A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling

Mitra Nasri

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
mitra@mpi-sws.org

Geoffrey Nelissen

CISTER Research Centre, Polytechnic Institute of Porto (ISEP-IPP), Porto, Portugal
grrpn@isep.ipp.pt

Björn B. Brandenburg

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

An effective way to increase the timing predictability of multicore platforms is to use non-preemptive scheduling. It reduces preemption and job migration overheads, avoids intra-core cache interference, and improves the accuracy of worst-case execution time (WCET) estimates. However, existing schedulability tests for global non-preemptive multiprocessor scheduling are pessimistic, especially when applied to periodic workloads. This paper reduces this pessimism by introducing a new type of sufficient schedulability analysis that is based on an exploration of the space of possible schedules using concise abstractions and state-pruning techniques. Specifically, we analyze the schedulability of non-preemptive job sets (with bounded release jitter and execution time variation) scheduled by a global job-level fixed-priority (JLFP) scheduling algorithm upon an identical multicore platform. The analysis yields a lower bound on the best-case response-time (BCRT) and an upper bound on the worst-case response time (WCRT) of the jobs. In an empirical evaluation with randomly generated workloads, we show that the method scales to 30 tasks, a hundred thousand jobs (per hyperperiod), and up to 9 cores.

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

Keywords and phrases global multiprocessor scheduling, schedulability analysis, non-preemptive tasks, worst-case response time, best-case response time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.9

Related Version An extended version of this paper is available as a technical report [19], `http://www.mpi-sws.org/tr/2018-003.pdf`.

Acknowledgements The first author is supported by a post-doc fellowship awarded by the Alexander von Humboldt Foundation. The second author was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234). The authors would like to thank the anonymous reviewers for their insightful comments and suggestions.

1 Introduction

While modern multicore platforms offer ample processing power and a compelling price/performance ratio, they also come with no small amount of architectural complexity. Unfortunately, this complexity—such as shared caches, memory controllers, and other shared micro-architectural resources—has proven to be a major source of execution-time unpredictability, and ultimately a fundamental obstacle to deployment in safety-critical systems.

In response, the research community has developed a number of innovative approaches for managing such challenging hardware platforms. One particularly promising approach explored in recent work [1, 15, 24] is to split each job into three distinct phases: **(i)** a dedicated *memory-load* or *prefetching* phase, which transfers all of a job's required memory from the shared main memory to a core-local private cache or scratchpad memory; followed by **(ii)** the actual *execution* phase, in which the job executes *non-preemptively* and in an isolated manner without interference from the memory hierarchy as all memory references are served from a fast, exclusive private memory, which greatly enhances execution-time predictability; and finally **(iii)** a *write-back* phase in which any modified data is flushed to main memory. As a result of the high degree of isolation restored by this approach [20], a more accurate *worst-case execution time* (WCET) analysis becomes possible since the complete mitigation of inter-core interference during the execution phase allows existing uniprocessor techniques [25] to be leveraged. Recent implementations of the idea, such as Tabish et al.'s scratchpad-centric OS [24], have shown the phased-execution approach to indeed hold great promise in practice.

From a scheduling point of view, however, the phased-execution approach poses a number of difficult challenges. As jobs must execute non-preemptively—otherwise prefetching becomes impractical and there would be only little benefit to predictability—the phased-execution approach fundamentally requires a *non-preemptive real-time multiprocessor scheduling problem* to be solved. In particular, Alhammad and Pellizzoni [1] and Maia et al. [15] considered the phase-execution model in the context of non-preemptive *global scheduling*, where pending jobs are allocated simply to the next available core in order of their priorities.

Crucially, to make schedulability guarantees, Alhammad and Pellizzoni [1] and Maia et al. [15] rely on existing state-of-the-art analyses of global non-preemptive scheduling as a foundation for their work. Unfortunately, as we show in Sec. 6, this analytical foundation—i.e., the leading schedulability tests for global non-preemptive scheduling [4, 10, 11, 13]—suffers from substantial pessimism, especially when applied to periodic hard real-time workloads.

To attack this analysis bottleneck, we introduce a new, much more accurate method for the schedulability analysis of *finite sets of non-preemptive jobs* under *global job-level fixed-priority* (JLFP) scheduling policies. Our method, which can be applied to *periodic real-time tasks* (and other recurrent task models with a repeating hyperperiod), is based on a novel state-space exploration approach that can scale to realistic system parameters and workload sizes. In particular, this work introduces a new abstraction for representing the space of possible non-preemptive multiprocessor schedules and explains how to explore this space in a practical amount of time with the help of novel state-pruning techniques.

Related work. Global non-preemptive multiprocessor scheduling has received much less attention to date than its preemptive counterpart. The first sufficient schedulability test for global non-preemptive scheduling was proposed by Baruah [4]. It considered sequential sporadic tasks scheduled with a non-preemptive *earliest-deadline-first* (G-NP-EDF) scheduling algorithm. Later, Guan et al. [10, 11] proposed three new tests; one generic schedulability

test for any *work-conserving* global non-preemptive scheduling algorithm, and two response-time bounds for G-NP-EDF and global non-preemptive *fixed-priority* (G-NP-FP) scheduling. Recently, Lee et al. [13, 14] proposed a method to remove unnecessary carry-in workload from the total interference that a task suffers. These tests for sporadic tasks have been used in various contexts such as the schedulability analysis of periodic parallel tasks with non-preemptive sections [21] and systems with shared cache memories [26] or with transactional memories [1, 24]. However, these tests become needlessly pessimistic when applied to periodic tasks as they fail to discount many execution scenarios that are impossible in a periodic setting. Moreover, these tests do not account for any release jitter that may arise due to timer inaccuracy, interrupt latency, or networking delays.

To the best of our knowledge, no exact schedulability analysis for global job-level fixed-priority non-preemptive scheduling algorithms (including G-NP-EDF and G-NP-FP) either for sporadic or for periodic tasks has been proposed to date. The exact schedulability analysis of global *preemptive* scheduling for sporadic tasks has been considered in several works [3, 5, 6, 9, 23]. These analyses are mainly based on exploring all system states that can be possibly reached using model checking, timed automata, or linear-hybrid automata. These works are inherently designed for a preemptive execution model, where no lower-priority task can block a higher-priority one, and hence are not applicable to non-preemptive scheduling. The second limitation of the existing analyses is their limited scalability. They are affected by the number of tasks, processors, and the granularity of timing parameters such as periods. For example, the analysis of Sun et al. [23] can only handle up to 7 tasks and 4 cores, while the solution by Guan et al. [9] is applicable only if task periods lie between 8 and 20.

In our recent work [16], we have introduced an exact schedulability test based on a schedule-abstraction model for uni-processor systems executing non-preemptive job sets with bounded release jitter and execution time variation. By introducing an effective state-merging technique, we were able to scale the test to task sets with more than 30 tasks or about 100000 jobs in their hyperperiod for any job-level fixed-priority scheduling algorithm. The underlying model and the test's exploration rules, however, are designed for, and hence limited to, uniprocessor systems and cannot account for any scenarios that may arise when multiple cores execute jobs in parallel.

Contributions. In this paper, we introduce a sufficient schedulability analysis for global job-level fixed-priority scheduling algorithms considering a set of non-preemptive jobs with bounded release jitter and execution time variation. Our analysis derives a lower bound on the best-case response time (BCRT) and an upper bound on the worst-case response time (WCRT) of each job, taking into account all uncertainties in release and execution times. The proposed analysis is not limited to the analysis of periodic tasks (with or without release jitter), but can also analyze any system with a known job release pattern, e.g., bursty releases, multi-frame tasks, or any other application-specific workload that can be represented as a recurring set of jobs.

The analysis proceeds by exploring a graph, called *schedule-abstraction graph*, that contains all possible schedules that a given set of jobs may experience. To render such an exploration feasible, we aggregate all schedules that result in the same order of start times of the jobs and hence significantly reduce the search space of the analysis and makes it independent from the time granularity of the timing parameters of the systems. Moreover, we provide an efficient path-merging technique to collapse redundant states and avoid non-required state explorations. The paper presents an algorithm to explore the search space, derives merge rules, and establishes the soundness of the solution.

2 System Model and Definitions

We consider the problem of scheduling a finite set of non-preemptive *jobs* $\mathcal{J}$ on a multicore platform with m identical cores. Each job $J_i = ([r_i^{min}, r_i^{max}], [C_i^{min}, C_i^{max}], d_i, p_i)$ has an earliest-release time r_i^{min} (a.k.a. *arrival time*), latest-release time r_i^{max}, *absolute* deadline d_i, best-case execution time (BCET) C_i^{min}, WCET C_i^{max}, and priority p_i. The priority of a job can be decided by the system designer at design time or by the system's JLFP scheduling algorithm. We assume that a numerically smaller value of p_i implies higher priority. Any ties in priority are broken by job ID. For ease of notation, we assume that the "$<$" operator implicitly reflects this tie-breaking rule. We use $\mathbb{N}$ to represent the natural numbers including 0. We assume a discrete-time model and all job timing parameters are in $\mathbb{N}$.

At runtime, each job is *released* at an *a priori* unknown time $r_i \in [r_i^{min}, r_i^{max}]$. We say that a job J_i is *possibly released* at time t if $t \geq r_i^{min}$, and *certainly released* if $t \geq r_i^{max}$. Such release jitter may arise due to timer inaccuracy, interrupt latency, or communication delays, e.g., when the task is activated after receiving data from the network. Similarly, each released job has an *a priori* unknown execution time requirement $C_i \in [C_i^{min}, C_i^{max}]$. Execution time variation occurs because of the use of caches, out-of-order-execution, input dependencies, program path diversity, state dependencies, etc. We assume that the absolute deadline of a job, i.e., d_i, is fixed *a priori* and not affected by release jitter. Released jobs remain pending until completed, i.e., there is no job-discarding policy.

Each job must execute sequentially, i.e., it cannot execute on more than one core at a time. Hence, because jobs are non-preemptive, 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 length of the interval between the *arrival* and completion of the job [2], i.e., $t + C_i - r_i^{min}$. We say that a job is *ready* at time t if it is released and did not yet start its execution prior to time t.

In this paper, we assume that shared resources that must be accessed in mutual exclusion are protected by FIFO spin locks. Since we consider a non-preemptive execution model, it is easy to obtain a bound on the worst-case time that any job spends spinning while waiting to acquire a contested lock; we assume the worst-case spinning delay is included in the WCETs.

Throughout the paper, we use $\{\cdot\}$ to denote a set of items in which the order of elements is irrelevant and $\langle\cdot\rangle$ to denote an enumerated set of items. In the latter case, we assume that items are indexed in the order of their appearance in the sequence. 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 return the usual maximum and minimum values in X, respectively.

The schedulability analysis proposed in this paper can be applied to periodic tasks. A thorough discussion of how many jobs must be considered in the analysis for different types of tasks with release offset and constrained or arbitrary deadlines has been presented in [16].

We consider a non-preemptive global JLFP scheduler upon an identical multicore platform. The scheduler is invoked whenever a job is released or completed. In the interest of simplifying the presentation of the proposed analysis, we make the modeling assumption that, without loss of generality, at any invocation of the scheduler, at most one job is picked and assigned to a core. If two or more release or completion events occur at the same time, the scheduler is invoked once for each event. 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.

We allow for a non-deterministic core-selection policy when more than one core is available for executing a job, i.e., when a job is scheduled, it may be scheduled on any available core. The reason is that requiring a deterministic tie-breaker for core assignments would impose a large synchronization overhead, e.g., to rule out any race windows when the scheduler is invoked concurrently on different cores at virtually the same time, and hence no such rule is usually implemented in operating systems.

We say that 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.

▶ **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 $r_i \in [r_i^{min}, r_i^{max}]$.

We exclusively focus on work-conserving, and priority-driven 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 the scheduler overhead and to account for any micro-architectural interference (e.g., cache or memory bus interference).

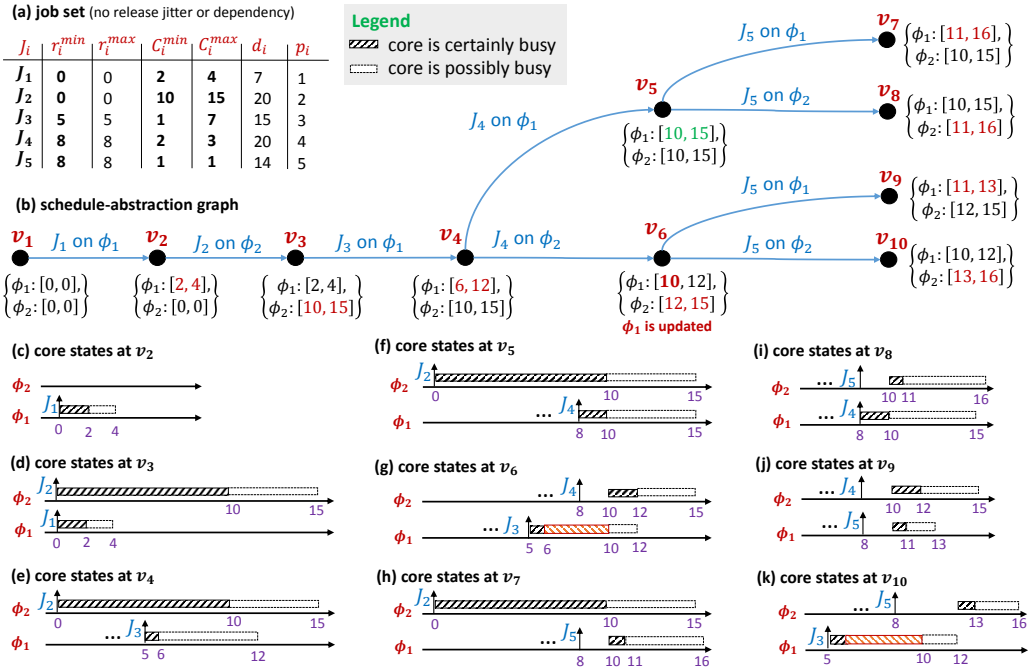
3 Schedule-Abstraction Graph

Our schedulability analysis derives a safe upper bound on the WCRT and a safe lower bound on the BCRT of each job by exploring a superset of all possible schedules. Since the number of schedules depends on the space of possible execution scenarios, which is a combination of release times and execution times of the jobs, it is intractable to naively enumerate all distinct schedules. To solve this problem, we aggregate schedules that lead to the *same ordering* of job start times (a.k.a. dispatch times) on the processing platforms. To this end, in the rest of this section, we introduce an abstraction of job orderings that encodes possible finish times of the jobs.

To represent possible job orderings we use an acyclic graph whose edges are labeled with jobs. Thus, each path in the graph represents a dispatch order of jobs in the system. Fig. 1-(b) shows an example of such a graph. For example, the path from v_1 to v_9 means that the jobs $\langle J_1, J_2, J_3, J_4, J_5 \rangle$ have been scheduled one after another. The length of a path P, denoted by $|P|$, is the number of jobs scheduled on that path.

To account for the uncertainties in the release times and execution times of jobs, which in turn result in different schedules, we use intervals to represent the state of a core. For example, assume that there is only one core in the system and consider a particular job J_i. Assume that the release interval and execution requirement of J_i are $[0, 5]$ and $[10, 15]$, respectively. In a job ordering where J_i is the first job dispatched on the core, the resulting core interval will become $[10, 20]$, where $10 = r_i^{min} + C_i^{min}$ and $20 = r_i^{max} + C_i^{max}$ are the *earliest finish time* (EFT) and *latest finish time* (LFT), respectively, of the job on the core. Here, the interval $[10, 20]$ means that the core will be *possibly available* at time 10 and will be *certainly available* at time 20. Equivalently, any time instant t in a core interval corresponds to an execution scenario in which the core is busy until t and becomes available at t.

Using the notion of core intervals, we define a system state as a set of m core intervals. System states are vertices of the graph and represent the states of the cores after a certain set of jobs has been scheduled in a given order.



J_i	r_i^{min}	r_i^{max}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	0	0	2	4	7	1
J_2	0	0	10	15	20	2
J_3	5	5	1	7	15	3
J_4	8	8	2	3	20	4
J_5	8	8	1	1	14	5

Figure 1 A schedule-abstraction graph G for five jobs that are scheduled on two cores: **(a)** shows the job set information (jobs do not have release jitter), **(b)** shows the schedule-abstraction graph, **(c)** to **(k)** show the state of the two cores at system states v_2 to v_{10}, respectively.

3.1 Graph Definition

The schedule-abstraction graph is a directed acyclic graph $G = (V, E)$, where V is a set of system states and E is the set of labeled edges. A system state $v \in V$ is a multiset of m core intervals denoted by $\{\phi_1, \phi_2, \ldots, \phi_m\}$. A core interval $\phi_k = [EFT_k, LFT_k]$ is defined by the EFT and LFT of a job that is scheduled on the core, denoted by EFT_k and LFT_k, respectively. Equivalently, EFT_k is the time at which the core becomes *possibly available* and LFT_k is the time at which the core becomes *certainly available*. Since cores are identical, the schedule-abstraction graph does not distinguish between them and hence does not keep track of the physical core on which a job is executing.

The schedule-abstraction graph contains all possible orderings of job start times in any possible schedule. This ordering is represented by directed edges. Each edge $e = (v_p, v_q)$ from state v_p to state v_q has a label representing the job that is scheduled next after state v_p. The sequence of edges in a path P represents a possible sequence of scheduling decisions (i.e., a possible sequence of job start times) to reach the system state modeled by v_p from the initial state v_1.

3.2 Example

Fig. 1-(b) shows the schedule-abstraction graph that includes all possible start-time orders of the jobs defined in Fig. 1-(a) on a two-core processor. In the initial state v_1, no job is scheduled. At time 0, two jobs J_1 and J_2 are released. Since $p_1 < p_2$, the scheduler first schedules J_1 on one of the available cores. For the sake of clarity, we have numbered the cores in this example, however, they are identical from our model's perspective.

Fig. 1-(c) shows the state of both cores after job J_1 is scheduled. The dashed rectangle that covers the interval $[0, 2)$ shows the time during which the core is certainly not available for other jobs since $C_1^{min} = 2$. In this state, the EFT of ϕ_1 is 2 and its LFT is 4, as shown by the white rectangle, i.e., ϕ_1 may possibly become available at time 2 and will certainly be available at time 4. From the system state v_2, only v_3 is reachable. The transition between these two states indicates that job J_2 is scheduled on the available core ϕ_2 starting at time 0.

As shown in Fig. 1-(d), core ϕ_1 is certainly available from time 4. Thus, when job J_3 is released at time 5, the scheduler has no other choice but to schedule job J_3 on this core. The label of this transition shows that J_3 has been scheduled.

From system state v_4, two other states are reachable depending on the finish times of jobs J_2 and J_3.

State v_5. If core ϕ_1 becomes available before core ϕ_2, then J_4 can start its execution on ϕ_1. This results in state v_5 (Fig. 1-(f)). The core intervals of v_5 are obtained as follows. According to the intervals of v_4, the earliest time at which ϕ_1 becomes available is 6, while the release time of J_4 is 8, thus, the earliest start time of J_4 on core ϕ_1 is 8, which means that its earliest finish time is 10. The latest start time of J_4 such that it is still scheduled on core ϕ_1 is time 12. The reason is that J_4 is released at time 8 and hence is pending from that time onward. However, it cannot be scheduled until a core becomes available. The earliest time a core among ϕ_1 and ϕ_2 becomes available is at time 12 (which is the latest finish time of J_3). Since the scheduling algorithm is work-conserving, it will certainly schedule job J_4 at 12 on the core that has become available. Consequently, the latest finish time of J_4 is $12 + 3 = 15$.

State v_6. In state v_4, if core ϕ_2 becomes available before ϕ_1, then job J_4 can be scheduled on ϕ_2 and create state v_6 (Fig. 1-(g)). In this case, the earliest start time of J_4 is at time 10 because, although it has been released before, it must wait until core ϕ_2 becomes available, which happens only at time 10. As a result, the earliest finish time of J_4 will be time $10 + 2 = 12$. On the other hand, the latest start time of J_4 such that it is scheduled on core ϕ_2 is 12 because at this time, job J_4 is ready and a core (ϕ_1) becomes available. Thus, if J_4 is going to be scheduled on ϕ_2, core ϕ_2 must become available by time 12. Note that since our core-selection policy is non-deterministic, if ϕ_2 becomes available at time 12, J_4 may be dispatched on either core. Consequently, the latest finish time of J_4 when scheduled on ϕ_2 is $12 + 3 = 15$. Furthermore, system state v_6 may arise only if core ϕ_1 has not become available before time 10, as otherwise job J_4 will be scheduled on ϕ_1 and create state v_5. Thus, state v_6 can be reached only if ϕ_1 does not become available before time 10. To reflect this constraint, the core interval of ϕ_1 must be updated to $[10, 12]$. The red dashed rectangle in Fig. 1-(g) illustrates this update. According to the schedule-abstraction graph in Fig. 1-(b), there exist three scenarios in which J_5 finishes at time 16 and hence misses its deadline. These scenarios are shown in Figs. 1-(h), (i) and (k), and are reflected in states v_7, v_8, and v_{10}, respectively.

4 Schedulability Analysis

This section explains how to build the schedule-abstraction graph. Sec. 4.1 presents the high-level description of our search algorithm, which consists of alternating *expansion*, *fast-forward*, and *merge* phases. These phases will be discussed in details in Sec. 4.2, 4.3, and 4.4, respectively. Sec. 5 provides a proof of correctness of the proposed algorithm.

4.1 Graph-Generation Algorithm

During the expansion phase, (one of) the shortest path(s) P in the graph from the root to a leaf vertex v_p is expanded by considering all jobs that can possibly be chosen by the JLFP scheduler to be executed next in the job execution sequence represented by P. For each such job, the algorithm checks on which core(s) it may execute. Finally, for each core on which the job may execute, a new vertex v'_p is created and added to the graph, and connected via an edge directed from v_p to v'_p.

After generating a new vertex v'_p, the fast-forward phase advances time until the next scheduling event. It accordingly updates the system state represented by v'_p.

The merge phase attempts to moderate the growth of the graph. To this end, the terminal vertices of paths that have the same set of scheduled jobs (but not necessarily in the same order) and core states that will lead to similar future scheduling decisions by the scheduler, are merged into a single state whose future states cover the set of all future states of the merged states. The fast-forward and merge phases are essential to avoid redundant work, i.e., to recognize that two or more states are similar early on before they are expanded. The algorithm terminates when there is no vertex left to expand, that is, when all paths in the graph represent a valid schedule of all jobs in $\mathcal{J}$.

Algorithm 1 presents our iterative method to generate the schedule-abstraction graph in full detail. A set of variables keeping track of a lower bound on the BCRT and an upper bound on the WCRT of each job is initialized at line 1. These bounds are updated whenever a job J_i can possibly be scheduled on any of the cores. The graph is initialized at line 2 with a root vertex v_1. The expansion phase corresponds to lines 6–21; line 13 implements the fast-forward, and lines 14–18 realize the merge phase. These phases repeat until every path in the graph contains $|\mathcal{J}|$ distinct jobs. We next discuss each phase in detail.

4.2 Expansion Phase

Assume that P is a path connecting the initial state v_1 to v_p. The sequence of edges in P represents a sequence of scheduling decisions (i.e., a possible sequence of job executions) to reach the system state modeled by v_p from the initial state v_1. We denote by $\mathcal{J}^P$the set of jobs scheduled in path P. To expand path P, Algorithm 1 evaluates for each job $J_i \in \mathcal{J} \setminus \mathcal{J}^P$ that was not scheduled yet whether it may be the next job picked by the scheduler and scheduled on any of the cores. For any job J_i that can possibly be scheduled on a core $\phi_k \in v_p$ *before any other job starts executing*, a new vertex v'_p is added to the graph (see lines 6–12 of Algorithm 1).

To evaluate if job J_i is a potential candidate for being started next in the dispatch sequence represented by P, we need to know:

1. The earliest time at which J_i may start to execute on core ϕ_k when the system is in the state described by vertex v_p. We call that instant the *earliest start time* (EST) of J_i on core ϕ_k, and we denote it by $EST_{i,k}(v_p)$.
2. The time by which J_i must have certainly started executing if it is to be the *next job* to be scheduled by the JLFP scheduler on the processing platform. This second time instant is referred to as the *latest start time* (LST) of J_i and is denoted by $LST_i(v_p)$.

$LST_i(v_p)$ represents the latest time at which a work-conserving JLFP scheduler schedules J_i next after state v_p. Note that $LST_i(v_p)$ is a global value for the platform when it is in state v_p, while $EST_{i,k}(v_p)$ is related to a specific core ϕ_k.

Algorithm 1: Schedule Graph Construction Algorithm.

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

$\forall J_i \in \mathcal{J}, BCRT_i \leftarrow \infty, WCRT_i \leftarrow 0$;
Initialize G by adding a root vertex $v_1 = \{[0,0],[0,0],\ldots,[0,0]\}$, where $|v_1| = m$;
while $\exists$ *a path* P *from* v_1 *to a leaf vertex* v_p *s.th.* $|P| < |\mathcal{J}|$ **do**
  $P \leftarrow$ a path from v_1 to a leaf with the least number of edges in the graph;
  $v_p \leftarrow$ the leaf vertex of P;
  for *each job* $J_i \in \mathcal{J} \setminus \mathcal{J}^P$ **do**
    for *each core* $\phi_k \in v_p$ **do**
      if J_i *can be dispatched on core* ϕ_k *according to* (1) **then**
        Build v'_p using (10);
        $BCRT_i \leftarrow \min\{EFT'_k - r_i^{min}, BCRT_i\}$;
        $WCRT_i \leftarrow \max\{LFT'_k - r_i^{min}, WCRT_i\}$;
        Connect v_p to v'_p by an edge with label J_i;
        Fast-forward v'_p according to (13);
        while $\exists$ *path* Q *that ends to* v_q *such that the condition defined in Definition 4 is satisfied for* v'_p *and* v_q **do**
          Update v'_p using Algorithm 2;
          Redirect all incoming edges of v_q to v'_p;
          Remove v_q from V;
        end
      end
    end
  end
end

A job J_i can be the next job scheduled in the job sequence represented by P if there is a core ϕ_k for which the earliest start time $EST_{i,k}(v_p)$ of J_i on ϕ_k is not later than the latest time at which this job must have started executing, i.e., before $LST_i(v_p)$ (see Lemma 7 in Sec. 5 for a formal proof). That is, J_i may commence execution on ϕ_k only if

$$EST_{i,k}(v_p) \leq LST_i(v_p). \tag{1}$$

For each core ϕ_k that satisfies (1), a new vertex v'_p is created, where v'_p represents the state of the system after dispatching job J_i on core ϕ_k.

Below, we explain how to compute $EST_{i,k}(v_p)$ and $LST_i(v_p)$. Then we describe how to build a new vertex v'_p for each core ϕ_k and job J_i that satisfies (1). Finally, we explain how the BCRT and WCRT of job J_i are updated according to its $EST_{i,k}(v_p)$ and $LST_i(v_p)$, respectively. To ease readability, from here on we will not specify any more that ϕ_k, $EST_{i,k}(v_p)$ and $LST_i(v_p)$ are related to a specific vertex v_p when it is clear from context, and will instead use the short-hand notations $EST_{i,k}$ and LST_i.

Earliest start time. To start executing on a core ϕ_k, a job J_i has to be released and ϕ_k has to be available. Thus, the earliest start time $EST_{i,k}$ of a job J_i on a core ϕ_k is given by

$$EST_{i,k} = \max\{r_i^{min}, EFT_k\}, \tag{2}$$

where r_i^{min} is the earliest time at which J_i may be released and EFT_k is the earliest time at which ϕ_k may become available.

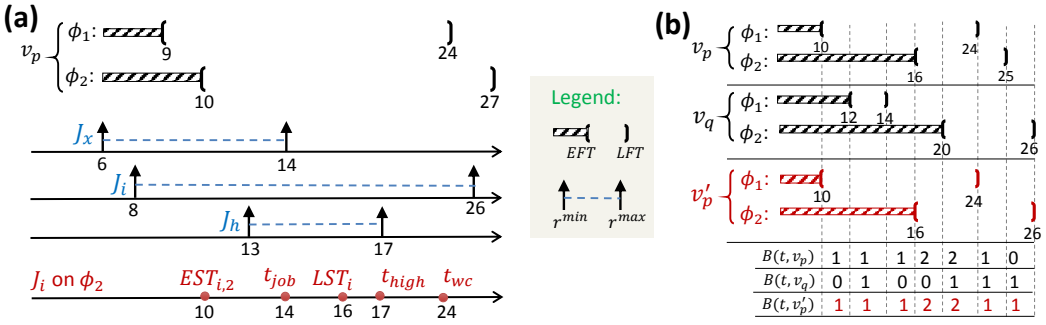


Figure 2 **(a)** Expansion scenario for J_i and ϕ_2, where $p_h < p_i < p_x$. **(b)** An example merge.

Latest start time. Because we assume a work-conserving JLFP scheduling algorithm, two conditions must hold for job J_i be the *next job* scheduled on the processing platform: **(i)** J_i must be the highest-priority ready job (because of the JLFP assumption), and **(ii)** for every job J_j released before J_i, either J_j was already scheduled earlier on path P (i.e., $J_j \in \mathcal{J}^P$), or all cores were busy from the release of J_j until the release of J_i.

If (i) is not satisfied, then a higher-priority ready job is scheduled instead of J_i. Therefore the latest start time LST_i of J_i must be earlier than the earliest time at which a not-yet-scheduled higher-priority job is certainly released, that is, $LST_i < t_{high}$, where

$$t_{high} = \min_{\infty}\{r_x^{max} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P \ \wedge \ p_x < p_i\}. \tag{3}$$

If (ii) is not satisfied, then an earlier released job J_j will start executing on an idle core before J_i is released. Therefore the latest start time LST_i of J_i cannot be later than the earliest time at which both a core is certainly idle and a not-yet-scheduled job is certainly released. Formally, $LST_i \leq t_{wc}$, where

$$t_{wc} \triangleq \max\{t_{core}, t_{job}\}, \tag{4}$$

$$t_{core} \triangleq \min\{LFT_x \mid 1 \leq x \leq m\}, \text{ and} \tag{5}$$

$$t_{job} \triangleq \min_{\infty}\{r_y^{max} \mid J_y \in \mathcal{J} \setminus \mathcal{J}^P\}. \tag{6}$$

In the equations above, t_{core} is the earliest time at which a core is certainly idle and t_{job} is the earliest time at which a not-yet-scheduled job is certainly released.

Combining $LST_i < t_{high}$ and $LST_i \leq t_{wc}$, we observe that J_i must start by time

$$LST_i = \min\{t_{wc}, \ t_{high} - 1\}. \tag{7}$$

▶ **Example 2.** Fig. 2-(a) shows how $EST_{i,k}$ and LST_i are calculated when job J_i is scheduled on core ϕ_2. In this example, $t_{job} = 14$ since job J_x becomes certainly available at that time. However, the earliest time at which a core (in this case, core ϕ_1) becomes available is $t_{core} = 24$, thus, $t_{wc} = 24$. On the other hand, the earliest time at which a job with a higher priority than J_i is certainly released is $t_{high} = 17$. Thus, $LST_i = t_{high} - 1 = 16$.

Building a new system state. If Inequality (1) holds, it is possible that job J_i is the next successor of path P and is scheduled on core ϕ_k at any $t \in [EST_{i,k}, LST_i]$ (Lemma 7 in Sec. 5 proves this claim). Our goal is to generate a single new vertex for the schedule-abstraction graph that aggregates all these execution scenarios.

Let v'_p denote the vertex that represents the new system state resulting from the execution of job J_i on core ϕ_k. The earliest and latest times at which ϕ_k may become available after executing job J_i is obtained as follows:

$$EFT'_k = EST_{i,k} + C_i^{min} \quad \text{and} \quad LFT'_k = LST_i + C_i^{max}. \tag{8}$$

Furthermore, because the latest scheduling event in the system state v'_p occurs no earlier than $EST_{i,k}$, no other job in $\mathcal{J} \setminus \mathcal{J}^P$ may possibly be scheduled before $EST_{i,k}$.

▶ **Property 3.** *If job J_i is the next job scheduled on the platform, and if it is scheduled on core ϕ_k, then no job $\in \mathcal{J} \setminus \mathcal{J}^P$ starts executing on any core $\phi_x, 1 \leq x \leq m$ before $EST_{i,k}$.*

Proof. By contradiction. Assume a job $J_j \in \mathcal{J} \setminus \mathcal{J}^P$ starts executing on a core ϕ_x before $EST_{i,k}$. Because J_i cannot start executing on ϕ_k before $EST_{i,k}$, J_j must be different from J_i and hence J_j starts to execute before J_i. That contradicts the assumption that J_i is the first job in $\mathcal{J} \setminus \mathcal{J}^P$ to be scheduled on the platform. ◀

To ensure that Property 3 is correctly enforced in the new system state represented by v'_p, we update the core intervals in state v'_p as follows

$$\phi'_x \triangleq \begin{cases} [EFT'_k, LFT'_k] & \text{if } x = k, \\ [EST_{i,k}, EST_{i,k}] & \text{if } x \neq k \ \wedge \ LFT_x \leq EST_{i,k}, \\ [\max\{EST_{i,k}, EFT_x\}, LFT_x] & \text{otherwise.} \end{cases} \tag{9}$$

The first case of (9) simply repeats (8) for job J_i. The second and third cases ensure that no job in $\mathcal{J} \setminus \mathcal{J}^P$ can be scheduled on those cores before $EST_{i,k}$. This is done by forcing ϕ_x's earliest availability time to be equal to $EST_{i,k}$. Finally, for cores that would certainly be idle after $EST_{i,k}$ (i.e., the second case in (9)), we set LFT_k (i.e., the time at which it becomes certainly available) to $EST_{i,k}$.

Finally, the new vertex v'_p is generated by applying (9) on all cores, i.e.,

$$v'_p = \{\phi'_1, \phi'_2, \ldots, \phi'_m\}. \tag{10}$$

Deriving the BCRT and WCRT of the jobs. Recall that the BCRT and the WCRT of a job are relative to its *arrival time*, i.e., r_i^{min}, and not its actual *release time*, which can be any time between r_i^{min} and r_i^{max}. In other words, release jitter counts towards a job's response time. As stated earlier, the earliest finish time of J_i on core ϕ_k cannot be smaller than EFT'_k and the latest finish time of J_i on core ϕ_k cannot be larger than LFT'_k (obtained from (8)). Using these two values, the BCRT and WCRT of job J_i are updated at lines 10 and 11 of Algorithm 1 as follows.

$$BCRT_i \leftarrow \min\{EFT'_k - r_i^{min}, BCRT_i\} \tag{11}$$

$$WCRT_i \leftarrow \max\{LFT'_k - r_i^{min}, WCRT_i\} \tag{12}$$

If the algorithm terminates, then $WCRT_i$ and $BCRT_i$ contain an upper bound on the WCRT and a lower bound on the BCRT of job J_i, respectively, over all paths. Since the graph considers all possible execution scenarios of $\mathcal{J}$, it considers all possible schedules of J_i. The resulting WCRT and BCRT estimates are therefore safe bounds on the actual WCRT and BCRT of the job, respectively. This property is proven in Corollary 18 in Sec. 5.

The quality of service of many real-time systems depends on both the WCRT and response-time jitter [7] of each task, i.e., the difference between the BCRT and WCRT of that

Algorithm 2: Algorithm that merges v_p and v_q, and creates v'_p.

1 Sort and re-index the core intervals $\phi_k(v_p)$ of v_p in a non-decreasing order of their EFTs, such that $EFT_1(v_p) \leq EFT_2(v_p) \leq \dots EFT_m(v_p)$;
2 Sort and re-index v_q's core intervals in a non-decreasing order of their EFTs such that $EFT_1(v_q) \leq EFT_2(v_q) \leq \dots EFT_m(v_q)$;
3 Pair each two core intervals $\phi_x(v_p)$ and $\phi_x(v_q)$ to create $\phi_x(v'_p) \triangleq [\min\{EFT_x(v_p), EFT_x(v_q)\}, \max\{LFT_x(v_p), LFT_x(v_q)\}]$;

task. One of the advantages of our schedule-abstraction graph is that it not only provides a way to compute those quantities, but also allows to extract the maximum variation between the response times of successive jobs released by the same task, hence allowing a more accurate analysis of (for instance) sampling jitter in control systems.

4.3 Fast-Forward Phase

As shown in lines 6 and 7, one new state will be added to the graph for each not-yet-scheduled job that can be scheduled next on one of the cores. This situation can lead to an explosion in the search space if the number of states is not reduced. In this work, we merge states to avoid redundant future explorations. To aid the subseqent merge phase, the fast-forward phase advances the time until a job may be released. We denote that instant by $t_{min} \triangleq \min_\infty \left\{ r_x^{min} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P \setminus \{J_i\} \right\}$. The fast-forward phase thus updates each core interval $\phi'_x \in v'_p$ as follows:

$$\phi'_x = \begin{cases} [t_{min}, t_{min}] & LFT_x \leq t_{min}, \\ [\max\{t_{min}, EFT_x\}, LFT_x] & \text{otherwise.} \end{cases} \tag{13}$$

The first case of (13) relies on the fact that from LFT'_x onward (i.e., the time at which a core ϕ'_x becomes certainly available), ϕ'_x remains available until a new job is scheduled on it. Since the earliest time at which a job can be scheduled is t_{min}, this core remains available at least until t_{min}. Thus, it is safe to update its interval to $[t_{min}, t_{min}]$, which denotes that the core is certainly free by t_{min}. Similarly, the second case of (13) is based on the fact that a core ϕ_x that is possibly available at EFT'_x remains possibly available either until reaching LFT'_x (where it certainly becomes free) or until a job may be scheduled on ϕ_x, which does not happen until t_{min} at the earliest. Lemma 9 in Sec. 5 proves that fast-forwarding state v'_p will not change any of the future states that can be reached from v'_p before applying (13).

4.4 Merge Phase

The merge phase seeks to collapse states to avoid redundant future explorations. The goal is to reduce the size of the search space such that the computed BCRT of any job may never become larger, the computed WCRT of any job may never become smaller, and all job scheduling sequences that were possible before merging states are still considered after merging those states. The merge phase is implemented in lines 14–18 of Algorithm 1, where the condition defined below in Definition 4 is evaluated for paths with length $|P| + 1$.

Since each state consists of exactly m core intervals, merging two states requires finding a matching among the two sets of intervals to merge individual intervals. Let states v_p and v_q be the end vertices of two paths P and Q. In order to merge v_p and v_q into a new state v'_p, we apply Algorithm 2. Next, we establish our merging rules, which will be proven to be safe in Corollary 15 in Sec. 5.

▶ **Definition 4.** Two states v_p and v_q can be merged if **(i)** $\mathcal{J}^P = \mathcal{J}^Q$, **(ii)** $\forall\ \phi_i(v_p), \phi_i(v_q)$, $\max\{EFT_i(v_p), EFT_i(v_q)\} \leq \min\{LFT_i(v_p), LFT_i(v_q)\}$, and **(iii)** at any time t, the number of possibly-available cores in the merged state must be equal to the number of possibly-available cores in v_p or v_q, i.e.,

$$\forall t \in T, B(t, v'_p) = B(t, v_p) \ \vee \ B(t, v'_p) = B(t, v_q), \tag{14}$$

where $B(t, v_x)$ counts the number of core intervals of a state v_x that contain t, i.e.,

$$B(t, v_x) = \left| \left\{ \phi_y(v_x) \mid t \in \left[EFT_y(v_x), LFT_y(v_x) \right] \right\} \right|, \tag{15}$$

and where T is the set of time instants at which the value of $B(\cdot)$ may change, i.e.,

$$T = \{EFT_x(v_p)|\ \forall x\} \ \cup \ \{LFT_x(v_p)|\ \forall x\} \ \cup \ \{EFT_x(v_q)|\ \forall x\} \ \cup \ \{LFT_x(v_q)|\ \forall x\}. \tag{16}$$

▶ **Example 5.** Fig. 2-(b) shows two states v_p and v_q that are merged to create state v'_p. As shown, for any $t \in T$, $B(t, v'_p)$ is equal to $B(t, v_p)$ or $B(t, v_q)$.

Notably, any merge rule that respects condition (i) in Definition 4 is *safe* (see Corollary 1 in Sec. 5.3). The role of conditions (ii) and (iii) is to trade-off between the accuracy and performance of the analysis by evading the inclusion of impossible execution scenarios in the resulting state. We leave the investigation of more accurate (or more eager) merging conditions, as well as the applicability of abstraction-refinement techniques, to future work.

5 Correctness of the Proposed Solution

In this section, we show that the schedule-abstraction graph constructed by Algorithm 1 correctly includes all job schedules that can arise from any possible execution scenario, i.e., for any possible execution scenario, there exists a path in the graph that represents the schedule of those jobs in that execution scenario (Theorem 17). The proof has two main steps: we first assume that the fast-forward and merge phases are not executed and show that the EFT and LFT of a job obtained from Equation (8) are correct lower and upper bounds on the finish time of a job scheduled on a core (Lemma 6) and that for an arbitrary vertex v_p, Inequality (1) is a necessary condition for a job to be scheduled next on core ϕ_k (Lemma 7). From these lemmas, we conclude that without fast-forwarding and merging, for any execution scenario there exists a path in the schedule graph that represents the schedule of the jobs in that execution scenario (Lemma 8).

In the second step, we show that the fast-forward and merge phases are *safe*, i.e., these phases will not remove any potentially reachable state from the original graph (Lemma 9 and Corollary 16). Finally, we establish that Algorithm 1 correctly derives an upper bound on the WCRT and a lower bound on the BCRT of every job (Corollary 18).

5.1 Soundness of the Expansion Phase

In this section, we assume that neither the fast-forward nor the merge phase is executed.

▶ **Lemma 6.** *For any vertex $v_p \in V$ and any successor v'_p of v_p such that job $J_i \in \mathcal{J} \setminus \mathcal{J}^P$ is scheduled on core ϕ_k between v_p and v'_p, $EFT_k(v'_p)$ and $LFT_k(v'_p)$ (as computed by (8)) are a lower bound and an upper bound, respectively, on the completion time of J_i.*

Proof. If neither the fast-forward nor the merge phases are executed, (9) is the only equation used to build a new state v'_p. In this lemma, we first prove that the EST and LST of the job obtained from (2) and (7) are a lower and an upper bound on the start time of job J_i on ϕ_k after the scheduling sequence represented by P. Then, we conclude that $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are safe bounds on the finish time of J_i on ϕ_k. The proof is by induction.

Base case. The base case is for any vertex v'_p that succeeds to the root vertex v_1 where all cores are idle. Hence in v'_p, job J_i is scheduled on one of the idle cores, say ϕ_k. Since all cores are idle at time 0, Equation (2) yields $EST_{i,k}(v_1) = r_i^{min}$, which is by definition the earliest time at which job J_i may start. Consequently, the earliest finish time of J_i cannot be smaller than $EFT_k(v'_p) = r_i^{min} + C_i^{min}$.

Similarly, (7) yields $LST_i(v_1) = \min\{t_{high} - 1, t_{job}\}$ (recall that $t_{core} = 0$ since all cores are idle in v_1). J_i cannot start later than $LST_i(v_1) = t_{job}$ if it is the first scheduled job as all cores are idle and hence as soon as a job is certainly released, it will be scheduled right away on one of the idle cores. Similarly, J_i cannot start its execution if it is not the highest-priority job anymore, i.e., at or after time t_{high}. As a result, the latest finish time of J_i cannot be larger than $LFT_k(v'_p) = \min\{t_{job}, t_{high} - 1\} + C_i^{max}$. Therefore, $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are safe bounds on the finishing time of J_i on ϕ_k after the scheduling sequence $P = \langle v_1, v'_p \rangle$.

For all other cores ϕ_x such that $x \neq k$, (9) enforces that $EFT_x(v'_p) = LFT_x(v'_p) = EST_{i,k}(v_1) = r_i^{min}$ (recall that $EFT_k(v_1) = LFT_k(v_1) = 0$), which is indeed the earliest time at which any job may start on ϕ_x if J_i is the first job executing on the platform and J_i is not released before r_i^{min}.

Induction step. Assume now that each core interval on every vertex from v_1 to v_p along path P provides a lower bound and an upper bound on the time at which that core will possibly and certainly be available, respectively, to start executing a new job. We show that in the new vertex v'_p obtained from scheduling job J_i on core ϕ_k after P, (8) provides a safe lower and upper bound on the finish time of J_i, and for other cores, the new core intervals computed by (9) are safe, i.e., no new job can start its execution on a core ϕ_x before EFT_x and the core cannot remain busy after LFT_x.

EFT. The earliest start time of J_i on core ϕ_k, i.e., $EST_{i,k}(v_p)$, cannot be smaller than $EFT_k(v_p)$ since, by the induction hypothesis, $EFT_k(v_p)$ is the earliest time at which core ϕ_k may start executing a new job. Moreover, a lower bound on $EST_{i,k}(v_p)$ is given by r_i^{min}, because J_i cannot execute before it is released. This proves (2) for ϕ_k. Further, if J_i starts its execution at $EST_{i,k}(v_p)$, it cannot finish before $EST_{i,k}(v_p) + C_i^{min}$ since its minimum execution time is C_i^{min}. Thus, the EFT of job J_i on ϕ_k in system state v'_p cannot be smaller than $EST_{i,k}(v_p) + C_i^{min}$, which proves the correctness of (8) for $EFT_k(v'_p)$.

The EFTs of all other cores ϕ_x in v'_p cannot be smaller than $EFT_x(v_p)$ in state v_p since no new job is scheduled on them. Furthermore, according to Property 3, job J_i can be scheduled on core ϕ_k (instead of any other core) only if no other job in $\mathcal{J} \setminus \mathcal{J}^P$ has started executing on any other core than ϕ_k until $EST_{i,k}(v_p)$. Hence, $\max\{EST_{i,k}(v_p), EFT_x(v_p)\}$ is a safe lower bound on the EST of a job in state v'_p (as computed by (9)).

LFT. Next, we show that $LST_i(v_p)$ cannot exceed $t_{high} - 1$ or t_{wc} as stated by (7). First, consider t_{high} and suppose $t_{high} \neq \infty$ (otherwise the claim is trivial). Since a higher-priority job is certainly released at the latest at time t_{high}, job J_i is no longer the highest-priority job at time t_{high}. Consequently, it cannot commence execution under a JLFP scheduler at or after time t_{high} if it is to be the next job scheduled after P. Hence,

job J_i will be a direct successor of path P *only if* its execution starts no later than time $t_{high} - 1$. Now, consider t_{wc}. At time t_{wc}, a not-yet-scheduled job is certainly released and a core is certainly available. Hence a work-conserving scheduler will schedule that job at t_{wc}, thus, job J_i will be a direct successor of path P *only if* its execution starts no later than time t_{wc}. Since $LST_i(v_p)$ is the upper bound on the time at which job J_i can start its execution while being the next job scheduled after path P, the latest finish time of J_i on core ϕ_k cannot be larger than $\min\{t_{high} - 1, t_{wc}\} + C_i^{max}$, which proves the correctness of (8) for $LFT_k(v'_p)$.
Since in state v'_p job J_i is scheduled on core ϕ_k other cores cannot be available before $EST_{i,k}$, otherwise a work-conserving scheduler would schedule J_i on one of those cores instead of on ϕ_k. Equation (9) ensures that if J_i is the next job to be scheduled and if ϕ_k is the core on which J_i is scheduled, no other core will *certainly* be available by $EST_{i,k}(v_p)$, i.e., $EFT_x(v'_p) \geq EST_{i,k}(v_p)$.
By induction on all vertices in V, we have that $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are safe bounds on the finish time of any job scheduled between any two states v_p and v'_p, including J_i. ◀

▶ **Lemma 7.** *Job J_i can be scheduled next on core ϕ_k after jobs in path P only if* (1) *holds.*

Proof. If job J_i is released at time r_i^{min} and the core ϕ_k becomes available at EFT_k, then it can be dispatched no earlier than at time $EST_{i,k} = \max\{r_i^{min}, EFT_k\}$. If (1) does not hold, then t_{high} or t_{wc} (or both) are smaller than $EST_{i,k}$. This implies that either a higher-priority job other than job J_i is certainly released before $EST_{i,k}$ or a job other than J_i is certainly released before $EST_{i,k}$ and a core is certainly available before $EST_{i,k}$. In both cases, a work-conserving JLFP scheduling algorithm will not schedule job J_i until that other job is scheduled. Consequently, job J_i cannot be the next successor of path P. ◀

▶ **Lemma 8.** *Assuming that neither the fast-forward nor the merge phases are executed in Algorithm 1, for any execution scenario such that a job $J_i \in \mathcal{J}$ completes at some time t on core ϕ_k (under the given scheduler), there exists a path $P = \langle v_1, \ldots, v_p, v'_p \rangle$ in the schedule-abstraction graph such that J_i is the label of the edge from v_p to v'_p and $t \in [EFT_k(v'_p), LFT_k(v'_p)]$, where $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are given by Equation* (8).

Proof. Since Algorithm 1 creates a new state in the graph for every job J_i and every core ϕ_k that respects Condition (1), the combination of Lemmas 6 and 7 proves that all possible system states are generated by the algorithm when the fast-forward and merge phases are not executed. Further, Lemma 6 proves that $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are safe bounds on the finishing time of J_i, meaning that if J_i finishes at t in the execution scenario represented by path P, then t is within $[EFT_k(v'_p), LFT_k(v'_p)]$. ◀

5.2 Soundness of the Fast-Forward Phase

We prove that fast-forwarding will not affect any of the successor states of an updated state.

▶ **Lemma 9.** *Updating the core intervals of vertex v_p during the fast-forwarding phase does not affect any of the states reachable from v_p.*

Proof. Let v_p be the original state and v_q be the updated state after applying (13). Let path P denote the path from v_1 to v_p. Note that state v_q shares the same path P as v_p. We show that for any arbitrary job $J_i \in \mathcal{J} \setminus \mathcal{J}^P$ (i.e., those that are not scheduled in path P) and any arbitrary core $\phi_k(v_p) \in v_p$, the EST and LST of job J_i is the same as for core

$\phi_k(v_q) \in v_q$. From this we conclude that all system states reachable from v_p are reachable from v_q and that those reachable states remain unchanged. More precisely, we show that, $\forall k$, **(i)** $EST_{i,k}(v_p) = EST_{i,k}(v_q)$ and **(ii)** $LST_k(v_p) = LST_k(v_q)$.

Claim (i). From (2), we have $EST_{i,k}(v_p) = \max\{r_i^{min}, EFT_k(v_p)\}$. If the EFT of $\phi_k(v_q)$ has not been updated by (13), i.e., $EFT_k(v_p) > t_{min}$, then we trivially have $EST_{i,k}(v_q) = EST_{i,k}(v_p)$. Otherwise, if $EFT_k(v_q)$ has been updated, it must be true that $EFT_k(v_p) \leq t_{min}$ and $EFT_k(v_q) = t_{min}$. In this case, $EST_{i,k}(v_q) = \max\{r_i^{min}, t_{min}\} = \max\{r_i^{min}, EFT_k(v_p)\} = EST_{i,k}(v_p)$ since $EFT_k(v_p) \leq t_{min} \leq r_i^{min}$ (from the definition of t_{min}). Thus, in both cases, $EST_{i,k}(v_p) = EST_{i,k}(v_q)$.

Claim (ii). From (13) we know that if the LFT of a core $\phi_k(v_p)$ is being updated, $LFT_k(v_p) < t_{min}$ and $LFT_k(v_q) = t_{min}$. By definition, $t_{min} = \min\{r_x^{min} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P\} \leq \min\{r_x^{max} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P\} = t_{job}(v_p)$ (the last equality is due to (6)). Moreover, by (5) we have $t_{core}(v_p) \leq LFT_k(v_p) < LFT_k(v_q) = t_{min} \leq t_{job}(v_p)$ and $t_{core}(v_q) \leq LFT_k(v_q) = t_{min} \leq t_{job}(v_q)$ (because t_{job} only depends on path P and v_p and v_q share the same path). Therefore, by (7), $LST_k(v_p) = \min\{t_{high}(v_p) - 1,\ \max\{t_{job}(v_p),\ t_{core}(v_p)\}\} = \min\{t_{high}(v_p) - 1, t_{job}(v_p)\}$ and $LST_k(v_q) = \min\{t_{high}(v_q) - 1,\ \max\{t_{job}(v_q),\ t_{core}(v_q)\} = \min\{t_{high}(v_q) - 1, t_{job}(v_q)\}$. Since t_{job} and t_{high} only depend on path P, and v_p and v_q share the same path, the LST in both states is identical, i.e., $LST_k(v_p) = LST_k(v_q)$. ◀

5.3 Soundness of the Merge Phase

We now establish that merging two states is safe, i.e., it neither removes a possible job sequence from the graph (Corollary 16), nor does it decrease the upper bound on the WCRT (or increase the lower bound on the BCRT) of any job in $\mathcal{J}$ (Corollary 18).

We first define the notion of a "mutated" vertex as follows: v'_p is a *mutated* version of v_p if it has the same set of scheduled jobs as the original state v_p and $\forall x$, $EFT_x(v'_p) \leq EFT_x(v_p)$ and $\forall x$, $LFT_x(v_p) \leq LFT_x(v'_p) \vee LFT_x(v_p) \leq t_{job}(v_p)$. We assume that a mutated state v'_p sits in place of the original state v_p in the schedule-abstraction graph.

Next, for any such mutated vertex, we prove that any job that was a direct successor of the original state is also a direct successor of the mutated vertex (Lemma 10). Moreover, we show that the direct successors of mutated states are also mutated (Lemma 11 and 12). This property is then used to prove the main claim that merging is safe. Due to space limitations, we provide the proofs of Lemmas 10 to 14 in an online technical report [19].

▶ **Lemma 10.** *For a vertex v'_p created by mutating v_p, any job J_i that can be scheduled on core $\phi_k(v_p)$ according to* (1)*, can still be scheduled on core $\phi_k(v'_p)$ according to* (1)*.*

▶ **Lemma 11.** *Let v'_p be created by mutating v_p, and let v_q and v'_q be the vertices resulting from scheduling job J_i on core $\phi_k(v_p)$ and $\phi_k(v'_p)$, respectively. $\forall x$, $LFT_x(v'_q) \geq LFT_x(v_q)$ or $LFT_x(v_q) \leq t_{job}(v_q)$.*

▶ **Lemma 12.** *Let v'_p be created by mutating v_p, and let v_q and v'_q be the vertices resulting from scheduling job J_i on core $\phi_k(v_p)$ and $\phi_k(v'_p)$, respectively. $\forall x$, $EFT_x(v'_q) \leq EFT_x(v_q)$.*

▶ **Lemma 13.** *If v'_p is a vertex created by mutating v_p, then all the system states reachable from v_p are also reachable from v'_p.*

▶ **Lemma 14.** *Let v_q and v_p be two vertices such that $\mathcal{J}^P = \mathcal{J}^Q$ (i.e., the set of jobs scheduled until reaching v_q is equal to the set of jobs scheduled until reaching v_p), then the state v'_p resulting from merging v_p and v_q with Algorithm 2 is a mutated version of both v_p and v_q.*

By successively applying Lemmas 13 and 14, we obtain the following corollary.

▶ **Corollary 15.** *Let v_q and v_p be two vertices such that $\mathcal{J}^P = \mathcal{J}^Q$ (i.e., the set of jobs scheduled until reaching v_q is equal to the set of jobs scheduled until reaching v_p), all system states reachable from v_p and v_q are also reachable from the merged state v'_p.*

▶ **Corollary 16.** *For two states that are merged by Algorithm 1, all system states reachable from either of them are also reachable from the merged state.*

Proof. Since for two states v_p and v_q, Definition 4 enforces that $\mathcal{J}^P = \mathcal{J}^Q$, the resulting merged state satisfies the requirement of Corollary 15 and hence proves the claim. ◀

5.4 Soundness of Algorithm 1

By successively applying Lemmas 8 and 9 and then Corollary 16, we obtain that the analysis is safe, as stated in Theorem 17 and its corollary below.

▶ **Theorem 17.** *For any execution scenario such that a job $J_i \in \mathcal{J}$ completes at some time t on core ϕ_k (under the given scheduler), there exists a path $P = \langle v_1, \ldots, v_p, v'_p \rangle$ in the schedule-abstraction graph such that J_i is the label of the edge from v_p to v'_p and $t \in [EFT_k(v'_p), LFT_k(v'_p)]$, where $EFT_k(v'_p)$ and $LFT_k(v'_p)$ are given by Equation (8).*

▶ **Corollary 18.** *Lines 10 and 11 of Algorithm 1 calculate a lower and an upper bound on the BCRT and WCRT, respectively, of every job in $\mathcal{J}$.*

Proof. Lines 10 and 11 obtain a job's response time directly from (8), which provides correct bounds on the earliest and latest finish times of a job according to Lemma 6. Since according to Theorem 17, for any execution scenario, there is a path in the graph, Algorithm 1 includes all possible schedules of a job and hence the obtained values are correctly lower-bounding and upper-bounding the actual BCRT and WCRT of that job. ◀

5.5 Inexactness of Algorithm 1

The following example shows that the abstraction that we use to represent core states may reflect impossible execution scenarios. Therefore, Algorithm 1 is sufficient but not exact.

Assume that a system state v_p contains two core intervals $\phi_1 = [5, 10]$ and $\phi_2 = [1, 10]$ and that there is an unscheduled job J_1 with $C_1^{min} = C_1^{max} = 5$, $r_1^{min} = r_1^{max} = 1$, and $d_1 = 30$. Further, assume that during the expansion phase of Algorithm 1, J_1 is dispatched to ϕ_1, which results in $\phi_1 = [10, 15]$ and $\phi_2 = [5, 10]$ (after the update phase). According to this new system state, it may happen that core ϕ_2 becomes available at time $5 \in [5, 10]$, and that core ϕ_1 remains busy until time $15 \in [10, 15]$. However, this scenario is actually impossible. If ϕ_1 remains busy until time 15, then J_1 must have started to execute at time 10, implying that both ϕ_1 and ϕ_2 must have been busy until time 10. Otherwise, job J_1 would have been dispatched on ϕ_2 rather than ϕ_1. In other words, ϕ_1 may become available at time 15 only if ϕ_2 becomes available no earlier than time 10. This example shows a dependency between the availability time of the cores, which is ignored in the current system state abstraction to keep the system state encoding simple, and to increase the number of states that can be merged. This design decision, however, makes the analysis inexact since it considers all possible but also some impossible execution scenarios.

6 Empirical Evaluation

We conducted experiments to answer two main questions: **(i)** does our test yield better schedulability; and **(ii)** is the runtime of our analysis practical? To answer the first question, we applied Algorithm 1 to two global non-preemptive scheduling policies: G-NP-FP and G-NP-EDF. As we are unaware of any schedulability analysis for non-preemptive job sets (or periodic tasks) for the aforementioned global scheduling policies, we used the existing tests designed for sporadic non-preemptive task sets as a baseline. These tests include the schedulability test of Baruah [4] for G-NP-EDF (denoted by Baruah-EDF), two tests of Guan et al. [10] for any global non-preemptive work-conserving scheduler (denoted by Guan-Test1-WC), and for G-NP-FP (denoted by Guan-Test2-FP), and the recent schedulability test of Lee (denoted by Lee-FP) [13]. For the sake of comparison, we used simple rate-monotonic priorities for the fixed-priority tests since we did not observe substantial differences when trying out other heuristics such as laxity-monotonic priorities.

To randomly generate a periodic task set with n tasks and a given utilization U, we first randomly generated n period values in the range [10000, 100000] microseconds with log-uniform distribution (and a granularity of $5000\mu s$ as suggested by Emberson et al. [8]. We then used the RandFixSum [22] algorithm to generate n random task-utilization values that sum to U. From the task utilization, we obtained C_i^{max} and set C_i^{min} to be $0.1 \cdot C_i^{max}$. Tasks were assumed to have implicit deadlines. We discarded any task set that had more than 100000 jobs per hyperperiod. Although, in theory, a hyperperiod may contain many more jobs, in industrial settings, e.g., automotive systems [12], periods are usually chosen such that the hyperperiod includes only at most a couple of thousand jobs.

The experiments were performed by varying **(i)** the total system utilization U (for 4 cores and 10 tasks), **(ii)** the number of tasks n (for 4 cores and $U = 2.8$, which is 70% of the capacity of the cores), **(iii)** the number of cores m (for 10 tasks and $U = 2.8$), and **(iv)** the total task utilization U while tasks had 100 microseconds release jitter (10 tasks and 4 cores). This roughly represents jitter magnitudes that can be expected due to interrupt handling delays. For each combination of n, m, and U, 1000 random task sets were generated.

To evaluate schedulability of a task set, we implemented Algorithm 1 as a single-threaded C++ program and performed the analysis on a cluster of hosts having an Intel Xeon E7-8857 v2 processor clocked at 3 GHz and 1.5 TiB RAM. In the experiments, a task set was claimed unschedulable as soon as either an execution scenario with a deadline miss was found or a timeout of four hours was reached. Fig. 3 reports the observed schedulability ratio and runtime of Algorithm 1 for different setups. The schedulability ratio is the ratio of task sets deemed to be schedulable divided by the number of generated task sets.

Schedulability results. Figs. 3-(a) to (c) show a significant gap between the schedulability ratio of our solution and the state-of-the-art tests. For example, while Lee-FP could only identify 8% of schedulable task sets for $U = 2.4$, our test shows that at least 72% of them are schedulable. Similar patterns are seen when for increasing task and core counts. Fig. 3-(b) shows that schedulability improves as the number of tasks increases. This is since, by keeping U constant, increasing n decreases per-task utilization, which in turn reduces WCETs and blocking. Thus, more task sets become schedulable. Of the existing tests, however, only Lee-FP and Guan-Test2-FP benefit from this effect, and only by up to 16% (for $n = 30$).

With the increase in the number of cores, blocking scenarios caused by tasks with large execution times are less likely to occur and hence more task sets are deemed schedulable. However, as shown in Fig. 3-(c), the current tests are quite pessimistic, e.g., Lee-FP could

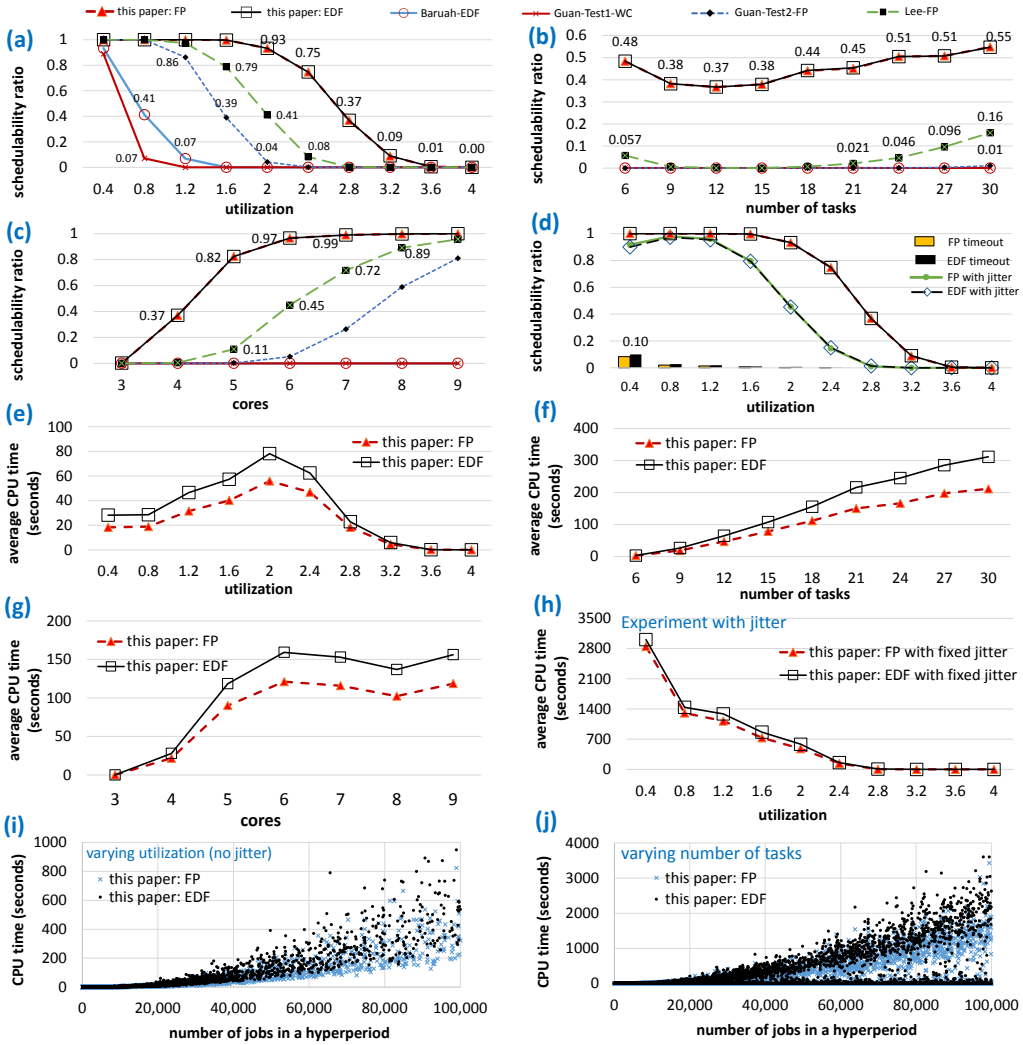


Figure 3 Experimental results for various parameters. **(a, b, c, d)** Schedulability ratio. **(e, f, g, h)** Average analysis runtime. **(i, j)** Analysis runtime vs. the number of jobs in a hyperperiod.

identify only 11% of the task sets as schedulable when (at least) 82% of the task sets are schedulable on 5 cores. From Figs. 3-(a) to (c), we conclude that our analysis is able to reclaim a large portion of pessimism in the baseline analyses (when applied to periodic tasks).

Fig. 3-(d) shows the effect of jitter on schedulability. Since jitter increases the number of possible interleavings between the start time of the tasks, more blocking scenarios become possible and hence tasks with tight deadlines may become unschedulable. This behavior can be observed in the average runtime of the analysis reported in Fig. 3-(h). Yet, our analysis achieves a substantially higher schedulability ratio than the baselines.

It is worth noting that for $U = 0.4$, the counterintuitive drop in schedulability for tasks with jitter is due to the timeout. The bar chart shown at the bottom of Fig. 3-(d) represents the ratio of task sets that could not be analyzed within the four-hour limit. The reason is that for $U = 0.4$, tasks have a small WCET and thus more combinations of job orderings may require analysis before Algorithm 1 is able to merge the branches. In the future, we plan to develop techniques to handle lower (or higher) utilization tasks differently, e.g., by designing more eager merge rules that combine paths with different job sets.

Moreover, we observed that the gap between the schedulability ratio of EDF and FP is small because most of the deadline misses are due to the work-conserving nature of the policy rather than the priority assignment. Namely, since a work-conserving scheduler cannot leave the processor idle, it will schedule any lower-priority job before the next higher-priority job is released. As a result, high-frequency tasks with tight deadlines will miss their deadline before the priority assignment method can play a significant role in improving the order of executions. We conclude that there is a need for a global scheduling algorithm that is able to avoid such blocking scenarios, for instance by being non-work-conserving. While such non-work-conserving non-preemptive scheduling algorithms have recently been proposed for uniprocessor systems [17, 18], currently no such solution exists for multiprocessor platforms.

Runtime of the analysis. Fig. 3-(e) shows that the average analysis runtime increases with increasing task-set utilization, since busy windows become longer. Consequently, paths that have the same set of jobs are merged only at later stages. For larger utilizations such as for $U \geq 2.8$, however, identifying *unschedulable* task sets becomes easy due to the presence of tasks with large WCETs that can block all cores for a long time. Since we stop the analysis as soon as a deadline miss is found, not-schedulable task sets with large utilization can be identified quickly. The analysis runtime hence decreases rapidly for larger utilization values.

Figs. 3-(f) and (g) show that the analysis runtime grows with increasing tasks and core counts because more states are generated in the expansion phase. It is worth noting that unlike the effect pertaining to the number of tasks, increasing the number of cores will not increase the runtime monotonically. The reason is that, as shown in Fig. 3-(c), for a workload with $U = 2.8$ and 10 tasks, almost all task sets are schedulable on 6 cores or more. That is, the number of cores *per se* only has a limited effect on the runtime of the algorithm; however, larger platforms are likely to host large task sets, with a potentially large number of jobs per hyperperiod, and our analysis is sensitive to such increases in workload size.

Figs. 3-(i) and 3-(j) report the runtime of the analysis for each task set w.r.t. the number of jobs in a hyperperiod for two scenarios: varying utilization and varying the number of tasks, respectively. As shown by the figures, the runtime of the analysis grows with the increase in the number of jobs in a hyperperiod. We also observe that with an increase in the number of tasks from 10 (Fig. 3-(i)) to up to 30 (Fig. 3-(j)), the largest observed runtime of the analysis grows linearly, i.e., from 1000 to 4000.

Since a naive analysis without path merging does not scale even for a uniprocessor system, as shown in [16], we did not perform a separate experiment to show the efficiency of the path merging technique. In the future, we plan to further explore the design space for different merge conditions and their efficiency for different task set types and utilizations.

Overall, we conclude that: **(i)** the proposed analysis is practical for realistic workload sizes, **(ii)** it identifies a significantly larger portion of schedulable tasks in comparison with state-of-the-art tests for sporadic tasks, and **(iii)** even when jitter is considered (which allows for more blocking scenarios and uncertainties), our analysis still achieves much higher schedulability than the baseline tests (which, to be clear, are designed for sporadic task sets).

In terms of limitations, we also observed that the runtime of the analysis grows quickly (e.g., more task sets hit the four-hour timeout) for larger systems (e.g., when $n \geq 20$ and $m \geq 16$). This is due to the increase in the number of tasks and the number of ways a task can be assigned to a core in the expansion phase of the algorithm. To scale to such large systems, a more efficient abstraction is needed that allows for more eager merging techniques.

7 Conclusion

The paper provides a sufficient schedulability analysis for global job-level fixed-priority scheduling algorithms and non-preemptive job sets. We have presented a technique for deriving an upper bound on the WCRT and a lower bound on the BCRT by exploring an abstraction of all possible schedules of a job set that reflects the uncertainties in job execution and release times. We developed the notion of a schedule-abstraction graph for global schedulers and introduced two key techniques, namely path merging and fast-forwarding, to slow the state-space growth and proved the analysis to be sound.

Our empirical evaluation using periodic workloads shows significant schedulability improvements w.r.t. the state-of-the-art tests in all experimental setups. The observed runtime of the analysis ranged from a couple of seconds to a couple of hours for realistic system setups, e.g., up to 30 tasks, up to 9 cores, and up to 100000 jobs per hyperperiod, which is an acceptable performance for an offline, design-time analysis.

Furthermore, our current implementation is sequential. We expect that parallelizing the analysis, so that naturally independent scenarios are explored in parallel, would yield a substantial speedup. To this end, we hope to derive rules that allow maximum paralellism between independent exploration frontiers. Moreover, we will investigate different merge rules to reduce the runtime of the analysis. We also plan to extend the solution presented here to analyze systems with more complicated properties such as precedence constraints and preemption points, and to other scheduling problems such as gang scheduling.

References

1. 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.
2. 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.
3. 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. Springer, 2007.
4. Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168, 2006.
5. Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. *Algorithmica*, 63(4):763–780, 2012.
6. 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)*, 2015.
7. Anton Cervin, Bo Lincoln, Karl-Erik Arzen, and Giorgio Buttazzo. The Jitter Margin and Its Application in the Design of Real-Time Control Systems. In *International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, pages 1–9, 2004.
8. 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.
9. 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.

10 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.

11 Nan Guan, Wang Yi, Zonghua Gu, Qingxu Deng, and Ge Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 137–146, 2008.

12 S. Kramer, D Ziegenbein, and A Hamann. Real world automotive benchmark for free. In *International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS)*, 2015.

13 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.

14 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.

15 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.

16 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.

17 Mitra Nasri and Gerhard Fohler. Non-work-conserving non-preemptive scheduling: motivations, challenges, and potential solutions. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 165–175, 2016.

18 Mitra Nasri and Mehdi Kargahi. Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks. *Real-Time Systems*, 50(4):548–584, 2014.

19 Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. Technical Report MPI-SWS-2018-003, Max Planck Institute for Software Systems, Germany, 2018. URL: `http://www.mpi-sws.org/tr/2018-003.pdf`.

20 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.

21 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.

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

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

24 Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric OS for multi-core embedded systems. In *IEEE Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, 2016.

25 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The

worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, 2008.

26 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)*, 2017.