Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks

Citation for published version (APA):
Nelissen, G., Igual, J. M. I., & Nasri, M. (2022). Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks. In M. Maggio (Ed.), *34th Euromicro Conference on Real-Time Systems, ECRTS 2022* (pp. 12:1-12:22). Article 12 (Leibniz International Proceedings in Informatics, LIPIcs; Vol. 231). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECRTS.2022.12

DOI:
10.4230/LIPIcs.ECRTS.2022.12

Document status and date:
Published: 28/06/2022

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks

Geoffrey Nelissen
Eindhoven University of Technology, The Netherlands

Joan Marcè i Igual
Eindhoven University of Technology, The Netherlands

Mitra Nasri
Eindhoven University of Technology, The Netherlands

Abstract

Gang scheduling has long been adopted by the high-performance computing community as a way to reduce the synchronization overhead between related threads. It allows for several threads to execute in lock steps without suffering from long busy-wait periods or be penalized by large context-switch overheads. When combined with non-preemptive execution, gang scheduling significantly reduces the execution time of threads that work on the same data by decreasing the number of memory transactions required to load or store the data. In this work, we focus on two main types of gang tasks: *rigid* and *moldable*. A moldable gang task has a presumed known minimum and maximum number of cores on which it can be executed at runtime, while a rigid gang task always executes on the same number of cores. This work presents the first response-time analysis for non-preemptive moldable gang tasks. Our analysis is based on the notion of schedule abstraction; a new approach for response-time analysis with the promise of high accuracy. Our experiments on periodic rigid gang tasks show that our analysis is 4.9 times more successful in identifying schedulable tasks than the existing *utilization-based test* for rigid gang tasks.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases schedulability analysis, response time analysis, moldable gang tasks, rigid gang tasks, schedule abstraction graph, multiprocessor, non-preemptive

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.12

Funding *Joan Marcè i Igual*: This work was made with the support of the NWO SAM-FMS project (project number 17931) as a part of the MASCOT program.
Mitra Nasri: This work was made with the support of the EU ECSEL Joint Undertaking under grant agreement no 101007260 (project TRANSACT).

1 Introduction

Gang scheduling is a scheduling approach that groups and executes parallel threads as a "gang". A gang of threads reserves a set of cores for their execution. The threads have exclusive access to those cores from the moment they start to execute until they complete or get preempted. Since the early 80s, gang scheduling has been adopted by the high-performance computing community [23] as a way to reduce synchronization overheads between related threads, and to optimize the access time to shared data in data-intensive applications [12]. It allows for many threads to execute in lock steps without suffering from long busy-wait periods or be penalized by large context-switch overheads. Furthermore, it reduces the number of memory transactions by allowing the application to load its data only once for all threads rather than once per thread.

34th Euromicro Conference on Real-Time Systems (ECRTS 2022).
Editor: Martina Maggio; Article No. 12; pp. 12:1–12:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Gang scheduling is a versatile model and has various applications. It is used to assign resources to applications in data centers, to schedule hardware tasks on *field programmable gate arrays* (FPGAs) [5], and is also used in *graphics processor units* (GPUs) [2] where each kernel thread-block needs a fixed number of cores before its execution starts.

The efficiency of gang scheduling improves significantly when it is combined with non-preemptive execution. It then results in smaller execution times as it eradicates the need to reload data into memory after each preemption. This is usually a crucial factor to optimize the worst-case execution time of a gang task, which often treats large amounts of data. Despite these benefits, to the best of our knowledge, no theoretical analysis exists for obtaining a safe upper bound on the worst-case response time of gang tasks under *non-preemptive* scheduling.

In this work, we provide the first *worst-case response-time* (WCRT) analysis for two generic classes of gang tasks, namely, *rigid* and *moldable*, where tasks are scheduled by a non-preemptive *job-level fixed-priority* (JLFP) scheduling policy (e.g., non-preemptive deadline monotonic (DM), earliest-deadline first (EDF), etc.). A *rigid* gang task requires a fixed number of cores to execute its threads. Thus, a job released by a rigid gang task cannot start its execution until at least the number of cores it requires are available. A *moldable* gang task, however, uses a minimum and a maximum number of cores on which it may be executed. Each job of a moldable task may then be allocated a different number of cores by the scheduler depending on how many cores are available at its start time. As a result, the actual execution time of a job of a moldable task depends on the actual number of cores allocated by the scheduler to that job at runtime. It is worth noting that rigid gang is only a particular case of moldable gang. Yet, due to its potentials to adapt its level of parallelism according to the number of cores available, the analysis of moldable gang tasks is a much more challenging problem than that of rigid gang tasks.

Related work. Gang tasks, or so called “coscheduled threads”, were introduced by Ousterhout et al. [23] in 1982. From the late 80s, it has been known that the optimal scheduling of non-preemptive gang (NPGang) tasks is an NP-complete problem as it requires solving a bin packing problem at the core [6]. Since then, further studies have focused on designing more efficient scheduling algorithms either to improve the average-case response time [32] or to reduce the effect of fragmentation [11], which happens when the number of idle cores is not enough to start executing any of the pending jobs.

Work on real-time rigid gang. Goossens et al. [13] show that preemptive gang scheduling under a JLFP policy is not *sustainable* [7] w.r.t. execution time variation. Since then, two schedulability tests [9,13] and one optimal scheduling policy [15] have been introduced for preemptive rigid gang under FP scheduling. However, neither of the tests nor the optimal scheduling policy are applicable on non-preemptive gang tasks. Recently, Dong and Liu [10] introduced a utilization-based test for non-preemptive sporadic gang tasks. That work is the closest to ours as it considers non-preemptive gang tasks. However, it cannot be used to compute an upper bound on the response-time of the tasks as it only outputs a yes/no answer that indicates whether the task set is schedulable or not. It is also limited to rigid gang tasks and cannot analyze moldable gang tasks.

Work on real-time moldable and malleable gang. Malleable gang tasks are a generalization of moldable gang tasks according to which a job may change its level of parallelism during its execution. Kato et al. [16] and Richard et al. [26] propose sufficient schedulability tests for preemptive moldable gang scheduling under global EDF. Berten et al. [4] propose a greedy scheduling algorithm and its schedulability test for preemptive moldable gangs. Collette et al. [8] present a feasibility test as well as a scheduling algorithm that minimizes the number of cores required to schedule malleable gang tasks. In their work, they also show that EDF is not an optimal policy for malleable gang. However, these results are again limited to preemptive tasks and have not been extended (and cannot be easily adapted) to non-preemptive scheduling.

Work on bundle scheduling. In 2019, Wasly et al. [30] have extended the classical rigid gang task model and proposed the *bundled task model* (BTM) along with a sufficient schedulability test. Bundle tasks are modelled as a succession of "bundles" with precedence constraints between them. Each bundle is preemptively scheduled and follows a rigid gang model. However, each bundle may request a different number of cores than other bundles, thereby providing a mean to change the parallelism of the task throughout its execution. Nevertheless, similar to prior work, the existing tests for bundle scheduling have been designed for preemptive execution and are not applicable to non-preemptive scheduling.

Work on schedule-abstraction-based analyses. Our response-time analysis for non-preemptive gang tasks is based on the notion of *schedule abstraction*, a new type of response-time analysis that provides highly accurate schedulability results (as shown in [31]). It was first proposed in 2017 [18] and, since then, has been applied to various response-time analysis problems for non-preemptive scheduling on single-core [18,24,25] and multicore platforms [19,20,22], and is also being extended to the analysis of Ethernet TSN [28].

Contributions. In this paper, we propose a new analysis that uses the idea of schedule abstraction [19] to derive the best-case and worst-case response times (BCRT and WCRT) of a set of periodic rigid and/or moldable gang tasks scheduled by a non-preemptive JLFP scheduling policy (detailed in Section 2). To the best of our knowledge, this is the first response-time analysis for non-preemptive moldable (and rigid) gang tasks.

2 System Model

2.1 Platform and Task Model

We assume a platform made of m identical cores on which we execute a set of n non-preemptive periodic moldable gang tasks. We consider periodic tasks since they are present in more than 80% of real-time systems according to the recent survey of Akesson et al. [1] on the industry practice in real-time systems. Each task τ_k ($1 \leq k \leq n$) periodically releases non-preemptive moldable gang *jobs*. Whenever the scheduler dispatches a job J_i, it can chose on how many cores to execute J_i out of a set $\mathcal{P}_i$ (see Section 2.2 for a detailed description of the scheduler). The set $\mathcal{P}_i$ contains all the valid options to parallelize the execution of a job J_i. For each possible number of cores $p \in \mathcal{P}$ that may be allocated to J_i, job J_i will execute for a minimum of $C_i^{\min}(p)$ and a maximum of $C_i^{\max}(p)$ time units before completing its execution. In other words, $C_i^{\min}(p)$ and $C_i^{\max}(p)$ are the best- and worst-case execution times (BCET and WCET) of the job as a function of the number of cores on which it executes. We assume that jobs may experience a bounded release jitter, that may arise from timer inaccuracy, interrupt latency, or queuing delays. That is, a job J_i may be released at any time within an interval $[r_i^{\min}, r_i^{\max}]$ where $r_i^{\min}$ is its *earliest release time* and $r_i^{\max}$ is its *latest release time.*

In sum, each job $J_i \in \mathcal{J}$ is then defined by the tuple $([r_i^{\min}, r_i^{\max}], \bar{C}_i^{\min}, \bar{C}_i^{\max}, \mathcal{P}_i, d_i)$ where $r_i^{\min}$ and $r_i^{\max}$ are the earliest and latest release times of J_i; d_i is its absolute deadline; $\mathcal{P}_i$ is the set of core counts on which J_i may execute; $\bar{C}_i^{\min}$ and $\bar{C}_i^{\max}$ are vectors such that each entry $C_i^{\min}(p)$ and $C_i^{\max}(p)$ contains the BCET and WCET of J_i on p cores, respectively.

For ease of notation, we define $m_i^{\min} = \min\{p \mid p \in \mathcal{P}_i\}$ and $m_i^{\max} = \max\{p \mid p \in \mathcal{P}_i\}$ as the minimum and maximum number of cores on which job J_i may execute. We also define the function $next_i(p)$ for all values $p < m_i^{\max}$ as a function that returns the smallest number of cores larger than p on which job J_i can execute. That is, $next_i(p) = \min\{k \in \mathcal{P}_i \mid k > p\}$. Without loss of generality, we assume that $1 \leq m_i^{\min} \leq m_i^{\max} \leq m$ for all jobs. That is, job J_i cannot execute on less than one core and cannot request more cores than the number of cores in the platform. We also assume that the values in $\bar{C}_i^{\min}$ and $\bar{C}_i^{\max}$ are monotonically decreasing. That is, a job may execute on more cores only if it decreases its execution time.

As mentioned before, rigid gang is a special form of moldable gang, namely, if $m_i^{\min} = m_i^{\max}$ (and thus $|\mathcal{P}_i| = 1$) for a job J_i, then the job is said to be **rigid**, otherwise, it is **moldable**. If all jobs released by a task are rigid, then the task is rigid.

Our response-time analysis is based on schedule abstraction which must be applied on a finite observation window (or job set). For **periodic tasks** with synchronous releases, constrained deadlines (i.e., deadlines smaller than or equal to periods) and with or without release jitter, the length of the observation window is the task set's hyperperiod H, i.e, the least common multiple of all tasks' periods [14]. Thus, the job set $\mathcal{J}$ must include all jobs released by all tasks during the interval $[0, H)$. Further discussions on how to create job sets for periodic tasks with offsets or arbitrary deadlines are found in [18] and [14].

It is worth noting that despite focusing on periodic tasks, our analysis is applicable to any arbitrary job set as well. This is helpful, for example, to analyze tasks with bursty release patterns or with complex arrival models such as generalized multi-frame tasks [21].

2.2 Scheduler Model

Jobs are scheduled non-preemptively using a work-conserving global job-level fixed priority (JLFP) scheduling algorithm that is assumed to follow the following set of rules:

▶ **Rule 1.** *A job J_i is considered* ready *at time t if and only if it is released at or before t, it is not yet completed at t and it is not already executing at time t.*

▶ **Rule 2.** *A job J_i is considered* eligible *to be dispatched at time t if and only if it is ready at time t and there are at least $m_i^{\min}$ cores available at time t.*

▶ **Rule 3.** *The scheduler is invoked whenever a job is released or a job completes.*

▶ **Rule 4.** *At every invocation of the scheduler, the highest priority* eligible *job is chosen to be dispatched.*

▶ **Rule 5.** *The dispatched job is assigned a number of cores equal to the maximum value in $\mathcal{P}_i$ that is smaller than or equal to the number of free cores at the time at which it is dispatched (i.e., it always executes on as many cores as possible so as to maximize its parallelism).*

▶ **Rule 6.** *The number of cores allocated to a job cannot change during its execution*

▶ **Rule 7.** *The execution of a job cannot be preempted once it started.*

▶ **Rule 8.** *No core may remain idle as long as there are eligible jobs to be dispatched.*

Since we assume a JLFP scheduling algorithm, we use the notations hp_i and lp_i to refer to the set of higher and lower priority jobs than job J_i, respectively.

3 Worst-Case Response-Time Analysis

To check the schedulability of a task set, we compute an upper bound on the worst-case response time (WCRT) of each task by calculating an upper bound on the WCRT of each job of the tasks in the observation window (e.g., hyperperiod). If the WCRT of a task is not larger than its deadline, then the task is deemed *schedulable.*

As already mentioned, we use a *schedule abstraction*-based analysis [18–20] as opposed to, e.g., a *critical-instant*-based analysis [3], in order to compute the WCRT of every job in a job set $\mathcal{J}$. In this section, we first explain preliminaries on schedule abstraction (Sec. 3.1) and the challenges to build such an analysis for gang tasks (Sec. 3.2). We then elaborate on how we model the *system state* for gang tasks (Sec. 3.3) and provide a *top-down description* of the analysis in Sec. 3.4. We then discuss the details of the algorithm in Sec. 4 and 5

3.1 Preliminaries on Schedule-Abstraction Technique

Motivation. Schedule abstraction [18–20] is a recently developed technique to analyze the best- and worst-case response times of a set of jobs. It efficiently explores the set of "possible" schedules that can be generated by the job set under a given scheduling policy. A recent comparison between the schedule abstraction and an exact response-time analysis in UPPAAL (a generic formal verification tool) on a set of independent non-preemptive periodic tasks scheduled by global fixed-priority scheduling shows that it is more than *three-order-of-magnitude* faster than UPPAAL while having almost 100% accuracy [31], i.e., it is able to detect almost all schedulable task sets. Since then, further improvements using partial-order reduction techniques [24,25] have yet accelerated the analysis by five additional orders-of-magnitude in comparison to [18]. This impressive gain, which now allows to analyze hundreds of tasks non-preemptively scheduled on a single core platform in a matter of seconds, was achieved at the cost of a negligible added pessimism on the WCRT estimation. However, this improved version of the analysis is currently limited to the analysis of single-core platforms and is therefore not directly applicable nor extendable to the analysis of gang scheduling on multiprocessor platforms yet. For this reason, in this section, we focus on the original idea of the schedule-abstraction graph (SAG) as it was presented in [19].

In terms of accuracy, it has been shown that the SAG analysis is far less pessimistic than critical-instant-based analyses when applied on more complex problems such as the response-time analysis of parallel tasks (with directed-acyclic graph dependencies) on multiprocessor platforms [19]. Namely, for platforms with 4 to 16 cores, the schedule-abstraction technique is *2 to 4.3 times more successful* in identifying schedulable task sets than the analysis of [27] based on the critical-instant theory.

Key idea. Assuming a deterministic scheduling policy and no uncertainty in the release and execution times of the jobs, a job set will have only one possible schedule (which can also be obtained by *simulating* the schedule of the job set during the observation window). Under uncertainties, however, multiple schedules could occur. The schedule-abstraction technique combines these schedules whenever they relate to the same job dispatching ordering on the platform. To defer the state-space explosion and hence scale to reasonably large system sizes, the schedule-abstraction technique uses two key ideas: (i) combining (and abstracting) schedules that share the same jobs dispatching ordering and (ii) introducing merging techniques that allow combining partial schedules (or job orderings) whose future can be explored together (e.g., because they are followed b the same future scheduling decisions).

Schedule-abstraction graph (SAG). It abstracts all possible schedules of a given job set $\mathcal{J}$ in the form of a directed graph $\mathcal{G} = \langle V, E \rangle$, where V is the set of vertices (referred to as nodes) and E is the set of edges each of which connects a pair of vertices to each other. A path in the graph $\mathcal{G}$ represents a possible order of scheduling decisions taken by the scheduler. Each node $v \in V$ represents *the set of system states that may result from the scheduling decisions encoded on the paths that reach to v.* A directed edge connecting a node v to a node v' in $\mathcal{G}$ represents a scheduling decision (i.e., dispatching of a job) taken by the scheduler that brings the set of system states represented by v to a subset of the system states represented by v'. A scheduling decision is equivalent to *dispatching* a job on the platform. Hence, each edge in the graph is labeled by a job. Since the existing analyses that use the schedule-abstraction graph are designed for non-preemptive jobs, the best- and worst-case completion time of a job only depends on its start time, and thus on the system state v after which it is dispatched. The earliest and latest completion time of the dispatched job is also recorded on the edge of the graph.

Given the potential uncertainty on the release and execution times of the jobs, a job J_i may appear in different places in the schedule abstraction graph (i.e., after different sequences of scheduling decisions). Therefore, the WCRT of a job is given by the largest completion time recorded on all edges referencing that job in the SAG.

3.2 Challenges in Analyzing Gang Tasks using Schedule Abstraction

Two of the clear differences between this work and previous works on schedule abstraction [18–20,22] is that: (1) a job of a gang task may need more than one core to start executing, hence cores may remain idle even when there is pending workload, and (2) a single job may release (or free) more than one core simultaneously. These two particularities imply that the time at which different cores start and stop executing workload is somewhat synchronized. Thus, there is a need for system state representation that allows us to keep track of such synchronization between cores in order to correctly and efficiently analyze possible schedules. Furthermore, due to reason (1), the existing rules for analyzing global JLFP scheduling policies are not applicable to the gang scheduling algorithm introduced in Sec. 2.2. This means that new expansion and merge rules must be designed.

3.3 System-State Representation

To keep track of the number of cores claimed by a gang job at the time it is being dispatched on the platform and hence to address challenge (2) in Sec. 3.2, we encode additional information on the edges of the graph. In our new schedule abstraction, an edge between nodes v and v' will not only be labeled with the job J_i that is being dispatched after state v but it also includes the number of cores that are claimed by J_i at the time it has been dispatched.

To address challenge (1) in Sec. 3.2, we develop a new model to encode a system state using four data: (i) the set $\mathcal{S}(v)$ of all jobs that have already been dispatched to reach the system state represented by node v, (ii) the set of all possible instants at which cores may become available to execute new workload, and (iii) the number of cores that might be freed at the exact same time, and (iv) the time at which those cores become available (recall that a gang job executes on p parallel cores and thus releases p cores when it completes).

To encode (ii), we use m intervals. Each interval $A_k(v) = [A_k^{\min}(v), A_k^{\max}(v)]$ (with $1 \leq k \leq m$) encloses all the time instants at which k cores may become available to execute new workload in node v in $\mathcal{G}$. That is, $A_k^{\min}(v)$ is the time until which there are certainly *less than* k cores available, and $A_k^{\max}(v)$ is the time by which *at least* k cores are certainly

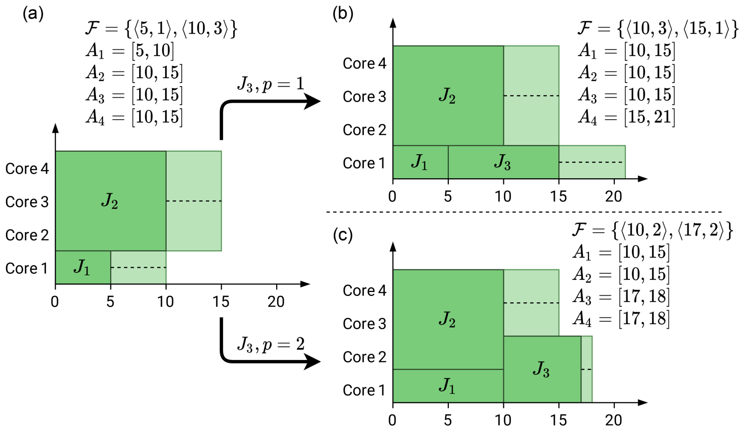

Figure 1 Example of two possible execution scenarios for J_3 and their resulting system states. **(a)** initial state, **(b)** J_3 scheduled with $p = 1$, **(c)** J_3 scheduled with $p = 2$.

available to execute new jobs. We call $A_k^{\min}(v)$ and $A_k^{\max}(v)$ the earliest and latest availability time of k cores for system state v, and we call $A_k(v)$ the availability interval of k cores in state v. In the following, when there is no ambiguity, we do not explicitly write the system state v when referring to A_k, $A_k^{\min}$ and $A_k^{\max}$.

To encode (iii) and (iv) and thus know how many cores may be freed simultaneously by a single job at what time, we store a set of pairs of values $\mathcal{F} = \big\{\big\langle f_1(v), M_1(v)\big\rangle, \big\langle f_2(v), M_2(v)\big\rangle, \dots\big\}$ such that each pair $F_\ell(v) = \big\langle f_\ell(v), M_\ell(v)\big\rangle$ has the following meaning: at least $M_\ell(v)$ cores will be freed by a single job no earlier than time $f_\ell(v)$. By definition, we have that the total number of cores that may be freed is equal to m, i.e., $\sum_{\ell>0} M_\ell(v) = m$, and the earliest time $f_\ell(v)$ at which a group of cores may be freed must also correspond to the earliest time at which some core may become available, i.e., $\forall \ell, \exists k$ s.t. $f_\ell(v) = A_k^{\min}(v)$.

▶ **Example 1.** Figure (1a) shows a system with $m = 4$ cores where two jobs have been scheduled: J_1 on one core, and J_2 on three cores. J_1 must finish within the interval $[5, 10]$, and J_2 must finish within $[10, 15]$. Therefore, one core becomes possibly available at time 5 and three additional cores become possibly available simultaneously at time 10. Similarly, one core is certainly available at time 10 and three more cores become certainly available at time 15. Thus, $\mathcal{F} = \{\langle 5, 1\rangle, \langle 10, 3\rangle\}$, $A_1 = [5, 10], A_2 = [10, 15], A_3 = [10, 15]$ and $A_4 = [10, 15]$.

Now, assume that a job J_3 is released at time 1 with $\mathcal{P}_3 = \{1, 2\}$, $C_3^{\min}(1) = 10$, $C_3^{\min}(2) = 7$, $C_3^{\max}(1) = 11$ and $C_3^{\max}(2) = 8$. Two execution scenarios are possible, hence two new system states are created. If J_1 finishes before J_2 then one core will be freed and J_3 will be scheduled with $p = 1$. This means that J_3 starts executing at the earliest at time 5 and at the latest at time 10. For $p = 1$ we know that the best-case and worst-case execution time of J_3 is 10 and 11, respectively. Therefore, the finish time interval of J_3 is $[15, 21]$. Then, as shown in Figure (1b), three cores become possibly available at time 10 and one additional core becomes possibly available at time 15. Therefore, we have $\mathcal{F} = \{\langle 10, 3\rangle, \langle 15, 1\rangle\}$, and the availability intervals become $A_1 = [10, 15]$, $A_2 = [10, 15]$, $A_3 = [10, 15]$ and $A_4 = [15, 20]$.

However, in another execution scenario where J_1 and J_2 finish at the same time, J_3 will be dispatched on $p = 2$ cores. This can only happen at time 10. For $p = 2$, the execution-time interval of J_3 is $[7, 8]$, leading to the finish-time interval $[17, 18]$. Thus, the new availability intervals are $A_1 = [10, 15]$, $A_2 = [10, 15]$, $A_3 = [17, 18]$, $A_4 = [17, 18]$ and $\mathcal{F} = \{\langle 10, 2\rangle, \langle 17, 2\rangle\}$ as shown in Figure (1c).

Algorithm 1 Algorithm to generate a schedule abstraction graph.

```
   Input   : Job set $\mathcal{J}$
   Outputs : Bounds on the BCRT and WCRT of every job in $\mathcal{J}$;
1  $\forall J_i \in \mathcal{J}, BCRT_i \leftarrow \infty$, $WCRT_i \leftarrow 0$;
2  initialize $\mathcal{G}$ with a root node $v_1$ with $\mathcal{S}(v_1) = \emptyset$, $A_k(v_1) = [0, 0], \forall 1 \leq k \leq m$, and
     $\mathcal{F}(v_1) = \{(0, m)\}$;
3  while $\exists$ a leaf node $v$ s.t. $\mathcal{S}(v) \neq \mathcal{J}$ do
4      $P \leftarrow$ the shortest path from $v_1$ to a leaf node $v$;
5      $v \leftarrow$ the leaf vertex of $P$;
6      for each job $J_i \in \mathcal{J} \setminus \mathcal{S}$ do
7          for $\forall p \in \mathcal{P}_i$ do
8              if $J_i$ may be dispatched next on $p$ cores then
9                  Compute the earliest and latest finish time $EFT_i^p$ and $LFT_i^p$ of $J_i$ on $p$ cores;
10                 $BCRT_i \leftarrow \min\{EFT_i^p - r_i^{min}, BCRT_i\}$;
11                 $WCRT_i \leftarrow \max\{LFT_i^p - r_i^{min}, WCRT_i\}$;
12                 Build the next states using Alg. 2;
13                 Try to merge the new system states with other nodes in $\mathcal{G}$ (Sec. 5);
14 return $BCRT_i$, $WCRT_i$ for all $J_i \in \mathcal{J}$;
```

3.4 Constructing the Schedule Abstraction Graph

The schedule-abstraction graph for a job set $\mathcal{J}$ is built according to Algorithm 1 following a breadth-first strategy. The algorithm starts by building an initial node v_1 representing the state of the system when no job started to execute yet. Therefore, v_1 is initialized with an empty set of scheduled jobs ($\mathcal{S}(v_1) = \emptyset$), with all cores potentially and certainly available at time 0 (i.e., $A_k(v_1) = [0, 0], \forall 1 \leq k \leq m$,) and with all m cores being freed simultaneously at time 0 (i.e., $\mathcal{F}(v_1) = \{\langle 0, m \rangle\}$).

Then, for each node in the graph that has not been analyzed yet (Line 3), Algorithm 1 checks which jobs that have not been scheduled yet may be dispatched next by the scheduler and on how many cores they may be executed (Lines 6 to 8). For each such job J_i and number of cores p, the earliest and latest completion times EFT_i^p and LFT_i^p of the job are computed (Line 9). If the computed completion times result in larger (smaller, respectively) worst-case (best-case, respectively) response times for J_i than those computed on other path of the graph (i.e., for other sequences of scheduling decisions), then it updates the recorded values $WCRT_i$ and $BCRT_i$ for their WCRT and/or BCRT (Lines 10 to 11). Finally, Algorithm 1 uses Algorithm 2 presented later in Section 4.3 to build *all* system states that may result from scheduling J_i on p cores in state v (Line 12) and hence expand the graph. Section 4 provides more details and explanations about this *expansion phase.*

To defer a potential state-space, Algorithm 1 tries to merge the newly created nodes with existing ones and hence reduce the number of branches in the graph (Algorithm 1). Section 5 provides more details about this *merge phase.* Finally, the algorithm stops when no more job can be added to any of the leaf nodes, namely, when the set of scheduled jobs in each leaf node is equal to the set of input jobs (i.e., $\mathcal{S}(v) = \mathcal{J}$).

4 Expansion Phase

The expansion phase has three consecutive steps: **(1)** for each job J_i that was not dispatched yet (i.e., $J_i \notin \mathcal{S}(v)$) and for each possible number of cores $p \in \mathcal{P}_i$, check whether J_i may be the next job dispatched by the scheduler on exactly p cores in state v, **(2)** if J_i may be dispatched next, compute the earliest and latest finish times of J_i, and finally, **(3)** build the new system states resulting from the scheduler dispatching J_i on p cores in state v. We discuss each of those steps in Sections 4.1–4.3.

4.1 Dispatch Condition

To check whether a job J_i may be the next job dispatched by the scheduler on p cores in system state v, we first compute the earliest time $EST_i^p(v)$ at which that job would be starting to execute on p cores if it was the only job left to execute. Then, we compute the latest time $LST_i^p(v)$ at which it must have started *in order to be the first job dispatched by the scheduler* considering all the other pending jobs in the system. It is crucial to understand that $LST_i^p(v)$ is defined under the condition that J_i is going to be the first job being dispatched after the state v. Scenarios where J_i is not the first job to be dispatched after the state v will be automatically explored during future expansions of the graph.

If $LST_i^p(v)$ is larger than or equal to $EST_i^p(v)$, then there exists an execution scenario in which J_i may be the next job dispatched on p cores by the scheduler. Otherwise, if $LST_i^p(v) < EST_i^p(v)$, then either J_i cannot be dispatched on p cores or there will always be *another job* dispatched *before* J_i.

4.1.1 Earliest Start Time

The earliest start time $EST_i^p(v)$ of J_i on p cores ($p \in \mathcal{P}_i$) depends on the following properties:

(i) by Rule 2, J_i cannot start before it is released (i.e, $EST_i^p(v) \geq r_i^{\min}$);

(ii) there must be at least p cores available to start to execute J_i on p cores; and

(iii) by Rule 5, if $p < m_i^{\max}$, then less than $next_i(p)$ cores may be available when J_i is dispatched (otherwise, by Rule 5, it would be dispatched on more than p cores).

Hence, as proven in Lemma 2 below, we can compute $EST_i^p(v)$ as follows

$$EST_i^p(v) = \max\{r_i^{\min}, t_{gang}^p(v)\} \tag{1}$$

$$t_{gang}^p(v) = \begin{cases} A_p^{\min}(v) & \text{if } p = m_i^{\max}, \\ A_p^*(v) & \text{otherwise.} \end{cases} \tag{2}$$

where we define $A_p^*(v)$ as the earliest time at which *at least* p cores but less than $next_i(p)$ cores may become available. Note that $A_p^*(v)$ is different from $A_p^{\min}(v)$ in the sense that $A_p^{\min}(v)$ only ensures that at least p cores are available but does not enforce that there are less than $next_i(p)$ available cores. Section 4.1.2 will explain how to compute $A_p^*(v)$.

▶ **Lemma 2.** *A job J_i cannot start executing on exactly p cores before time $EST_i^p(v)$.*

Proof. We analyze two cases:

Case 1. If $p = m_i^{\max}$, job J_i cannot start before to be released (i.e., before $r_i^{\min}$) and cannot start until at least p cores are available (i.e., at $A_p^{\min}(v)$). Thus, J_i cannot start before $\max\{r_i^{\min}, A_p^{\min}(v)\}$, thus proving the claim for the case $p = m_i^{\max}$ in Equation (1).

Case 2. If $p < m_i^{\max}$, again, job J_i cannot start before $r_i^{\min}$ and at least p cores are available. Furthermore, by Rule 5, J_i cannot start executing on p cores if $next_i(p)$ or more cores are available. Thus, J_i cannot start before $\max\{r_i^{\min}, A_p^*(v)\}$. ◀

4.1.2 Computing $A_p^*(v)$

Let $A_k^{exact}(v)$ be the earliest time at which *exactly* k cores may become available. Then, the earliest time at which *at least* p cores but less than $next_i(p)$ cores may become available is given by $A_p^*(v) = \min_k\{A_k^{exact}(v) \mid p \leq k < next_i(p)\}$.

The value of $A_k^{exact}(v)$ can be computed from the information available in $\mathcal{F}(v)$. Specifically, we must find a subset $\mathcal{F}' \subseteq \mathcal{F}(v)$ such that $\sum_{F_\ell \in \mathcal{F}'} M_\ell = k$ and for which the time at which the latest core is freed (i.e., the time given by $\max_{F_\ell \in \mathcal{F}'}\{f_\ell\}$) is minimum. The earliest time $A_k^{exact}(v)$ at which exactly k cores may become available is then equal to the time at which the last core in $\mathcal{F}'$ is freed, i.e., $A_k^{exact}(v) = \max_{F_\ell \in \mathcal{F}'}\{f_\ell\}$.

If there is no subset $\mathcal{F}' \subseteq \mathcal{F}(v)$ such that $\sum_{F_\ell \in \mathcal{F}'} M_\ell = k$, then there is no possibility for exactly k cores to become simultaneously available in system state v, i.e., there will always be more cores or less cores available at any time. In that case, we set $A_k^{exact}(v) = +\infty$.

Note that to avoid computing all combinations of values in $\mathcal{F}(v)$ to compute $A_k^{exact}(v)$, one can use text-book solutions that solve the subset-sum problem. Namely, using dynamic programming [17] for the subset-sum problem, one can compute $A_p^*(v)$ with a complexity $O(s \cdot N)$ where s is the maximum sum to find, and N is the number of elements in the set $\mathcal{F}$. In our case, both s and the size of $\mathcal{F}$ are upper-bounded by the number of cores m resulting to an $O(m^2)$ complexity.

4.1.3 Latest Start Time

The latest time LST_i^p at which job J_i may start to execute on p cores assuming that it is the next job that is dispatched by the scheduler depends on three factors:

(i) The time $t_{avail}^p(v)$ at which at least $next_i(p)$ cores become available, since, if $m_i^{\max} > p$, the scheduler would then dispatch J_i on more than p cores (instead of p cores);

(ii) The time $t_{wc}(v)$ at which another job than J_i certainly becomes eligible for execution, since J_i will not be dispatched *first* if it has not been dispatched before $t_{wc}(v)$;

(iii) The time $t_{high}^p(v)$ at which a higher-priority job may become eligible, since to be dispatched before any other job, J_i must be dispatched before time $t_{high}^p(v)$.

We explain how to compute bounds on those three time instants.

First, according to Rule 5, if J_i starts to execute on p cores at time LST_i^p, then either p is the maximum number of cores on which J_i may execute, i.e., $p = m_i^{\max}$, or there are no more than p cores available at time LST_i^p. Since $A_{next_i(p)}^{\max}(v)$ denotes the time by which $next_i(p)$ cores will certainly become available, we have that

$$LST_i^p(v) \leq t_{avail}^p(v) \tag{3}$$

where

$$t_{avail}^p(v) = \begin{cases} A_{next_i(p)}^{\max}(v) - 1 & \text{if } p < m_i^{\max}, \\ +\infty & \text{otherwise.} \end{cases} \tag{4}$$

Second, if J_i is the first job dispatched by the scheduler until time LST_i^p, then according to Rules 3 and 8, there must be no other job that was eligible to be dispatched before LST_i^p. Since by Rule 2, a job J_j is eligible only if it is ready and there are at least $m_j^{\min}$ cores available, we must have

$$LST_i^p(v) \leq t_{wc}(v) \tag{5}$$

with

$$t_{wc}(v) = \min_{J_j \notin \mathcal{S}(v)} \{\max\{r_j^{\max}, A^{\max}_{m_j^{\min}}(v)\}\}, \tag{6}$$

where $r_j^{\max}$ is the latest time at which a job J_j that was not scheduled yet (i.e., $J_j \notin \mathcal{S}(v)$) may be released, and $A^{\max}_{m_j^{\min}}(v)$ is the latest time by which the minimum number of cores $m_j^{\min}$ requested by J_j will be available to execute J_j.

Third, according to Rule 4, if job J_i is dispatched at time LST_i^p and it is the first job dispatched by the scheduler in system state v, then J_i must be the highest priority eligible job until time LST_i^p. That is,

$$LST_i^p(v) < t^p_{high}(v), \tag{7}$$

with

$$t^p_{high}(v) = \overset{\infty}{\min_{J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}}} \left\{ t_h^p(J_i, J_j) \right\}, \tag{8}$$

where $\min^{\infty}_{x \in S}\{x\} = +\infty$ if $S = \emptyset$, otherwise, $\min^{\infty}_{x \in S}\{x\} = \min_{x \in S}\{x\}$, and

$$t_h^p(J_i, J_j) = \begin{cases} r_j^{\max} & \text{if } m_j^{\min} \le p, \\ \max\{r_j^{\max}, A^{\max}_{m_j^{\min}}\} & \text{otherwise.} \end{cases}. \tag{9}$$

▶ **Lemma 3.** *J_i will not be the first job dispatched in system state v or will not be dispatched on exactly p cores, if it did not start to execute before time $t^p_{high}(v)$ as defined by Equation (8).*

Proof. We prove that a not-yet-dispatched higher-priority job J_j (i.e., $J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}$) will be dispatched before J_i if J_i did not start executing before $t_h^p(J_i, J_j)$. It then directly follows that J_i will not be the first job dispatched on p cores if J_i did not start to execute before $t^p_{high}(v) = \min^{\infty}_{J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}} \left\{ t_h^p(J_i, J_j) \right\}$, hence proving the lemma. We consider two cases:

Case 1. If $m_j^{\min} \le p$, then the higher-priority job J_j requires fewer cores than the number of cores requested by job J_i. Thus, if job J_j is released when J_i become eligible, then according to Rule 2, J_j is also eligible, and because J_j has a higher priority than J_i, the scheduler will dispatch J_j instead of J_i (Rule 4). Therefore, J_i cannot be scheduled before J_j on p cores if it did not start to execute before $r_j^{\max}$. This proves that J_j will be dispatched before J_i if J_i did not start to execute before $t_h^p(J_i, J_j)$.

Case 2. If $m_j^{\min} > p$, then, according to Rule 2, the higher-priority job J_j becomes eligible when it is released and when $m_j^{\min}$ cores are available. This happens at the latest at time $\max\{r_j^{\max}, A^{\max}_{m_j^{\min}}(v)\}$. Then, because J_j has a higher priority than J_i, the scheduler will dispatch J_j if J_i did not start to execute before $\max\{r_j^{\max}, A^{\max}_{m_j^{\min}}(v)\}$ (Rule 4). Thus, we proved that J_j will be dispatched before J_i if J_i did not start to execute before $t_h^p(J_i, J_j)$. ◀

▶ **Corollary 4.** *Job J_i cannot be dispatched on p cores and be the first job dispatched in state v later than $LST_i^p(v) = \min\{t^p_{avail}(v), t_{wc}(v), t_{high}(v) - 1\}$.*

Proof. It directly follows from the combination of Equations (3), (5) and (7). ◀

4.1.4 Dispatch Condition

A job J_i may be dispatched on p cores (with $p \in \mathcal{P}_i$) and may be the first job dispatched by the scheduler in a system state v only if the earliest time at which it may be dispatched on p cores is no later than the latest time at which it may be the first job to be dispatched. That is, it must respect the following inequality:

$$EST_i^p(v) \leq LST_i^p(v) \tag{10}$$

▶ **Theorem 5.** *A job J_i may be dispatched on p cores and be the first job dispatched by the scheduler in system state v only if $EST_i^p(v) < \infty$ and Inequality* (10) *is respected.*

Proof. It is obvious that the earliest start time $EST_i^p(v)$ of J_i must be smaller than ∞ to ensure that J_i may start to execute in system state v. Hence, we focus on Inequality (10). By contradiction, assume that (i) a job J_i is the first job dispatched by the scheduler in system state v, that (ii) J_i is assigned p core by the scheduler and that (iii) J_i does not respect Inequality (10). Let t_s be the time at which J_i starts executing. By Lemma 2, we have that $t_s \geq EST_i^p(v)$. Thus, by assumption (iii) and the definition of $LST_i^p(v)$ given in Corollary 4, we have $t_s > t_{avail}^p$ or $t_s > t_{wc}$ or $t_s \geq t_{high}$. We analyse each case independently.

- $t_s > t_{avail}^p$. Since by Equation (4), $t_{avail}^p \geq A_{next_i(p)}^{\max}(v) - 1$ and because $t_s > t_{avail}^p$, we have $t_s \geq A_{next_i(p)}^{\max}(v)$. Therefore, at least $next_i(p)$ cores are available at time t_s. Thus, by Rule 5, J_i is dispatched on at least $next_i(p)$ cores. This contradicts the assumption (ii) that J_i is dispatched on p cores.
- $t_s > t_{wc}$. By definition of t_{wc}, a job certainly became eligible to be dispatched by time t_{wc}. Therefore, a job must have been dispatched by the scheduler at or before t_{wc}. This contradicts the assumption (i) that J_i is the first job dispatched by the scheduler and J_i is dispatched at time t_s.
- $t_s \geq t_{high}^p$. By Lemma 3, J_i is not the highest-priority eligible job at time t_s. Thus, by Rule 4, it is not the first job dispatched by the scheduler, hence contradicting the assumption (i).

We thus reached a contradiction in all cases, which proves the claim. ◀

4.2 Job Finish Times

The earliest time at which a job J_i may complete its execution when dispatched on p cores is when it starts at the earliest (i.e., at $EST_i^p(v)$) and executes for its best-case execution time on p cores (i.e., for $C_i^{\min}(p)$). That is,

$$EFT_i^p(v) = EST_i^p(v) + C_i^{\min}(p) \tag{11}$$

Similarly, the latest time at which a job J_i may complete its execution when it is the next job dispatched and it is dispatched on p cores is when it starts as late as possible (i.e., at $LST_i^p(v)$) and it runs for its WCET on p cores (i.e., for $C_i^{\max}(p)$). That is,

$$LFT_i^p(v) = LST_i^p(v) + C_i^{\max}(p) \tag{12}$$

4.3 Building New System States

If job J_i satisfies the dispatch condition for p cores in state v, then there are execution scenarios in which the scheduler may dispatch J_i on p cores in system state v. For each such scenario, we build a new node v' representing the system state resulting from scheduling J_i on p cores. Apart from adding J_i to the set of scheduled jobs $\mathcal{S}(v')$, there are two data structures that must be updated. The set of availability intervals, and the set of earliest simultaneous core releases $\mathcal{F}$. We discuss these in the following sub-sections.

4.3.1 New Set of Earliest Simultaneous Core Releases $\mathcal{F}$

Our discussion has two parts. We first cover the case where the number of cores p assigned to J_i is smaller than its maximum parallelism $m_i^{\max}$, and then cover the case where $p = m_i^{\max}$.

4.3.1.1 $p < m_i^{\max}$

If $p < m_i^{\max}$, then exactly p cores must be available when J_i starts to execute (Rule 5). Yet, any combination of simultaneously released cores that sum to p and are possibly released between the earliest and latest start time of J_i may be used to execute J_i. Because there may be more than one such combination, we first identify every subset $\mathcal{F}_k^{=p}$ of elements in $\mathcal{F}(v)$ such that $\sum_{F_\ell \in \mathcal{F}_k^{=p}} M_\ell(v) = p$ and $\forall F_\ell \in \mathcal{F}_k^{=p}$, $f_\ell(v) \leq LST_i(v)$. Then, for each subset $\mathcal{F}_k^{=p} \subseteq \mathcal{F}(v)$ that meets those conditions, we create a new node v'_k in the graph that represents the system state resulting from dispatching J_i on the specific p cores contained in $\mathcal{F}_k^{=p}$. The new set of earliest simultaneous core releases $\mathcal{F}(v'_k)$ in the new state v'_k is then built according to Lemma 6.

▶ **Lemma 6.** *Let node v'_k result from executing J_i on the p cores in $\mathcal{F}_k^{=p}$, then the set of earliest simultaneous core releases is $\mathcal{F}(v'_k) = \left\{\langle EFT_i^p(v), p\rangle\right\} \cup \left\{\mathcal{F}(v) \setminus \mathcal{F}_k^{=p}\right\}$.*

Proof. Since v'_k considers a system state that results from dispatching job J_i on p cores, p cores will be released simultaneously by J_i when it finishes its execution. This happens no earlier than the earliest finish time $EFT_i^p(v)$ of J_i. Therefore, $\mathcal{F}(v'_k) \supseteq \left\{\langle EFT_i^p(v), p\rangle\right\}$.

Furthermore, because by assumption J_i executes on the cores in $\mathcal{F}_k^{=p}$, the time at which the cores in $\mathcal{F}(v) \setminus \mathcal{F}_k^{=p}$ are released is not impacted by the execution of J_i. Thus, $\mathcal{F}(v'_k) \supseteq \left\{\mathcal{F}(v) \setminus \mathcal{F}_k^{=p}\right\}$. ◀

4.3.1.2 $p = m_i^{\max}$

When the number of cores p assigned to J_i is equal to its maximum parallelism $m_i^{\max}$, there must be at least p but also potentially more than p cores available when J_i starts to execute. Thus, differently from the case covered above, we identify every subset $\mathcal{F}_k^{\geq p}$ of $\mathcal{F}(v)$ whose elements sum up to *at least* p. That is, $\sum_{F_\ell \in \mathcal{F}_k^{\geq p}} M_\ell(v) \geq p$ and $\forall F_\ell \in \mathcal{F}_k^{\geq p}$, $f_\ell(v) \leq LST_i(v)$. As before, for each subset $\mathcal{F}_k^{\geq p}$, we create a new node v'_k whose set of earliest simultaneous core releases $\mathcal{F}(v'_k)$ is computed according to Lemmas 7 and 8.

▶ **Lemma 7.** *If all the cores in $\mathcal{F}_k^{\geq p}$ are released when J_i starts to execute, then J_i starts no earlier than $t_k = \max\limits_{F_\ell \in \mathcal{F}_k^{\geq p}} \{f_\ell\}$.*

Proof. By definition of F_ℓ, the M_ℓ cores modeled by F_ℓ are all released at the earliest at time f_ℓ. Thus, all the cores in $\mathcal{F}_k^{\geq p}$ are available no earlier than $\max_{F_\ell \in \mathcal{F}_k^{\geq p}}\{f_\ell\}$. Since J_i starts when all cores in $\mathcal{F}_k^{\geq p}$ are available, this proves the claim. ◀

▶ **Lemma 8.** *Let node v'_k result from executing J_i on p of the cores in $\mathcal{F}_k^{\geq p}$, then the set of earliest simultaneous core releases is $\mathcal{F}(v'_k) = \left\{\langle EFT_i^p(v), p\rangle\right\} \cup \left\{\langle t_k, (s-p)\rangle\right\} \cup \left\{\mathcal{F}(v) \setminus \mathcal{F}_k^{\geq p}\right\}$ where s is the number of cores in $\mathcal{F}_k^{\geq p}$, i.e, $s = \sum_{F_\ell \in \mathcal{F}_k^{\geq p}} M_\ell(v)$.*

Proof. Since v'_k considers a system state that results from dispatching job J_i on p cores, p cores will be released simultaneously by J_i when it finishes its execution. This happens no earlier than the earliest finish time $EFT_i^p(v)$ of J_i. Therefore, $\mathcal{F}(v'_k) \supseteq \left\{\langle EFT_i^p(v),\, p\rangle\right\}$.

Algorithm 2 Build all system states resulting from dispatching J_i on p cores in v.

```
for $\forall \mathcal{F}_k^p \subseteq \mathcal{F}(v)$ s.t. conditions of Sec. 4.3.1 are respected do
    Add a node $v'_k$ to the sched.-abstraction graph $\mathcal{G}$;
    $\mathcal{S}(v'_k) \leftarrow \mathcal{S}(v) \cup \{J_i\}$;
    Compute $PA$ and $CA$ according to Lemmas 9 and 11;
    Sort $PA$ and $CA$ in non-decreasing order ;
    $\forall x,\ 1 \le x \le m,\ A_k(v'_k) = [PA_x,\ CA_x]$;
    Compute $\mathcal{F}(v'_k)$ according to Lemmas 6 and 8;
    Connect $v$ to $v'_k$ with an edge;
```

Furthermore, by assumption, all cores in $\mathcal{F}_k^{\ge p}$ are free when J_i starts to execute. Therefore, all $(s-p)$ cores in $\mathcal{F}_k^{\ge p}$ on which J_i does not execute are free from J_i's start time onward. By Lemma 7, J_i starts no earlier than t_k. Hence, $\mathcal{F}(v'_k) \supseteq \Big\{ \langle t_k,\ (s-p) \rangle \Big\}$.

Finally, because J_i executes on the cores in $\mathcal{F}_k^{\ge p}$, the time at which the cores in $\mathcal{F}(v) \setminus \mathcal{F}_k^{\ge p}$ are released is not impacted by the execution of J_i. Thus, $\mathcal{F}(v'_k) \supseteq \Big\{ \mathcal{F}(v) \setminus \mathcal{F}_k^{\ge p} \Big\}$. ◀

4.3.2 New Availability Intervals

To construct the availability intervals $A_x(v'_k)$ $(1 \le x \le m)$ of a system state v'_k reachable from v, we build the set PA of all instants at which each core may *potentially* be available, and the set CA of the latest possible times at which each core will certainly become available after dispatching J_i on p cores in $\mathcal{F}_k^{=p}$ or $\mathcal{F}_k^{\ge p}$ (depending on whether $p < m_i^{\max}$ or $p = m_i^{\max}$ as discussed above). We do so using Lemmas 9 and 11.

▶ **Lemma 9.** *A set of lower bounds on the time instants at which each core may potentially become available to execute new workload in v'_k is given by*

$$PA = \Big\{ p \times \{EFT_i^p(v)\} \Big\} \cup \Big\{ \max(A_x^{\min}(v),\ t_k) \mid p < x \le m \Big\}$$

where $p \times \{EFT_i^p(v)\}$ means that $EFT_i^p(v)$ appears p times in the set.

Proof. First, since v'_k considers a system state that results from dispatching job J_i on p cores, at least p cores will become available no earlier than the earliest finish time $EFT_i^p(v)$ of J_i. Therefore, PA must contain p times $EFT_i^p(v)$.

Second, by Rule 8, J_i will always be dispatched on the p first cores that become available. Therefore, the earliest time at which the $(m-p)$ remaining cores may become available is the earliest time at which the $(m-p)$ latest cores may become available before dispatching J_i. By definition of the availability intervals, those times are $\{A_x^{\min}(v) \mid p < x \le m\}$.

Finally, since job J_i is the first job dispatched by the scheduler in state v, and because by Lemma 7, t_k is the earliest time at which J_i is dispatched, cores can start to execute new workload no earlier than t_k in v'_k. Combining the three facts above prove the claim. ◀

▶ **Corollary 10.** *A lower bound on the time at which x cores are potentially available to execute new workload in v'_k (i.e., $A_x^{\min}(v'_k)$) is given by the x^{th} element in the non-decreasingly ordered set PA.*

Proof. Since PA contains a lower bound on the availability time of every core in state v'_k, the x^{th} element in the ordered set is a lower bound on the availability time of x cores. ◀

▶ **Lemma 11.** *A set of upper bounds on the time instants at which each core will certainly become available to execute new workload in v'_k is given by*

$$CA = \Big\{p \times \{LFT_i^p(v)\}\Big\} \cup \Big\{\max\{A_x^{\max}(v),\, t_k\} \mid p < x \leq m\Big\}.$$

Proof. Since v'_k represents a system state resulting from dispatching job J_i on p cores, there must be at least p cores that will become available to execute new workload no later than the latest finish time of J_i. That is, there must be p values no smaller than $LFT_i^p(v)$ in CA, i.e., $CA \supseteq \Big\{p \times \{LFT_i^p(v)\}\Big\}$.

Furthermore, all $(m-p)$ cores that do not execute J_i will be freed no later than the certain availability time of the $(m-p)$ latest cores that become available in the initial system state v (i.e., the system state before dispatching J_i). Those times are given by $\Big\{A_x^{\max}(v) \mid p < x \leq m\Big\}$.

Finally, since job J_i is the first job dispatched by the scheduler in v, and because t_k is the earliest time at which J_i is dispatched (Lemma 7), cores can start to execute new workload no earlier than t_k in v'_k. Combining all the above, we prove the lemma. ◀

▶ **Corollary 12.** *An upper bound on the time at which x cores are certainly available to execute new workload in v'_k (i.e., $A^{\max}(v'_k)$) is given by the x^{th} element in the non-decreasingly ordered set CA.*

Proof. Same proof as Corollary 10, replacing PA with CA. ◀

The complete procedure to build the system states resulting from dispatching J_i on p cores in state v is summarized in Algorithm 2.

5 Merge Phase

The merge phase aims at merging a newly created node v_k with a previously existing node v_q (and create a combined state v_z) when they have the same set of scheduled jobs and their core-availability intervals intersects:

▶ **Rule 9.** *If v_k and v_q are two nodes such that $\mathcal{S}(v_k) = \mathcal{S}(v_q)$ and $\forall x,\ 1 \leq x \leq m,\ A_x(v_k) \cap A_x(v_q) \neq \emptyset$, then v_k and v_q are merged into a single state v_z.*

The availability intervals of the merged state v_z are then computed so that they enclose the availability intervals of both states v_k and v_q. That is, $\forall x \mid 1 \leq x \leq m$:

$$A_x(v_z) = \big[\min\{A_x^{\min}(v_k), A_x^{\min}(v_q)\},\quad \max\{A_x^{\max}(v_k), A_x^{\max}(v_q)\}\big]. \tag{13}$$

This way, all possible combinations of instants at which cores become available in either state v_k or v_q is also possible in v_z.

Additionally, the set of earliest simultaneous core releases $\mathcal{F}(v_z)$ of the merged state is computed using Algorithm 3. In essence, for both initial states v_k and v_q, Algorithm 3 sorts the groups of cores that are simultaneously released in a non-decreasing order with respect to the time at which they are released. It then breaks the groups of simultaneously released cores in smaller ones so that the size of the groups match in both states (lines 3–10), i.e., after the transformation we have $|\mathcal{F}'(v_k)| = |\mathcal{F}'(v_q)|$ and $\forall x, 1 \leq x \leq |\mathcal{F}'(v_k)|,\ M'_x(v_k) = M'_x(v_q)$. It then keeps the groups of cores that are released the earliest and assign them to $\mathcal{F}(v_z)$ (lines 10–14), i.e., $|\mathcal{F}(v_z)| = |\mathcal{F}'(v_k)| = |\mathcal{F}'(v_q)|$ and $\forall x, 1 \leq x \leq |\mathcal{F}'(v_k)|,\ M_x(v_z) = M'_x(v_k) = M'_x(v_q)$ and $f_x(v_z) = \min\{f'_x(v_k), f'_x(v_q)\}$.

Algorithm 3 Merge of $\mathcal{F}(v_k)$ and $\mathcal{F}(v_q)$ into $\mathcal{F}(v_z)$.

```
   input  : $\mathcal{F}(v_k)$ and $\mathcal{F}(v_q)$
   output : $\mathcal{F}(v_z)$
1  $\mathcal{F}'(v_k) = \mathcal{F}'(v_q) = \emptyset$;
2  while $\mathcal{F}(v_k) \neq \emptyset \wedge \mathcal{F}(v_q) \neq \emptyset$ do
3      Extract the pair $\langle f_K, M_K \rangle$ such that $f_K$ is the minimum value in $\mathcal{F}(v_k)$ and, in case of
         tie, $M_K$ is the minimum among the tying values. For $\mathcal{F}(v_q)$ extract $\langle f_Q, M_Q \rangle$ using the
         same rule;
4      $M_{new} \leftarrow \min\{M_K, M_Q\}$;
5      Add $\langle f_K, M_{new} \rangle$ to $\mathcal{F}'(v_k)$;
6      Add $\langle f_Q, M_{new} \rangle$ to $\mathcal{F}'(v_q)$;
7      $M_K \leftarrow M_K - M_{new}$;
8      $M_Q \leftarrow M_Q - M_{new}$;
9      Add $\langle f_K, M_K \rangle$ to $\mathcal{F}(v_k)$ if $M_K > 0$ ;
10     Add $\langle f_Q, M_Q \rangle$ to $\mathcal{F}(v_q)$ if $M_Q > 0$ ;
11 forall $1 \leq x \leq |\mathcal{F}'(v_k)|$ do
12     $f_x(v_z) = \min\{f'_x(v_k), f'_x(v_q)\}$;
13     $M_x(v_z) = M'_x(v_k)$;
14     Add $\langle f_x(v_z), M_x(v_z) \rangle$ to $\mathcal{F}(v_z)$;
15 return $\mathcal{F}(v_z)$;
```

We now prove that all simultaneous core release patterns that are possible in one of the two initial states v_k or v_q is also possible in the new merged state v_z.

▶ **Lemma 13.** *If exactly p cores may be available at time t in either v_k or v_q, then exactly p cores may be available at time t in v_z.*

Proof. Assume that v refers to either v_k or v_q. Each group of cores $F_\ell(v) \in \mathcal{F}(v)$ is subdivided in one or several smaller groups of cores in $\mathcal{F}(v_z)$ (lines 3–10 in Algorithm 3), that is, $\exists \mathcal{F}' \subseteq \mathcal{F}(v_z)$, $\sum_{F_x(v_z) \in \mathcal{F}'} M_x(v_z) = M_\ell(v)$. Furthermore, each group of cores in the subset $\mathcal{F}'$ has an *earliest* release time that is earlier than or at the same time as that of F_ℓ (lines 10–14), i.e., $\forall F_x(v_z) \in \mathcal{F}',\ f_x(v_z) \leq f_\ell(v)$. Since for every group of cores that can be simultaneously released at a given time t in v there is a set $\mathcal{F}'$ in v_z composed of the same number of cores, each with an earliest release time no later than t, it then holds that the cores in $\mathcal{F}'$ can also be simultaneously released at t. This proves the lemma. ◀

6 Proof of Correctness

Now that the complete algorithm for building the schedule-abstraction graph has been presented, we prove that the analysis covers all possible execution scenarios and hence returns safe bounds on the BCRT and WCRT of each job in the analyzed job set $\mathcal{J}$.

▶ **Theorem 14.** *For any possible execution scenario such that J_i executes on p cores and finishes at time t, there is a path $\langle v_1, \ldots, v_k \rangle$ in the schedule-abstraction graph such that J_i passes the dispatch condition on p cores in v_k and $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$.*

Proof. Assume that the availability intervals and the set of earliest simultaneous core releases $\mathcal{F}(v_k)$ of state v_k safely model the actual availability times and simultaneous releases of the m cores resulting from the sequence of scheduling decisions encoded in the path $\langle v_1, \ldots, v_k \rangle$.

We prove that $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$, that J_i passes the dispatch condition in v_k and that each state v'_k created by Algorithm 2 because of executing J_i on p cores in v_k, correctly models the actual availability times and simultaneous releases of the cores after executing J_i on p cores.

Under the inductive assumption stated above, Lemma 2 and Corollary 4 prove that $EST_i^p(v_k)$ and $LST_i^p(v_k)$ are safe lower- and upper-bounds on the start time of J_i on p cores in v_k, respectively. Furthermore, since gang jobs are non-preemptive, Equations (11) and (12) are safe lower- and upper-bounds on t (i.e., $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$). Moreover, since $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$, it must hold that $EFT_i^p(v_k) \leq LFT_i^p(v_k)$, and thus the condition of Equation (10) is respected. Then, Lemmas 6 and 8 and Corollaries 10 and 12 prove that the simultaneous releases of the cores and their availability is correctly modeled in each newly created state v'_k resulting from scheduling J_i on p cores. Therefore, the inductive assumption is respected for v'_k. Also, according to Lemma 13, potentially merging v'_k with another node in Algorithm 1 maintains the validity of the inductive assumption.

Finally, since all cores are assumed to be free in the initial system state, the inductive assumption (i.e., correct availability intervals and simultaneous core releases) obviously holds for v_1 and thus follows by induction on all the states created by Algorithm 1. ◀

7 Empirical Evaluation

We performed experiments to: (i) evaluate whether the proposed analysis improves schedulability in comparison to the state of the art, (ii) understand the influence of $m^{\min}$ and $m^{\max}$ on schedulability of moldable gang tasks, and (iii) evaluate the runtime of our analysis.

The experiments were conducted by applying Algorithm 1 to the analysis of rigid and moldable gang tasks under a non-preemptive JLFP policy (some experiments use non-preemptive G-EDF and others G-RM as reported in Table 1). We compared our results with the test by Dong and Liu [10] as it is the only existing test for non-preemptive gang tasks. It is worth noting that the test of Dong and Liu considers sporadic tasks while we have performed the analysis on periodic tasks. We, however, decided to keep this comparison since it is currently the only available test that can be applied on periodic gang tasks.

We implemented Algorithm 1 in C++ and performed the analysis on a cluster using AMD Ryzen Threadripper 2920X 12-Core and Intel Core i9-9900K processors. All machines are equipped with 64 GiB of RAM. Roughly, 60% of the experiments ran on the AMD and 40% on the Intel machines. We report the *CPU time* as the runtime of the analysis.

7.1 Experiments on Synthetic Task Sets

We generate periodic task sets using the same established method used in prior studies [15,19,20]. We randomly generated n utilization values with a total sum of $m \times U$, where U is the system utilization and m is the number of cores. This was carried out using Stafford's RandFixSum algorithm [29] (where we ensure that the utilization U_i of each task is in the interval $[0.001, m_i^{\min}]$). To avoid cases where the hyperperiod is impractically large due to incompatible task periods, we choose the period values with a log-uniform distribution in the interval $[10000, 100000]$ with a granularity of 5000 (as in [20]). Additionally, we discard every task set that contains more than 100,000 jobs in its hyperperiod. To allow comparison with the state-of-the-art, release jitter is set to 0. Note that this favourably impacts our analysis runtime since the schedule abstraction graph branches less often for such setting.

Table 1 Specification of the experiments performed.

Experiment	m	n	$m_i^{\min}$	$m_i^{\max}$	**max** U_i	**Policy**
a-seq-random	8	20	1	$\{1,2,3,\ldots,m\}$	1	NP G-RM
a-seq-divisor				$\{1, 2, 4, 8\}$		
a-gang-random			$\{1,2,3,\ldots,m\}$	$\{1,2,3,\ldots,m\}$	$m \times U$	
a-gang-divisor			$\{1, 2, 4, 8\}$	$\{1, 2, 4, 8\}$		
b	8	20	$m^{\min} = m^{\max}$	$\{1, 2, 4, 6, 8\}$	$m \times U$	NP G-EDF
c	8	8–24	1	$\{1, 2, \ldots, 8\}$	1	NP G-RM
d	4	10	1	$\{1, 2, 3, 4\}$	1	NP G-RM
e	8	20	1	$\{1, 2, \ldots, 8\}$	1	NP G-RM
f	16	32	1	$\{1, 2, \ldots, 16\}$	1	NP G-RM

To evaluate the impact of $m_i^{\min}$ and $m_i^{\max}$ on schedulability, we assign different values for $m_i^{\min}$ and $m_i^{\max}$ for each experiment depending on its purpose, as detailed in Table 1. The $\max U_i$ value specifies the maximum utilization that a single task may have in the specified experiment. The values of $m_i^{\min}$ and $m_i^{\max}$ in experiments *a-seq-random* and *a-gang-random* are selected randomly with a uniform distribution from the set $\{1,2,\ldots,m\}$. In experiments *a-seq-divisor* and *a-gang-divisor*, these values are selected randomly from the set $\{1,2,4,8\}$ which is composed of the divisors of the number of cores $m = 8$, we always ensure that $m_i^{\min} < m_i^{\max}$ when picking random values. Experiment (b) assumes a rigid gang model, thus, $m_i^{\min} = m_i^{\max}$ for all tasks. Finally, in experiments (c)– (f) we show results for the generation methods seq-random, gang-random and when all tasks share the same $m_i^{\min}$ and $m_i^{\max}$ values. In the latter case, $m_i^{\min} = 1$ and $m_i^{\max}$ varies from 1 to 16.

In our experiments, jobs of a task τ_i can execute on any number of cores within $[m_i^{\min}, m_i^{\max}]$. Their BCET and WCET on p cores (with $p \in [m_i^{\min}, m_i^{\max}]$) are set to $\left\lfloor \frac{U_i \times T_i}{2 \times p} \right\rfloor$ and $\left\lfloor \frac{U_i \times T_i}{p} \right\rfloor$, respectively. Hence, execution time decreases with an increasing number of cores, and BCET is half the WCET.

Figure 2 shows the results of each experiment. For each data point in the plots, we generate 450 random task sets and report the *schedulability ratio* (i.e., the percentage of task sets deemed schedulable by the analysis). Additionally, we report the runtime of the schedulability analysis for each task set tested in experiments (d), (e), and (f) as a function of the number of jobs in their hyperperiod.

7.2 Schedulability Results

Impact of system utilization on rigid gang tasks. As shown in Figure 2b, The SAG analysis clearly outperforms the utilization-based test of Dong and Liu for any value of $m^{\max}$. For instance, Dong and Liu's test does not detect any schedulable task set at U=40% while our analysis confirms that between 95 and 100% (depending on the maximum task parallelism $m_i^{\max}$) of these task sets are in fact schedulable. More importantly, our analysis identifies **4.9 times** more schedulable task sets than [10] (over all $m_i^{\max}$ values from 1 to 8). This value is computed by taking the ratio between the surface below all schedulability curves obtained with our analysis and the surface below all schedulability curves of Dong and Liu's test. Another observation is that rigid gang tasks with $m_i^{\max} = 8$ have the highest schedulability in comparison to rigid gangs with $m_i^{\max} < 8$. When $m_i^{\max} = 8$, tasks can only start when all 8 cores are available, hence, the schedulability problem boils down to a schedulability analysis for a uniprocessor platform, where the SAG analysis is highly accurate (see [18]).

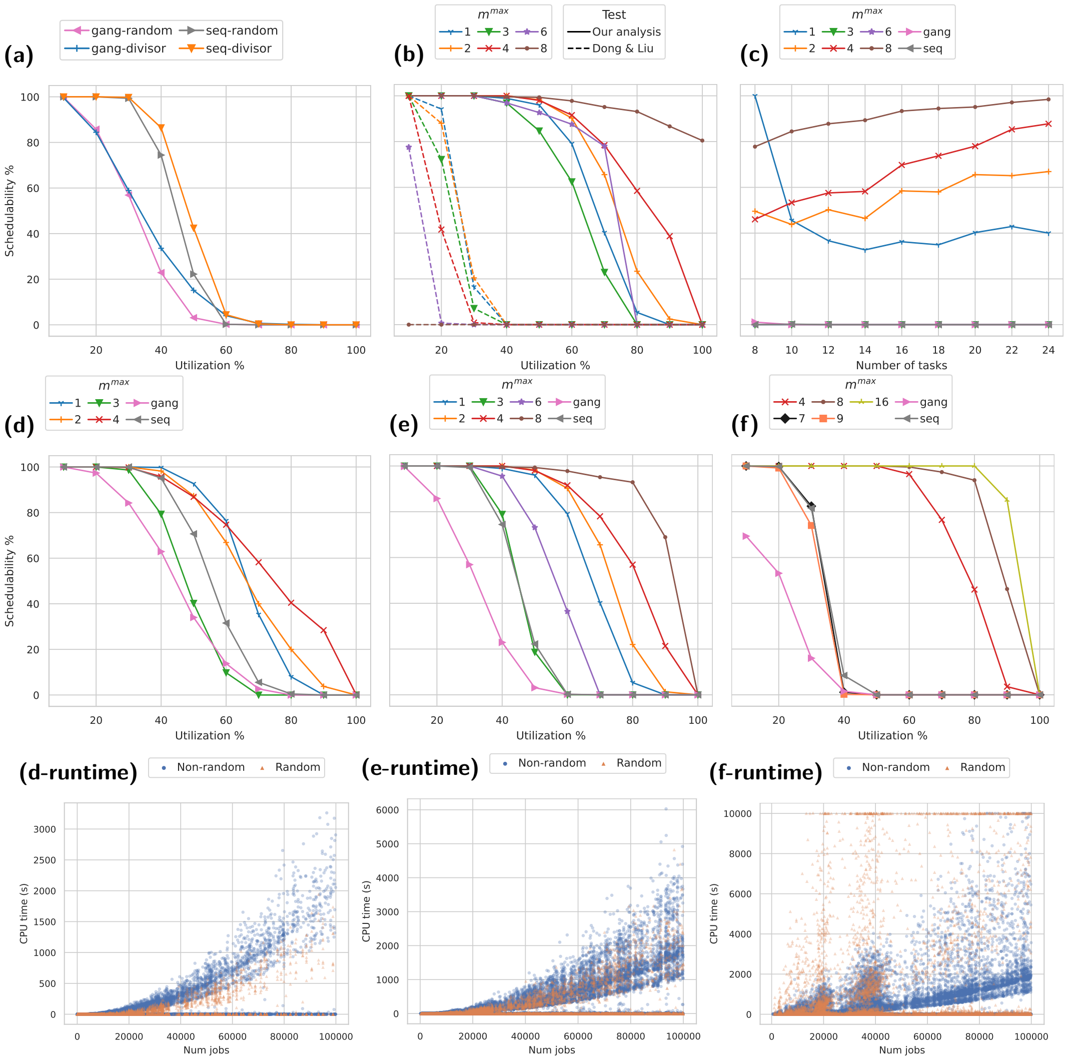

Figure 2 Experimental results. **(a)** moldable gang tasks with m=8, **(b)** rigid gang tasks with m=8, dashed lines are the Dong and Liu's test [10] and the continuous lines are ours, **(c)** moldable gang tasks with m=8, U=0.7 and $m^{\min} = 1$, **(d)** moldable gang tasks with m=4 and $m^{\min} = 1$, **(e)** moldable gang tasks with m=8 and $m^{\min} = 1$, **(f)** moldable gang tasks with m=16 and $m^{\min} = 1$.

Impact of utilization and task parallelism on moldable gang tasks. As shown in Figure 2a (see explanation of the curves in experimental setup), task sets with $m_i^{\min}$ set to 1 for all tasks (i.e., curves *a-seq-random* and *a-seq-divisor*) have a higher-schedulability ratio because they can be dispatched as soon as one or more cores become free. If $m_i^{\min} \geq 1$, however, tasks may experience longer blocking (waiting for their minimum number of cores to be freed) and frequent priority inversions with lower-priority tasks "stealing" available cores from higher-priority ones as it can be seen in *a-gang-random* and *a-gang-divisor* curves.

Furthermore, we compared the difference between choosing $m^{\max}$ to be a random value from 1 to m (the number of cores) and be a random value that is a divisor of m, i.e., $m^{\max} \in \{1, 2, 4, 8\}$. In the latter case (see curves *a-seq-divisor* and *a-gang-divisor*), the schedulability ratio slightly improves in comparison to the former. This slight improvement is caused by having a few less scenarios where cores are left idle with pending workload. However, the impact remains rather small.

Impact of the number of tasks and task parallelism on moldable gang tasks. Figure 2c shows the effect of the number of *moldable* gang tasks when $U = 70\%$. When $m^{\max} = 1$, the results are identical to non-preemptive global scheduling since each task can claim only one core. Also, when $m^{\max} = 8$, the scheduler described in Section 2.2 will execute all jobs on $p = 8$ cores, which is equivalent to single-core scheduling. Thus, as the number of tasks increases, the execution time of the tasks decreases and the results become closer to those of single-core preemptive scheduling. That is why the schedulability ratio increases.

Similarly, when $m_i^{\max}$ is set to 2 or 4 for all tasks, then the scheduler of Section 2.2 behaves identically to a non-preemptive global scheduler on 4 and 2 cores respectively, hence explaining the typical tendency witnessed for such systems. From this experiment, we can conclude that a larger maximum task parallelism is beneficial for schedulability. When $m_i^{\max}$ is set to 3, 6 or is randomly chosen for each task, many jobs cannot be dispatched with their maximum number of cores due to $m_i^{\max}$ not being a divisor of the number of cores. Therefore, consistently with what is seen in Figure (2e) discussed later, the schedulability ratio falls to 0.

Impact of the number of cores and task parallelism on moldable gang tasks. Figure 2d shows that in a system with four processors, configuring $m^{\max}$ to 3 causes a significant lower schedulability ratio than with other values. With eight cores (Figure 2e), setting $m^{\max}$ to 3 or 6 also yields lower schedulability ratios (the same is true for $m^{\max} = 5$ and $m^{\max} = 7$, even though we do not show the results here to avoid clutter). The same effect is visible when $m^{\max}$ is chosen randomly for each task. This shows the positive impact of using a same $m^{\max}$ value that is a divisor of the number of cores for all tasks. When it is the case, all the jobs will always be scheduled with $p = m^{\max}$ because, as soon as a job finishes, it frees exactly the same number of cores as the next job needs to execute with $m^{\max}$ cores. This eliminates the problem of some cores being available but not used due to all pending jobs requesting more cores than available. When $m^{\max}$ is not a divisor of m, some jobs may be executing with $m^{\max}$ cores while others may execute with smaller values of p, this causes an imbalance in execution times that leads to more frequent deadline misses.

Runtime of the analysis. Figure 2 shows the runtime of the SAG analysis when all tasks share the same $m^{\max}$ value (blue) and when $m^{\max}$ is assigned randomly (orange) for the experiments (d)–(f). It shows that the runtime is well below $1000\,s$ in a vast majority of experiments, and the worst-case runtime is below 100 minutes for task sets with 20 tasks scheduled on platforms with 8 cores (see Figure 2e-runtime). For 16 cores executing 32 tasks, we start to see experiments timing out when the number of jobs increases and $m^{\max}$ is assigned randomly (note that those are reported as being deemed unschedulable in Figure (2f)).

Comparison with existing tests for global scheduling of sequential tasks. For the case where the schedulability problem can be reduced to an equivalent global JLFP scheduling problem of non-preemptive sequential tasks, we compared our results with those of the SAG-based test of [19], i.e., the most accurate analysis we are aware of for such systems. All task sets that were detected as schedulable by the test of [19] were also deemed schedulable by our analysis. This suggests that our new analysis reduces to that of [19] in the special case where all tasks share a same $m^{\max}$ value that divides the number of cores m. A comparison against other sufficient tests for global JLFP scheduling can be found in [19].

8 Summary and Conclusion

We proposed a new response-time analysis for rigid and moldable gang tasks scheduled under a non-preemptive JLFP scheduling policy. As far as we know, our work is the first effort to provide sound worst-case (and best-case) response time bounds for such systems. Our analysis is based on the notion of schedule abstraction and efficiently explores all possible sequences of scheduling decisions that may happen during the runtime of the system.

Our experiments show that for periodic rigid gang tasks, our analysis is able to identify 4.9 times more schedulable task sets than the state of the art analysis. Moreover, our experiments revealed the importance of choosing proper values for the minimum and maximum parallelism assigned to moldable gang tasks. We observed that assigning the same value $m_i^{\max}$ to all tasks yields the best performance, specially when $m_i^{\max}$ is a divisor of the number of cores.

We plan on extending our analysis to add support for precedence constraints between moldable gang jobs, and on improving the runtime and memory consumption of our analysis by extending the very promising partial order reduction techniques presented in [24,25]. Among other steps, this will require to develop a fast sufficient schedulability test for non-preemptive moldable gang tasks.

References

1 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–9, 2020.

2 Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.

3 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

4 Vandy Berten, Pierre Courbin, and Joël Goossens. Gang fixed priority scheduling of periodic moldable real-time tasks. In *Junior Researcher Workshopon Real-Time Computing (JRWRTC)*, pages 9–12, 2011.

5 A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2016.

6 Jacek Blazewicz, Mieczyslaw Drabowski, and Jan Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, c-35(5):389–393, 1986.

7 Felipe Cerqueira, Geoffrey Nelissen, and Björn B Brandenburg. On strong and weak sustainability, with an application to self-suspending real-time tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 26–1, 2018.

8 Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008.

9 Zheng Dong and Cong Liu. Analysis techniques for supporting hard real-time sporadic gang task systems. *Real-Time Systems*, 55(3):641–666, 2019.

10 Zheng Dong and Cong Liu. Work-in-progress: Non-preemptive scheduling of sporadic gang tasks on multiprocessors. In *Work-in-Progress of IEEE Real-Time Systems Symposium (WiP-RTSS)*, pages 512–515. IEEE, 2019.

11 Dror G. Feitelson. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 89–110, 1996.

12 Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

13 Joël Goossens and Vandy Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 189–196, 2010.

14 Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-Time Systems*, 52(6):808–832, 2016.

15 Joël Goossens and Pascal Richard. Optimal scheduling of periodic gang tasks. *Leibniz transactions on embedded systems*, 3(1):4:1–4:18, 2016.

16 Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 459–468, 2009.

17 Konstantinos Koiliaris and Chao Xu. Faster Pseudopolynomial Time Algorithms for Subset Sum. *ACM Transactions on Algorithms*, 15(3):1062–1072, 2019.

18 Mitra Nasri and Björn B. Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2017.

19 Mitra Nasri, Nelissen Geoffrey, and Björn B. Brandenburg. Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133, pages 21:1–21:23, 2019.

20 Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 106, pages 9:1–9:23, 2018.

21 Saranya Natarajan, Mitra Nasri, David Broman, Björn B. Brandenburg, and Geoffrey Nelissen. From code to weakly hard constraints: A pragmatic end-to-end toolchain for timed C. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 167–180, 2019.

22 Suhail Nogd, Geoffrey Nelissen, Mitra Nasri, and Björn B. Brandenburg. Response-Time Analysis for Non-Preemptive Global Scheduling with FIFO Spin Locks. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 115–127, 2020.

23 John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.

24 Sayra Ranjha, Mitra Nasri, and Geoffrey Nelissen. Work-in-progress: Partial-order reduction in reachability-based response-time analyses. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 544–547, 2021.

25 Sayra Ranjha, Geoffrey Nelissen, and Mitra Nasri. Partial-order reduction for schedule-abstraction-based response-time analyses of non-preemptive tasks. In *IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 121–132, 2022.

26 Pascal Richard, Joël Goossens, and Shinpei Kato. Comments on "Gang EDF Schedulability Analysis", 2017. `arXiv:1705.05798`.

27 Maria A. Serrano, Alessandra Melani, Sebastian Kehr, Marko Bertogna, and Eduardo Quiñones. An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling. In *IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, pages 193–202, 2017.

28 Srinidhi Srinivasan, Geoffrey Nelissen, and Reinder J. Bril. Work-in-progress: Analysis of tsn time-aware shapers using schedule abstraction graphs. In *Real-Time Systems Symposium (RTSS)*, pages 508–511, 2021.

29 Roger Stafford. Random vectors with fixed sum. Technical report, University of Oxford, 2006. URL: `http://www.mathworks.com/matlabcentral/fileexchange/9700`.

30 Saud Wasly and Rodolfo Pellizzoni. Bundled scheduling of parallel real-time tasks. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 130–142, 2019.

31 Beyazit Yalcinkaya, Mitra Nasri, and Björn B. Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*, pages 1222–1227, 2019.

32 Yanyong Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, 2003.