Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling

Nasri, Mitra; Nelissen, Geoffrey; Brandenburg, Björn B.

DOI
10.4230/LIPIcs.ECRTS.2019.21

Publication date
2019

Document Version
Final published version

Published in
31st Euromicro Conference on Real-Time Systems, ECRTS 2019

Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling

Mitra Nasri
Delft University of Technology (TUDelft), Delft, The Netherlands

Geoffrey Nelissen
CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP), Portugal

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Abstract

Most recurrent real-time applications can be modeled as a set of sequential code segments (or blocks) that must be (repeatedly) executed in a specific order. This paper provides a schedulability analysis for such systems modeled as a set of parallel DAG tasks executed under any limited-preemptive global job-level fixed priority scheduling policy. More precisely, we derive response-time bounds for a set of jobs subject to precedence constraints, release jitter, and execution-time uncertainty, which enables support for a wide variety of parallel, limited-preemptive execution models (e.g., periodic DAG tasks, transactional tasks, generalized multi-frame tasks, etc.). Our analysis explores the space of all possible schedules using a powerful new state abstraction and state-pruning technique. An empirical evaluation shows the analysis to identify between 10 to 90 percentage points more schedulable task sets than the state-of-the-art schedulability test for limited-preemptive sporadic DAG tasks. It scales to systems of up to 64 cores with 20 DAG tasks. Moreover, while our analysis is almost as accurate as the state-of-the-art exact schedulability test based on model checking (for sequential non-preemptive tasks), it is three orders of magnitude faster and hence capable of analyzing task sets with more than 60 tasks on 8 cores in a few seconds.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

Keywords and phrases parallel DAG tasks, global multiprocessor scheduling, schedulability analysis, non-preemptive jobs, precedence constraints, worst-case response time, OpenMP

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.21

Supplement Material The source code of the analysis tool is available at `https://github.com/brandenburg/np-schedulability-analysis`.

Funding This work was partially supported by national funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UID/CEC/04234); by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement, through the European Regional Development Fund (ERDF), and by national funds through the FCT, within project POCI-01-0145-FEDER-029119 (PReFECT); as well as by the European Union through the Clean Sky 2 Joint Undertaking, under H2020 (H2020-CS2-CFP08-2018-01) grant agreement number 832011 (THERMAC).

1 Introduction

With the proliferation of multicore and many-core processing platforms, the embedded systems world is steadily moving towards developing critical applications as (highly) parallel programs. In embedded real-time systems in particular, parallel programming approaches allow for more efficient use of a computing platform's resources, resulting in lower response times and improved power consumption. For instance, the automotive industry adopted

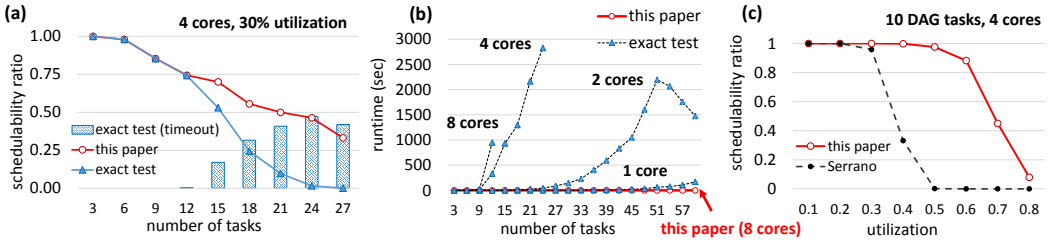

Figure 1 **(a)** schedulability ratio of the exact test [46] vs. our test for independent non-preemptive periodic tasks, **(b)** average runtime of the exact test [46] vs. our test for *schedulable* independent non-preemptive periodic task sets with total utilization of 30%, **(c)** schedulability ratio of Serrano et al.'s test [39] vs our test for DAG tasks. In inset (a), given a one-hour budget, starting from 12 tasks the *exact* test reports *lower* schedulability than the proposed test due to frequent timeouts. See Sec. 5 for a detailed description of the experimental setup.

multicore processors already more than six years ago, and their applications are routinely composed of thousands of *runnables* executing in parallel [24]. Such runnables are sequential code segments that perform simple operations, which are composed to produce complex applications by imposing precedence constraints that must be respected at runtime (to enforce a predictable ordering and to respect data dependencies). Because of the application domain, additional constraints on release and completion times are also associated with runnables to ensure temporal correctness, control performance, ease of synchronization and, in consequence, ease of integration of concurrent applications on multicore platforms.

Similarly to the automotive case, a wide variety of industrially relevant systems boil down to the execution of a set of functions with precedence constraints where a function is simply a *sequential* execution segment of a bigger, potentially *parallel*, application. Such applications (henceforth called *tasks*) may be modeled with *Directed Acyclic Graphs* (DAGs). Nodes of a DAG represent sequential code segments, and edges model their precedence constraints. Each application represented by a DAG releases *jobs* based on timer events or inputs regularly received from the environment following periodic or non-periodic activation patterns (e.g., multi-frame or multi-rate tasks [16, 6, 20]). Robotics applications executed upon the ROS middleware, machine learning algorithms developed with the TensorFlow or Pytorch frameworks, or applications developed with OpenMP are other notable examples of systems that are often time-driven and that may be naturally modeled with DAGs. To summarize, parallel DAG tasks are the characteristic real-time workload of the multicore age and thus of central interest for schedulability analysis.

In this work, we consider a limited-preemptive task model, where nodes of a DAG must execute non-preemptively, but higher-priority workload may preempt the execution of a DAG between the execution of any two of its nodes. This execution model is motivated by many previous studies [11, 34, 2, 29, 37, 39] that have shown that non-preemptive (or limited-preemptive) scheduling improves the timing predictability of jobs running on a multicore platform, since it reduces the number of context switches, increases cache predictability [45], and improves the accuracy of *worst-case execution time* (WCET) estimates and worst-case blocking bounds (e.g., due to contention for shared resources).

Two types of frameworks exist for the schedulability analysis of such systems today. Exact solutions based on model checkers or constraint programming [46, 42], and sufficient (but inexact) solutions usually based on some sort of response-time analysis [39, 15, 13, 14].

It has been demonstrated that exact analyses based on constrained programming or model checkers such as Uppaal do not scale well [46, 42]. For example, Figure 1(b) shows the time required to deem a simple non-preemptive periodic task set schedulable using Uppaal

as a function of the number of tasks for several different core counts on a 3.3 GHz Intel Xeon machine with 256 GiB RAM. Even for such a simple model without intra-task parallelism or precedence constraints, Uppaal requires an average of 45 minutes to analyze 24 tasks on 4 cores used at only 30% of their capacity (i.e., the total platform utilization is 30%), with nearly 50% of the tested workloads timing out after 1 hour (see Figure 1(a)). Worse, it cannot solve the problem at all in less than one hour for 12 tasks (or more) on 8 cores with a total utilization of only 30%. Clearly, such a solution can realistically be used only for very small systems, which limits practicality.

On the other hand, classic sufficient schedulability analyses following the standard response-time analysis paradigms are usually fast but very pessimistic. For instance, as seen in Figure 1(c), the only sufficient test existing for the schedulability of limited-preemptive DAGs scheduled by a global fixed-priority scheduler (proposed by Serrano et al. [39]) cannot detect that *any* of the generated task sets with a total utilization larger than 50% (4 cores, 10 DAG tasks) is schedulable, when in fact at least 90% of them are. This pessimism reaches a level that calls into question the utility of such tests in industrial settings.

In this paper, we propose a new approach for the schedulability analysis of *limited-preemptive DAG tasks* that presents a more balanced tradeoff between runtime and accuracy. Case in point, w.r.t. analysis speed, in the scenario shown in Figure 1(a), our solution solves the schedulability problem of non-preemptive tasks *almost* optimally (empirically, almost all schedulable workloads are in fact deemed schedulable) in less than 10 seconds on average, while Uppaal needed tens of minutes to reach the same conclusion (and frequently exceeded the one-hour timeout). Furthermore, w.r.t. analysis accuracy for DAG tasks, the proposed analysis clearly increases the number of workloads successfully detected as being schedulable in comparison to the solution of Serrano et al. by a substantial margin (see Figure 1(c)).

The analysis presented in this paper covers any *global job-level fixed-priority* (JLFP) scheduler (e.g., global limited-preemptive earliest-deadline first (G-LP-EDF) or fixed-priority (G-LP-FP) scheduling). Specifically, each node of each DAG instance released by a task can have a distinct priority, a distinct release time, and is assumed to execute non-preemptively. We allow for the practical, but analytically challenging complication that each node may experience release jitter and execution-time uncertainty, which in combination with non-preemptivity results in scheduling anomalies that are notoriously difficult to analyze precisely.

To strike a good balance between accuracy and runtime, our analysis constructs a *schedule-abstraction graph* that abstracts all possible orderings of job dispatch times resulting from the underlying JLFP scheduling policy, based on which we derive bounds on the best- and worst-case response time of each job. This approach requires: **(i)** a *system-state abstraction* that represents the state in which the system may be after a given sequence of scheduling events, **(ii)** sound *exploration rules* that reflect how new system states may be reached from a given state, and **(iii)** *merging rules* for the aggregation of similar states to defer, as long as possible, the usual state-space explosion problem.

As a key technical contribution, this paper introduces a *new system-state abstraction* in which the number of newly created states at the end of each exploration step is *independent of the number of cores*, which ensures scalability to large multicore platforms. Furthermore, our new abstraction also allows for aggressive merging rules, and hence greatly reduces the number of system states that must be investigated to cover all relevant job schedules. Based on this novel technique, **(i)** we devise a schedule-abstraction graph generation algorithm that considers the precedence constraints of DAG tasks and ensures a small per-state memory footprint and low per-state computational costs, **(ii)** we prove the system state-space exploration and merging rules to be sound, and **(iii)** we report results on extensive

experiments involving both synthetic DAGs and actual DAGs from parallel benchmark applications. The experiments show the proposed method to scale to systems with up to 64 cores, to be able to identify up to 90 percentage points more schedulable task sets in comparison to the state-of-the-art response-time analysis for limited-preemptive sporadic DAG tasks [39], and to be three orders of magnitude faster than model-checking approaches [46].

2 Related Work

The schedulability analysis of a set of independent non-preemptive *sporadic* tasks scheduled by a global scheduling policy such as G-LP-EDF or G-LP-FP has been studied in several works [5, 19, 23, 22, 11]. These analyses, however, do not account for release jitter and become needlessly pessimistic when applied to *periodic* tasks or jobs with regular, yet not necessarily periodic, activation patterns [33] as they fail to discount many execution scenarios that are impossible in such systems commonly found in industry.

In response to the need for supporting task models with more complicated job activation patterns, Stigge et al. [41] and Abdullah et al. [1] provided schedulability analyses for non-preemptive digraphs and digraphs with a mixed set of preemptive and non-preemptive nodes, respectively. The digraph model was later extended to support a rendezvous synchronization mechanism [31]. However, to the best of our knowledge, there is no result yet on digraphs with non-preemptive nodes and complex inter-task precedence constraints.

To work around the lack of a schedulability test for non-preemptive DAGs, Saifullah et al. [37] provided solutions to convert a DAG to a set of independent jobs whose arrival times and deadlines are assigned in a way that respects the DAG's given precedence constraints. This job set is then converted to an equivalent periodic task set and evaluated using Baruah's [5] or Guan's [19] schedulability analyses for independent, non-preemptive tasks. This approach, however, suffers from the pessimism inherent in the *decomposition* step, i.e., regardless of the accuracy of the underlying schedulability tests, many schedulable DAG tasks will be deemed unschedulable simply because the decomposition technique may not be able to find feasible parameters for the decomposed independent tasks.

Liu and Anderson extensively studied sporadic processing pipelines and DAGs under global scheduling in a soft real-time context [25, 26, 27, 28], showing that deadline tardiness remains bounded as long as the system is not overloaded (i.e., DAG instances may miss deadlines, but are guaranteed to complete within an *a priori* fixed interval after their deadline). In contrast to Liu and Anderson's focus on establishing (non-tight) tardiness bounds, our goal is to determine as accurate as possible response-time bounds given (possibly) hard deadlines.

Serrano et al. [39] proposed an analysis for limited-preemptive DAG tasks. This is the closest work to our problem as it explicitly considers precedence constraints and limited-preemptive global scheduling at the same time. Our work improves upon this result by: **(i)** providing a much more accurate analysis for periodic DAGs and other types of tasks with regular, yet non-periodic release patterns, **(ii)** including all JLFP global scheduling policies in one uniform analysis framework, and **(iii)** supporting inter-task dependencies (rather than only precedence constraints within individual DAG tasks).

Several works have proposed exact analyses for global *preemptive* sporadic tasks *without* precedence constraints [4, 8, 9, 18, 43]. These analyses generally explore all system states that can possibly be reached using model checking, timed automata, or linear-hybrid automata. These solutions, however, are limited to the preemptive execution model and have limited scalability w.r.t. the number of tasks, processors, and the granularity of time. For instance, the analysis of Sun et al. [43] is reported to be limited to 7 tasks and 4 cores, and Guan et al.'s approach [18] is applicable only if task periods are integers in the range from 8 to 20.

In our own prior work [33], we considered the schedulablity analysis of a set of *independent* (i.e., non-DAG), non-preemptive sequential jobs scheduled with a global JLFP scheduling policy. While this paper superficially resembles [33] in that it uses a similar general approach – namely, the generation of a schedule-abstraction graph [32] – it actually follows a substantially different design needed to support limited-preemptive parallel DAG tasks. Specifically, in order to scale to non-trivial DAG tasks, the system state abstraction, exploration rules, and merge rules presented in this paper are entirely novel, and in fact even incomparable, to those previously used in [33]. Case in point, extensive experiments (see Sec. 5) revealed that the solution presented in this paper is up to two orders-of-magnitude faster than [33] when non-preemptive sequential tasks are analyzed, which reflects the nontrivial scalability advantages of the novel approach introduced in this paper.

3 System Model and Definitions

We consider the problem of globally scheduling a set of limited-preemptive parallel tasks with known arrival patterns upon a multiprocessor platform composed of m unit-speed processors. Each task is modeled by a DAG (V, E), where V is the set of execution segments, and E is the set of precedence constraints between execution segments in V. Each execution segment $v_j \in V$ has an execution time, and may (or may not) be assigned a relative release offset and relative deadline with respect to the arrival time of the task. For each arrival of a task, every execution segment in V releases a *job*. Even though we assume that tasks have known arrival patterns, we allow their execution segments, and hence their jobs, to be subject to release jitter. Similarly, the exact execution time of each job is *a priori* unknown. In addition, we allow precedence constraints to be specified among execution segments of different DAGs, thereby allowing for arbitrary inter-task precedence constraints.

As the arrival pattern of each task is known, our problem reduces to the analysis of a finite set of non-preemptive jobs $\mathcal{J}$ on an observation window whose length can be computed *a priori*. For periodic tasks with constrained deadlines, release jitter and synchronous releases, the observation window is equal to one hyperperiod (i.e., the least common multiple of all periods). Bounds on the observation window length for periodic tasks with release offsets, precedence constraints, and arbitrary deadlines were established by Goossens et al. [17].

Each job $J_i = ([r_i^{min}, r_i^{max}], [C_i^{min}, C_i^{max}], d_i, p_i, pred_i))$ released in the observation window has an *earliest-release time* r_i^{min}, a *latest-release time* r_i^{max}, a *best-case execution time* (BCET) C_i^{min}, a WCET C_i^{max}, an *absolute* deadline d_i, a priority p_i, and a set of predecessors $pred_i \subset \mathcal{J}$, i.e., a set of jobs that must complete before J_i may start executing. The set of successors of a job J_i is denoted by $succ_i = \{J_x \mid J_i \in pred_x\}$.

Each job is assigned a priority by a given job-level fixed-priority (JLFP) scheduling policy. We assume that a numerically smaller value of p_i implies higher priority. Any ties in priority are broken arbitrarily in a deterministic way. For ease of notation, we assume that the "$<$" operator implicitly reflects this tie-breaking rule. We assume a discrete-time model, i.e., all job timing parameters are integer multiples of a basic time unit such as a processor cycle.

At runtime, each job is *released* at an *a priori* unknown time $r_i \in [r_i^{min}, r_i^{max}]$. The release bounds r_i^{min} and r_i^{max} are computed based on the arrival pattern (e.g., periodic, multi-rate, or bursty) of J_i's task, its offset, and its release jitter. We also assume that each job J_i has an *a priori* unknown execution time requirement $C_i \in [C_i^{min}, C_i^{max}]$. We assume that a job's absolute deadline d_i is fixed and not affected by release jitter. We say that a job J_i is *possibly released* at time t if $t \geq r_i^{min}$, and *certainly released* if $t \geq r_i^{max}$.

Any two jobs that are neither directly nor indirectly predecessor/successor of each other are said to be independent. Independent jobs may execute in parallel. Each individual job must execute sequentially, i.e., it cannot execute on more than one core at a time and must

run to completion once started. A job J_i that starts its execution on a core at time t occupies that core during the interval $[t, t + C_i)$. In this case, we say that job J_i *finishes by* time $t + C_i$. At time $t + C_i$, the core used by J_i becomes available to start executing other jobs. A job's *response time* is defined as the difference between the earliest-release time and the actual completion time of the job[1], i.e., $t + C_i - r_i^{min}$. We say that a job is *ready* at time t if it is released, did not start its execution before time t, and all of its predecessors have finished by time t. Further, we assume that the system does not have a job-discarding policy, i.e., released jobs remain pending until their execution is finished.

The paper assumes that shared resources that must be accessed in mutual exclusion are protected by FIFO spin locks. Since jobs execute non-preemptively, it is easy to obtain a bound on the worst-case time that any job spends spinning while waiting to acquire a contested lock [44]; we assume the worst-case spin delay is included in each job's WCET.

For ease of notation, we use $\max_0\{X\}$ and $\min_\infty\{X\}$ over a set of positive values $X \subseteq \mathbb{N}$ that is completed by 0 and ∞, respectively. That is, if $X = \emptyset$, then $\max_0\{X\} = 0$ and $\min_\infty\{X\} = \infty$. Otherwise they yield the usual maximum and minimum values in X.

We consider any non-preemptive global JLFP scheduler upon an identical multiprocessor platform. The scheduler is invoked whenever a job is released or completed. To simplify the presentation of the proposed analysis, we make the modeling assumption that, without loss of generality, at any invocation of the scheduling algorithm, at most one of the pending jobs is picked by the scheduler and assigned to a core. The scheduler is invoked once for each event if two or more release or completion events occur at the same time. The actual scheduler implementation in the analyzed system need not adhere to this restriction and may process more than one event during a single invocation. Our analysis remains safe if the assumption is relaxed in this manner.

In this paper, we exclusively focus on priority-driven and work-conserving scheduling algorithms, i.e., the scheduler dispatches a job only if the job has the highest priority among all ready jobs, and it does not leave a core idle if there exists a ready job. We assume that the WCET of each job is padded to cover all scheduling overheads and to account for any micro-architectural interference (e.g., competition for shared caches or memory bandwidth).

A job set $\mathcal{J}$ is *schedulable* under a given scheduling policy if no execution scenario of $\mathcal{J}$ results in a deadline miss, where an execution scenario is defined as follows [32].

▶ **Definition 1.** An *execution scenario* $\gamma = \{(r_1, C_1), (r_2, C_2), \ldots, (r_n, C_n)\}$, where $n = |\mathcal{J}|$, is an assignment of execution times and release times to the jobs of $\mathcal{J}$ such that, for each job J_i, $C_i \in [C_i^{min}, C_i^{max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$.

4 Schedulability Analysis

The schedulability analysis proceeds by exploring the space of all possible schedules using the notion of a *schedule-abstraction graph* [32]. Each path in this graph reflects a sequence of job-dispatch decisions made by the underlying scheduling policy. As discussed in Sec. 2, a key innovation of this paper is a new system-state abstraction that more richly aggregates the necessary information in each state and, ultimately, reduces the number of edges in the final graph. After introducing the new abstraction (Sec. 4.2), we explain how to build the graph (Sec. 4.3), define exploration rules for work-conserving global JLFP scheduling policies (Sec. 4.4), describe how to soundly construct a new state (Sec. 4.5), and finally show how to merge similar states to reduce the size of the graph (Sec. 4.6). A proof of correctness of the analysis is presented in Sec. 4.7.

[1] Any release jitter is counted as part of the job's response time, as introduced by Audsley et al. [3].

4.1 Job Finish Times and System-Availability Intervals

Because jobs experience release jitter and execution time variation, exponentially many execution scenarios exist, and the exact finishing time of each job cannot be known a priori. Therefore, we compute an interval $[EFT_i, LFT_i]$ in which a job J_i will finish after a given sequence of scheduling decisions taken by the scheduler. This interval is lower-bounded by J_i's *earliest finish time* EFT_i and upper-bounded by its *latest finish time* LFT_i, that is, J_i may possibly finish at or after EFT_i and is certainly finished at LFT_i. A key challenge is that this uncertainty in job finish times introduces uncertainty in processor availability, which in turn affects the finish-time intervals of subsequently scheduled jobs.

To address this challenge, in our new abstraction, a state represents the state of the system after a possible sequence of scheduling decisions (corresponding to a subset of execution scenarios) by indicating when one, two, three, ..., m cores will *possibly* and *certainly* become available. Namely, each state includes a set of *system-availability intervals*, denoted $A = \{A_1, A_2, \ldots, A_m\}$, where $A_x = [A_x^{min}, A_x^{max}]$ means that x cores are *possibly available* (PA) starting at time A_x^{min} and *certainly available* (CA) no later than at time A_x^{max}.

▶ **Example 1.** Consider a system with $m = 3$ cores and suppose that three jobs are scheduled, with the following finish-time intervals: $[10, 45]$, $[30, 40]$, and $[15, 25]$. In this example, one core becomes possibly available at time 10. Two cores can possibly be available from time 15 onward. Similarly, one core becomes certainly available at time 25, and two cores become certainly available at time 40. Thus, $A_1 = [10, 25]$, $A_2 = [15, 40]$, and $A_3 = [30, 45]$.

4.2 Graph Definition

We define the schedule-abstraction graph as a directed-acyclic graph $G = (V, E)$, where V is a set of system states and E is the set of labeled edges. An edge $e \in E$ is defined as $e = (v_p, v_q, J_i)$, where v_p and v_q are the source and destination vertices of the edge, and the label J_i is the job that, by being scheduled, evolves state v_p to state v_q. We say job J_i is dispatched *next* after v_p or *succeeds* v_p if it is on an outgoing edge from a state v_p.

A path P from the initial state v_1 to a state v_p represents a possible sequence of job-dispatching events (or scheduling decisions) that lead to state v_p from the initial state v_1, which represents the initial idle system at time zero before any job is scheduled. The length of a path refers to the number of jobs scheduled on that path, i.e., $|P| \triangleq |\mathcal{J}^P|$, where $\mathcal{J}^P$ is the set of jobs that appear as labels on the edges of path P.

In graph G, it is possible to have more than one incoming edge to a vertex v_p. However, in that case, the following property must hold for any two paths that connect v_1 to v_p.

▶ **Property 1.** *For any two arbitrary paths P and Q that connect v_1 to v_p, $\mathcal{J}^P = \mathcal{J}^Q$.*

Having defined edges and paths, we next define a system state $v \in V$ as a three-tuple that contains: **(i)** the set of m system-availability intervals as defined in Sec. 4.1, denoted $A(v)$, **(ii)** a set $\mathcal{X}(v)$ of jobs that are *certainly executing* on the platform in state v, and **(iii)** a set of finish-time intervals $\{[EFT_x(v), LFT_x(v)] \mid J_x \in \mathcal{X}(v)\}$, where $EFT_x(v)$ and $LFT_x(v)$ represent the time at which job J_x is possibly and certainly finished considering the sequence of job-dispatch events that led to state v.

The motivation for including the set of certainly running jobs $\mathcal{X}(v)$ is that, given precedence constraints, the ready time of a job depends on the completion time of its predecessors. This creates a challenge as storing the EFT and LFT of *every* job on *every* path would require an exponentially increasing amount of memory w.r.t. the number of jobs scheduled. As a tradeoff, to improve the accuracy of the analysis, we maintain the set of

Algorithm 1: Schedule Graph Construction Algorithm.

Input : Job set $\mathcal{J}$
Output : Schedule graph $G = (V, E)$

1 $\forall J_i \in \mathcal{J}, BR_i \leftarrow \infty, WR_i \leftarrow 0$;
2 Initialize G by adding $v_1 = \big(\{[0,0], \ldots, [0,0]\}, \mathcal{X} = \emptyset, \emptyset\big)$;
3 **while** $\exists$ *path* P *from* v_1 *to a leaf vertex s.th.* $|P| < |\mathcal{J}|$ **do**
4 $P \leftarrow$ the shortest path from v_1 to a leaf vertex v_p;
5 $\mathcal{R}^P \leftarrow$ set of ready jobs obtained with Eq. (1);
6 **for** *each job* $J_i \in \mathcal{R}^P$ **do**
7 **if** J_i *can be dispatched after* v_p *according to Eq.* (9) **then**
8 Build v'_p using Algorithm 2;
9 $BR_i \leftarrow \min\{EFT_i(v'_p) - r_i^{min}, BR_i\}$;
10 $WR_i \leftarrow \max\{LFT_i(v'_p) - r_i^{min}, WR_i\}$;
11 Connect v_p to v'_p by an edge with label J_i;
12 **while** $\exists$ *path* Q *that ends to* v_q *such that Rule 1 is satisfied for* v'_p *and* v_q **do**
13 Merge v'_p and v_q by updating v'_p using Eq. (15);
14 Redirect all incoming edges of v_q to v'_p;
15 Remove v_q from V;
16 **end**
17 **end**
18 **end**
19 **end**

certainly running jobs $\mathcal{X}(v)$ and their finishing time intervals in each system state v. Since there are at most m such jobs per state, the amount of memory required per state remains constant. This property of the algorithm is discussed in detail in Sec. 4.4.

4.3 Graph-Generation Algorithm

We next introduce the main state-space exploration algorithm for finding the schedule-abstraction graph for a given workload and platform. We first provide an informal high-level overview, and then present the algorithm more precisely as pseudocode in Algorithm 1.

The schedule-abstraction graph is built iteratively in two alternating phases: *expansion* and *merging*. The expansion phase, expands (one of) the shortest path(s) P in the graph by considering all jobs that can possibly be dispatched *next* in the job-dispatch sequence represented by P. For each such job J_i, a new vertex v'_p is created and added to the graph via a directed edge from v_p to v'_p. The new state v'_p is generated from v_p by updating the core availability intervals and the set of certainly running jobs (and their finish-time intervals) when the execution of J_i is considered.

The merge phase slows down the growth of the graph by merging, whenever possible, the terminal vertices of paths that have the same set of dispatched jobs. As a key soundness condition, the merge phase guarantees that any possible execution scenario that can be generated from two un-merged states v_p and v_q can still be generated after they are merged.

The search ends when there is no vertex left to expand, that is, when all paths represent a valid schedule of all jobs in $\mathcal{J}$, which implies that all possible schedules have been explored.

Algorithm 1 presents our iterative breadth-first method for generating the schedule-abstraction graph in full detail. A set of variables keeping track of the smallest and largest response times (BR_i and WR_i, respectively) observed for each job in all execution scenarios explored so far is initialized in line 1; these bounds are updated whenever a job J_i can possibly be dispatched on a core (lines 9 and 10). The graph is initialized in line 2 with a

root vertex v_1 that represents m idle cores at time 0. The expansion phase corresponds to lines 6–18 and lines 12–16 implement the merge phase. These phases repeat until every path in the graph contains $|\mathcal{J}|$ distinct jobs (line 3). We next discuss each phase in detail.

4.4 Expansion Phase

In this section, we explain how to expand a path P ending in v_p, as found in line 4 in Algorithm 1, by dispatching an eligible job after the scheduling sequence represented by P.

Overview

The expansion phase starts by obtaining the set of potentially *ready jobs* for system state v_p, i.e., jobs whose predecessors have been dispatched previously on path P.

For each ready job J_i, we calculate the earliest and latest time at which J_i can be dispatched on the platform after state v_p. These times are called the *earliest start time* (EST) and the *latest start time* (LST) of the job, denoted by $EST_i(v_p)$ and $LST_i(v_p)$, respectively.

If the earliest time at which the job can potentially start executing, i.e., $EST_i(v_p)$, is earlier than the latest time at which a work-conserving JLFP scheduler would allow that job to start if it is to be the next scheduled job, i.e., $LST_i(v_p)$, then the job is *eligible* to be dispatched after state v_p. For each eligible job, a new state v'_p is created and appended to path P after state v_p.

We next explain in detail, and precisely define, each step of the expansion phase.

Ready Interval

As stated in Sec. 3, a job is ready only if it is released and all of its predecessors have been completed. Thus, potentially ready jobs for path P are those that are not yet dispatched and all of their predecessors are in $\mathcal{J}^P$, i.e.,

$$\mathcal{R}^P \triangleq \{J_i \mid J_i \in \mathcal{J} \setminus \mathcal{J}^P \ \wedge \ pred(J_i) \subseteq \mathcal{J}^P\}. \tag{1}$$

Since each job J_i may suffer release jitter and because the exact finish times of J_i's predecessors are not known, the exact time at which J_i becomes ready is also unknown. For that reason, we compute a lower bound on the time at which a job $J_i \in \mathcal{R}^P$ is *possibly ready*, denoted R_i^{min}, and an upper bound on the time at which J_i is *certainly ready*, denoted R_i^{max}. Since a job can start its execution only if **(i)** it is released, and **(ii)** all its predecessors have completed, R_i^{min} is the minimum of r_i^{min} and the earliest time at which all predecessors of J_i have possibly completed, and R_i^{max} is the maximum of r_i^{max} and the time at which all predecessors of J_i have certainly completed, i.e.,

$$R_i^{min} \triangleq \max\left\{r_i^{min}, \max_0\{EFT_x^*(v_p) \mid J_x \in pred(J_i)\}\right\}, \text{ and} \tag{2}$$

$$R_i^{max} \triangleq \max\left\{r_i^{max}, \max_0\{LFT_x^*(v_p) \mid J_x \in pred(J_i)\}\right\}, \tag{3}$$

where $EFT_x^*(v_p)$ and $LFT_x^*(v_p)$ are safe bounds (defined next) on the earliest and latest finish time of J_x for all execution scenarios that lead to v_p. The use of $\max_0$ in Eqs. (2) and (3) ensures that the ready interval of jobs with no precedence constraint is equal to their release jitter interval, i.e., $R_i^{min} = r_i^{min}$ and $R_i^{max} = r_i^{max}$ if J_i does not have predecessors.

For the predecessors of J_i that are *certainly running* in system state v_p, i.e., any job $J_x \in \mathcal{X}(v_p) \cap pred(J_i)$, the bounds $EFT_x^*(v_p)$ and $LFT_x^*(v_p)$ can safely assume the values $EFT_x(v_p)$ and $LFT_x(v_p)$ saved in state v_p. However, for predecessors of J_i that are not certainly running in state v_p, i.e., any job J_x that is not in $\mathcal{X}(v_p)$, there is no bound on

$EFT_x(v_p)$ and $LFT_x(v_p)$ saved in v_p (which, to recall, is an intentional space optimization). Therefore, we instead use the current values of BR_x and WR_x (see Algorithm 1) as they are safe bounds on the EFT and LFT of J_x for all system states explored up to this point (lines 9 and 10 of Algorithm 1), which also includes v_p.

To summarize, if a job J_x belongs to $\mathcal{X}(v_p)$, then EFT_x^* and LFT_x^* are equal to $EFT_x(v_p)$ and $LFT_x(v_p)$, respectively. Otherwise, they are equal to BR_x and WR_x, respectively.

Earliest and Latest Start Times

Consider a job $J_i \in \mathcal{R}^P$, i.e., all the precedence constraints of J_i are respected. Job J_i cannot start executing prior to the earliest time at which it may become ready, i.e., R_i^{min}, nor can it start executing before the earliest time at which a core may become available, which is given by A_1^{min}. Thus, the earliest time at which J_i can start its execution after path P is given by

$$EST_i = \max\{R_i^{min}, A_1^{min}\}. \tag{4}$$

The latest start time of J_i after path P is decided by two factors: **(i)** the scheduler follows a JLFP scheduling policy, and **(ii)** the scheduler is work-conserving.

Considering factor (i), since a JLFP scheduling policy always dispatches the highest-priority ready job, the latest start time of J_i is upper-bounded by $t_{high} - 1$, where t_{high} is the earliest point in time from which on J_i certainly is not the highest-priority ready job anymore. An upper bound on t_{high} is given by Eq. (5) as proven in Lemma 2.

$$t_{high} \triangleq \min_{\infty}\{th_x(J_i) \mid J_x \in \mathcal{R}^P \ \wedge \ p_x < p_i\}, \text{ where} \tag{5}$$

$$th_x(J_i) \triangleq \max\left\{r_x^{max}, \max_{0}\{LFT_y^*(v_p) \mid J_y \in pred(J_x) \setminus pred(J_i)\}\right\}. \tag{6}$$

▶ **Lemma 2.** *Job J_i will not be the highest-priority ready job in $\mathcal{R}^P$ for system state v_p at any time later than $t_{high} - 1$.*

Proof. Suppose that $t_{high} \neq \infty$ (otherwise the claim is trivially true as it does not actually constrain J_i). Let $J_x \in \mathcal{R}^P$ be the job with higher priority than J_i such that $th_x(J_i) = t_{high}$.

At time $th_x(J_i)$, job J_x is certainly released (since according to Eq. (6), $th_x(J_i) \geq r_x^{max}$) and all predecessors of J_x that are not predecessors of J_i have been certainly completed (since $\forall J_y \in pred(J_x) \setminus pred(J_i)$, $th_x(J_i) \geq LFT_y^*(v_p)$ according to Eq. (6)). If $pred(J_x) \cap pred(J_i) = \emptyset$, then according to Eq. (3), J_x is certainly ready at $th_x(J_i)$ and J_i cannot be the highest-priority ready job from $th_x(J_i)$ onward.

If $pred(J_x) \cap pred(J_i) \neq \emptyset$, then, at the first point in time $t \geq th_x(J_i)$ such that all precedence constraints of J_i are respected, all precedence constraints of J_x are also respected (recall that the precedence constraints of J_x that are not common with J_i were already respected before or at time $th_x(J_i)$). In other words, if J_i becomes ready at or after $th_x(J_i)$ then J_x also becomes ready and J_i is not the highest-priority ready job. ◀

Additionally, considering factor (ii), if there is a time where a core is certainly available (which is the case from time A_1^{max} onward), and a job is certainly ready, a work-conserving scheduler must dispatch the job at that time, which is denoted t_{wc} and obtained as follows.

$$t_{wc} \triangleq \max\left\{A_1^{max}, \ \min_{\infty}\{R_x^{max} \mid J_x \in \mathcal{R}^P\}\right\} \tag{7}$$

▶ **Lemma 3.** *Job $J_i \in \mathcal{R}^P$ will not be dispatched next after v_p at any time later than t_{wc}.*

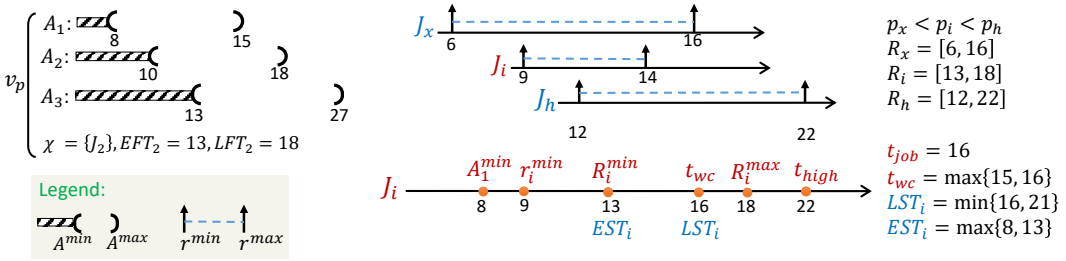

Figure 2 Calculating EST_i and LST_i for a successor job J_i of a certainly running job J_2.

Proof. Assume that $t_{wc} \neq \infty$; otherwise the claim is trivial. At time t_{wc}, a not-yet-dispatched job J_x whose precedence constraints are satisfied is certainly ready (because $t_{wc} \geq \min_\infty\{R_x^{max} \mid J_x \in \mathcal{R}^P\}$), and a core is certainly available (because $t_{wc} \geq A_1^{max}$). Hence, a work-conserving scheduler will dispatch J_x at t_{wc}. Consequently, J_i will be a direct successor of state v_p *only if* it starts no later than t_{wc}. ◀

Combining the facts that $LST_i \leq t_{high} - 1$ (Lemma 2) and $LST_i \leq t_{wc}$ (Lemma 3), we observe that J_i may be the next job scheduled after path P only if it starts no later than

$$LST_i = \min\{t_{wc},\ t_{high} - 1\}. \tag{8}$$

▶ **Example 4.** Figure 2 shows how EST_i and LST_i are calculated for J_i. The earliest time at which one core becomes ready is 8, and J_i is released at the earliest at time $r_i^{min} = 9$. However, since J_i must wait for its predecessor J_2 to finish before it becomes ready, we have $EST_i = 13$, which is the earliest finish time of J_2. Since J_x is certainly ready at time 16, and since at least one core is certainly available from time 15 onward, the latest time at which job J_i can be dispatched next after v_p is 16; otherwise, a work-conserving scheduler would schedule J_x after v_p instead. In this example, t_{high} is 22, where a higher priority job J_h is certainly released. However, since $t_{wc} < t_{high} - 1$, the LST_i is bounded by $t_{wc} = 16$.

Eligibility Condition

A job $J_i \in \mathcal{R}^P$ can be dispatched next after path P if its earliest start time EST_i is not later than its latest start time LST_i, i.e., if

$$EST_i \leq LST_i. \tag{9}$$

▶ **Lemma 5.** *Job J_i is a direct successor of v_p only if Inequality (9) holds.*

Proof. According to Lemmas 2 and 3, LST_i is an upper bound on the time at which J_i can be dispatched after v_p. Therefore, if J_i cannot be dispatched by LST_i, then it cannot be a direct successor of v_p. Since EST_i is the earliest time at which J_i can be dispatched after v_p, if $EST_i > LST_i$, J_i cannot be a direct successor of v_p. ◀

If a job J_i is dispatched next after v_p, its earliest and latest finish times are trivially

$$EFT_i = EST_i + C_i^{min} \text{ and} \tag{10}$$
$$LFT_i = LST_i + C_i^{max}. \tag{11}$$

Algorithm 2: Create a new state v'_p by dispatching job J_i after state v_p.

```
1 Initialize PA and CA using Eqs. (12) and (13);
2 for each J_x ∈ X(v_p) ∩ pred(J_i) do
3 |   if LST_i < LFT_x(v_p) ∧ LFT_x(v_p) ∈ CA then
4 |   |   replace LFT_x(v_p) with LST_i in CA;
5 |   end
6 end
7 Sort PA and CA in non-decreasing order;
8 ∀x, 1 ≤ x ≤ m, A_x(v'_p) ← [PA_x, CA_x];
9 X(v'_p) is obtained from Eq. (14);
```

4.5 Creating a New State

If job $J_i \in \mathcal{R}^P$ satisfies Inequality (9), it can be dispatched next after v_p and a new system state v'_p is created to reflect this possibility. Algorithm 2 presents the procedure for creating a new state v'_p for job J_i. Line 1 creates two lists called PA and CA that contain bounds on the instants at which each core becomes possibly and certainly available after dispatching job J_i, respectively. Those lists are built using the following two lemmas.

▶ **Lemma 6.** *Lower bounds (respectively, upper bounds) on the instants at which each core becomes possibly (respectively, certainly) available after dispatching job J_i in system state v_p are given by PA (respectively, CA) defined as follows.*

$$PA \triangleq \left\{ \max\{EST_i, A_x^{min}(v_p)\} \mid 2 \leq x \leq m \right\} \cup \{EFT_i\} \tag{12}$$

$$CA \triangleq \left\{ \max\{EST_i, A_x^{max}(v_p)\} \mid 2 \leq x \leq m \right\} \cup \{LFT_i\} \tag{13}$$

Proof. We rely on the following four facts:

Fact 1. Since J_i is the first job starting to execute after system state v_p is reached, and because J_i's earliest start time is $EST_i(v_p)$, either all cores are busy until $EST_i(v_p)$, or no other job is released until $EST_i(v_p)$. In either case, after J_i is dispatched and the new system state v'_p is reached, none of the cores start executing another job before $EST_i(v_p)$. Therefore, for each core, its earliest and latest availability times for jobs other than J_i in the new state v'_p are no smaller than $EST_i(v_p)$.

Fact 2. At most x cores are available in the interval $[A_x^{min}(v_p), A_{x+1}^{min}(v_p))$ for $1 \leq x < m$, and no core is available for J_i to execute prior to $A_1^{min}(v_p)$ (by definition of $A_x^{min}(v_p)$). Therefore, each instant $A_x^{min}(v_p)$ is a lower bound on the availability time of a different core.

Fact 3. x cores are certainly available in the interval $[A_x^{max}(v_p), A_{x+1}^{max}(v_p))$ for $1 \leq x < m$, and all cores are certainly available after $A_m^{max}(v_p)$, by definition of $A_x^{max}(v_p)$. Each instant $A_x^{max}(v_p)$ is thus an upper bound on the availability time of a different core.

Fact 4. When J_i starts executing, it starts on the *first* available core (whichever physical core it is), and will occupy it until its finish time.

From Facts 1 and 2, the availability times of the cores in the new state v'_p are lower-bounded by $\{\max\{EST_i, A_x^{min}(v_p)\} \mid 1 \leq x \leq m\}$. Furthermore, from Facts 2 and 4, J_i starts its execution at the earliest at time $A_1^{min}(v_p)$ and keeps the core that was potentially available at $A_1^{min}(v_p)$ *certainly busy* until $EFT_i(v_p)$. Equivalently, that core will be possibly available at the earliest at $EFT_i(v_p)$ in the new system state v'_p. Therefore, the earliest

times at which cores are potentially available in the new state v'_p are lower-bounded by $\{\max\{EST_i, A_x^{min}(v_p)\} \mid 2 \leq x \leq m\} \cup \{EFT_i\}$. This proves Eq. (12).

Similarly, from Facts 3 and 4, J_i starts executing on the first available core, which becomes certainly available at the latest at time A_1^{max}. J_i keeps that core *possibly busy* until $LFT_i(v_p)$, or equivalently said, the core that became available no later than A_1^{max} will be certainly available at $LFT_i(v_p)$ in the new system state v'_p. Therefore, considering Facts 1 and 3, the times at which cores are certainly available in the new state v'_p are upper-bounded by $\{\max\{EST_i, A_x^{max}(v_p)\} \mid 2 \leq x \leq m\} \cup \{LFT_i\}$. This proves Eq. (13). ◀

▶ **Lemma 7.** *If J_i is the job scheduled after v_p, all cores running predecessors of J_i in system state v_p become available by time $LST_i(v_p)$.*

Proof. As all predecessors of J_i must complete before J_i starts executing, and as the latest start time of J_i is $LST_i(v_p)$, all running predecessors of J_i must complete before time $LST_i(v_p)$. Hence, all cores running predecessors of J_i become available by time $LST_i(v_p)$. ◀

Line 1 in Algorithm 2 computes PA and CA according to Lemma 6. Lines 2–6 further ensure that the availability times of cores that are certainly executing predecessors of J_i are not larger than $LST_i(v_p)$, hence complying with Lemma 7.

Finally, Algorithm 2 computes the system-availability intervals for v'_p by sorting the lists PA and CA in non-decreasing order (lines 7–8). The correctness of this step is proven next.

▶ **Lemma 8.** *For any state v'_p built with Algorithm 1, let us define $t(v'_p)$ as follows: if $v_p = v_1$ then $t(v'_p) = 0$, otherwise $t(v'_p)$ is the EST of the last job dispatched to reach v'_p. For $1 \leq x \leq m$, x cores cannot be simultaneously available within $[t(v'_p),\ A_x^{min}(v'_p))$, and x cores are certainly available after time $A_x^{max}(v'_p)$.*

Proof. We prove the claim by structural induction on the states in the schedule-abstraction graph. The base case is state v_1, in which all cores are idle and, for $1 \leq x \leq m$, $A_x^{min}(v_1) = A_x^{max}(v_1) = 0$. The claim trivially holds as the interval $[t(v_1),\ A_x^{min}(v_1)) = [0, 0)$ is empty, and x cores are certainly available at time $A_x^{max}(v_1) = 0$, for all $1 \leq x \leq m$.

Next, in the inductive step, assume the claim holds for state v_p, that is x cores cannot be simultaneously available within $[t(v_p),\ A_x^{min}(v_p))$, and x cores are certainly available after time $A_x^{max}(v_p)$ for all $1 \leq x \leq m$. We prove that the claim holds in state v'_p resulting from dispatching J_i after v_p.

Assuming that $A_x^{min}(v_p)$ and $A_x^{max}(v_p)$ were safe bounds in state v_p (which holds by the induction hypothesis), Lemmas 6 and 7 prove that PA and CA provide safe lower bounds (resp., upper bounds) on the potential (resp., certain) availability times of each core in system state v'_p following the dispatch of job J_i, which happens no earlier than $t(v'_p) = EST_i$. Therefore, the x^{th} smallest element in PA is a lower bound on the time at which the x^{th} core may become available after $t(v'_p)$. Hence, the x^{th} smallest element in PA is also a lower bound on the time at which x cores may be simultaneously available after $t(v'_p)$. Since Algorithm 2 assigns the x^{th} smallest element in PA to $A_x^{min}(v'_p)$, the inductive claim holds for $A_x^{min}(v'_p)$. Similarly, the x^{th} smallest element in CA is an upper bound on the time at which an x^{th} core becomes certainly available in state v'_p. Hence, the x^{th} smallest element in CA is an upper bound on the time at which x cores are certainly available in v'_p. Since $A_x^{max}(v'_p)$ is assigned the x^{th} smallest element in CA, the inductive claim holds for $A_x^{max}(v'_p)$. ◀

Finally, Algorithm 2 updates the set of jobs that are certainly running in system state v'_p using the following property.

▶ **Property 2.** *If the earliest finish time of a running job $J_x \in \mathcal{X}(v_p)$ is later than J_i's latest start time, then J_x is still certainly running after J_i starts executing.*

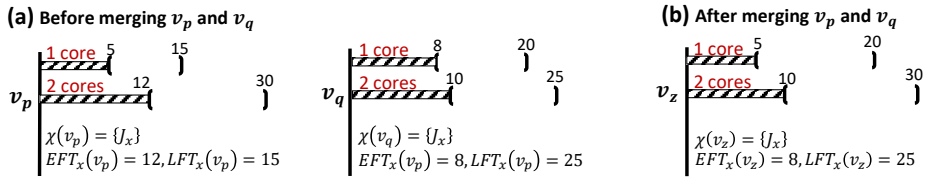

Figure 3 States v_p and v_q **(a)** before and **(b)** after merging.

Therefore, certainly running jobs in state v'_p include J_i and all jobs that were running in state v_p and respect Property 2, i.e., $\{J_x \mid J_x \in \mathcal{X}(v_p) \ \wedge \ LST_i \leq EFT_x(v_p)\}$. Moreover, all predecessors of J_i must have been completed by LST_i. Hence, Eq. (14) below excludes the predecessors of J_i from the list of jobs that are certainly running in state v'_p.

$$\mathcal{X}(v'_p) \leftarrow \{J_i\} \cup \{J_x \mid J_x \in \mathcal{X}(v_p) \setminus pred(J_i) \ \wedge \ LST_i \leq EFT_x(v_p)\} \tag{14}$$

4.6 Merge Phase

To slow down the growth of the graph (in terms of the number of system states generated), we merge paths with intersecting availability intervals that have the same set of jobs.

▶ **Rule 1** (Merge Rule). Two states v_p and v_q can be merged if $\mathcal{J}^P = \mathcal{J}^Q$ and $\forall x,\ 1 \leq x \leq m$, $A_x(v_p) \cap A_x(v_q) \neq \emptyset$.

When two states v_p and v_q are merged, the system-availability intervals $A_x(v_z)$ in the merged state v_z are set to include the availability intervals of both v_p and v_q:

$$A_x(v_z) = \left[\min\{A_x^{min}(v_p), A_x^{min}(v_q)\},\ \max\{A_x^{max}(v_p), A_x^{max}(v_q)\}\right]. \tag{15}$$

Eq. (15) expresses the fact that x cores become potentially available in the merged state v_z when x cores become potentially available in either of the original states v_p or v_q, and x core are certainly available in v_z when x cores are certainly available in both v_p and v_q.

Additionally, it is easy to see that the set of certainly running jobs in the merged state v_z comprises the jobs that were certainly running in both v_p and v_q, that is,

$$\mathcal{X}(v_z) = \{J_x \mid J_x \in \mathcal{X}(v_p) \cap \mathcal{X}(v_q)\}. \tag{16}$$

The finish time interval of each job J_x that is certainly running in v_z is updated to consider the bounds that were previously derived for all execution scenarios that lead to either v_p or v_q, and hence also to the merged state v_z. Therefore, we have that the EFT of J_x in v_z is the minimum between the EFTs in v_p and v_q. Similarly, the LST of J_x in v_z is the maximum LST reported for J_x in v_p and v_q, that is,

$$\begin{aligned} &EFT_x(v_z) = \min\{EFT_x(v_p), EFT_x(v_q)\} \text{ and} \\ &LFT_x(v_z) = \max\{LFT_x(v_p), LFT_x(v_q)\}. \end{aligned} \tag{17}$$

Figure 3 shows two states before and after merging. Lemma 9 proves that merging is safe.

▶ **Lemma 9.** *Merging two states v_p and v_q according to Rule 1 and Eqs.* (15), (16) *and* (17) *is* safe, *i.e., it does not remove any potentially reachable system state from the graph.*

Proof. First, Rule 1 enforces that the set of jobs scheduled on the path to v_p and v_q is identical for v_p and v_q. Therefore, the set of jobs that remain to be dispatched after v_z is the same as for v_p and v_q.

Second, removing jobs from the set of certainly running jobs $\mathcal{X}(\cdot)$ as done by Eq. (16), only increases the uncertainty in state v_z and therefore the set of system states reachable from v_z. Similarly, increasing the size of the possible finish intervals of the certainly running jobs (as done by Eq. (17)) increases the number of possible execution scenarios covered by v_z in comparison to v_p and v_q.

Finally, by Eq. (15) the system-availability intervals of the merged state v_z include the availability intervals of v_p and v_q. Therefore, all possible combinations of times at which a given number of cores is available either in state v_p or in state v_q are also possible in v_z. Thus, all sequences of dispatch events that are possible in v_p and v_q are possible in v_z and the system states reachable from v_z include all system states reachable from v_p and v_q. ◀

4.7 Correctness of the Proposed Solution

This section establishes the correctness of our analysis by showing that, for any possible execution scenario, there exists a path in the graph created by Algorithm 1 that represents the schedule of all jobs in the given scenario (i.e., Algorithm 1 is sound, but not exact).

▶ **Theorem 10.** *For any execution scenario such that a job $J_i \in \mathcal{J}$ finishes at some time t, there exists a path $P = \langle v_1, \ldots, v_p, v'_p \rangle$ in the schedule-abstraction graph generated by Algorithm 1 such that J_i is the label of the edge from the state v_p to the state v'_p and $t \in [EFT_i(v_p), LFT_i(v_p)]$.*

Proof. Initially, assume that the path $\langle v_1, \ldots, v_p \rangle$ respects the claim for all jobs dispatched before J_i in the execution scenario that led J_i to finish at time t. Furthermore, assume that **(i)** the availability intervals of state v_p correctly bound the availability time of x simultaneous cores in state v_p, $\forall x, 1 \leq x \leq m$, **(ii)** $\mathcal{X}(v_p)$ correctly includes a subset of the jobs that are certainly running on the platform before J_i is dispatched, and **(iii)** for each job $J_x \in \mathcal{X}(v_p)$, the interval $[EFT_x(v_p), LFT_x(v_p)]$ safely lower- and upper-bounds (i.e., contains) the completion time of J_x. We prove that there exists a vertex v'_p that is directly connected to v_p with an edge labeled J_i, that all three requirements (i)–(iii) hold for state v'_p, and that the interval $[EFT_i(v_p), LFT_i(v_p)]$ contains the completion time of J_i.

Under the assumption that hypotheses (i)–(iii) hold for v_p, Lemma 5 proves that Algorithm 1 expands the graph for any job that can possibly be dispatched next after v_p, hence also for J_i. Further, as proven in Sec. 4.4, Eq. (10) and Eq. (11) provide a lower and an upper bound on the completion time of J_i, respectively. Moreover, by Lemma 8, the availability intervals of v'_p correctly bound the simultaneous availability of x cores for all $1 \leq x \leq m$.

Eq. (14) computes the set $\mathcal{X}(v'_p)$ of certainly running jobs in state v'_p. Therefore, Requirement (ii) directly follows from Property 2 and the discussion of its role in obtaining Eq. (14) in Section 4.5. Requirement (iii) is the consequence of the assumption that the interval $[EFT_x(v_p), LFT_x(v_p)]$ computed for every job J_x dispatched before J_i in a state reached prior to v_p (and certainly running in v_p) is correct. Finally, according to Lemma 9, merging two states as in lines 12–16 of Algorithm 1 does not invalidate Requirements (i)-(iii).

Crucially, requirements (i)–(iii) are true for any state v_p' that is a direct successor of the initial system state v_1 because **(a)** in the initial state no job has been dispatched yet and all cores are available, and **(b)** Algorithm 1 initializes v_1's availability intervals to $[0, 0]$ (satisfying (i)), and sets the certainly running jobs set $\mathcal{X}(v_1)$ to $\emptyset$ (thus also satisfying (ii) and (iii)). The claim thus follows by induction on the states created by Algorithm 1. ◀

5 Empirical Evaluation

We conducted experiments to answer two main questions: **(i)** does the proposed test detect more schedulable workloads than state-of-the-art schedulability tests? And **(ii)** is the runtime of our analysis practical? We applied Algorithm 1 to the global limited-preemptive scheduling policy G-LP-FP. For the sake of simplicity, we used simple rate-monotonic priorities. As a baseline, we compared our results with the schedulability test of Serrano et al. [39] (identified as Serrano in the graphs) designed for *sporadic* limited-preemptive DAG tasks as it is the only available schedulability test in the state of the art for global limited-preemptive scheduling of DAG tasks based on the classical response-time analysis approach.

Since our test may also be used to analyze the special case of independent sequential non-preemptive tasks (NPR), we also performed experiments on such task sets, and compared our results to the test of Baruah for G-NP-EDF [5] (denoted by Baruah in the graphs), the test of Guan et al. for G-NP-FP [19] (denoted by Guan), the test of Lee for G-NP-FP [22] (denoted by Lee), and the test of Nasri et al. [33] for any G-NP-JLFP scheduling algorithm (denoted by Nasri18). Finally, we compare against the test of Yalcinkaya et al. [46] (denoted as Exact), an exact UPPAAL-based schedulability test for G-NP-FP (and EDF) that is designed for periodic tasks with fixed-preemption points and segmented self-suspensions.

We note that the Serrano, Baruah, Guan, and Lee tests are designed for sporadic DAG tasks; hence, we expect them to be pessimistic when applied to periodic workloads since sporadic tasks can generate more interference scenarios than periodic tasks. However, we believe that quantifying this pessimism serves to signify the need for schedulability tests that take task-activation patterns into account in order to provide more accurate results.

We implemented Algorithm 1 as a multi-threaded C++ program and performed the analysis on a cluster of machines each equipped with 256 GiB RAM and Intel Xeon E5-2667 v2 processors clocked at 3.3 GHz. We parallelized the breadth-first exploration of the schedule-abstraction graph using Intel's open-source Thread Building Blocks (TBB) library. Specifically, the while-loop in lines 3–19 can be easily parallelized since each iteration works on a different path. We report the *CPU time* as the runtime of the analysis, i.e., the total sum of the runtime of all threads used to analyze a task set. In the experiments, a task set was claimed unschedulable as soon as an execution scenario with a deadline miss was found.

Experiments on synthetic task sets. We generated tasks using the same established techniques as used in prior studies [30, 35, 38, 39, 10]. The method generates series-parallel DAGs with nested fork-joins by recursively expanding blocks (a.k.a. non-terminal nodes) to either terminal nodes or parallel sub-graphs until a maximum depth of recursion (which limits the number of nested branches), a maximum length of the critical path, or a maximum number of nodes in the DAG is reached. The generation algorithm allows to define the branching factor, i.e., the maximum number of branches of parallel sub-graphs (denoted by n_{par}). In our experiment, the probability that a node is terminal, i.e., that it does not immediately fork a new branch, was set to $p_{term} = 0.4$, the probability of adding a random edge (precedence constraint) between two siblings was set to $p_{add} = 0.1$, the maximum number of nested branches was 3, the maximum number of nodes in the DAG was 50, and the maximum critical path length was set to 10 nodes. The WCET of each node was selected uniformly at random from the range [1, 50]. The BCET of each node was set to be 70% of the WCET.

To generate periodic DAG tasks with total utilization U, we used uUniFast [7] to generate random utilization values with a total sum U, and then assigned a period to each task using $\max\{\bar{C}_i/U_i, \bar{C}_i\}$, where $\bar{C}_i$ is the total sum of the WCET of all nodes of the

task and U_i is the task utilization. To avoid cases where periods are co-prime and hence systems for which the hyperperiod is impractically large, we scaled the obtained periods so that they are contained in the interval $[500, 100000]$ with possible values given by the set $\{x \cdot 10^y \mid 1 \leq x \leq 9,\ 3 \leq y \leq 5\}$ms. This covers periods that are three orders of magnitude apart with a log-uniform distribution and includes periods commonly used by the automotive industry [21]. After assigning periods, we proportionally scale the WCET of the nodes so that tasks keep their intended utilization U_i (as assigned by uUniFast). We assumed that all tasks are first released at time 0 and that their deadlines are equal to their periods. Moreover, we assumed that all DAG nodes have an arrival time equal to the arrival time of the corresponding task. To filter out trivial task sets, we discarded task sets that cannot be successfully scheduled by G-LP-FP when each node of each task executes for its WCET. That is, we simulated the schedule of one hyperperiod using the WCET of each node and checked if there is a deadline miss (note that this initial test is only a necessary schedulability test, not a sufficient one because of the schedulability anomalies that exist under non-preemptive and limited preemptive scheduling).

The experiments were performed by varying **(Exp1)** the total system utilization U for 10 DAG tasks on 4 cores (Figures 4(a) and (d)), and **(Exp2)** the number of cores m and DAG tasks $n \in \{5, 10, 15, 20\}$ with $U = 0.5 \cdot m$ (Figures 4(b) and (e)). For each combination of parameters (e.g., DAG tasks with $U = 30\%, n = 10, m = 4$), more than 100 random task sets were generated. For each setup, we report the *schedulability ratio* (ratio of schedulable task sets to the number of task sets generated for that setup) and the *runtime* of Algorithm 1 for the task sets that were deemed *schedulable*. We excluded the runtime of unschedulable task sets since it would otherwise favor our solution and bias the results due to the fact that we stop the analysis as soon as a deadline miss is found. In other words, we only report the runtime of experiments that ran to the end, which is the worst case from an analysis runtime perspective. Since the runtime of the Serrano test never exceeded one second, it was omitted from all diagrams depicting runtimes.

Figure 4(a) shows a significant gap between the schedulability ratio determined by our solution and the baseline analysis for DAG tasks. For example, the Serrano test could only identify 10% of schedulable task sets for $U = 0.3$, while our test shows that at least 99% of them are schedulable. Furthermore, with the increase in the number of cores, the Serrano test becomes more pessimistic, e.g., it cannot find any schedulable task set with $U \geq 0.3$ when there are 16 cores, while the proposed test still finds schedulable task sets until $U = 0.6$.

Figure 4(b) shows the schedulability ratio as a function of the ratio between the number of DAG tasks and the number of cores (denoted by n/m). We observe that schedulability decreases when the number of tasks is close to the number of cores (i.e., the ratio n/m is around 1). We explain this trend by the fact that when there are more tasks than cores ($n/m > 1$), the per-task utilization and hence the blocking times caused by nodes of lower-priority tasks are smaller. As a result, the schedulability ratio is larger. This can be easily seen for $m = 4$ and $m = 8$ (since most values of n are larger than 4 or 8). The effect of smaller blocking times shows itself for $m = 16$, too, as an increase in the schedulability ratio for $n = 20$. When there are more cores than tasks ($n/m < 1$), there are enough cores to execute all tasks in parallel, hence the increase in schedulability. Further, more cores for a fixed number of tasks implies increased opportunity for tasks to execute their nodes in parallel; hence their response times approach their critical path lengths. This can be seen for larger values of m, e.g., 16 to 64. For instance the schedulability ratio for $m = 64$ is 100% for 10, 15 and 25 tasks, while it varies between 30% and 75% for $m = 16$.

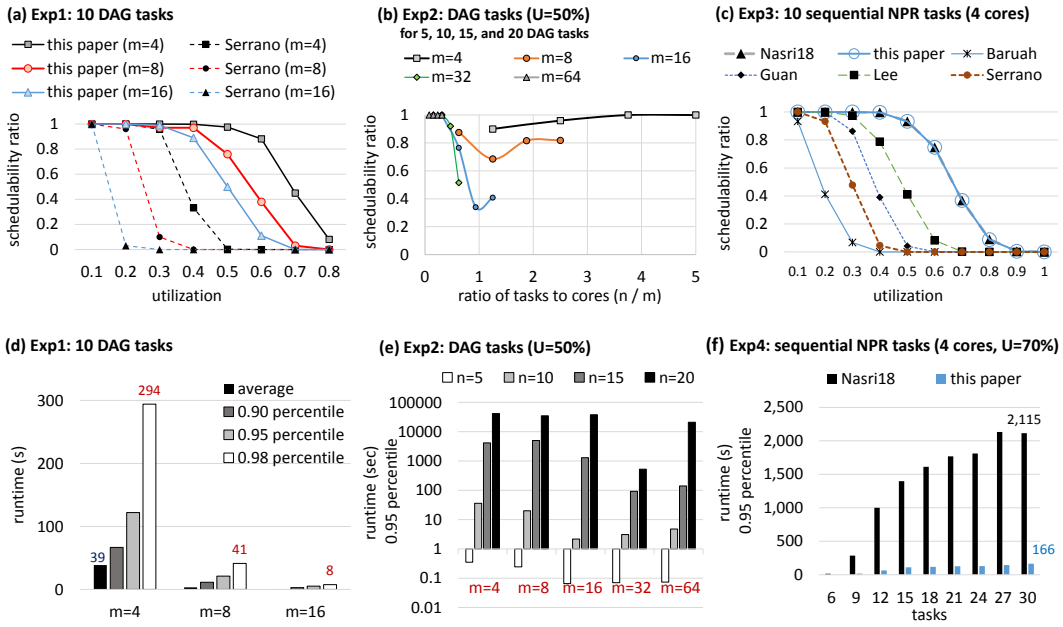

Figure 4 Experimental results for synthetic task sets for different experiments: **(a, d)** Exp1, **(b, e)** Exp2, **(c)** Exp3, and **(f)** Exp4. In (b), all task sets with $m = 64$ are schedulable. Hence, the curve overlaps with other curves (prior to the point $n/m < 0.4$).

Figures 4(d) and (e) show either a decrease or only a small linear increase in the runtime of the analysis w.r.t. to the increase in the number of cores in all experiments. Thanks to our new system state abstraction, the number of direct successors of a state does not depend on the number of cores in the system (unlike Nasri18 [33]) and hence the dependence on m is limited to the cost of calculating t_{high}, t_{wc}, etc. for each state.

For DAG tasks, with an increase in the number of tasks, the runtime of our analysis increases rapidly as the number of nodes and hence the number of jobs increases. While our analysis efficiently handles most task sets with up to 15 tasks within a couple of hours, it becomes notably slower for larger numbers of tasks. This, in particular, affects systems with a smaller number of cores, e.g., 4 and 8 cores, because when the system has insufficient cores to fully exploit the available task parallelism, the number of pending nodes in each system state increases. Since all nodes exhibit execution time variation, this drastically increases the number of possible scheduling decisions. As a result, the schedule abstraction graph grows rapidly since it must consider all possible interleavings.

In Figure 4(e) we observe a decrease in the runtime of the analysis from $m = 16$ to $m = 32$ and then an increase from $m = 32$ to $m = 64$. This decrease is due to the decrease in blocking times and an increase in the number of available cores (e.g., from 16 to 32 for 20 tasks). As a result, the busy windows become shorter, and hence paths merge very quickly as there are only relatively few interleavings to consider. On the other hand, the increase in the runtime of the analysis for $m = 64$ comes from the fact that, in a task set with 20 DAG tasks with $U = 50\%$, there are more tasks with large per-task utilizations. This situation increases the length of busy windows since tasks have only little slack. Moreover, due to the execution time variation of the tasks, there will be more scenarios that must be covered in the graph, which leads to an increase in the runtime of the analysis.

Experiments on non-preemptive sequential tasks (NPR). For the experiments on independent sequential non-preemptive tasks, we used the same task set generation setup as in [33]. To randomly generate a non-preemptive periodic task set with n tasks and a given utilization U, we used Emberson et al. [12] method to select the periods with a log-uniform distribution from the range [10000, 100000] microseconds with a granularity of $5000\mu s$. We then used the RandFixSum [40] algorithm to generate n random task utilizations that sum to U. From the task utilization, we obtained C_i^{max} from the task utilization and the period and then set C_i^{min} to be 10% of C_i^{max}. Tasks were assumed to have implicit deadlines and any task set that had more than 100,000 jobs per hyperperiod was discarded from the experiment.

The experiments were performed by varying **(Exp3)** the total system utilization U ($n = 10$ and $m = 4$) for sequential non-preemptive tasks (Figure 4(c)), and **(Exp4)** the number of tasks n ($U = 70\%$ and $m = 4$) for sequential non-preemptive tasks (Figure 4(f)).

For sequential NPR tasks, as seen in Figure 4(c), our test performs similarly to Nasri18 (event though those tests are incomparable since task sets may be deemed schedulable by one and unschedulable by the other and vice versa). Both tests find many more schedulable task sets than the tests of Baruah, Guan, Lee, and Serrano. For example, for $U = 0.6$, our test and Nasri18 improve accuracy by 66 percentage points over the other baseline tests.

We have tried to run the exact test of Yalcinkaya et al. [46] on the data from Exp3. However, due to the scalability issue discussed in the introduction, the test could not complete the analysis of enough task sets to extract any meaningful results. Instead, we ran our analysis on the dataset used by Yalcinkaya et al. for sequential NPR task sets (see Exp2 in [46] for details). The results are depicted in Figure 1(a); the difference in accuracy between our test and the exact test is indistinguishable for the considered NPR task sets.

Figure 4(f) shows a neat improvement of our new analysis w.r.t. Nasri18, i.e., the best known analysis for G-NP-JLFP scheduling. For example, the 95^{th} percentile runtime of Nasri18 [33] for 4 cores and 30 NPR tasks is more than 2,115 s while the 95^{th} percentile runtime of the analysis presented in this paper is 166 s (i.e., a more than one order-of-magnitude difference). The maximum runtime of Nasri18 on all experiments that finished was 3027 s and one task set reached the time out of 1 h, while the maximum runtime of our new analysis was 275 s. The average runtime of the Nasri18 test was 327 s while our new analysis took an average of 25s only. These numbers strongly suggest that the proposed analysis is at least one order of magnitude faster than the Nasri18 test.

Experiments on benchmark task sets. We used the StreamIT benchmarks, which consist of a set of *digital signal processing* applications to evaluate the performance of our analysis on a realistic application workload. We used the DAG structure and WCET information of the tasks obtained by Rouxel et al. [36]. Table 1 reports the number of DAG nodes, width of the DAG graph (i.e., maximum number of parallel nodes), and the number of fork/join nodes. This table also presents the number of states, edges, and the runtime of the analysis for each of the benchmark applications when executed on a 4-core platform. As it can be seen, the analysis takes less than a minute even when there are more than 400 nodes in the DAG or when there are 80 fork/join constructs.

Discussion. Overall, we conclude that: **(i)** the proposed analysis is practical for realistic workload sizes and benchmarks, **(ii)** it has high accuracy when compared with the state-of-the-art exact schedulability analysis of sequential non-preemptive tasks with a global scheduling policy while being able to scale to much bigger systems (i.e., with more tasks

Table 1 Analysis of Benchmark Tasks.

Benchmark	#nodes	w	forks	**states**	**edges**	**runtime (ms)**
802.11a	119	7	17	10,164	28,656	**483.15**
Audiobeam	20	15	1	20	20	**0.18**
BeamFormer	56	12	2	6,036	29,686	**494.45**
CFAR	4	1	0	4	4	**0.05**
Complex-FIR	3	1	0	3	3	**0.04**
DCT2	40	16	2	40	40	**0.63**
DES	423	8	80	2,343	4,983	**849.63**
FFT2	26	2	1	74	122	**1.24**
FFT4	42	2	10	42	42	**0.27**
Filterbank	52	6	1	810,425	5,293,419	**25,339.02**
FMRadio	43	12	7	53,199	258,781	**1,402.83**

and more cores), **(iii)** it identifies a significantly larger portion of schedulable DAG tasks in comparison to the existing test, and **(iv)** the new system state abstraction allows a significant improvement in terms of scalability in comparison to the state-of-the-art test Nasri18.

However, even though the new abstraction allows scaling to much larger workloads, the treatment of execution time variation still needs further improvement. In the presence of precedence constraints, the impact of the response-time jitter of a job on its successors is the same as if the successors had a large release jitter. This induced jitter accumulates over chains of jobs with precedence constraints and greatly increases the degree of non-determinism in the graph exploration, and eventually forces the algorithm to consider all combinations of job orderings. This, for example, happens often in highly parallel DAG tasks or when the number of DAG tasks increases. Consequently, new techniques will have to be developed to allow the analysis to scale to highly parallel DAGs with large execution-time jitter.

6 Conclusion

We have considered the problem of analyzing the schedulability of a set of limited-preemptive DAG tasks with internal parallelism and precedence constraints scheduled upon a multicore platform using a global *job-level fixed-priority* (JLFP) scheduling policy. Our analysis conceptually enumerates all possible schedules using a novel system state abstraction that keeps track of the times at which a certain number of cores will become available. We have shown how the space of possible schedules can be explored with the abstraction, provided a proof of correctness, and conducted extensive experiments to assess the efficiency of the solution. Our analysis finds between 10 and 90 percentage points more schedulable task sets for most system configurations, in comparison with the best available baseline. It also scales to systems with up to 64 cores and 20 DAG tasks. A comparison with the state-of-the-art exact schedulability test for sequential non-preemptive tasks scheduled by a global JLFP scheduling policy has shown our analysis to scale much better while being almost as accurate as the exact test. The proposed analysis, however, does not yet scale to highly parallel DAG tasks or systems with a large number of cores (e.g., more than 64). In the future, we will investigate better ways of managing jitter, e.g., by applying partial-order reduction to skip over redundant paths that do not contribute to the worst-case response time of a task.

References

1. Jakaria Abdullah, Morteza Mohaqeqi, Gaoyang Dai, and Wang Yi. Schedulability Analysis and Software Synthesis for Graph-Based Task Models with Resource Sharing. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–270, 2018.
2. Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability Analysis of Global Memory-predictable Scheduling. In *ACM International Conference on Embedded Software*, pages 20:1–20:10, 2014.
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. Theodore P. Baker and Michele Cirinei. Brute-Force Determination of Multiprocessor Schedulability for Sets of Sporadic Hard-Deadline Tasks. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 62–75, 2007.
5. Sanjoy Baruah. The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors. *Real-Time Systems*, 32(1):9–20, 2006.
6. Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 222–231, 2015.
7. Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
8. Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility Analysis of Sporadic Real-time Multiprocessor Task Systems. In *ESA*, pages 230–241. Springer, 2010.
9. Artem Burmyakov, Enrico Bini, and Eduardo Tovar. An Exact Schedulability Test for Global FP Using State Space Pruning. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 225–234, 2015.
10. Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio C. Buttazzo. Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.
11. UmaMaheswari Devi and James H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *IEEE International Real-Time Systems Symposium (RTSS)*, pages 12–341, 2005.
12. Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques For The Synthesis Of Multiprocessor Tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, 2010.
13. José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling. In *Proceedings of the 25th international conference on real-time networks and systems*, pages 28–37. ACM, 2017.
14. José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability Analysis of DAG Tasks With Arbitrary Deadlines Under Global Fixed-Priority Scheduling. *Real-Time Systems*, 55(2):387–432, April 2019.
15. José Fonseca, Geoffrey Nelissen, Vincent Nelis, and Luís Miguel Pinho. Response Time Analysis of Sporadic DAG Tasks Under Partitioned Scheduling. In *11th IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2016.
16. José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luís Miguel Pinho. A multi-DAG Model for Real-time Parallel Applications with Conditional Execution. In *Annual ACM Symposium on Applied Computing (SAC)*, pages 1925–1932, 2015.
17. Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-Time Syst.*, 52(6):808–832, 2016.
18. Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. In *Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, pages 263–272, 2007.

19 Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536–546, 2011.

20 Zhishan Guo, Ashikahmed Bhuiyan, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-Efficient Multi-Core Scheduling for Real-Time DAG Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 22:1–22:21, 2017.

21 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmark for free. In *International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS)*, 2015.

22 Jinkyu Lee. Improved Schedulability Analysis Using Carry-In Limitation for Non-Preemptive Fixed-Priority Multiprocessor Scheduling. *IEEE Transactions on Computers*, 66(10):1816–1823, 2017.

23 Jinkyu Lee and Kang G. Shin. Improvement of Real-Time Multi-CoreSchedulability with Forced Non-Preemption. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1233–1243, 2014.

24 Robert Leibinger. Software Architectures for Advanced Driver Assistance Systems (ADAS), 2015. URL: `https://people.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-talk-keynote.pdf`.

25 Cong Liu and James H Anderson. Supporting pipelines in soft real-time multiprocessor systems. In *21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 269–278. IEEE, 2009.

26 Cong Liu and James H Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 23–32. IEEE, 2010.

27 Cong Liu and James H Anderson. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *31st IEEE Real-Time Systems Symposium (RTSS)*, pages 3–13. IEEE, 2010.

28 Cong Liu and James H Anderson. Supporting graph-based real-time applications in distributed systems. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, volume 1, pages 143–152. IEEE, 2011.

29 Cláudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2017.

30 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *Euromicro Conference on Real-Time Systems, (ECRTS)*, pages 211–221, 2015.

31 Morteza Mohaqeqi, Jakaria Abdullah, Nan Guan, and Wang Yi. Schedulability Analysis of Synchronous Digraph Real-Time Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 176–186, 2016.

32 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.

33 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)*, pages 9:1–9:23, 2018.

34 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 269–279, 2011.

35 Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit Preemption Placement for Real-Time Conditional Code. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 177–188, 2014.

36 Benjamin Rouxel and Isabelle Puaut. STR2RTS: Refactored streamit benchmarks into statically analysable parallel benchmarks for WCET estimation & real-time scheduling. In *OASIcs-OpenAccess Series in Informatics*, volume 57, pages 1:1–1:12, 2017.

37 Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Parallel Real-Time Scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

38 Maria A. Serrano, Alessandra Melani, Marko Bertogna, and Eduardo Quiñones. Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In *Europe Conference on Design, Automation & Test & Exhibition (DATE)*, pages 1066–1071, 2016.

39 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.

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

41 Martin Stigge and Wang Yi. Combinatorial Abstraction Refinement for Feasibility Analysis of Static Priorities. *Real-Time Systems*, 51(6):639–674, 2015.

42 Youcheng Sun and Marco Di Natale. Assessing the Pessimism of Current Multicore Global Fixed-Priority Schedulability Analysis. Research report, University of Oxford, 2017. URL: `https://hal.archives-ouvertes.fr/hal-01468067`.

43 Youcheng Sun and Giuseppe Lipari. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling. *Real-Time Systems Journal*, 52(3):323–355, 2016.

44 Alexander Wieder and Björn B. Brandenburg. On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 45–56, 2013.

45 Jun Xiao, Sebastian Altmeyer, and Andy Pimentel. Schedulability Analysis of Non-preemptive Real-time Scheduling for Multicore Processors with Shared Caches. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 199–208, 2017.

46 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.