobs, we use the dynamic programming approach given in Algorithm 2.

The `RTA` test time-complexity is $O(n^3 M^2 D_{max}^2)$, where $D_{max} = \max\{D_i | \tau_i \in \tau\}$ is the longest relative deadline among all tasks. Computing the worst-case response time using fixed-point iteration on Equation (18) can take at most $n \cdot S_k < n \cdot D_k$ steps for each task τ_k and at each iteration step we solve the knapsack problem using Algorithm 2 having the

Algorithm 3: Response time analysis

```
Input: Task set under analysis τ;
Output: Schedulability of τ;
1  sched ← True, updated ← True, ∀ τ_k ∈ τ : ŝ_k ← S_k;
2  while sched is False and updated is True do
3      updated ← False; sched ← True;
4      foreach τ_k ∈ τ do
5          s_k ← 1;
6          while s_k ≤ ŝ_k do
7              W_k ← min (LHS Eq. (7) and (9)) with Δ=s_k;
8              if W_k/M_k ≥ s_k then
9                  s_k ← ⌊W_k/M_k⌋ + 1;
10             else
11                 if s_k < ŝ_k then
12                     updated ← True;
13                     ŝ_k ← s_k;
14                 break;
15             if s_k > S_k then
16                 sched ← False;
17 return sched;
```

worst-case time complexity of $O(nM^2)$. We repeat the entire procedure as long as there is at least one unschedulable task and as long as we can obtain improvement on at least one task worst-case response time. In such a way, the schedulability analysis of the entire task set can be performed multiple times which can be upper-bounded by $n \cdot D_{max}$.

F. Priority assignment

A priority assignment policy $\mathcal{P}$ is optimal with respect to a schedulability test $\mathcal{S}$, if and only if there are no task sets that are deemed schedulable by test $\mathcal{S}$ using another priority assignment policy but not deemed schedulable by test $\mathcal{S}$ using policy $\mathcal{P}$ [32]. An Optimal Priority Assignment (OPA) algorithm was proposed for uniprocessor fixed-priority feasibility analysis by Audsley [33]. Later, OPA was extended to the multiprocessor case in [34] by proving that OPA is compatible to a global fixed-priority schedulability test $\mathcal{S}$ if and only if the following three conditions hold:

1) The schedulability of a task τ_k may, according to test $\mathcal{S}$, depend on any independent properties of tasks with priorities higher than k, but not on any properties of those tasks that depend on their relative priority ordering.
2) The schedulability of a task τ_k may, according to test $\mathcal{S}$, depend on any independent properties of tasks with priorities lower than k, but not on any properties of those tasks that depend on their relative priority ordering.
3) When the priorities of any two tasks of adjacent priority are swapped, the task being assigned the higher priority cannot become unschedulable according to test $\mathcal{S}$, if it was previously schedulable at the lower priority.

According to the above three conditions, [34] classifies global fixed-priority schedulability tests into OPA-compatible and OPA-incompatible. For OPA-compatible tests, Audsley's OPA algorithm should be used. For OPA-incompatible tests, it is recommended to use a priority assignment heuristic DkC, which was demonstrated to be highly effective in multiprocessor scheduling and applicable to any schedulability test [34].

Next, we show that schedulability tests `Fixed` and `RTA` are unfortunately OPA-incompatible due to a conflict between schedulability condition (9) and the above condition 3). Therefore, we use the priority assignment heuristic DkC recommended by [34] for performance evaluation in Section V.

Theorem IV.10. *The proposed NP-FP Gang schedulability tests* `Fixed` *and* `RTA` *are OPA-incompatible.*

Proof: It suffices to show that condition 3) does not hold for (9). We consider two tasks τ_x and τ_y with $m_x < m_y$. Tasks τ_x and τ_y are initially at priorities k and $k+1$, respectively. τ_y is schedulable, and $\tau_x \notin \tau^0(y)$. If we swap the priorities of the two tasks, task τ_x moves from set $hplev(y)$ to $lplv(y)$. As a result, when checking the schedulability of τ_y, the interference workload generated by task τ_x needs to be excluded from the first term $\sum_{\tau_i \in hplev(y)} W^{NC}_{y\leftarrow i}(\Delta)$ and included in the second term $\sum_{\tau_i \in hphv(y) \cup lplv(y)} W^{CI}_{y\leftarrow i}(\Delta)$. (The third term $\sum_{\tau^0(y)} W^{diff}_{y\leftarrow i}(\Delta)$ does not change because $\tau_x \notin \tau^0(y)$.) Since $W^{CI}_{y\leftarrow x}(\Delta) \geq W^{NC}_{y\leftarrow x}(\Delta)$, the LHS of (9) may increase, which may turn task τ_y from schedulable to unschedulable. ■

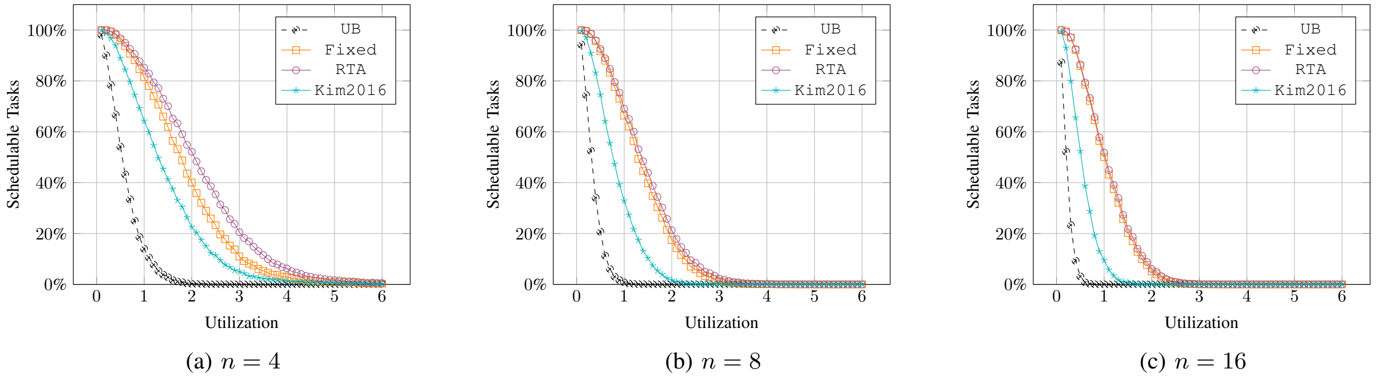
V. Performance Evaluation

This section evaluates the schedulability performance of our proposed schedulability tests. We first perform simulations on synthetic task sets generated randomly with various parameters using a standard task set generation technique. Then, we conduct a case study on task sets generated from real-world Edge TPU benchmarks with eight representative convolutional neural networks widely used in computer vision applications. For each task set, we compare the acceptance ratio (the percentage of task sets deemed schedulable) achieved by our proposed schedulability tests and the one proposed in [6], which is, to our best knowledge, the only schedulability test proposed for non-preemptive fixed-priority rigid gang scheduling. In the following subsections, we use `UB` to denote the utilization bound test proposed in Section III, `Fixed` and `RTA` introduced in Section IV to denote the schedulability test with fixed time window and the response time analysis with fixed-point iteration, respectively. We use `Kim2016` to denote the schedulability test proposed in [6].

As discussed in Section IV-F, the proposed schedulability tests `Fixed` and `RTA` are not OPA-compatible, thus we use DkC [34] as the priority assignment for `Fixed` and `RTA` as recommended by [34]. However, since `Kim2016` is OPA-compatible (proof see Appendix), we use OPA to provide the optimal priority ordering for `Kim2016`.

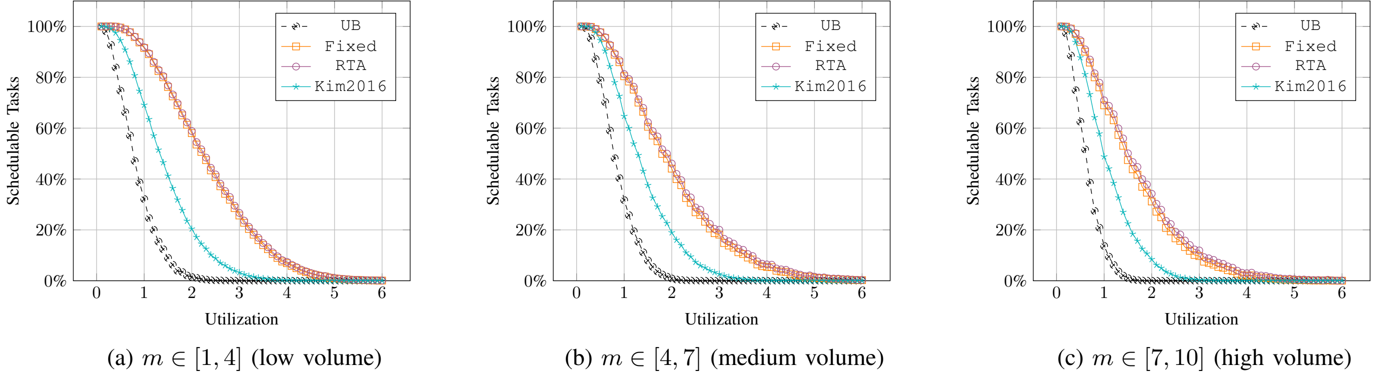
A. Performance evaluation on synthetic task sets

We generate synthetic task sets using `DRS` [35], a generalized version of the `UUnifast` [36] and `RandFixedSum` [37] algorithms. To evaluate the schedulability performance in various scenarios, we consider



(a) $n = 4$ (b) $n = 8$ (c) $n = 16$

Fig. 5: Schedulability ratios on synthetic task sets with $M = 8, m \in [1, 8]$.



(a) $m \in [1, 4]$ (low volume) (b) $m \in [4, 7]$ (medium volume) (c) $m \in [7, 10]$ (high volume)

Fig. 6: Schedulability ratios on synthetic task sets with $M = 16, n = 16$.

two sets of problem settings: (i) we fix the number of available processors $M = 8$, sample the task volume m from the range $[1, 8]$, and vary the task set size $n \in \{4, 8, 16\}$; (ii) we set $M = 16, n = 16$ and vary the task volume sampling range from $[1, 4]$ (low volume), $[4, 7]$ (median volume), and $[7, 10]$ (high volume). Additionally, the task set utilization U is varied from $[0.1, M]$ with a step of 0.1 in both sets of problem settings.

We generate 10,000 random task sets for each combination of the above parameters. For each task set, we first set the U and then generate utilization for every task using `DRS` such that their sum equals U. The utilization of every single task should not be larger than the maximum volume m_{max} of the task set. Then we generate volume for each task from range $[\max(m_{min}, \lceil U_i \rceil), m_{max}]$ with uniform distribution. We take the maximum of m_{min} and $\lceil U_i \rceil$ as the left side of the range since the utilization of each task should be no bigger than the volume. Otherwise, the task is not schedulable ($C_i > T_i$). Next, we generate C_i for each task from a uniform distribution in the range $[10, 100]$ ms. We select this range since it is close to the benchmark results in our case study. Finally, we calculate the period of each task by $T_i = \lceil C_i \cdot m_i / U_i \rceil$.

Figure 5 shows the evaluation results with varied task set sizes. From the figure, we can see that the proposed schedulability tests `Fixed` and `RTA` outperform `Kim2016` for all task set sizes. Specifically, test `RTA` achieves up to 30.3%, 38.4%, and 46.5% additional task sets deemed schedulable compared to `Kim2016` for $n = 4$, $n = 8$, and $n = 16$, respectively. Moreover, the improvement of `RTA` over `Kim2016` increases with the task set size n because `RTA` can limit more carry-in jobs when there are more tasks in the task set, while `Kim2016` does not limit any carry-in jobs. By comparing `Fixed` with `RTA`, we can see that the performance gain of `RTA` over `Fixed` is getting smaller for larger task sets.

Figure 6 shows the evaluation results with varied task volume ranges. It can be seen from the figure that both `Fixed` and `RTA` outperform `Kim2016` for all task volume levels. Specifically, `RTA` achieves up to 39.1%, 28.5%, and 29.1% additional task sets deemed schedulable compared to `Kim2016` for low, medium, and high volume task sets, respectively. Moreover, it can be observed that `UB` gets more pessimistic for higher volume tasks. This can be interpreted by Inequality (6). The lower the RHS value of (6), the more pessimistic the utilization bound test. From (6), it can be seen that the RHS value decreases with the increase of both the task volume m_k and period T_i. According to $U_i = C_i \cdot m_i / T_i$, a task with higher volume tends to have a larger period under the same utilization and WCET distribution. As a result, the RHS value of in Inequality (6) decreases with the task volumes, and therefore, `UB` is more pessimistic for higher volume tasks.

We note that the above results of the proposed tests `Fixed` and `RTA` can be further improved by using an optimal priority

TABLE I: Edge TPU benchmarking results

Model	Inception-v1	Inception-v2	Inception-v3	Inception-v4	ResNet-50	ResNet-101	ResNet-152	Inception-ResNet-v2
Model Size (MB)	5.72	10.19	21.56	40.90	23.40	42.46	57.53	54.13
WCET (ms)	6	10	15	31	24	44	55	40
Volume	1	2	4	6	4	6	9	9

assignment policy, which we plan to study in the future work.

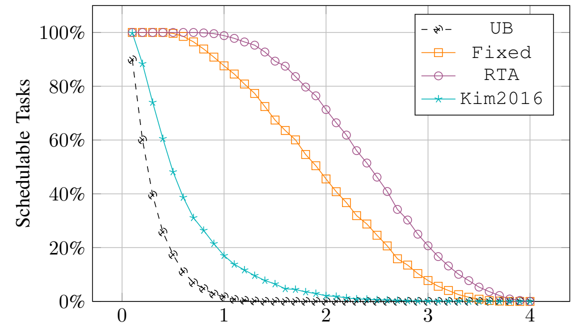
B. Case study on Edge TPU benchmarks

To evaluate the performance of our proposed schedulability tests in real-world applications, we benchmark the execution times of 8 representative deep neural networks on COTS Edge TPU devices and generate task sets based on the benchmarked task profiles. We use two hardware configurations in the benchmarking. One is ASUS AI Accelerator card CRL-G18U-P3D integrated with 8 Edge TPUs. The other one is ASUS AI Accelerator card CRL-G116U-P3D integrated with 16 Edge TPUs[1]. Both cards are connected via PCIe to a workstation with Intel(R) Xeon(R) Silver 4216 CPU @ 2.10 GHz. It should be noted that although we use a desktop platform to host the Edge TPUs, the neural networks will be running on the Edge TPUs entirely, as long as the operations[2] defined in the neural network models are supported by Edge TPU. To this end, we select eight representative neural networks fully compatible with Edge TPU. The list of the neural network models used in our benchmark is reported in Table I.

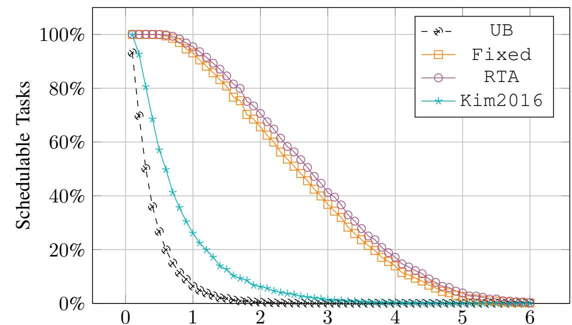
The table also shows that the selected neural networks have a wide range of sizes, from 5.72 MB to 57.53 MB. Recall that the benefit of running a model on multiple Edge TPUs is to cache the entire neural network parameters on the Edge TPU on-chip memory and avoid memory transaction latency. Our preliminary experiments found that the first six neural network models can be fully cached on 8 Edge TPUs. However, the last two neural networks *i.e.,* ResNet-152 and Inception-ResNet-v2, will occupy more than 8 Edge TPUs. Based on this observation, we benchmark the first six models with the hardware integrated with 8 Edge TPUs and benchmark the last two models with the hardware integrated with 16 Edge TPUs.

The benchmarking process is as follows. For each neural network, we run it on the pipelines with different number of Edge TPUs varied from 1 to the maximum number of Edge TPUs available on the card (*i.e.*, 8 for CRL-G18U-P3D and 16 for CRL-G116U-P3D). For each number of Edge TPUs, we perform 1,000 inferences and take the maximum inference time as its WCET. Then, we select the lowest WCET and the corresponding Edge TPU number as the network setup used in the case study. Specific results are shown in Table I. Since the neural networks are benchmarked on two hardware configurations with different number of available Edge TPUs, we generate two suites of neural network task sets accordingly. In the first suite of task sets, the available number of Edge TPUs M is set as 8, and each task set includes 6 tasks corresponding to the first six neural network models in Table I.

[1]https://iot.asus.com/products/AI-accelerator/AI-Accelerator-PCIe-Card/
[2]https://coral.ai/docs/edgetpu/models-intro/#supported-operations/



(a) 8 Edge TPUs, 6 neural network inference tasks



(b) 16 Edge TPUs, 8 neural network inference tasks

Fig. 7: Schedulability ratios on Edge TPU benchmark tasks.

In the second suite, M is set to 16, and each task set includes all eight benchmarked neural networks. In each test suite, the target task set utilization U varies in $[0.1, M]$ with a step of 0.1. For each target utilization, we generate 10,000 task sets with random task utilization generated by `DRS`. The upper bound of each task utilization is set as its volume to ensure $C_i/T_i \leq 1, \forall \tau_i \in \tau$. The task periods are calculated as $T_i = \lceil C_i \cdot m_i / U_i \rceil$ accordingly.

Figure 7 shows the schedulability results on the Edge TPU benchmark task sets. It can be observed from the figure that our proposed `Fixed` and `RTA` outperform the existing test `Kim2016` on both test suites. In particular, `RTA` achieves up to 85.7% additional schedulable task sets compared to `Kim2016` on the test suite with 8 Edge TPUs, and 73.2% on the test suite with 16 Edge TPUs. The improvement of `RTA` over `Kim2016` on the Edge TPU benchmark task sets is larger than synthetic task sets. This is as expected because, in the Edge TPU benchmark task sets, a large-size neural network usually requires a large WCET and large volume (as shown in Table I). As a result, the variance of the task execution demands (*i.e.*, $C_i \times m_i$) is larger on the neural network task

sets. In `Fixed` and `RTA`, the interference workload generated by extremely high-demand tasks can be effectively limited by the developed gang task carry-in limitation technique, which is not investigated in `Kim2016`.

VI. Related work

The problem of rigid gang scheduling was shown to be NP-hard [38]. Goossens et al. [39] provided and proved an exact schedulability test for preemptive fixed-priority rigid gang scheduling. Further, Goossens and Richard [40] used the idea of *proportionate progress* [41], [42] to propose an optimal scheduling algorithm for periodic implicit deadline rigid gang tasks. Collette et al. [43] also leveraged the idea of proportional progress and proposed a theoretically optimal scheduling algorithm for *malleable* (*i.e.,* the scheduler can change the number of processors the job is running on at any point during the job execution) periodic tasks with implicit deadlines. Berten et al. [44] considered *moldable* (*i.e.,* the scheduler can decide the number of processors allocated to a parallel job and this decision is made before the job starts) gang scheduling and showed that by using the information on the past and future scheduling events, the scheduler can better adjust the number of processors dedicated to each job. Nelissen et al. [45] proposed the response time analysis under the job-level fixed-priority policy for moldable gang tasks with a periodic activation model. Kato and Ishikawa [46] apply *Earliest Deadline First (EDF)* to the gang scheduling for preemptive rigid tasks while Dong and Liu [47], [48] studied *Global EDF* for gang task systems. Dong et al. [2] also explored the gang scheduling problem for soft real-time systems and proposed a tardiness bound under *Global EDF*. Ueter et al. [49] proposed a stationary gang scheduling algorithm and showed that it can be reduced to the single processor self-suspension scheduling problem. Ali et al. [50] proposed a real-time preemptive gang scheduling framework that enforces only one gang executing at any time in the system to limit memory interference. They further explored how to form virtual-gangs from a group of parallel real-time tasks in [51]. They proposed gang formation algorithms and an intra-gang synchronization framework for multicore platforms.

The sufficient schedulability tests for sporadic sequential tasks executing on multiprocessors were proposed for non-preemptive *Global EDF* [20], [30], [52] and *Fixed-Priority* [19], [20], [30], [31], [53], [54]. For non-preemptive rigid gang scheduling, Dong and Liu [22] derived a utilization-based schedulability test under *Global EDF*. The closest work to ours is Kim et al. [6], who consider the non-preemptive fixed-priority scheduling of rigid gang tasks in power systems. Our analysis enhances their work mainly by reducing the number of carry-in interfering jobs and applying a classic iterative response time analysis.

VII. Conclusion

In this paper, we propose new schedulability analysis techniques for non-preemptive rigid gang scheduling. For any work-conserving non-preemptive gang scheduling algorithm, we propose a utilization bound test with linear-time complexity. For NP-FP Gang scheduling, we propose two schedulability tests based on the concept of *problem window* and carry-in workload limitation. The first test is of quadratic time complexity, and the second test is pseudo-polynomial since we further refine the analysis by applying fixed-point iteration. We evaluate our proposed schedulability analysis on both synthetic task sets and neural network task benchmarks on multi-TPU edge AI accelerators. The evaluation results demonstrate the effectiveness of the proposed analyses by comparing them with the state-of-the-art schedulability test proposed for NP-FP Gang scheduling.

Future directions include extending the proposed schedulability analysis to moldable and malleable gang scheduling algorithms. Additionally, designing optimal priority assignment algorithms for the proposed NP-FP gang schedulability tests is also interesting to study.

Appendix

Theorem A.1. *The NP-FP Gang schedulability test* `Kim2016` *(19) is OPA-compatible.*

$$\sum_{\tau_i \in \tau \setminus lephev(k)} W^{CI}_{k \leftarrow i}(\Delta) + \sum_{\tau_i \in lphev(k)} W^{one}_{k \leftarrow i}(\Delta) < M_k \cdot \Delta \qquad (19)$$

Proof: We show that the three conditions for OPA compatibility hold for (19). In (19), the computation of workload $W^{CI}_{k \leftarrow i}(\Delta)$ and $W^{one}_{k \leftarrow i}(\Delta)$ depend on independent properties of task $\tau_i \in \tau \setminus lephev(k)$ and $lphev(k)$, respectively, but not on their relative priority ordering. Therefore, the first two conditions hold. For the third condition, we consider two tasks τ_x and τ_y initially at priorities k and $k+1$, respectively, and τ_y is schedulable. There are three circumstances to consider:

1) $m_x < m_y$. If we swap the priorities of the two tasks, task τ_x moves from $hplev(y)$ to $lplv(y)$.
2) $m_x = m_y$. If we swap the priorities of the two tasks, task τ_x moves from $hplev(y)$ to $lphev(y)$.
3) $m_x > m_y$. If we swap the priorities of the two tasks, task τ_x moves from $hphv(y)$ to $lphev(y)$.

For circumstance 1), the computation of the LHS of (19) does not change since $hplev(y)$ and $lplv(y)$ are both within set $\tau \setminus lephev(y)$. For circumstance 2) and 3), the workload of task τ_x needs to be excluded from $\sum_{\tau_i \in \tau \setminus lephev(k)} W^{CI}_{y \leftarrow i}(\Delta)$ and included in $\sum_{\tau_i \in lphev(k)} W^{one}_{y \leftarrow i}(\Delta)$. Since $W^{CI}_{y \leftarrow x}(\Delta) \geq W^{one}_{y \leftarrow x}(\Delta)$, the change will result in an equal or smaller LHS of (19), thus cannot turn task τ_y from schedulable to unschedulable. Hence, the third condition holds. ∎

Acknowledgment

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

[1] 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.

[2] Z. Dong, K. Yang, N. Fisher, and C. Liu, “Tardiness bounds for sporadic gang tasks under preemptive global EDF scheduling,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2867–2879, 2021.

[3] S. Wasly and R. Pellizzoni, “Bundled scheduling of parallel real-time tasks,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 130–142.

[4] 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.

[5] M. A. Jette, “Performance characteristics of gang scheduling in multiprogrammed environments,” in *ACM/IEEE Conference on Supercomputing*, 1997, pp. 1–12.

[6] E. Kim, J. Lee, L. He, Y. Lee, and K. G. Shin, “Offline guarantee and online management of power demand and supply in cyber-physical systems,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 89–98.

[7] “Edge TPU,” https://cloud.google.com/edge-tpu, accessed: 2022-10-25.

[8] “Edge TPU Performance Benchmarks,” https://coral.ai/docs/edgetpu/benchmarks, accessed: 2022-10-25.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[10] S. Christian, V. Vincent, S. Ioffe, S. Jon, and W. Zbigniew, “Rethinking the Inception architecture for computer vision,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[11] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in *Thirty-first AAAI Conference on Artificial Intelligence*, 2017.

[12] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[13] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, “A comprehensive survey of industry practice in real-time systems,” *Real-Time Systems*, vol. 58, no. 3, pp. 358–398, 2021.

[14] M. Bertogna and M. Cirinei, “Response-time analysis for globally scheduled symmetric multiprocessor platforms,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 149–160.

[15] T. P. Baker, “Multiprocessor EDF and Deadline Monotonic schedulability analysis,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2003, pp. 120–129.

[16] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor scheduling for real-time systems*. Springer Cham, 2015.

[17] S. Baruah, “Techniques for multiprocessor global schedulability analysis,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 119–128.

[18] N. Guan, M. Stigge, W. Yi, and G. Yu, “New response time bounds for fixed priority multiprocessor scheduling,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2009, pp. 387–397.

[19] N. Guan, W. Yi, Q. Deng, Z. Gu, and G. Yu, “Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling,” *Journal of Systems Architecture*, vol. 57, no. 5, pp. 536–546, 2011.

[20] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu, “New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2008, pp. 137–146.

[21] H. Leontyev and J. H. Anderson, “A unified hard/soft real-time schedulability test for global EDF multiprocessor scheduling,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2008, pp. 375–384.

[22] Z. Dong and C. Liu, “Work-in-progress: Non-preemptive scheduling of sporadic gang tasks on multiprocessors,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 512–515.

[23] R. J. Bril, J. J. Lukkien, R. I. Davis, and A. Burns, “Message response time analysis for ideal controller area network (CAN) refuted,” *the 5th International Workshop on Real-Time Networks*, pp. 5–10, 2006.

[24] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, “Controller area network (can) schedulability analysis: Refuted, revisited and revised,” *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[25] S. Lauzac, R. Melhem, and D. Mosse, “Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor,” in *the 10th EUROMICRO Workshop on Real-Time Systems*, 1998, pp. 188–195.

[26] B. Andersson and J. Å. Jönsson, “Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2000, pp. 53–56.

[27] G. B. Dantzig, “Discrete-variable extremum problems,” *Operations Research*, vol. 5, no. 2, pp. 266–288, 1957.

[28] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[29] E. Balas and E. Zemel, “An algorithm for large zero-one knapsack problems,” *Operations Research*, vol. 28, no. 5, pp. 1130–1154, 1980.

[30] J. Lee and K. G. Shin, “Improvement of real-time multi-coreschedulability with forced non-preemption,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1233–1243, 2014.

[31] J. Lee, “Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling,” *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1816–1823, 2017.

[32] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns, “A review of priority assignment in real-time systems,” *Journal of Systems Architecture*, vol. 65, pp. 64–82, 2016.

[33] N. C. Audsley, “On priority assignment in fixed priority scheduling,” *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[34] R. I. Davis and A. Burns, “Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems,” *Real-Time Systems*, vol. 47, pp. 1–40, 2011.

[35] D. Griffin, I. Bate, and R. I. Davis, “Generating utilization vectors for the systematic evaluation of schedulability tests,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 76–88.

[36] E. Bini and G. C. Buttazzo, “Measuring the performance of schedulability tests,” *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[37] P. Emberson, R. Stafford, and R. I. Davis, “Techniques for the synthesis of multiprocessor tasksets,” in *the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010, pp. 6–11.

[38] M. Kubale, “The complexity of scheduling independent two-processor tasks on dedicated processors,” *Information Processing Letters*, vol. 24, no. 3, pp. 141–147, 1987.

[39] J. Goossens and V. Berten, “Gang FTP scheduling of periodic and parallel rigid real-time tasks,” *arXiv preprint arXiv:1006.2617*, 2010.

[40] J. Goossens and P. Richard, “Optimal scheduling of periodic gang tasks,” *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 04:1–04:18, 2016.

[41] S. Baruah, “Fairness in periodic real-time scheduling,” in *IEEE Real-Time Systems Symposium (RTSS)*, 1995, pp. 200–209.

[42] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, “Proportionate progress: A notion of fairness in resource allocation,” *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[43] 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.

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

[45] G. Nelissen, J. Marcè i Igual, and M. Nasri, “Response-time analysis for non-preemptive periodic moldable gang tasks,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, vol. 231, 2022, pp. 12:1–12:22.

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

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

[48] Z. Dong and C. Liu, “Analysis techniques for supporting hard real-time sporadic gang task systems,” *Real-Time Systems*, vol. 55, no. 3, pp. 641–666, 2019.

[49] N. Ueter, M. Günzel, G. von der Brüggen, and J.-J. Chen, “Hard real-time stationary gang-scheduling,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021, pp. 10:1–10:19.

[50] W. Ali and H. Yun, “RT-Gang: Real-time gang scheduling framework for safety-critical systems,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 143–155.

[51] W. Ali, R. Pellizzoni, and H. Yun, “Virtual gang scheduling of parallel real-time tasks,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 270–275.

[52] S. K. Baruah, “The non-preemptive scheduling of periodic tasks upon multiprocessors,” *Real-Time Systems*, vol. 32, no. 1–2, p. 9–20, 2006.

[53] H. Baek and J. Lee, “Improved schedulability test for non-preemptive fixed-priority scheduling on multiprocessors,” *IEEE Embedded Systems Letters*, vol. 12, no. 4, pp. 129–132, 2020.

[54] J. Lee and K. G. Shin, “Controlling preemption for better schedulability in multi-core systems,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2012, pp. 29–38.