Response-Time Analysis for Non-Preemptive Global Scheduling with FIFO Spin Locks

Nogd, Suhail; Nelissen, Geoffrey; Nasri, Mitra; Brandenburg, Bjorn B.

DOI
10.1109/RTSS49844.2020.00021

Publication date
2020

Document Version
Final published version

Published in
Proceedings - 2020 IEEE 41st Real-Time Systems Symposium, RTSS 2020

Response-Time Analysis for Non-Preemptive Global Scheduling with FIFO Spin Locks

Suhail Nogd*, Geoffrey Nelissen†, Mitra Nasri† and Björn B. Brandenburg‡
* Delft University of Technology (TUDelft), The Netherlands
† Eindhoven University of Technology (TU/e), The Netherlands
‡ Max Planck Institute for Software Systems (MPI-SWS), Germany

2020 IEEE Real-Time Systems Symposium (RTSS) | 978-1-7281-8324-4/20/$31.00 ©2020 IEEE | DOI: 10.1109/RTSS49844.2020.00021

***Abstract*—Motivated by the lack of response-time analyses for non-preemptive global scheduling that consider shared resources, this paper provides such an analysis for global job-level fixed-priority (JLFP) scheduling policies and FIFO-ordered spin locks. The proposed analysis computes response-time bounds for a set of resource-sharing jobs subject to release jitter and execution-time uncertainties by implicitly exploring all possible execution scenarios using state-abstraction and state-pruning techniques. A large-scale empirical evaluation of the proposed analysis shows it to be substantially less pessimistic than simple execution-time inflation methods, thanks to the explicit modeling of contention for shared resources and scenario-aware blocking analysis.**

***Index Terms*—real-time systems, response-time analysis, shared resources, global multiprocessor scheduling**

I. Introduction

An essential capability provided by virtually all multitasking real-time operating systems (or runtime environments) is the ability to share data or software resources without running into race conditions. Multiprocessor real-time systems typically use two types of locking protocols for this purpose: suspension-based and spin-based protocols, which offer various tradeoffs. Although the literature provides ample analyses of both of these approaches for globally scheduled *preemptive* tasks (e.g., see a recent survey [10]), to date only little attention has been paid to high-accuracy response-time analysis for globally scheduled *non-preemptive* tasks that share resources.

To close this gap, we propose a blocking-aware response-time analysis for non-preemptive tasks that are scheduled by a work-conserving global *job-level fixed-priority* (JLFP) scheduling policy on a multicore platform. We focus on spin-lock protocols since they are a natural fit for non-preemptive scheduling: they are relatively simple to implement, require virtually no OS support, and are compatible with run-to-completion (or single-stack) runtime systems. Furthermore, spin locks incur significantly lower runtime overhead (relative to suspension-based locking protocols) [9], so that the cost of suspending and resuming a task can easily outweigh the cost of busy-waiting (i.e., spinning) if critical sections are relatively short (which they usually are in well-designed systems). Finally, while it is notoriously difficult to predict cache contents after suspensions (usually, it is pessimistically assumed that any suspension results in a complete loss of cache affinity), a spinning job effectively *protects* the state of its assigned processor and cache. This makes the system not only more time-predictable, but also goes a long way towards enabling a sound *worst-case execution time* (WCET) analysis.

More specifically, in **this paper**, we propose a new schedulability analysis for periodic tasks and other workloads with a repeating pattern of job releases (such as multi-frame tasks). This choice is motivated by the fact that many practical workloads are periodic (e.g., in automotive systems, in control applications, in video and audio processing, in computer vision etc.). In fact, a recent survey of 120 industry practitioners by Akesson et al. [4] indicated that 82% of the respondents work on systems that include periodic tasks, and 21% of the respondents answered that their system is exclusively comprised of tasks with periodic and/or table-driven activation.

The analysis proposed in this paper is based on the notion of schedule abstraction, a concept initially introduced by Nasri et al. [21]–[23]. However, whereas Nasri et al.'s prior analyses do not consider locking-induced delays, we assume—and model in detail—that tasks coordinate mutually exclusive access to shared resources by means of *FIFO-ordered spin locks*, a common choice in multiprocessor real-time systems [10]. Our analysis is generic in nature and covers all work-conserving global non-preemptive JLFP schedulers such as *non-preemptive earliest-deadline first* (NP-EDF) and *non-preemptive fixed-priority* (NP-FP) scheduling. Furthermore, our analysis is *sufficient*, i.e., it covers any sequence of job finish times that may occur in the modeled system.

The proposed analysis implicitly explores all possible orders of job start times as well as their accesses to shared resources in a *schedule-abstraction graph* [21]–[23]. Each such ordering results in a different sequence of *system states*. The efficiency of the analysis (in terms of runtime and memory footprint) and accuracy of this exploration strongly depends on the level of abstraction used to encode the system states. To this end, we propose a completely *new system-state abstraction* that accurately models shared resource accesses by means of FIFO spin locks. This new system state representation requires the design (and a proof of soundess of) a whole new set of expansion and merging rules, which are used to build the schedule-abstraction graph. Ultimately, we prove in Section IV-I that all possible system states are indeed explored and that our analysis yields a safe *worst-case response time* (WCRT) bound for each job, which reflects the worst-case blocking suffered by those jobs.

Notably, our analysis does *not* just pessimistically account

2576-3172/20/$31.00 ©2020 IEEE
DOI 10.1109/RTSS49844.2020.00021

for the *union* of all critical sections that could theoretically overlap with a job in *some* schedule; rather, our analysis is *execution-scenario-aware* and accounts only for contention that can actually arise in a specific interleaving of resource accesses by the jobs. For example, if a job incurs *either* shared-resource contention (if it is scheduled right away) *or* interference from higher-priority jobs (but not both), then the proposed analysis will reflect the worse of the two scenarios, but *not* an infeasible combination of both (scheduling interference *and* resource contention). We are not aware of any prior response-time analysis with a comparable level of accuracy in the accounting of synchronization delays.

In fact, somewhat surprisingly, we found that there is hardly any directly related **prior work** [10]. While FIFO-ordered spin locks in real-time systems have long been recognized to have favorable properties for worst-case analysis [3,5]–[7,9,12,13, 15,18,20,27], prior work in this space has focused on either *preemptive* global scheduling [3,7,9,12,15,20] or partitioned scheduling [5]–[7,9,18,27].

In the context of preemptive global scheduling, the most relevant locking protocols and blocking analyses are Brandenburg's *holistic blocking analysis* of non-preemptive FIFO spin locks [9] as used in Block et al.'s FMLP for *short* critical sections [7], and the *suspension-based* global OMLP [11], FMLP for *long* critical sections [7,28], and PIP [16,25,28]. We compare against each of these in our evaluation (Sec. V).

For a review of real-time locking in general, we refer the reader to Brandenburg's recent comprehensive survey of the area [10].

Finally, we note that, while it might be tempting to compare the timing analysis of FIFO-ordered spin lock accesses to queuing theory or the analysis of networks, our given problem is fundamentally different from packets traversing a network since dealing with locks in real-time systems is essentially a multi-resource scheduling problem (i.e., jobs need to get access to both the processor and the shared resources at the same time but for different durations). More precisely, we cannot solely model our problem as a network of queues to be traversed because the queuing delays for access to processors and shared resources interact in nontrivial ways. This makes a direct comparison to the domain of queuing theory, Real-Time Calculus [26], or Network Calculus [8] inapplicable.

To our knowledge, there is no prior work on the central contribution of this paper: a *blocking-aware* response-time analysis for *non-preemptive* global scheduling with shared resources protected by FIFO-ordered spin locks.

II. System Model and Definitions

We consider a work-conserving global JLFP scheduling policy to schedule a set of *non-preemptive* tasks on a multicore platform with m identical cores. Each task releases jobs according to a specific pattern (e.g., periodic, rate-based or bursty), and each job has a fixed priority. The JLFP scheduler dispatches ready jobs in order of decreasing priority.

A. Workload Model

A job $J_i = ([r_i^{min}, r_i^{max}], d_i, p_i, \langle J_{i,1}, \ldots, J_{i,n_i} \rangle)$ has an earliest release time r_i^{min}, a latest release time r_i^{max}, an absolute deadline d_i, and a priority p_i. We assume that timing parameters are discrete, i.e., integer multiples of a clock tick.

A job's execution is modeled by its sequence of *job segments* $\langle J_{i,1}, \ldots, J_{i,n_i} \rangle$, where $J_{i,1}$ and J_{i,n_i} are its first and last segment, respectively. Each job segment $J_{i,j}$ is identified by $J_{i,j} = ([C_{i,j}^{min}, C_{i,j}^{max}], \eta_{i,j}, [L_{i,j}^{min}, L_{i,j}^{max}])$, where $C_{i,j}^{min}$ is the best-case execution time (BCET), $C_{i,j}^{max}$ is the worst-case execution time (WCET), $\eta_{i,j}$ is the set of resources that the segment *requests*, and $L_{i,j}^{min}$ and $L_{i,j}^{max}$ are the minimum and maximum length of the critical section associated with $\eta_{i,j}$, respectively. We obviously must have that $L_{i,j}^{min} \leq C_{i,j}^{min}$ and $L_{i,j}^{max} \leq C_{i,j}^{max}$ since the critical section length is part of the execution time of the segment. In this work, we assume that each segment $J_{i,j}$ can access at most one resource protected by a lock (i.e., $|\eta_{i,j}|$ is equal to 0 or 1). Without any loss of generality, we further assume that the critical section of a segment is located at the beginning of the segment. The segment must first be granted the lock protecting the shared resource in $\eta_{i,j}$ (if any) to start executing.

All segments of a job inherit the job's priority. For ease of notation, we use $hp(J_{i,j})$ to refer to the set of segments with a higher priority than $J_{i,j}$. We assume that ties in priority are broken arbitrarily but consistently.

Each job must execute sequentially (on one core) with run-to-completion semantics, but may compete for shared resources with other jobs running in parallel on other cores. We do not consider inter-job precedence constraints in this paper.

B. Shared Resources

We let $\mathcal{L}$ denote the set of shared resources protected by locks. A shared resource $\ell_x \in \mathcal{L}$ is *available* iff no job is currently accessing it. When a job segment $J_{i,j}$ is given access to a resource, we say that the resource is *granted* to $J_{i,j}$. A segment $J_{i,j}$ cannot start executing until the resource $\ell_x \in \eta_{i,j}$ (if any) is granted to $J_{i,j}$. Resource ℓ_x will be released as soon as the segment's critical section completes (i.e., within $[L_{i,j}^{min}, L_{i,j}^{max}]$ time units after the resource was granted). If a segment $J_{i,j}$ requests access to a resource ℓ_x that is already granted to another job, we say that J_i is *blocked* on ℓ_x. The job J_i *busy-waits* (i.e., *spins*) until it is granted ℓ_x.

In this paper, we assume FIFO spin locks, i.e., resources are granted in the order in which requests arrive. If two jobs request the same resource at the exact same time, we assume that the tie is broken arbitrarily but consistently (e.g., in our experiments, ties are broken in favor of lower job IDs).

C. Execution Model

A job is *ready* at time t if it has been released and has not yet started executing. Due to the non-preemptive execution model, a job must run to completion without preemption once it has started execution. Thus, once the first segment $J_{i,1}$ of a job J_i begins its execution on a core, the core is exclusively serving J_i until J_i completes its execution, including any blocking

delays. We say that a job J_i has *claimed* a core if it started executing and has not finished yet. Upon completion we say that J_i *releases* the core, making it again available to other jobs. Hence, a core is either *claimed* (busy executing a job) or *free* for a new job to start executing.

Given the *finish time* f_i and the earliest-release time r_i^{min} of J_i, we compute the *response time* of the job as $f_i - r_i^{min}$.

D. Problem Definition

We seek to bound the worst-case response time of each member of a finite set of non-preemptive jobs $\mathcal{J}$ that access shared resources. The job set $\mathcal{J}$ is the set of all jobs released in an *a priori* computed *observation window*. The input job set in the observation window must be a representative workload for the system, namely, either because the observation window is long enough to capture the whole workload (e.g., in batch scheduling), or because the release patterns of the jobs in the observation window repeats for the rest of the lifetime of the system so that it is sufficient to only analyze one instance. For example, the observation window of synchronous periodic task sets with implicit deadlines that exhibit no deadline misses is one *hyperperiod* (i.e., the least integer multiple of the periods); further safe observation windows for various types of periodic (or multi-frame) task models can be found in [19,21,24].

Our analysis deems a job set $\mathcal{J}$ *schedulable* only if no execution scenario of $\mathcal{J}$ (Definition 1 below) leads to some job exhibiting a response time exceeding its deadline. We extend the original definition of execution scenario [21] to reflect shared-resource accesses.

Definition 1. *An execution scenario γ is a mapping of jobs to release times, segment execution times, and critical-section lengths such that $\forall J_i \in \mathcal{J}$, $r_i \in [r_i^{min}, r_i^{max}]$ and $\forall j, 1 \leq j \leq n_i$, $C_{i,j} \in [C_{i,j}^{min}, C_{i,j}^{max}]$ and $L_{i,j} \in [L_{i,j}^{min}, L_{i,j}^{max}]$.*

III. Schedulability Analysis

The proposed schedulability analysis builds a *schedule-abstraction graph* (SAG) [21] to implicitly explore all possible execution scenarios of a job set. In this section, we define our notion of a *resource-aware* SAG and explain its construction and use in bounding each job's worst-case response time.

A. Schedule-Abstraction Graph

A SAG is a directed acyclic graph $G = (V, E)$, where V indicates the set of (abstract) states reachable by the system and E represents the set of scheduling decisions leading from one (abstract) system state to another. An edge $e = (v_p, v_q, J_{i,j})$ from v_p to v_q with label $J_{i,j}$ indicates that executing $J_{i,j}$ in state v_p evolves the system to state v_q. We say that a job segment $J_{i,j}$ starts executing *next* in state v_p (or *succeeds* v_p) if there exists an outgoing edge from v_p with label $J_{i,j}$.

By convention [21], state v_1 represents the initial state of the system at time zero, where every core is idle and no segment has started executing yet. A path P from v_1 to a state v_p represents a possible job-segment start order that results in system state v_p. The length of such a path P indicates the number of segments that have started (and potentially already finished) their execution when the system reaches state v_p, i.e., $|P| \triangleq |\mathcal{J}^P|$, where $\mathcal{J}^P$ denotes the set of labels on the edges of path P. If a vertex v_p has multiple incoming edges, then the scheduling decisions that lead to v_p must involve the same set of job segments on any two paths from v_1 to v_p.

Property 1. *(adapted from [23]) For any two paths P and Q, if both P and Q start at v_1 and end in v_p, then $\mathcal{J}^P = \mathcal{J}^Q$.*

B. System State Representation

As previously stated, we consider that jobs can be subject to release jitter and that there can be variations in the execution times of its segments. As a consequence of this uncertainty, we must compute a finish time interval $[EFT_{i,j}, LFT_{i,j}]$ for each segment of a job J_i after a sequence of scheduling decisions taken by the scheduler. This interval is bounded by the job segment's earliest and latest finish times $EFT_{i,j}$ and $LFT_{i,j}$ in any execution scenario that complies with the assumed sequence of scheduling decisions. Thus, we say that a job segment $J_{i,j}$ of job J_i can possibly finish at or after $EFT_{i,j}$ and is certainly finished by $LFT_{i,j}$. This uncertainty in the finish times of segments introduces a challenge as it also means that we have uncertainty in processor and shared-resource availability times, which then affect the finish time intervals of subsequent job segments.

To address this challenge, we develop a new abstraction to encode information about the system state. In each vertex v_p, the system state abstraction records the availability of the cores and shared resources for any execution scenario that complies with the sequence of scheduling decisions modeled by the edges on the paths leading to v_p.

As discussed in Section II-C, a core is either free to execute a ready job or claimed by a job that was previously dispatched by the scheduler, i.e., a job has started executing on the core and not all its segments have finished executing yet. Thus, let $\mathcal{C}(v_p)$ denote the set of jobs that claimed a core (where $|\mathcal{C}(v_p)| \leq m$). A system state is then modeled as follows:

- **Claimed core availability intervals.** For each job $J_i \in \mathcal{C}(v_p)$, we record the interval $[Cl_i^{min}(v_p), Cl_i^{max}(v_p)]$, where $Cl_i^{min}(v_p)$ and $Cl_i^{max}(v_p)$ indicate when the core claimed by job J_i becomes *possibly* and *certainly* available to execute the next segment of J_i, respectively.
- **Free-core availability intervals.** We record $m - |\mathcal{C}(v_p)|$ intervals $A_x(v_p) = [A_x^{min}(v_p), A_x^{max}(v_p)]$ such that $1 \leq x \leq m - |\mathcal{C}(v_p)|$, which indicate when one, two, three, $\ldots$, $m - |\mathcal{C}(v_p)|$ free cores (i.e., those that are not already claimed by executing jobs) become possibly and certainly available to execute *ready jobs*.
- **Shared-resource availability intervals.** For each shared resource $\ell_x \in \mathcal{L}$, a state records the interval $[SR_x^{min}(v_p), SR_x^{max}(v_p)]$, where $SR_x^{min}(v_p)$ and $SR_x^{max}(v_p)$ denote the times at which ℓ_x become possibly and certainly available again, respectively.

For ease of reference, we also introduce the two notations $SR_{i,j}^{min}(v_p)$ and $SR_{i,j}^{max}(v_p)$ to refer to the availability interval of the resource accessed by segment $J_{i,j}$ (if any).

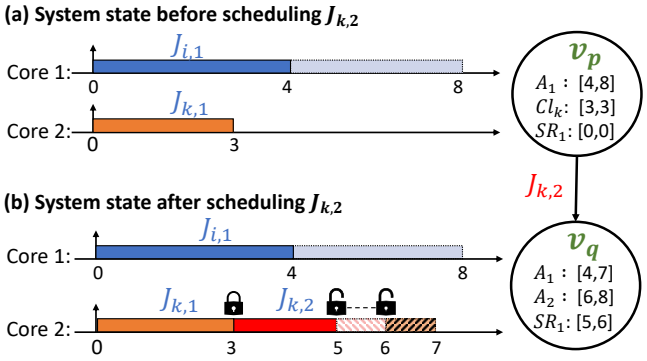


Fig. 1: Two successive system states for $m = 2$ and two jobs J_i (with one segment) and J_k (with two segments) released at time 0. Only the second segment of J_k requests a shared resource ℓ_1 with a critical section length $L_{k,2} \in [2, 3]$. The execution times of the segments are $C_{i,1} \in [4, 8]$, $C_{k,1} \in [3, 3]$ and $C_{k,2} \in [3, 4]$.

Hence, if $J_{i,j}$ accesses a shared resource (i.e., $\exists \ell_x \in \eta_{i,j}$), then $SR^{min}_{i,j}(v_p)$ and $SR^{max}_{i,j}(v_p)$ are the earliest and latest availability time $SR^{min}_{x}(v_p)$ and $SR^{max}_{x}(v_p)$ of the resource ℓ_x accessed by $J_{i,j}$. If $J_{i,j}$ does not access a resource, then simply $SR^{min}_{i,j}(v_p) = SR^{max}_{i,j}(v_p) = 0$.

Example. Fig. 1 shows how an abstract state v_p in the SAG evolves into a new state v_q when a ready job segment $J_{k,2}$ is scheduled. On the left, we show the status of the cores before and after scheduling $J_{k,2}$. The right side shows the contents of the abstract state v_p and how it evolves after scheduling $J_{k,2}$.

Since J_i has only one segment, its start time defines its finish time interval and thus the time at which Core 1 may become available for ready jobs (between times 4 and 8). Thus, v_p records that one core is free for a ready job to execute within the interval $[4, 8]$ (i.e., $A_1(v_p) = [4, 8]$). State v_p also records that one core is claimed by job J_k whose first segment $J_{k,1}$ will potentially and certainly complete by time 3 (i.e., $Cl_k = [3, 3]$). Moreover, since none of the two segments executing in v_p access a shared resource, the shared resource ℓ_1 is certainly available from time 0 (i.e., $SR_1 = [0, 0]$).

Prior to executing $J_{k,2}$, the lock protecting the shared resource it requires must be acquired. Since the shared resource is available, the resource is granted to $J_{k,2}$ as soon as the core already claimed by job J_k becomes available, i.e., at time 3. Since $J_{k,2}$'s critical section has a variable execution time, it may release the lock any time in the interval $[5, 6]$. This is recorded in the state v_q with the interval $SR_1 = [5, 6]$. Moreover, since $J_{k,2}$ is the last segment of its job, as soon as it finishes, Core 2 is released and becomes available for other jobs. Hence, in state v_q, no core is claimed anymore. Since the execution time of $J_{k,2}$ ranges from 3 to 4 time units, the core claimed by $J_{k,2}$ will become available in the interval $[6, 7]$. This means that one core becomes possibly available at time 4 (Core 1), one core is certainly available at time 7 (Core 2), two cores are possibly available at time 6, and two cores are certainly available at time 8. Thus, v_q records the availability intervals $A_1(v_q) = [4, 7]$ and $A_2(v_q) = [6, 8]$.

Algorithm 1: Construction of the SAG for job set $\mathcal{J}$.

Inputs : Job set $\mathcal{J}$
Output: Bounds on the BCRT and WCRT of every job in $\mathcal{J}$

$\forall J \in \mathcal{J}, BR_i \leftarrow \infty, WR_i \leftarrow 0$;
Initialize G by adding $v_1 = (\{[0,0], ..., [0,0]\}, \{[0,0], ..., [0,0]\})$;
while $\exists$ *path* P *from* v_1 *to a leaf vertex such that* $P \neq \mathcal{J}$ **do**
  P $\leftarrow$ the shortest path from v_1 to a leaf vertex v_p;
  $\mathcal{R}^P \leftarrow$ set of potentially ready segments obtained with Eq. (1);
  for *each segment* $J_{i,j} \in \mathcal{R}^P$ **do**
    if $J_{i,j}$ *can start executing after* v_p *(Theorem 1)* **then**
      Build v'_p using Algorithm 2;
      if $J_{i,j} = J_{i,n_i}$ **then**
        $BR_i \leftarrow \min\{EFT(v'_p) - r^{min}_i, BR_i\}$;
        $WR_i \leftarrow \max\{LFT(v'_p) - r^{max}_i, WR_i\}$;
      end
      Connect v_p to v'_p with an edge labeled $J_{i,j}$;
      if $\exists$ *path* Q *that ends in* v_q *such that Rule 1 is satisfied for* v'_p *and* v_q **then**
        Merge v'_p and v_q in v_z using Eq. (16)-(18);
        Redirect all incoming edges of v_q and v'_p to v_z;
        Remove v_q and v'_p from G;
      end
    end
  end
end

Note that every abstract system state records exactly m core availability intervals. They are split in $|\mathcal{C}(v_p)|$ intervals for claimed cores (those on which the first segment of a job was dispatched and its last segment was not dispatched yet) and $m - |\mathcal{C}(v_p)|$ intervals for free cores (i.e., any core that is not claimed). Therefore, the number of claimed and free cores does not depend on a specific time point but rather on the specific set of in-progress segments.

C. Schedule-Abstraction Graph Generation

The SAG is built iteratively. Each iteration comprises two phases, namely the *expansion phase* and the *merge phase*. In the expansion phase, (one of) the shortest path(s) P in the graph is picked and expanded for every job segment that can possibly start executing next in the state v_p at the end of P. For every such job segment $J_{i,j}$, we create a new vertex v'_p in the graph, which represents a new system state and is connected to v_p via a directed edge labeled with $J_{i,j}$. The information encoded in the new state v'_p contains an updated version of the free cores, claimed cores and shared-resource availability intervals reflecting that $J_{i,j}$ has now started to execute.

After the graph has been expanded with new states, the merge phase commences. The purpose of the merge phase is to slow down the growth of the graph, to postpone a potential state-space explosion for as long as possible. This is achieved by merging any two "similar" states that terminate paths with identical sets of job segments. To preserve soundness, every system state that is reachable from any of the two original states must also be reachable from the merged state, which ensures that no possible execution scenario is discarded.

The exploration completes when no vertex remains to be expanded, i.e., all job segments have been scheduled on every

path. At this point, each path represents a set of valid execution scenarios and every possible schedule has been explored.

The complete SAG construction procedure is given in Algorithm 1. Note that it uses two variables (BR_i and WR_i) for each job in $\mathcal{J}$ to keep track of its best-case (BCRT) and worst-case response time (WCRT) on any path in the graph. Those bounds are updated (lines 10–11) whenever the last segment of a job is scheduled on a path, hence indicating the completion of the job in that execution scenario. If, by the end of the exploration, no job has a WCRT exceeding its deadline, then the analysis deems the job set schedulable.

IV. Expansion Phase

We now consider the expansion and merge phases and show how a path ending in state v_p is expanded to a new state v_q.

A. Overview

First, we build a set of potentially *ready* job segments in state v_p, i.e., the segments that have not started executing on path P and for which all preceding segments have started and potentially completed. For each such segment $J_{i,j}$, we compute the earliest and latest time $EST_{i,j}(v_p)$ and $LST_{i,j}(v_p)$ at which the segment can start executing in v_p. That is, $EST(v_p)$ denotes the earliest time at which a global work-conserving JLFP scheduler would allow $J_{i,j}$ to start in v_p. Similarly, $LST_{i,j}(v_p)$ indicates the latest time at which $J_{i,j}$ must have certainly started executing if it is the next segment to succeed v_p. If $J_{i,j}$ has not started by $LST_{i,j}(v_p)$, a different segment must have started to execute after v_p. Hence, a segment is said to be eligible to be a successor of v_p only if its earliest start time $EST_{i,j}(v_p)$ is no later than its latest start time $LST_{i,j}(v_p)$. For each eligible job segment, a new vertex v'_p is added to the graph, where v'_p encodes the system state after $J_{i,j}$ started executing. In addition to deciding whether a job segment is eligible to start executing after state v_p, $EST_{i,j}(v_p)$ and $LST_{i,j}(v_p)$ also help compute the earliest and latest finish times $EFT_{i,j}(v_p)$ and $LFT_{i,j}(v_p)$ of $J_{i,j}$.

Next, each step of the expansion phase is explained in detail.

B. Ready Interval

We consider that a job segment is ready if the job has been released and all its preceding segments have completed. Thus, we define the set of *potentially ready* segments $\mathcal{R}^P$ for path P to be the set of segments that have not yet started to execute (i.e., $J_{i,j} \notin \mathcal{J}^{\mathcal{P}}$) and whose predecessor (if any) has already started and potentially completed its execution (i.e., either $J_{i,j}$ is the first segment of its job J_i, thereby meaning $j = 1$, or $J_{i,j-1} \subseteq \mathcal{J}^{\mathcal{P}}$).

$$\mathcal{R}^P \triangleq \{J_{i,j} \mid J_{i,j} \notin \mathcal{J}^{\mathcal{P}} \wedge (j = 1 \vee J_{i,j-1} \subseteq \mathcal{J}^{\mathcal{P}})\} \quad (1)$$

C. Earliest Start Time

Since we consider that jobs may suffer from release jitter and execution-time variation, the exact finishing times of preceding job segments is unknown. Therefore, the exact time at which a segment $J_{i,j}$ may start to execute is also unknown. For each segment $J_{i,j}$ in $\mathcal{R}^P$, we prove a lower bound $EST_{i,j}(v_p)$ and an upper bound $LST_{i,j}(v_p)$ on the time at which $J_{i,j}$ may start executing in v_p (Equations (2) and (7), respectively).

$$EST_{i,j}(v_p) = \begin{cases} \infty & \text{if } j = 1 \wedge |\mathcal{C}(v_p)| = m \\ \max\{r_i^{min},\ A_1^{min}(v_p),\ SR_{i,j}^{min}(v_p)\} & \text{if } j = 1 \wedge |\mathcal{C}(v_p)| < m \\ \max\{Cl_i^{min}(v_p),\ SR_{i,j}^{min}(v_p)\} & \text{if } j > 1 \end{cases} \quad (2)$$

Lemma 1. *Segment $J_{i,j} \in \mathcal{R}^P$ cannot start executing (as a successor of state v_p) before $EST_{i,j}(v_p)$.*

Proof: The first segment $J_{i,1}$ of a job J_i can start its execution only if (i) it is released, (ii) the shared resource ℓ_x it requests (if any) is available **and** (iii) a core is available. Thus, if all cores have already been claimed by other jobs (i.e., $|\mathcal{C}(v_p)| = m|$) then $J_{i,1}$ cannot be a successor of v_p and $EST_{i,j}(v_p) = \infty$. This proves the first case of Eq. (2).

However, if there is a free core (i.e., $|\mathcal{C}(v_p)| < m|$), then, by definition, A_1^{min} is the earliest time at which a core can potentially become available. Furthermore, r_i^{min} is the earliest release time of J_i and $SR_{i,j}^{min}(v_p)$ is the earliest time at which the shared resource accessed by $J_{i,j}$ may become available. Thus, the earliest time at which $J_{i,1}$ may start to execute is given by $EST_{i,j}(v_p) = \max\{r_i^{min},\ A_1^{min}(v_p),\ SR_{i,j}^{min}(v_p)\}$ if $j = 1$. This proves the second case of Eq. (2).

Any segment that is not the first segment of a job (i.e., a segment $J_{i,j}$ with $j > 1$) can start its execution only if (i) the core claimed by the preceding segments belonging to the same job is available, **and** (ii) the shared resource it requests (if any) is also available. Since $J_{i,j}$ is in $\mathcal{R}^P$, all the segments of J_i that precede $J_{i,j}$ must have started (and potentially finished) executing on the core claimed by J_i. Thus, the earliest time at which the core claimed by J_i may become available for $J_{i,j}$ is given by $Cl_i^{min}(v_p)$. Therefore, the earliest time at which $J_{i,j}$ may start to execute is $\max\{Cl_i^{min}(v_p),\ SR_{i,j}^{min}(v_p)\}$ if $j > 1$. This proves the last case of Eq. (2). ■

D. Latest Start Time

The latest start time $LST_{i,j}(v_p)$ of segment $J_{i,j}$ is computed considering that the scheduler is (i) work-conserving and (ii) that it follows a non-preemptive JLFP scheduling policy.

First, focus on (i). By definition, a work-conserving scheduler *must* execute a segment as soon as the segment is certainly ready and a core is certainly available. We can thus prove the following two lemmas and their corollary.

Lemma 2. *An upper bound on the time at which a segment $J_{y,z}$ can certainly start executing (as a successor of state v_p) is given by*

$$tw_{y,z} = \begin{cases} \infty & \text{if } z = 1 \wedge |\mathcal{C}(v_p)| = m; \\ \max\{r_y^{max},\ A_1^{max},\ SR_{y,z}^{max}(v_p)\} & \text{if } z = 1 \wedge |\mathcal{C}(v_p)| < m; \\ \max\{Cl_y^{max}(v_p),\ SR_{y,z}^{max}(v_p)\} & \text{if } z > 1. \end{cases} \quad (3)$$

Proof: Infinity is an obvious upper bound on the start time of $J_{i,j}$. Therefore, in this proof, we only focus on the cases where $tw_{y,z} \neq \infty$.

A starting segment $J_{y,1}$ *must* start to execute as soon as (i) it is released, (ii) the resource ℓ_x it requests (if any) is available and (iii) a core is available. By definition, r_y^{max} is an upper bound on the release time of $J_{y,1}$, $SR_{y,z}^{max}(v_p)$ is an upper bound on the availability time of the shared resource accessed by $J_{y,z}$ (if any) and $A_1^{max}(v_p)$ denotes the time at which a core is certainly available in state v_p. Thus, $J_{y,1}$ can certainly start executing at $\max\{r_y^{max},\ A_1^{max},\ SR_{y,1}^{max}(v_p)\}$ when $z = 1$ and there is at least one free core in v_p.

Any segment $J_{y,z}$ that is not the first segment of $J_{y,z}$ (i.e., with $z > 1$) can certainly start executing when (i) the resource it requests (if any) is available, and (ii) the segments of J_y that precede $J_{y,z}$ have all completed their execution on the core claimed by J_y. Since $J_{y,z}$ is in $\mathcal{R}^P$, all the segments of $J_{y,z}$ preceding $J_{y,z}$ have already started (and potentially finished) executing on the core claimed by J_y, and because $Cl_y^{max}(v_p)$ is an upper bound on the time at which the core claimed by J_y becomes available to execute the next segment of J_y, we have that $J_{y,z}$ certainly starts at $\max\{Cl_y^{max}(v_p),\ SR_{y,z}^{max}(v_p)\}$. ■

Lemma 3. *A work-conserving scheduler will start executing a job segment no later than*[1]

$$t_{wc} = \min_{\infty}\{tw_{y,z} \mid J_{y,z} \in \mathcal{R}^P\} \tag{4}$$

Proof: By Lemma 2, a job segment $J_{y,z} \in \mathcal{R}^P$ is certainly ready to start executing at time $tw_{y,z}$. Thus there is at least one job segment that is ready to execute at time $\min_{\infty}\{tw_{y,z} | J_{y,z} \in \mathcal{R}^P\}$. Thus, t_{wc} is an upper-bound on the time at which a work-conserving scheduler will start executing a job segment. ■

Corollary 1. *A segment $J_{i,j} \in \mathcal{R}^P$ cannot be a direct successor of a state v_p if it starts executing any later than t_{wc}.*

Proof: Since the scheduler will certainly schedule a job segment by time t_{wc}, $J_{i,j}$ must start executing at or before t_{wc} if it is the first job scheduled after v_p. ■

Now that we covered work conservation, we consider the impact of the JLFP scheduling policy.

First, assume that two segments $J_{i,j}$ and $J_{y,z}$ request the same resource. Since a FIFO spin lock provides access to the shared resource based on the order in which requests were made, the order in which those segments will start executing does not depend on their priority but on the order of their releases instead. Therefore, the JLFP scheduling policy does not impact the start time of segments that share the same resource. Then, let $\mathcal{H}$ denote the set of segments with a higher priority than $J_{i,j}$ that do not share a resource with $J_{i,j}$, i.e., $\mathcal{H} = \{J_{y,z} \mid J_{y,z} \in \{\mathcal{R}^P \cap hp(J_{i,j})\} \wedge \eta_{y,z} \cap \eta_{i,j} = \emptyset\}$. Let t_{high} be an upper-bound on the time at which any segment in $\mathcal{H}$ will certainly be ready to execute. We prove (Lemmas 4 and 5) that t_{high} can be computed with Eq. (5).

$$t_{high} = \min_{\infty}\{th_{y,z}(J_{i,j}) \mid J_{y,z} \in \mathcal{H}\} \tag{5}$$

where

$$th_{y,z}(J_{i,j}) = \begin{cases} \max\{r_y^{max},\ SR_{y,z}^{max}(v_p)\} & \text{if } z = j = 1 \\ \max\{A_1^{max}(v_p), r_y^{max},\ SR_{y,z}^{max}(v_p)\} & \text{if } z = 1 \wedge j > 1 \\ \max\{Cl_y^{max}(v_p),\ SR_{y,z}^{max}(v_p)\} & \text{if } z > 1 \end{cases} \tag{6}$$

Lemma 4. *Let $J_{y,z}$ be a segment in $\mathcal{H}$. If $J_{i,j}$ did not start to execute before $th_{y,z}(J_{i,j})$, then $J_{y,z}$ will start before $J_{i,j}$.*

Proof: We analyze the three cases of Eq. (6).

Case 1. Assume that both $J_{i,j}$ and $J_{y,z}$ are the first segment of their respective jobs (i.e., $j = z = 1$). For a starting segment to be able to execute, it needs to be (1) released, (2) the resource it requests (if it requests one) must be available and (3) a core must be available for it to execute on. By definition, r_y^{max} is an upper bound on (1) and $SR_{y,z}^{max}(v_p)$ is an upper bound on (2). Regarding (3), we note that because $J_{y,1}$ and $J_{i,1}$ are both starting segments, they both compete for the same available cores. Since $J_{y,1}$ has a higher priority than $J_{i,j}$, if both $J_{y,1}$ and $J_{i,j}$ are ready and have their shared resources available at the same time, then $J_{y,1}$ will certainly start before $J_{i,j}$. Therefore, only (1) and (2) decide whether $J_{y,1}$ will start to execute before $J_{i,j}$. Thus, if $J_{i,j}$ did not start before $\max\{r_y^{max},\ SR_{y,z}^{max}(v_p)\}$ when $z = j = 1$, then $J_{y,1}$ will be dispatched before $J_{i,j}$.

If $J_{i,j}$ is not a starting segment (i.e., $j > 1$), then it already has a reserved core. Therefore, it does not compete with $J_{y,z}$ for the same core (i.e., we must account for (3)).

Case 2. If $J_{y,z}$ is the first segment of the higher priority job J_y, then, by definition, $A_1^{max}(v_p)$ is a safe upper bound on (3). Thus, because $J_{y,z}$ has a higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{A_1^{max}(v_p),\ r_y^{max},\ SR_{y,z}^{max}(v_p)\}$ when $z = 1 \wedge j > 1$.

Case 3. If $J_{y,z}$ is not the first segment of the higher-priority job J_y, then job J_y is already released and it already claimed a core. Thus, by definition, $Cl_y^{max}(v_p)$ is an upper bound on (3). Hence, as $J_{y,z}$ has a higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{Cl_y^{max}(v_p),\ SR_{y,z}^{max}(v_p)\}$ when $z > 1$. ■

Lemma 5. *A segment $J_{i,j} \in \mathcal{R}^P$ cannot be the direct successor of a state v_p if it starts executing later than $t_{high} - 1$.*

Proof: According to Lemma 4, $\forall J_{y,z} \in \mathcal{H}$, $J_{y,z}$ will start before $J_{i,j}$ if $J_{i,j}$ did not start before $th_{y,z}(J_{i,j})$. Then, $J_{i,j}$ will not be the next segment to *succeed* v_p if $J_{i,j}$ did not start before $\min_{\infty}\{th_{y,z}(J_{i,j}) \mid J_{y,z} \in \mathcal{H}\}$. ■

It directly follows that an upper bound on the start time of $J_{i,j}$ can be computed according to Lemma 6.

Lemma 6. *A segment $J_{i,j} \in \mathcal{R}^P$ can be the first segment to start executing in state v_p only if it starts no later than*

$$LST_{i,j} = \min\{t_{wc},\ t_{high} - 1\} \tag{7}$$

[1] $\min_{\infty}\{X\} = \min\{X \cup \{\infty\}\}$ where X is a set of positive values.

Proof: Since $LST_{i,j} \leq t_{wc}$ (by Cor. 1) and $LST_{i,j} \leq t_{high} - 1$ (by Lemma 5), the claim holds. ∎

E. Eligibility Condition

Now that we have computed a lower bound $EST_{i,j}(v_p)$ on the earliest start time and an upper bound $LST_{i,j}(v_p)$ on the latest start time of $J_{i,j}$ in state v_p, it is rather obvious that $J_{i,j}$ can be a successor to v_p only if

$$EST_{i,j}(v_p) \leq LST_{i,j}(v_p). \tag{8}$$

Lemma 7. *A segment $J_{i,j}$ is a direct successor of v_p only if $EST_{i,j}(v_p) < \infty$ and Inequality* (8) *holds.*

Proof: According to Eq. (2), if $EST_{i,j}(v_p) = \infty$ then $J_{i,j}$ is the first segment of job J_i and no free core is available to start executing $J_{i,j}$ (i.e., every core is claimed by an unfinished job). Since we assume a non-preemptive system, the scheduler cannot dispatch $J_{i,j}$ on a core in v_p. Further, from Lemma 1, we know that $EST_{i,j}(v_p)$ is a lower bound on the earliest time at which $J_{i,j}$ can start executing in v_p. From Lemma 6, we have that $LST_{i,j}(v_p)$ is an upper bound on the time at which $J_{i,j}$ can start executing in v_p. Hence, if $EST_{i,j} > LST_{i,j}$, it creates a contradiction, thereby meaning that $J_{i,j}$ cannot start to execute in v_p and thus cannot be a successor to v_p. ∎

Beside the eligibility condition stated in Lemma 7, systems using FIFO spin locks have a second eligibility condition.

Consider the case where two segments $J_{i,j}$ and $J_{y,z}$ compete for the same shared resource. Since access to the resource is granted in FIFO order, the order in which those job segments will start to execute depends on the order of their requests. Assume that we know a lower bound on the earliest time at which $J_{i,j}$ may request its shared resource, and that we further have an upper bound on the latest time at which $J_{y,z}$ may request the resource. We refer to those bounds as $ERT_{i,j}(v_p)$ and $LRT_{y,z}(v_p)$, respectively. We prove in Theorem 1 that if $ERT_{i,j}(v_p) > LRT_{y,z}(v_p)$, then $J_{y,z}$ must execute before $J_{i,j}$, thereby implying that $J_{i,j}$ cannot be the first segment dispatched in state v_p.

To prove Theorem 1, we first prove a lower and an upper bound on $ERT_{i,j}(v_p)$ and $LRT_{y,z}(v_p)$, respectively.

Lemma 8. *Segment $J_{i,j} \in \mathcal{R}^P$ cannot request a shared resource in v_p earlier than*

$$ERT_{i,j}(v_p) = \begin{cases} \max\{r_i^{min}, A_1^{min}(v_p)\} & \text{if } j = 1; \\ Cl_i^{min}(v_p) & \text{if } j > 1. \end{cases} \tag{9}$$

Proof: The first segment $J_{i,1}$ of job J_i cannot request a resource prior to its release, i.e., not before r_i^{min}, nor can it request a resource prior to the earliest time at which a core may become available to start its execution, i.e., not before $A_1^{min}(v_p)$. Thus, the earliest time at which $J_{i,1}$ may request its resource is lower-bounded by $\max\{r_i^{min}, A_1^{min}(v_p)\}$ (hence case 1 of Eq. (9)). If $J_{i,j}$ is not the first segment of J_i, then J_i is already released and $J_{i,j}$ requests the resource as soon as preceding segment completes, which is lower-bounded by $Cl_i^{min}(v_p)$ (hence case 2 of Eq. (9)). ∎

Lemma 9. *A job segment $J_{y,z} \in \mathcal{R}^P$ will have certainly requested its shared resource by time*

$$LRT_{y,z}(v_p) = \begin{cases} \max\{r_y^{max}, A_1^{max}(v_p)\} & \text{if } j = 1; \\ Cl_y^{max}(v_p) & \text{if } j > 1. \end{cases} \tag{10}$$

Proof: A segment $J_{y,z}$ will certainly request its shared resource when (1) the job it belongs to has been released and (2) the segment has a core to execute on.

By definition, the first segment $J_{y,1}$ of a job J_y is certainly released at time r_y^{max} and a core is certainly available by time $A_1^{max}(v_p)$. Thus, $J_{y,1}$ will have certainly requested its resource by time $\max\{r_y^{max}, A_1^{max}(v_p)\}$ (hence case 1 of Eq. (10)).

If $J_{y,z}$ is not the first segment of job J_y, then, because $J_{y,z} \in \mathcal{R}^P$, the predecessor of $J_{y,z}$ must have started to execute. Hence, J_y is already certainly released, and $LRT_{y,z}(v_p)$ is only bounded by the time at which the core claimed by the segment of J_y that precedes $J_{y,z}$ becomes available to execute $J_{y,z}$. That time is upper-bounded by $Cl_y^{max}(v_p)$. Therefore, segment $J_{y,z}$ will have certainly requested its resource by time $Cl_y^{max}(v_p)$ (hence case 2 of Eq. (10)). ∎

Now that the *earliest* time $ERT_{i,j}(v_p)$ at which $J_{i,j}$ may request its resource, and the *latest* time $LRT_{y,z}(v_p)$ at which another segment $J_{y,z}$ has certainly requested its resource have been bounded, we can prove the following necessary condition for $J_{i,j}$ to possibly be the next segment to start executing in state v_p.

Theorem 1. *A segment $J_{i,j}$ is a direct successor of v_p only if $EST_{i,j}(v_p) < \infty$, Condition* (8) *holds, and*

$$\forall J_{y,z} \in \mathcal{R}^P \text{ s.t. } \eta_{y,z} = \eta_{i,j} \neq \emptyset, ERT_{i,j}(v_p) \leq LRT_{y,z}(v_p). \tag{11}$$

Proof: The necessity of $EST_{i,j}(v_p) < \infty$ and Condition (8) were already proven in Lemma 7. Therefore, we focus on proving Condition (11). From Lemma 8, we know that $ERT_{i,j}(v_p)$ is the earliest time at which a job $J_{i,j}$ can request its shared resource. Similarly, from Lemma 9, we know that the latest time at which a different segment $J_{y,z}$ requests its own resource is given by $LRT_{y,z}(v_p)$. Therefore, if $J_{i,j}$ and $J_{y,z}$ request the same resource ℓ_x and $ERT_{i,j}(v_p) > LRT_{y,z}(v_p)$, $J_{y,z}$ must certainly be in front of $J_{i,j}$ in the FIFO queue regulating access to ℓ_x. In this case, $J_{y,z}$ will certainly execute before $J_{i,j}$, so $J_{i,j}$ cannot be a direct successor of v_p. ∎

F. Earliest and Latest Finish Times

Since segments execute non-preemptively, if $J_{i,j}$ is the segment dispatched in system state v_p, then its earliest finish time ($EFT_{i,j}(v_p)$) and latest finish time ($LFT_{i,j}(v_p)$) are:

$$EFT_{i,j}(v_p) \;=\; EST_{i,j}(v_p) + C_{i,j}^{min}(v_p), \tag{12}$$

$$LFT_{i,j}(v_p) \;=\; LST_{i,j}(v_p) + C_{i,j}^{max}(v_p). \tag{13}$$

Algorithm 2: Create a new state v'_p by executing job segment $J_{i,j}$ after v_p.

1 **if** $J_{i,j}$ *is the first segment of* J_i **then**
2 **if** $J_{i,j}$ *is* ***not*** *the last segment of* J_i **then**
  `// J_i claims a core`
3 Create the interval $Cl_i = [EFT_{i,j}(v_p), LFT_{i,j}(v_p)]$;
4 $\mathcal{C}(v'_p) \leftarrow \mathcal{C}(v_p) \cup Cl_i$;
5 **end**
6 **else**
7 **if** $J_{i,j}$ *is the last segment of* J_i **then**
  `// the core claimed by J_i is released`
8 $\mathcal{C}(v'_p) \leftarrow \mathcal{C}(v_p) \setminus Cl_i$;
9 **else**
  `// update the core claimed by J_i`
10 $Cl_i = [EFT_{i,j}(v_p), LFT_{i,j}(v_p)]$;
11 $\mathcal{C}(v'_p) \leftarrow \mathcal{C}(v_p)$;
12 **end**
13 **end**
14 Update $\mathcal{C}$ using Eqs. (14) and (15);
 `// Update the shared resource availability`
15 **if** $\eta_{i,j} \neq \emptyset$ **then**
16 Let $\ell_x = \eta_{i,j}$;
17 $SR_x^{min}(v'_p) = EST_{i,j}(v_p) + L_{i,j}^{min}$;
18 $SR_x^{max}(v'_p) = LST_{i,j}(v_p) + L_{i,j}^{max}$;
19 **end**
 `// Update the free cores availability intervals`
20 Initialize PA and CA using Lemma 11 ;
21 Sort PA and CA in non-decreasing order;
22 $\forall 1 \leq x \leq m - |\mathcal{C}(v'_p)|,\ A_x(v'_p) \leftarrow [PA_x, CA_x]$;

G. Creating a new State

As discussed in Sec. III-C, we expand the SAG for every segment $J_{i,j}$ that is eligible according to Theorem 1. For each such segment, we create a new node v'_p that represents the new state after dispatching $J_{i,j}$, as described in Alg. 2.

First, depending on whether $J_{i,j}$ is the first, last, or an intermediate segment of job J_i, it will claim a core, release the core it previously claimed, or keep executing on its claimed core, respectively. Therefore, in lines 1–13, Alg. 2 updates the set of claimed cores $\mathcal{C}$ by either adding, removing, or updating the claimed-core availability interval associated with job J_i. If a core is claimed or was claimed, then its availability interval is set to the finish time interval $[EFT_{i,j}, LFT_{i,j}]$ of the segment $J_{i,j}$ that starts to execute on it (Lines 3 and 10).

For all other claimed cores in $\mathcal{C}$, their availability interval is updated according to Equations (14) and (15) below.

Lemma 10. *If the first segment executed in system state v_p starts executing no later than $EST_{i,j}(v_p)$, then*

$$Cl_y^{min}(v'_p) = \max\{EST_{i,j}(v_p), Cl_y^{min}(v_p)\} \tag{14}$$

$$Cl_y^{max}(v'_p) = \max\{EST_{i,j}(v_p), Cl_y^{max}(v_p)\} \tag{15}$$

Proof: Since *the first* segment that starts to execute in system state v_p does so no later than $EST_{i,j}(v_p)$, the earlier time at which the next segment may start to execute is not before $EST_{i,j}(v_p)$. Thus, the cores are not available to execute new segments before $EST_{i,j}(v_p)$. This proves the lemma. ■

Thereafter, the availability interval of the resource accessed by $J_{i,j}$ (if any) is updated to align with the end of $J_{i,j}$'s critical section, i.e., $L_{i,j}^{min}$ time units after it started executing at the earliest, and $L_{i,j}^{max}$ time units after it started at the latest.

Finally, Alg. 2 uses Lemma 11 to create two sets PA and CA that store the times at which each free core becomes possibly and certainly available after $J_{i,j}$ started to execute (Line 20). Then, as discussed in Sec. III-B, the free cores availability intervals can be computed by sorting the sets PA and CA in a non-decreasing order and picking the x^{th} element of the sorted set PA (CA, resp.) as the lower bound (upper bound, resp.) on the availability interval A_x (Line 22).

Lemma 11. *The times at which each free core becomes possibly and certainly available in v'_p are contained in the sets PA and CA, respectively, computed as follows.*

$$PA = \begin{cases} \left\{\max\{EST_{i,j}, A_x^{min}\} \mid 2 \leq x \leq m - |\mathcal{C}|\right\} & \text{if } j = 1 \neq n_i \\ \left\{\max\{EST_{i,j}, A_x^{min}\} \mid 2 \leq x \leq m - |\mathcal{C}|\right\} \cup \left\{EFT_{i,n_i}\right\} & \text{if } j = 1 = n_i \\ \left\{\max\{EST_{i,j}, A_x^{min}\} \mid 1 \leq x \leq m - |\mathcal{C}|\right\} & \text{if } 1 < j < n_i \\ \left\{\max\{EST_{i,j}, A_x^{min}\} \mid 1 \leq x \leq m - |\mathcal{C}|\right\} \cup \left\{EFT_{i,n_i}\right\} & \text{if } j = n_i > 1 \end{cases}$$

$$CA = \begin{cases} \left\{\max\{EST_{i,j}, A_x^{max}\} \mid 2 \leq x \leq m - |\mathcal{C}|\right\} & \text{if } j = 1 \neq n_i \\ \left\{\max\{EST_{i,j}, A_x^{max}\} \mid 2 \leq x \leq m - |\mathcal{C}|\right\} \cup \left\{LFT_{i,n_i}\right\} & \text{if } j = 1 = n_i \\ \left\{\max\{EST_{i,j}, A_x^{max}\} \mid 1 \leq x \leq m - |\mathcal{C}|\right\} & \text{if } 1 < j < n_i \\ \left\{\max\{EST_{i,j}, A_x^{max}\} \mid 1 \leq x \leq m - |\mathcal{C}|\right\} \cup \left\{LFT_{i,n_i}\right\} & \text{if } j = n_i > 1 \end{cases}$$

Proof: We use the following facts.

Fact 1. As already proven for Lemma 10, because $EST_{i,j}(v_p)$ is the earliest time at which $J_{i,j}$ starts to execute in system state v_p, no segment dispatched after $J_{i,j}$ may start to execute prior to $EST_{i,j}(v_p)$. Therefore, no core may be available to execute new jobs before $EST_{i,j}(v_p)$. Thus, $EST_{i,j}(v_p)$ is a lower bound on the availability time of any free core in v'_p.

Fact 2. If $J_{i,j}$ is not the first segment of job J_i (i.e., $j > 1$), then it already has a claimed core and does not execute on a free core. Thus, the availability of each free core in v_p remains the same in v'_p. In combination with Fact 1, we get $PA \supseteq \left\{\max\{EST_{i,j}(v_p), A_x^{min}(v_p)\} \mid 1 \leq x \leq m - |\mathcal{C}(v_p)|\right\}$ and $CA \supseteq \left\{\max\{EST_{i,j}(v_p), A_x^{max}(v_p)\} | 1 \leq x \leq m - |\mathcal{C}(v_p)|\right\}$.

Fact 3. Because we assume a work-conserving scheduler, if $J_{i,j}$ is the first segment of J_i (i.e., $j = 1$), it will start executing on the first available free core in v_p. All the other cores will thus keep the same availability interval. Therefore, in combination with Fact 1, we get that $PA \supseteq \left\{\max\{EST_{i,j}(v_p), A_x^{min}(v_p)\} \mid 2 \leq x \leq m - |\mathcal{C}(v_p)|\right\}$ and $CA \supseteq \left\{\max\{EST_{i,j}(v_p), A_x^{max}(v_p)\} | 2 \leq x \leq m - |\mathcal{C}(v_p)|\right\}$.

Fact 4. If $J_{i,j}$ is not the first or last segment of job J_i (i.e., $1 < j < n_i$), then it does *not* release the core claimed by J_i. Thus, the set of free core in v'_p is the same as in v_p.

Fact 5. If $J_{i,j}$ is the last segment of J_i (i.e., $j = n_i$), then it releases the core claimed by J_i at the end of its execution. Thus, the core that was claimed by J_i in v_p becomes free for other jobs to execute on in v'_p. That released core is available

at the earliest at the EFT of $J_{i,j}$, and at the latest at the LFT of $J_{i,j}$. Hence, $PA \supseteq EFT_{i,j}(v_p)$ and $CA \supseteq LFT_{i,j}(v_p)$.

Fact 3 proves the first case in the definitions of PA and CA, respectively. The combination of Facts 3 and 5 prove the respective second cases. Fact 4 proves the third cases, and the combination of Facts 2 and 5 proves the fourth case. ■

H. Merge Phase

In order to delay a potential state-space explosion by considering every scenario in a different path, we introduce Rule 1 that merges two nodes of the graph into a single node that covers all states covered by the initial two nodes. This slows down the growth of the SAG (in terms of the number of nodes) while maintaining soundness.

Rule 1 (Merge rule)**.** *Two nodes v_p and v_q are merged if $\mathcal{J}^P = \mathcal{J}^Q$ and $\forall x,\ 1 \leq x \leq m - |\mathcal{C}|,\ A_x(v_p) \cap A_x(v_q) \neq \emptyset$.*

When two states v_p and v_q are merged into v_z, each free-core availability interval $A_x(v_z)$, claimed core availability interval $Cl_x(v_z)$, and shared resource availability interval $SR_x(v_z)$ in the merged state v_z are computed so as to fully cover the intervals of the initial states v_p and v_q.

$$A_x(v_z) = \big[\min\{A_x^{min}(v_p),\ A_x^{min}(v_q)\},\ \max\{A_x^{max}(v_p), A_x^{max}(v_q)\}\big] \tag{16}$$

$$Cl_x(v_z) = \big[\min\{Cl_x^{min}(v_p),\ Cl_x^{min}(v_q)\},\ \max\{Cl_x^{max}(v_p), Cl_x^{max}(v_q)\}\big] \tag{17}$$

$$SR_x(v_z) = \big[\min\{SR^{min}(v_p),\ SR^{min}(v_q)\},\ \max\{SR^{max}(v_p), SR^{max}(v_q)\}\big] \tag{18}$$

We now prove that a merge maintains soundness.

Lemma 12. *For two states v_p and v_q merged according to Rule 1, the set of claimed cores in v_p and v_q are assigned to the same job segments.*

Proof: Every job that has started and did not yet finish its execution until reaching state v_p has a claimed core in v_p. Since by Rule 1, the set of job segments in the path to v_p and v_q are identical (i.e., $\mathcal{J}^P = \mathcal{J}^Q$), all jobs that claimed core in v_p must also have a claimed core in v_q, and all jobs that claimed a core in v_q must also have claimed a core in v_p. ■

Lemma 13. *For two states v_p and v_q merged according to Rule 1, the number of free-core availability intervals are the same in v_p and v_q.*

Proof: Lemma 12 proves that the claimed cores in v_p and v_q are assigned to the same job segments. Thus, the number of claimed cores must be the same in v_p and v_q. Hence, the number of free cores must also be the same since the sum of claimed and free cores is always equal to m. It follows that we have as many free-core availability intervals in v_p as in v_q. ■

Theorem 2. *Merging two states v_p and v_q according to Rule 1 and Equations 16, 17, and 18 is safe, i.e., it does not remove any potentially reachable system state from the graph.*

Proof: Rule 1 ensures that the sets of segments that have started executing on the path to v_p and v_q are identical for v_p and v_q. Hence, the set of segments that still need to execute in the merged state v_z is the same as in v_p and v_q.
According to Lemmas 12 and 13, the set of claimed cores and the number of free cores in state v_p and v_q are the same, and are thus the same in the merged state too. Since the availability interval of shared resources and free and claimed cores in the merged state v_z are the union of those in v_p and v_q, any possible combination of a given number of cores and set of resources becoming available at a given time that is possible in either state v_p or v_q is also possible in the merged state v_z. Hence, all sequences of segment executions that may follow from v_p and v_q are also possible in v_z and the set of all system states reachable from v_z includes every state that is reachable from the original states v_p and v_q. ■

I. Correctness

We now put all the pieces together and establish soundness.

Theorem 3. *For any execution scenario such that segment $J_{i,j}$ completes at t, there is a path $\langle v_1, ..., v_p, v'_p\rangle$ in the SAG such that $J_{i,j}$ is the label of the edge connecting v_p to v'_p and $t \in [EFT_{i,j}(v_p), LFT_{i,j}(v_p)]$.*

Proof: Assume that there is a path $\langle v_1, ..., v_p\rangle$ such that the claim is respected for all segments that started to execute before $J_{i,j}$ in the execution scenario that led $J_{i,j}$ to finish at time t. Furthermore, assume that the availability intervals of state v_p correctly bound the actual availability times of the shared resources, free cores, and claimed cores.

We prove that $t \in [EFT_{i,j}(v_p), LFT_{i,j}(v_p)]$, that a new system state v'_p is created from scheduling $J_{i,j}$ in v_p, and that the availability intervals of state v'_p correctly bound the actual availability times of the shared resources, free cores, and claimed cores after executing $J_{i,j}$.

Under the inductive assumption stated above, Lemma 1 and Lemma 6 prove that $EST_{i,j}(v_p)$ and $LST_{i,j}(v_p)$ are safe lower- and upper-bounds on the start time of $J_{i,j}$, respectively. Because segments execute non-preemptively, Equations (12) and (13) are thus safe lower- and upper-bounds on t (i.e., we proved that $t \in [EFT_{i,j}(v_p), LFT_{i,j}(v_p)]$). Further, by the inductive assumption, the condition of Theorem 1 must hold for $J_{i,j}$ and Line 7 of Alg. 1 ensures that the graph is expanded with a new node v'_p. Then, by Lemmas 10 and 11 and the discussion of Alg. 2, the availability intervals of v'_p correctly bound the actual availability of shared resources and cores afters executing $J_{i,j}$. Therefore, the inductive assumption is respected for v'_p. Finally, according to Lemma 10, potentially merging v'_p with another node (lines 14 to 18 of Alg. 1) maintains the validity of the inductive assumption.

Crucially, the inductive assumption (i.e., correct availability intervals) obviously holds for v_1 (in which all cores and shared

resources are supposed to be available) and thus follows by induction on all the states created by Alg. 1. ■

V. Empirical Evaluation

We conducted a large-scale schedulability study to evaluate any gains in accuracy and to test for scalability limitations.

A. Setup and Workloads

To obtain a large corpus of diverse workloads with varied contention characteristics, we generated task sets as follows. For a given number of cores $m \in \{2, 4\}$, we generated $n \in \{m + 1, 2m, 3m, 5m\}$ periodic tasks that shared $n^r \in \{2m, 5m\}$ resources. Each task was configured to have a number of critical sections chosen uniformly at random from $\{0, 1, \ldots, n^{cs}\}$, for $n^{cs} \in \{5, 15\}$. The length of each critical section was drawn uniformly at random from either $[1\mu s, 15\mu s]$, $[10\mu s, 50\mu s]$, or $[50\mu s, 150\mu s]$ (*short*, *intermediate*, or *long* critical sections, respectively). Additionally, to obtain Fig. 2b, task sets for a specific setup with $m = 6$, $n = 12$, $n^{cs} = 5$ and $n^r = 12$ were generated as well.

For each considered combination of m, n, n^r, n^{cs}, and critical section lengths, we varied the total utilization U from 5% to 95% in steps of 5, and for each U, we generated 250 task sets, for a total of more than $450{,}000$ task sets.

For a given U and n, we generated n per-task utilizations u_1, u_2, $u_3, \ldots$ that sum to U using Emberson et al.'s *RandFixedSum* method [17]. To reflect that nonpreemptive scheduling is used in practice only for workloads with short jobs, for each task, an initial cost value C_i' was drawn from a normal distribution with mean $3ms$ and standard deviation $1.5ms$ (values of less than $10\mu s$ were re-drawn). To avoid unrealistic periods, we then selected the task period $T_i \in \{5, 8, 10, 12, 15, 20, 25, 30, 50, 75, 100, 150, 200, 250, 300, 500\}ms$ closest to C_i'/u_i and set the task's WCET to $u_i \cdot T_i$ (but no less than the sum of maximum critical section lengths).

The total WCET was randomly distributed across the task's segments $C_{i,1}^{max}, C_{i,2}^{max}, \ldots$ while respecting the corresponding maximum critical-section lengths (if any). Each segment was assigned a BCET $C_{i,j}^{min}$ by drawing a value uniformly at random from $[0.1C_{i,j}^{max}, 0.5C_{i,j}^{max}]$. Each critical section was assigned a minimum length of $0\mu s$ with probability 0.25, and otherwise chosen at random as $L_{i,j}^{min} \in [0.1L_{i,j}^{max}, 0.5L_{i,j}^{max}]$.

Finally, tasks were assigned unique priorities using the DkC heuristic for fixed-priority scheduling [14]. Afterwards, as a necessary test, one hyperperiod of the task set was simulated assuming that each job executes for its WCET (without any blocking). If the simulation already exhibited a deadline miss, then the task set was discarded to avoid generating task sets that are obviously infeasible under nonpreemptive scheduling.

B. Implementation and Baselines

We implemented the proposed analysis, which we denote as {**EDF, FP**}**-SAG-SR** in the following (under global nonpreemptive EDF and FP scheduling, respectively), by extending Nasri et al.'s open-source SAG tool [2,23].

As there exists no directly comparable analysis to compare against (the proposed solution is the first of its kind), we further considered the following loosely related baselines to provide context: 1) {**EDF, FP**}**-SAG-NO-BLOCKING**: Nasri et al.'s analysis [23] *without any blocking*. This analysis is not sound in the presence of shared resources; it merely serves to indicate an upper bound on attainable schedulability. 2) {**EDF, FP**}**-SAG-INF**: Nasri et al.'s analysis [23] with an *inflation-based blocking analysis* where each job's WCET is increased prior to response-time analysis to account for possible blocking based on the holistic blocking analysis approach [9, Ch. 5.4]. This analysis is sound, but structurally pessimistic (not scenario-aware), and thus provides a simple lower bound on schedulability that the proposed analysis should exceed. 3) **EDF-NO-BLOCKING**: the schedulability tests for *preemptive* global EDF provided in the open-source SchedCAT library [1] *without any blocking*, as an upper bound on schedulability under preemptive EDF scheduling. 4) **FP-NO-BLOCKING**: similarly, the schedulability tests for *preemptive* global FP scheduling in SchedCAT without any blocking. 5) {**EDF, FP**}**-FMLP-SHORT**: inflation-based holistic blocking analysis [9] of non-preemptive FIFO-ordered spin locks (i.e., "FMLP for short resources" [7]), as provided in SchedCAT. 6) {**EDF, FP**}**-OMLP**: inflation-based holistic blocking analysis of the *suspension-based* global OMLP locking protocol [11], as provided in SchedCAT. 7) **FP-FMLP–LONG**: Yang et al.'s analysis [28] of the suspension-based "FMLP for long resources" [7]. 8) **FP-PIP**: similarly, Yang et al.'s analysis [28] of the suspension-based PIP [16,25].

C. Results

In total, our experimental setup considered 14 individual analyses for over 95 different scenarios (i.e., parameter combinations). Due to space constraints, we cannot report the complete set of results here; a representative selection of our results is shown in Fig. 2 with the specific parameter choices given in each plot. To avoid clutter, only a subset of the 14 curves is shown in each graph.

Fig. 2a shows a comparison of the various SAG analyses for $m = 4$. First, there is little difference w.r.t whether jobs are prioritized according to fixed task-level DkC priorities [14] or by job-level EDF priorities. More importantly, however, the results show a large gain in schedulability relative to the inflation-based baseline. For example, while FP-SAG-INF only deemed about 15% of the task sets to be schedulable at 45% total utilization, FP-SAG-SR shows that more than 60% of the tested task sets are actually schedulable. Overall, the proposed analysis closes *more than half of the gap* between the inflation-based baseline and the upper bound on attainable schedulability represented by {EDF, FP}-SAG-NO-BLOCKING, which highlights a significant reduction in pessimism.

Fig. 2b focuses on EDF scheduling and shows similar trends for $m = 6$, with EDF-SAG-SR attaining much higher schedulability than EDF-SAG-INF. Furthermore, a comparison with EDF-OMLP and EDF-FMLP-SHORT shows the proposed analysis to be competitive with the state of the art for *preemptive* systems (for the considered workloads), especially considering that the EDF-NO-BLOCKING baseline

reveals that preemptive scheduling has a slight advantage in this scenario. Note that our evaluation does not reflect any differences in scheduling and runtime overheads, which can be expected to be (much) lower under nonpreemptive scheduling. Fig. 2c shows largely identical trends for $m = 4$.

Fig. 2d shows an analogous comparison for FP scheduling; here preemptive scheduling has a significant advantage and Yang et al.'s suspension-aware analysis [28] of the PIP and the "long" FMLP variant stand out as particularly effective. However, note again that this is a somewhat lopsided comparison due to workload differences (preemptive vs. nonpreemptive) and since we are discounting lower spin-lock overheads.

Fig. 2e shows the effect of increasing the number of tasks. Generally, schedulability drops as n increases, which is unsurprising as an increase in tasks is correlated with an increase in contention. Nonetheless, significant accuracy gains are apparent compared to the inflation-based baseline. For example, for $n = 5$ and $U = 60\%$, FP-SAG-SR shows more than 50% of the workloads to be schedulable, whereas FP-SAG-INF can show this for less than 20% of the workloads.

In Fig. 2f, we look at the effect of varying the number of cores. Here we clearly see that, for all investigated values of m, the proposed analysis performs better than the inflation-based baseline. Schedulability drops with increasing m since an increase in parallelism corresponds to an increase in contention (since $n = 2m$ in the shown scenarios).

In Figs. 2g and 2h, we investigate the runtime of the algorithm as a function of the total utilization, number of tasks and number of cores. Since the experiments have been run on several different machines (all being server-class machines with large amounts of DRAM), we cannot relate the runtime results to a single machine configuration. However, there are still interesting trends to be observed. For example, in Fig. 2g, the runtime increases with the utilization until $U \approx 30\%$. From 30% onwards, however, the runtime decreases. This is explained by the fact that when the utilization increases, we deal with increased contention, which results in more branching during the analysis, thereby leading to an increased runtime. However, when the utilization becomes larger, there is also an increasing chance to find more task sets to be unschedulable at an early stage of the analysis. Therefore, the graph construction is stopped earlier and the runtime decreases. We also observe in Fig. 2g that the runtime increases significantly with the number of tasks, which we link to an increasing number of jobs in the observation window and thus an increasing number of possible job segments execution orderings. In Fig. 2h we investigate the impact of varying the number of cores on the runtime (note that the number of tasks also changes in that experiment since $n = 2m$). Non-surprisingly, we observe that the runtime increases exponentially with the number of tasks and cores. However, even in the worst configuration shown in Fig. 2h, 95% of the generated task sets have an analysis runtime below 2.7 min. In the configurations of Fig. 2g, more than 95% of the task set are analyzed in less that 40 min. when $n \leq 12$, but task sets start to time out when $n = 20$.

Our evaluation also revealed some limitations. First, note how in Fig. 2e the results for FP-SAG-SR for $n = 20$ tasks are worse than for the baseline FP-SAG-INF, but *not* for lower task counts. This is a result of the increased runtime of the proposed analysis. In our experiments, we configured a timeout of 120 minutes per task set. As the runtime of the analysis increases with n, FP-SAG-SR started to exhibit a significant number of timeouts for $n = 20$. Scalability limitations also meant that while it is certainly possible to analyze a given system and even perform experiments for a chosen range of parameters when $m > 4$ (e.g., Fig. 2b shows results for $m = 6$), the general increase in runtime made an evaluation across the entire parameter space (i.e., *hundreds of thousands of workloads*, recall Sec. V-A) impractical for $m \geq 6$.

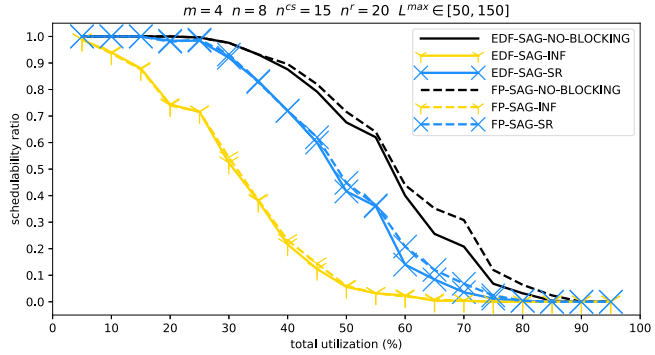
Surprisingly, we also found some rare cases where {EDF, FP}-SAG-SR perform (slightly) worse than the corresponding {EDF, FP}-SAG-INF analyses even in the absence of timeouts. Further investigation led us to discover a source of pessimism related to the encoding of shared-resource availability. Our current state abstraction assumes that the resource remains locked (and thus unavailable) for the whole time from SR^{min} until SR^{max} in the worst case. Although accurate in most cases, this interval may become very long in comparison to the actual critical section length in corner cases. There are thus rare situations where even an inflation based test may perform better. Such occasional instances can be easily handled by running both analyses and retaining the better result.

Nonetheless, we overall conclude that *almost* always the proposed analysis shows substantial accuracy gains in comparison to the state of the art and, for the considered workloads, is competitive with (or even superior to) solutions designed for preemptive systems. The positive results virtually across the board provide ample motivation to further improve accuracy and scalability of the proposed approach in future work.

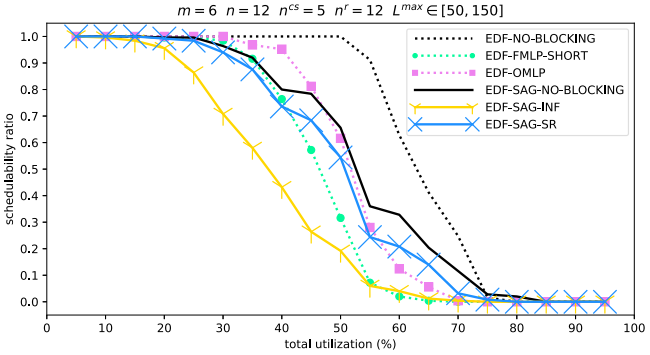
VI. Conclusion

We presented a new WCRT analysis for global JLFP scheduling and non-preemptive tasks with repeating job-release patterns that share resources protected by FIFO spin locks. To the best of our knowledge, it is the first solution to this problem. Our analysis implicitly explores all possible execution and resource access orderings using a novel system-state abstraction that explicitly models resource contention.

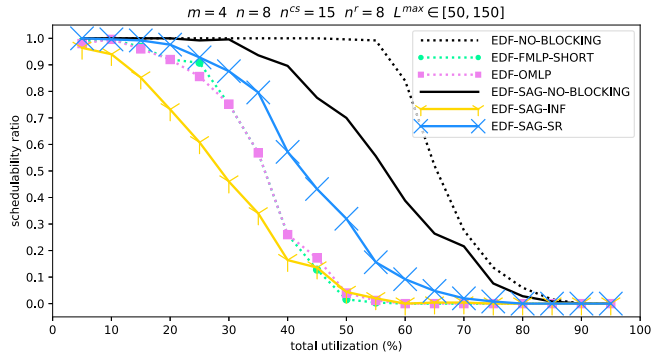
A comparison with inflation-based analyses has shown that our work is substantially less pessimistic. The empirical evaluation has also shown that, for the type of workloads considered in the experiments, our solution comes much closer to hypothetical blocking-free upper bounds and is competitive even with solutions for preemptive systems, suggesting that our analysis successfully discards many more scenarios that reflect impossible combinations of shared resources access orders, low-priority blocking and/or high-priority interference. We also identified an intermittent source of pessimism, which we plan on addressing at the same time as we work on scaling the analysis to larger numbers of core counts using more aggressive pruning techniques (such as partial-order reduction) to disregard scenarios that do not contribute to the worst case.



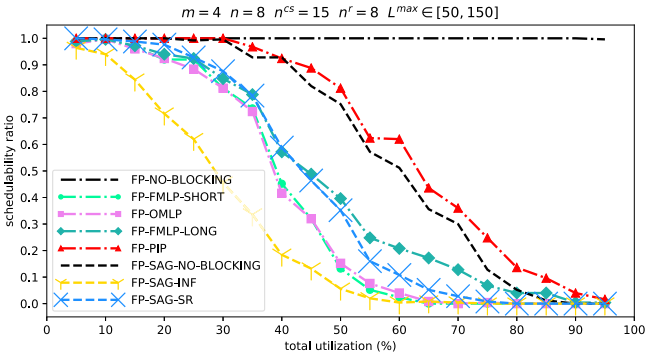
(a) comparison of SAG analyses for $m = 4$



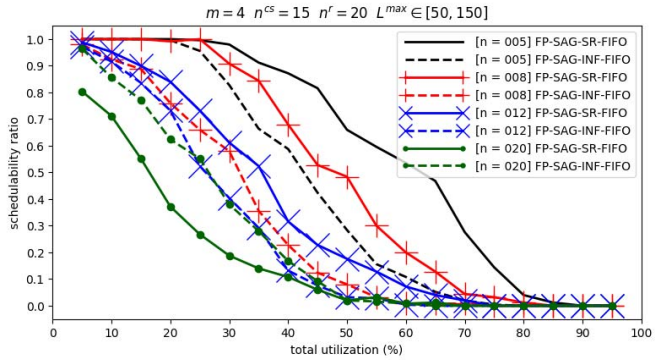
(b) comparison of EDF analyses for $m = 6$



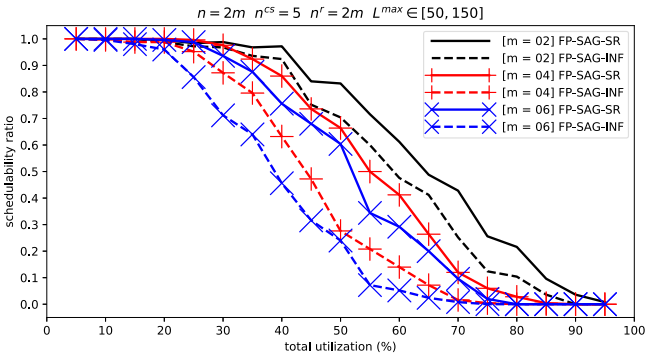
(c) comparison of EDF analyses for $m = 4$



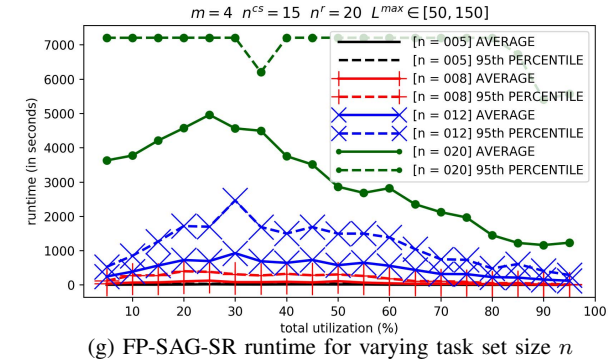
(d) comparison of FP analyses for $m = 4$



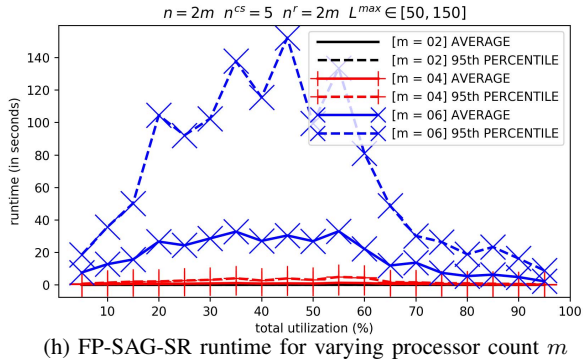
(e) schedulability for varying task set size n



(f) schedulability for varying processor count m



(g) FP-SAG-SR runtime for varying task set size n



(h) FP-SAG-SR runtime for varying processor count m

Fig. 2: Select experimental results. See Sec. V-B for an explanation of the considered analyses.

REFERENCES

[1] SchedCAT: The schedulability test collection and toolkit. https://github.com/brandenburg/schedcat.

[2] `nptest`: an implementation of schedule-abstraction graph analysis. https://github.com/brandenburg/np-schedulability-analysis.

[3] James H Anderson, Rohit Jain, and Kevin Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 346–355. IEEE, 1998.

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

[5] Alessandro Biondi and Björn B Brandenburg. Lightweight real-time synchronization under p-edf on symmetric and asymmetric multiprocessors. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 39–49, 2016.

[6] Alessandro Biondi, Björn B Brandenburg, and Alexander Wieder. A blocking bound for nested FIFO spin locks. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS)*, pages 291–302. IEEE, 2016.

[7] Aaron Block, Hennadiy Leontyev, Björn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE international conference on embedded and real-time computing systems and applications (RTCSA 2007)*, pages 47–56. IEEE, 2007.

[8] Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation*. John Wiley & Sons, 2018.

[9] Björn B Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[10] Björn B Brandenburg. Multiprocessor real-time locking protocols: A systematic review. *arXiv preprint arXiv:1909.09600*, 2019.

[11] Björn B Brandenburg and James H Anderson. Optimality results for multiprocessor real-time locking. In *2010 31st IEEE Real-Time Systems Symposium*, pages 49–60. IEEE, 2010.

[12] Yang Chang, Robert I Davis, and Andy J Wellings. Reducing queue lock pessimism in multiprocessor schedulability analysis. In *Proceedings of the 18th International Conference on Real-Time Networks and Systems (RTNS)*, pages 99–108, 2010.

[13] Travis S Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS)*, pages 148–157. IEEE, 1993.

[14] Robert I Davis and Alan Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 398–409. IEEE, 2009.

[15] UmaMaheswari C Devi, Hennadiy Leontyev, and James H Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 75–84, 2006.

[16] Arvind Easwaran and Björn Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 377–386. IEEE, 2009.

[17] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the Synthesis of Multiprocessor Tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'10)*, 2010.

[18] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 73–83. IEEE, 2001.

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

[20] Philip Holman and James H Anderson. Locking in pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 149–158. IEEE, 2002.

[21] Mitra Nasri and Björn B Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23. IEEE, 2017.

[22] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. A response-time analysis for non-preemptive job sets under global scheduling. In *30th Euromicro Conference on Real-Time Systems*, pages 9:1–9:24, 2018.

[23] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. Response-time analysis of limited-preemptive parallel dag tasks under global scheduling. In *31st Euromicro Conference on Real-Time Systems*, pages 21:1–21:23, 2019.

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

[25] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[26] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104. IEEE, 2000.

[27] Alexander Wieder and Björn B Brandenburg. On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks. In *2013 IEEE 34th Real-Time Systems Symposium (RTSS)*, pages 45–56. IEEE, 2013.

[28] Maolin Yang, Alexander Wieder, and Björn B Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *2015 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12. IEEE, 2015.