applied system innovation

MDPI

Article

Polynomial Exact Schedulability and Infeasibility Test for Fixed-Priority Scheduling on Multiprocessor Platforms

Natalia Garanina *,†, Igor Anureev † and Dmitry Kondratyev †

The Artificial Intelligence Research Center, Novosibirsk State University, 630090 Novosibirsk, Russia; i.anureev@g.nsu.ru (I.A.); d.kondratev2@g.nsu.ru (D.K.)
* Correspondence: n.garanina@g.nsu.ru
† These authors contributed equally to this work.

Abstract: In this paper, we develop an exact schedulability test and sufficient infeasibility test for fixed-priority scheduling on multiprocessor platforms. We base our tests on presenting real-time systems as a Kripke model for dynamic real-time systems with sporadic non-preemptible tasks running on a multiprocessor platform and an online scheduler using global fixed priorities. This model includes states and transitions between these states, allows us to formally justify a polynomial-time algorithm for an exact schedulability test using the idea of backward reachability. Using this algorithm, we perform the exact schedulability test for the above real-time systems, in which there is one more task than the processors. The main advantage of this algorithm is its polynomial complexity, while, in general, the problem of the exact schedulability testing of real-time systems on multiprocessor platforms is NP-hard. The infeasibility test uses the same algorithm for an arbitrary task-to-processor ratio, providing a sufficient infeasibility condition: if the real-time system under test is not schedulable in some cases, the algorithm detects this. We conduct an experimental study of our algorithms on the datasets generated with different utilization values and compare them to several state-of-the-art schedulability tests. The experiments show that the performance of our algorithm exceeds the performance of its analogues while its accuracy is similar.

Keywords: real-time systems; exact schedulability test; sufficient infeasibility test; fixed priority; non-preemptive tasks; Kripke model

check for updates

Academic Editor: Christos Douligeris

Received: 19 November 2024
Revised: 30 December 2024
Accepted: 15 January 2025
Published: 20 January 2025

Citation: Garanina, N.; Anureev, I.; Kondratyev, D. Polynomial Exact Schedulability and Infeasibility Test for Fixed-Priority Scheduling on Multiprocessor Platforms. *Appl. Syst. Innov.* **2025**, *8*, 15. https://doi.org/10.3390/asi8010015

1. Introduction

For safety critical real-time systems, the problem of determining whether tasks are completed on time is crucially important. In real-time systems theory, such a problem is called *the schedulability test*. The opposite test for failure to meet a deadline in a system is *the infeasibility test*. In particular, these concepts are closely related to a response-time analysis which the paper [1] proposes for the AUTOSAR (AUTomotive Open System ARchitecture), an open automotive software standard with a multi-core OS architecture specification. The authors analyze AUTOSAR inter-core task synchronization in the context of non-preemptive fixed-priority multi-core scheduling. The schedulability analysis of Google's multi-Tensor Processing Unit (TPU) edge AI accelerators is another important case for sporadic non-preemptive fixed-priority rigid gang scheduling [2].

For real-time systems with tasks running on a single processor or strictly assigned to one of the processors of a multiprocessor platform, algorithms for the schedulability test are well developed and studied [3]. However, for real-time systems running on multiprocessor platforms with an arbitrary allocation of tasks among processors, this problem is NP-hard for most schedulers [4]. At the same time, there are polynomial algorithms for the narrow classes of systems [3]. Therefore, various algorithms [5–8] are developed to test the sufficient conditions for the schedulability of systems, which evaluate real-time systems pessimistically: if the conditions are met, the system is considered to have passed the test, otherwise, its schedulability cannot be estimated. In [9], it is shown that often, with such an approach, a significant number of schedulable systems, up to 50%, can be rejected as probably non-schedulable. Similar results are shown by experiments in which sufficient tests are compared with each other [6–8]. Therefore, the development and improvement of exact methods remains a relevant task for various types of systems with different scheduling methods [9–14].

In addition to the development of specialized methods for exact schedulability testing, there are a number of works that use general formal methods from programs and systems analysis [15]. General formal methods can provide models, algorithms, and capabilities for the exact solution of the main problem of real-time system schedulability, which are not presented in specialized approaches. For example, Refs. [16,17] study a fixed-priority scheduler representation for a multiprocessor system and systems of non-preemptive self-suspending tasks using timed automata. The paper [18] models a special case of the system in the Promela language [19] of the SPIN verifier [20], and the paper [21] presents a Promela model for a single-processor system. In [22], the authors use graph games to simplify the reachability problem in the exact schedulability test. In a recent paper [14], we used the Kripke model to formalize a real-time system with an abstract scheduler. We implement this abstract formalization for the global fixed priority and the earliest-deadline priority schedulers, as well as preemptive and non-preemptive tasks, in Promela [23]. Our experiments with the model checker SPIN and Promela models for these real-time systems show that the exact schedulability test is only reliably completed for five processors and six tasks. As we increase the number of processors and tasks, the SPIN test begins ti take too long. This expected result motivates us to develop a more practical approach, as described in this paper.

In this paper, we study the problem of an exact schedulability test for systems in which sporadic tasks have a global fixed priority (GFP) and are not preemptive (NP). Such systems are known to be particularly sensitive to task parameters such as their execution time and relative deadline. At the same time, the class of real-time systems in which sporadic tasks should not be preempted and have a strictly defined priority—NP-GFP systems—is quite wide. In such systems, task preemption can be expensive due to the complexity or impossibility of restoring the context of interrupted jobs or computations. This class includes, for example, systems for distributing resources (memory, physical space in a warehouse) among occasionally arriving consumers (computing processes, cargo), message transmission systems with different message importance and/or urgency, big data processing systems, the trajectory and velocity planning of emergency rescue vehicles for collision avoidance [24], critical exception processing tasks in avionics and spacecraft industry control systems, etc. There are several works on schedulability tests for variants of such systems [1,2,6–8,17,25–28]. We discuss them in Section 5.

We propose an approach to the schedulability test problem that uses a formal representation of real-time systems as Kripke models, which are widely used in program and system verification [29]. This representation allows us to formulate the schedulability test problem as a reachability problem: any task of a real-time system may miss its deadline if the "bad state" corresponding to this situation is reachable in the corresponding Kripke model of the system. Using the inverse transition relation of the Kripke model for real-time systems in which there is one less processor than there are tasks, we distinguish 14 possible cases to analyse the backward reachability from the bad state. Five of these cases are unreachable from the initial state, which means that the system is schedulable. The remaining cases show that the missed deadline state is reachable, so in these cases, the system fails the schedulability test. Each of these cases can be described in terms of comparisons of task parameters, namely their execution times and deadlines. We integrate these comparisons into two first-order logical inequality predicates that take into account all the tasks in the system. The truth of these predicates corresponds to a negative verdict of the schedulability test for the real-time system under test. Thus, an algorithm that checks the truth of these predicates for the tasks of the system performs an exact schedulability test. The time complexity of this exact test is quadratic in the number of tasks. Since these predicates correspond to the exact non-schedulability of real-time systems with one less processor than there are tasks, we show that they can be used for an algorithm that partially solves the non-schedulability problem for systems with an arbitrary ratio of processors to tasks. This algorithm, which tests the sufficient condition for non-schedulability, also has quadratic time complexity. In addition, we propose an algorithm for partitioning the system into clusters of k tasks and $k-1$ processes, while simultaneously applying our schedulability test. Comparative experiments show that our schedulability and non-schedulability tests have a significant advantage in execution speed even compared to sufficient tests, and the non-schedulability testing algorithm is almost as accurate as the state-of-the-art algorithm from [28].

The rest of the paper is organized as follows. In Section 2, we recall the basic definitions of real-time systems and scheduling and formalize real-time systems with non-preemptive tasks and a global fixed priority scheduler as Kripke models. Section 3 presents the main results of our work: an analysis of the backward reachability from bad states and algorithms for schedulability and infeasibility tests that exploit this analysis. We describe our experiments in Section 4, providing details of the implementation. In Section 5, we discuss related work and our results. Our conclusion is given in Section 6.

2. A Real-Time Kripke Model for a Real-Time System with a Dynamic NP-GFP Scheduler

We consider that *a real-time system* is a set of tasks $T = (T_1, ..., T_n)$, where each *task* $T_i = (C_i, D_i, P_i)$ has *an execution time* C_i, *a relative deadline* D_i, and *a minimum period* P_i. Each task $T_i \in T$ can generate a potentially infinite number of *jobs*, each of which requires C_i units of time. These jobs must be completed before D_i time units after the release time. Release time instants are separated by at least P_i time units. If there are no other restrictions on the jobs' releases, these tasks are referred to as *sporadic tasks*. In this paper, we study the base case of real-time systems in which all task parameters are integers and $C_i < D_i \leq P_i$. All jobs are executed on *m processors*. If the number of processors is less then the number of tasks ($m < n$), we need *a scheduler* to decide which job to run next. We assume that scheduling decisions are taken at discrete time instants, starting from 0. The *schedulability test problem* is used to detect whether each job of each task in a real-time system is completed before its deadline. We say that *a real-time system is safe (schedulable)* if the schedulability test gives a positive verdict for it. If the schedulability test gives a negative verdict for a real-time system, then this system is *unsafe (infeasible)*. In the rest of the paper, we fix the real-time system T described above.

The scheduler must specify the conditions under which jobs are allowed to run on available processors. There are many types of schedulers that use different conditions and combinations of them. Some of the conditions are listed below.

- Global Fixed Priority (GFP). The set of tasks is ordered: T_1 has the highest priority, T_n has the lowest priority, and a job of a task with a lower number takes precedence over a job of a task with a higher number.
- Early Deadline-First (EDF). The job with an earlier deadline has a higher priority. This priority is dynamic, because tasks (and their jobs) can have different priorities at different points in time.
- Non-preemption. No job can be preempted by another job.
- Preemption. A job may be interrupted by another job with a higher priority.
- Dynamic scheduling. In a multiprocessor system, the jobs of each task can be executed on different processors (including the part of the job remaining after preemption).
- Static scheduling. In a multiprocessor system, all the jobs of each task are executed on one predetermined processor.

In our work, we focus on *a dynamic scheduler with global fixed priority for non-preemptive jobs (NP-GFP)*.

Kripke models are used, in particular, in formal methods for model checking programs and systems [29]. In this paper, we apply them to formalize the schedulability test problem as a problem of the (un)reachability of missed deadline states. Recall the definition of a Kripke model. Let a set of atomic propositions $Prop$ be given. *A Kripke model* is a tuple $M = (S, S_0, R, L)$, where

- S is a finite set of states of S;
- $S_0 \subseteq S$ is a set of initial states;
- $R \subseteq S \times S$ is a total transition relation between states;
- $L : Prop \longrightarrow 2^S$ is an evaluation function that connects the states of the model and the truth of atomic propositions.

Inspired by paper [4], we introduce the current values of the tasks' parameters as follows. For every task $i \in T$, let tuple $s_i = (i, C'_i, D'_i, P'_i, rel_i, bad_i)$ be *a state of task i*, where

- $C'_i \leq C_i$ is the time remaining until the job of this task is completed;
- $D'_i \leq D_i$ is the time until the deadline for completing a job of this task;
- $P'_i \leq P_i$ is the time until the next permissible release of a job of the task;
- $rel_i \in \mathbb{B}$ is a Boolean variable that indicates a job release: it becomes $true$ when task i releases a job, and it becomes $false$ when the job is completed.
- $bad_i \in \mathbb{B}$ is a Boolean variable that indicates that a job is about to miss its deadline: it is $false$ if $C'_i \leq D'_i$, and it becomes $true$ otherwise.

For brevity, we refer to the Boolean constants $true$ and $false$ as **1** and **0**, respectively. For representing the processors' load, we introduce variable $busy$: a number of jobs that are currently being executed ($busy \leq m$). Every global state s of our model is composed of tasks' states and the state for the processors' load. For global state s, let $s.C'_i$, $s.D'_i$, $s.P'_i$, $s.rel_i$, $s.bad_i$, and $s.busy$ be projections of s on its components for every $i \in [1..n]$.

In our Kripke model, for the dynamic non-preemptive GFP-scheduler, we use a predicate $go(i, s, t)$ that is $true$ if task i starts to execute the job at the s-successor state t, and $false$ otherwise: $go(i, s, t) \equiv (|Maj_i| + s.busy < m) \vee s.C'_i < C_i$, where $Maj_i = \{j \in [1..n] \mid j < i \wedge t.rel_j \wedge t.C'_j = t.C_j\}$ is the set of released jobs with higher GFP priority that have not yet started execution. Note that when $s.C'_i < C_i$, the job of task i is executed and since in this case $go(i, s, t) = true$, it continues to execute, i.e., it is not preempted.

To estimate the change in the processor load $busy$, we need to compute the amount of change $cl(s, t)$. This number takes into account the number of tasks whose jobs have just been approved for execution by the scheduler and the number of tasks whose jobs have just completed. To compute $cl(s, t)$, we introduce predicate $fin(i, s)$ for just completed jobs that is $true$ if task i completes its job in state s, and $false$ otherwise: $fin(i, s) \equiv s.C'_i = 0 \wedge s.rel_i$.

We treat $go(i,s,t)$ and $fin(i,s)$ as integer numbers (1 for *true* and 0 for *false*), so $cl(s,t) = \sum_{i=1}^{n}(go(i,s,t) - fin(i,s))$.

Let $Prop$ be a set of propositions consisting of arithmetic comparisons of current values of tasks' parameters and propositions about a number of running jobs.

We define the real-time system T with a dynamic non-preemptive GFP-scheduler as *the real-time Kripke model* $M^T = (S^T, s_0^T, R^T, L^T)$, where

- The finite set of states $S^T = \prod_{i=1}^{n}(\{i\} \times [0..C_i] \times [0..D_i] \times [0..P_i] \times \mathbb{B} \times \mathbb{B}) \times [0..m]$ for global state $s \in S^T$, $s_i = (i, C_i', D_i', P_i', rel_i, bad_i)$ is a projection of s on task i;
- The initial state $s_0^T = \prod_{i=1}^{n}\{(i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})\} \times \{0\}$ represents the one-element set of initial states;
- The total transition relation $R^T \in S^T \times S^T$ is defined by composing relations for i-task projections of global states s and t as follows.
  $(s,t) \in R^T$ if $t.busy = s.busy + cl(s,t)$ and one of the following points holds:
  1. $s_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$, and
     (a) $t_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$—task i does nothing;
     (b) $t_i = (i, C_i, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $\neg go(i,s,t)$—task i releases a job and the job is not started;
     (c) $t_i = (i, C_i - 1, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $go(i,s,t)$—task i releases a job and the job is immediately started;
  2. $s_i = (i, C_i', D_i', P_i', \mathbf{1}, \mathbf{0})$, $t_i = (i, C_i', D_i' - 1, P_i' - 1, \mathbf{1}, \mathbf{0})$ with $s.D_i' > 0$, and $\neg go(i,s,t)$—task i is waiting for permission to run its job;
  3. $s_i = (i, C_i', D_i', P_i', \mathbf{1}, \mathbf{0})$, $t_i = (i, C_i' - 1, D_i' - 1, P_i' - 1, \mathbf{1}, \mathbf{0})$ with $0 < s.C_i' < C_i$, $0 < s.D_i' < D_i$, $s.C_i' \leq s.D_i'$, and $go(i,s,t)$—task i executes its job;
  4. $s_i = (i, 0, D_i', P_i', \mathbf{1}, \mathbf{0})$, or $s_i = (i, C_i, D_i', P_i', \mathbf{0}, \mathbf{0})$, or $s_i = (i, C_i, D_i, P_i', \mathbf{0}, \mathbf{0})$, and
     (a) if $s.P_i' = 0$
        i. $t_i = (i, C_i - 1, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $go(i,s,t)$—task i completes its job normally, and it releases a new job, that starts immediately;
        ii. $t_i = (i, C_i, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $\neg go(i,s,t)$—task i completes its job normally, and it is releases a new job that does not start immediately;
        iii. $t_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$—task i completes its job normally, and it does not release a new job;
     (b) if $s.D_i' = 0$ and $s.P_i' > 0$, then $t_i = (i, C_i, D_i, P_i' - 1, \mathbf{0}, \mathbf{0})$—task i completes its job normally and waits for the next release;
     (c) if $s.D_i' > 0$, then $t_i = (i, C_i, D_i' - 1, P_i' - 1, \mathbf{0}, \mathbf{0})$—task i completes its job normally and waits for the next release;
  5. $s_i = (i, C_i', C_i' - 1, P_i', \mathbf{1}, \mathbf{0})$ and $t_i = (i, C_i', C_i' - 1, P_i', \mathbf{0}, \mathbf{1})$—a job of task i. Task i will definitely miss the deadline, so task i goes to the "bad" state;
  6. $s_i = (i, C_i', C_i' - 1, P_i', \mathbf{0}, \mathbf{1})$ and $t_i = (i, C_i', C_i' - 1, P_i', \mathbf{0}, \mathbf{1})$—task i is in this "bad" state forever.
- The evaluation function $L : Prop \longrightarrow 2^{S^T}$ is standard: it assigns comparison propositions to those states in which they are true.

In M^T, *a path* $\pi = s_0, s_1, \ldots$ is a sequence of states $s_i \in S^T$ such that $\forall i \geq 0$ $(s_i, s_{i+1}) \in R^T$. An *initial path* starts from the initial state. A path can be finite or infinite. We denote the finite path $\pi = s_0, \ldots, s_n$ as $\pi(s_0, s_n)$.

We refer to the state of the real-time system T in which $D_i' = C_i' - 1$ for some tasks i as *a bad state*, since there is no time left for task i to meet its deadline. Bad states in the Kripke model M^T are the set $Bad\_States = \{s \in S^T \mid \exists i \in [1..n] : s.bad_i = \mathbf{1}\}$. The proposition $Bad = \bigvee_{i=0}^{n}(s.bad_i = \mathbf{1})$ describes this set of states. In these settings, the exact schedulability test for the real-time system T is to satisfiability check the LTL formula $\Phi_T = \mathbf{G}(\neg Bad)$ in the real-time Kripke model M^T: the real-time system never reaches a state in which some tasks are in their bad state.

3. Backward Reachability-Based Schedulability Test

The idea of our backward reachability analysis for the exact schedulability test comes from model checking techniques for safety properties: if it is impossible to reach initial states of a model from "bad" model states using the inverse of the transition relation of this model, then "bad" states are unreachable and the model is safe. Note that the inverse of the transition relation may not be total, since the set of the model states may include states that are not reachable from the initial states. The set of bad states defined above can be represented as a union of particular bad states: $Bad\_States = Bad_1 \cup \ldots \cup Bad_n$, where $Bad_i = \{s \in Bad \mid s.bad_i = \mathbf{1}\}$. Note that in real-time systems with non-preemptive tasks under a GFP scheduler, when the number of processors is one less than the number of tasks $Bad_i = \{s \in Bad \mid s.bad_i = \mathbf{1} \wedge \forall j \in [1..n] : j \neq i \to s.bad_j = \mathbf{0}\}$, i.e., exactly one task can enter into a bad state, because otherwise there are some points where multiple released tasks do not occupy a free processor. To perform the exact schedulability test in these systems, it is important to check the unreachability of each set B_i. We also suppose that $P_j = D_j$ (actually, in real-time systems, it is often not necessary for a job to complete before it can release again. "This requirement is consistent with the throughput requirement that the system can keep up with all the work demanded of it at all times'' ([3] p. 41).). In the rest of our paper, we consider only such systems.

Let us fix a task i whose job misses its deadline, and the corresponding set Bad_i. In every state $s \in S^T$ which goes to a bad state $b_i \in Bad_i$ ($(s, b_i) \in R^T$), all the processors are busy; this is shown as $s.busy = m$, because in the opposite case, a i-job can capture some processors and start. Based on this observation, we conduct a backward reachability-based analysis. We consider 14 critical cases with different relative occurrences of tasks' release times and execution points closest to the bad state b_i. To carry this out, we analyze "lifelines" of the corresponding tasks in the next section. This analysis shows whether the bad state is reachable from the initial state or whether backward steps from the bad state lead to a state that is unreachable from the initial state. The former means that the real-time system is unsafe (infeasible) and the latter means that the system is safe (schedulable).

3.1. Backward Reachability-Based Case Analysis

To determine the safety (schedulability) of a real-time system under the above conditions, it is important to consider 14 key cases, as shown in Figure 1. In pictures, for every task j, the bold lines represent decreasing values of D'_j (the time remaining until the deadline of a released job) and dashed lines represent decreasing or standing values of C'_j (the time remaining until a job completion). We call these lines *the lifelines* of the corresponding task. The release of a job is indicated by a filled square for the deadline initial value D_j and a filled circle for the execution initial value C_j. The completion of a job is marked with an empty circle ($C'_j = 0$), and the job deadline is marked with an empty square ($D'_j = 0$). If the exact release or completion time of a job is not important, the job lifeline does not end in a square or circle. We consider four types of jobs and their corresponding tasks w.r.t. their lifelines: red, green, blue, and gray jobs/tasks.

- The red job i is a bad job that goes to bad state b^i with a red lifeline. Its release occurs in state r^i. The previous release and execution of the red task occurs in state e^i_p or re^i_p, with completion in state f^i_p.

- The green job j is a job with the last release and execution in state re^j before bad state b^i and with completion after the bad state. It has a green lifeline. Its previous release is in state r^j_p and its previous starting execution point is in state e^j_p.
- THe blue job k with the blue lifeline is the last one before state re^j, which is released and starts executing in state re^k and finishes after bad state b^i. All other jobs of the other tasks start even earlier and are supposed to be completed after bad state b^i. They are irrelevant for our analysis except for the fact that they occupy $m-2$ processors until the bad state.
- The gray job with the gray lifeline represents the red or blue job. It is started for execution in state e^l and is completed in state f^l.

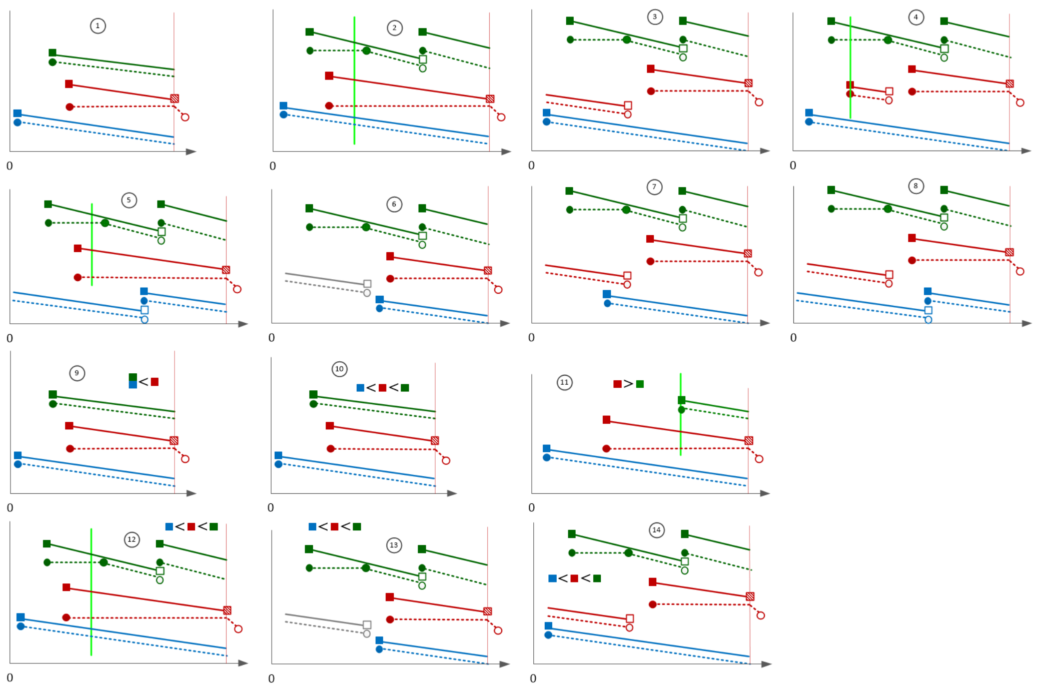


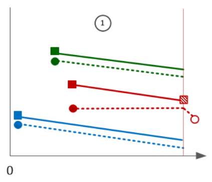
Figure 1. Backward reachability-based case analysis.

The vertical light red line in the pictures represents a bad state: the moment when the time until the deadline is less than the execution time (the diagonal square) but job execution has not started (the horizontal execution lifeline). The vertical light green line marks the impossible state when (1) at least two jobs are released but neither has started or (2) the priority is violated. This state cannot be reached from the initial state, hence conditions imposed on the deadline and time of execution for pictures with green lines are a criterion for safe (schedulable) real-time systems. Briefly speaking, every picture without the green line shows that there is a sequence of job releases that results in a missed deadline and every picture with the green line shows that every reverse sequence of transitions from a bad state results in an impossible system state that cannot be reached from the initial state (in fact, for each case there is only one possible reverse sequence).

The picture cases describe all the relevant combinations of the relative occurrence of release/start states for the red, green, and blue jobs (r^i, b^i, re^i_p, e^i_p, f^i_p, re^j, e^j_p, r^j_p, re^k, e^k_p, e^l, f^l) and their priorities ($i > j$ and $i > k$, $i < j$ and $i < k$, $k < i < j$, and the case $j < i < k$ is included in case $i < j$), since the number of processors is one less than the number of tasks. Let state s *be earlier than* s' ($s \preceq s'$) if there is a finite path from s to s' ($(s, s') \in R^{T*}$). When the order of states on a path does not matter, we write $s \sim s'$, and states with the same position on a path are considered equal $s = s'$. Each case can be described in terms of execution times and deadlines of these picture jobs. We say that *the case is safe* if there are some states in the picture case that are unreachable from the initial state of M^T. Otherwise, *the case is unsafe*.

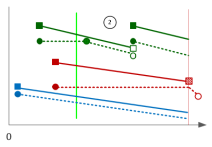
The first eight pictures in Figure 1 show the cases where the red task i is the last task: $i = n$. The last release of the red task before the bad state is in state r^i and none of the other tasks completes its job before b^i. Between r^i and b^i, the remaining execution time C_i' is stable and equal to C_i, but the time until deadline D_i' decreases.

Case 1.



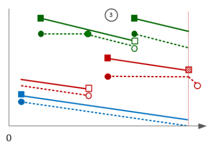
$i > j$ and $i > k$: $re^k \preceq re^j \preceq r^i \preceq b^i$—all jobs are released before the last release of the red job. This case is unsafe. Every task $j < i$ can start its job before (or in) state r^i, since there are m processors. Since none of them complete their job before b^i, i-job goes into the bad state due to its late release.
The condition for this picture is $\forall j \neq i: \; D_i < C_j$.

Case 2.



$i > j$ and $i > k$: $re^k \preceq r_p^j \preceq r^i \preceq e_p^j \preceq re^j \preceq b^i$—the green job is released after the last release of the red job.
This case is safe. Task $j > i$ releases and starts its job in state re^j between r^i and b^i. To prevent i-job from running in the state $prev(re^j)$ that precedes state re^j, task j must also run its job in that state. Therefore, $prev(re^j)_j = (j, 1, 1, \mathbf{1}, \mathbf{0})$ due to the period restriction $P_j = D_j$. Consider the state e_p^j which is in C_j transitions before re^j. This state is the beginning of the previous execution of j-job. During the path $\pi(e_p^j, re^j)$, $D_j' = C_j'$ due to the definition of relation R^T (non-preemptively). In $D_j - C_j$ transitions before e_p^j, state r_p^j keeps the previous release of j-job. On the path $\pi(r_p^j, e_p^j)$, C_j' is stable: $C_j' = C_j$ due to $D_j' = C_j' = C_j$ at state e_p^j. Let the other $m-1$ blue tasks start their jobs before r_p^j (other cases are in items 5–8 of this list). The last release of the red job may occur before or after e_p^j. The latter case is described in items 3, 4, and 6–8 of this list. The release of the red task before e_p^j corresponds to the state where the green task j and the red task i do not execute their tasks, despite the fact that one processor is free (the others are busy with other tasks). This state is unreachable from the initial state due to the transition relation R^T.
The condition for this picture is $\exists j \neq i: \; (D_i \geq 2 \cdot C_j \wedge (\forall k \neq j \wedge k \neq i: \; C_k \geq D_j + C_j))$.

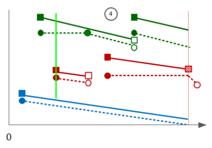
Case 3.



$i > j$ and $i > k$: $(re^k \sim e_p^i) \preceq r_p^j \preceq (e_p^j = f_p^i) \preceq r^i \preceq re^j \preceq b^i$—as Case 2—but the last red release occurs after the previous green execution state, and the previous red completion

occurs in the previous green execution state.
This case is unsafe. Here, we consider the last red release r^i between the green release re^j and the state e^j_p, after which the green job is executed. Certainly, the red task can start and then complete its job at e^j_p to prevent the green task from starting (the other $m-1$ blue tasks start their job before r^j_p). Therefore, there is a sequence of releases from the initial state that leads to the bad state.
The condition for this picture is $\exists j \neq i : \ (C_j \leq D_i < 2 \cdot C_j \wedge C_i > D_j - C_j \wedge (\forall k \neq j \wedge k \neq i : \ C_k \geq D_j + C_j))$.

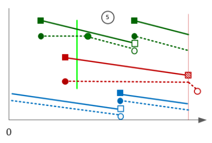
Case 4.



$i > j$ and $i > k$: $re^k \preceq r^j_p \preceq re^i \preceq (e^j_p = f^i_p) \preceq r^i \preceq re^j \preceq b^i$—as Case 3—but the previous red execution state occurs after the previous green release.
This case is safe. It is almost the same as the previous case, but the execution time of the red job is less than the stable period of the green-dashed lifeline. Therefore, the red release and execution cannot prevent green from executing because of the fixed priority: if the green and red tasks are ready to start jobs at the same time, the green job must start. There is no path in M^T that leads to such a situation.
The condition for this picture is $\exists j \neq i : \ (C_j \leq D_i < 2 \cdot C_j \wedge C_i \leq D_j - C_j \wedge (\forall k \neq j \wedge k \neq i : \ C_k \geq D_j + C_j))$.

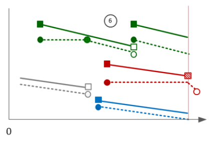
Similar to the above analysis of the lifelines of red and green tasks where the blue task started before r^j_p, the next four cases consider the releases of the blue task k that occur between the last two releases of the green task: $r^j_p \preceq re^k \preceq re^j$.

Case 5.



$i > j$ and $i > k$: $e^k_p \preceq r^j_p \preceq r^i \preceq e^j_p \preceq re^k \preceq re^j \preceq b^i$—as Case 2—but the last blue release occurs after the previous green execution state.
This case is safe. This case repeats the reasoning for Case 2 and it is easy to see that any release position of the blue task between r^j_p and re^j does not affect this reasoning.
The condition for this picture is $\exists j \neq i : \ (D_i \geq 2 \cdot C_j \wedge (\forall k \neq j \wedge k \neq i : \ C_k \geq C_j))$.

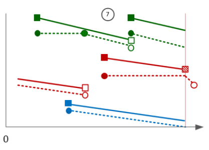
Case 6.



$i > j$ and $i > k$: $e^l \preceq r^j_p \preceq (e^j_p = f^l) \preceq re^k \preceq r^i \preceq re^j \preceq b^i$—as Case 3—but the last blue release occurs after the previous green execution state and before the last release of the red job.
This case is unsafe, and is similar to Case 3. Here, we consider the blue release between the red release and state e^j_p, after which the green job is executed. Since there are two releases after e^j_p, either a red or blue task can complete their job at e^j_p to prevent the green job from starting. Again, there is a sequence of task releases from the initial state that leads to the

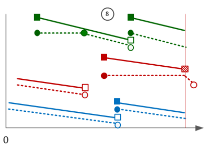
bad state.
The condition for this picture is $\exists j \neq i: \ (C_j \leq D_i < 2 \cdot C_j \wedge C_i > D_j - C_j \wedge (\forall k \neq j \wedge k \neq i: \ D_i \leq C_k \leq 2 \cdot C_j))$.

Case 7.



$i > j$ and $i > k$: $e^i_p \preceq r^j_p \preceq re^k \preceq (e^j_p = f^i_p) \preceq r^i \preceq re^j \preceq b^i$—as Case 6—but the last blue release precedes the previous green execution state.
This case is almost the same as the previous one, except that the last blue release occurs between r^j_p and e^j_p. This fact obliges the red task to prevent the green one from being started, as in Case 3. As before, there is a path from the initial state that leads to the bad state.
The condition for this picture is $\exists j \neq i \ : \ (C_j \leq D_i < 2 \cdot C_j \wedge C_i > D_j - C_j \wedge (\forall k > j : 2 \cdot C_j \leq C_k \leq D_j + C_j))$.

Case 8.



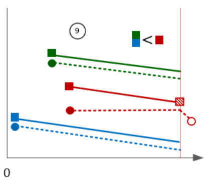
$i > j$ and $i > k$: $(e^k_p \sim e^i_p) \preceq r^j_p \preceq (e^j_p = f^i_p) \preceq r^i \preceq re^k \preceq re^j \preceq b^i$—as Case 7—but the last blue release follows the last release of the red job.
This case is unsafe. In the previous case, we move the blue release to a position between the red release and the green last release. Due to the stability of the red dotted lifeline, the reasoning for the previous blue states is similar to the reasoning for the previous green states (Case 2). Combined with the need to prevent the green job from being executed, as in the previous cases, we again obtain a sequence of releases from the initial state that leads to a bad state.
The condition for this picture is $\exists j \neq i: \ (C_j \leq D_i < 2 \cdot C_j \wedge C_i > D_j - C_j \wedge (\forall k \neq j \wedge k \neq i: \ C_j < C_k < D_i))$.

We resume the above reasoning regarding Cases 1–8 as follows.

Lemma 1. *If $i = n$, then T is unsafe if $\forall j \neq i: \ (D_i < C_j)$ or $\exists j \neq i: \ (C_j \leq D_i < 2 \cdot C_j \wedge C_i > D_j - C_j)$.*

The last six pictures in Figure 1 show the cases where the red task i is the greatest task: $i = 1$ (Picture 9) and the cases where the red task i is the middle task: $1 < i < n$ (pictures 10–14).

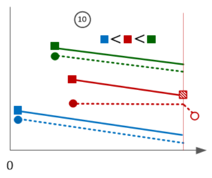
Case 9.



$i < j$ and $i < k$: $re^k \preceq re^j \preceq r^i \preceq b^i$—all jobs are released before the last release of the red job.
This case is unsafe. Every task $j > i$ can start its job before (or in) state r^i, since there are m processors. Since none of them complete their jobs before b^i, the i job goes into the bad

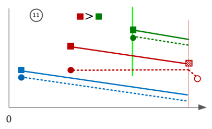
state due to its late release and non-preemptivity.
The condition for this picture is $\forall j \neq i: \ D_i < C_j$.

Case 10.



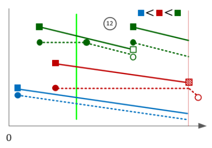
$k < i < j$—$re^k \preceq re^j \preceq r^i \preceq b^i$—all jobs are released before the last release of the red job.
This case is unsafe. We have exactly the same case as the previous one: again, each task $j < i$ starts its job before (or in) state r^i, and i job misses its deadline because there is no free processor to execute it.
The condition for this picture is $\forall j \neq i: \ D_i < C_j$.

Case 11.



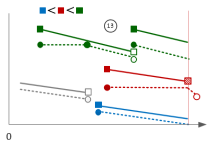
$i < j$: $re^k \preceq r^j_p \preceq r^i \preceq e^j_p \preceq re^j \preceq b^i$—the green job is released after the last release of the red job.
This case is safe. Let there be a green task $j > i$ that releases and starts its job in state re^j between r^i and b^i. However, this is an impossible situation, since the red task must start its job at re^j due to the fixed priority. The following pictures 12–14 consider only the cases where (blue) tasks with lower priority than the red task release and start executing their jobs before r^i.
The condition for this picture is $\exists j > i: \ (D_i \geq C_j \wedge (\forall k \neq j \wedge k \neq i: \ C_k \geq C_j))$.

Case 12.



$j < i < k$: $re^k \preceq r^j_p \preceq r^i \preceq e^j_p \preceq re^j \preceq b^i$—the green job is released after the last release of the red job.
This case is safe for the same reason as Case 2.
The condition for this picture is $\exists j < i: \ (D_i \geq 2 \cdot C_j \wedge (\forall k \neq j \wedge k \neq i: \ C_k \geq D_j + C_j))$.

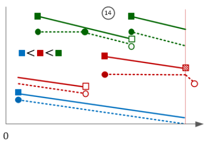
Case 13.



$j < i < k$: $e^k_p \preceq r^j_p \preceq r^i \preceq e^j_p \preceq re^k \preceq re^j \preceq b^i$—as Case 2—but the last blue release occurs after the previous green execution state.
This case is unsafe for the same reason as Case 6.

The condition for this picture is $\exists j < i : \; (C_j \le D_i < 2 \cdot C_j \land C_i > D_j - C_j \land (\forall k < i : \; D_i \le C_k \le 2 \cdot C_j))$.

Case 14.



$j < i < k$: $(re^k \sim e^i_p) \preceq r^j_p \preceq (e^j_p = f^i_p) \preceq r^i \preceq re^j \preceq b^i$—as Case 2—but the last red release is after the previous green execution state and the previous red completion is at the previous green execution state.
This case is unsafe for the same reason as Case 3.
The condition for this picture is $\exists j < i : \; (C_j \le D_i < 2 \cdot C_j \land C_i > D_j - C_j \land (\forall k < i : \; C_k \ge D_j + C_j))$.

We resume the above reasoning regarding Cases 9–14 as follows.

Lemma 2. *If $i < n$, then T is unsafe iff $\forall j \ne i : \; (D_i < C_j)$ or $\exists j < i : \; (C_j \le D_i < 2 \cdot C_j \land C_i > D_j - C_j)$.*

3.2. Algorithms for Schedulability and Infeasibility Tests

Summarizing Lemmas 1 and 2, we formulate a general criterion for checking the safety (schedulability) and unsafety (infeasibility) for real-time systems with n tasks executed on $n-1$ processors:

Theorem 1. *The real-time system T with n tasks executed on $n-1$ processors is unsafe (infeasible) if there exists task i, such that*

- $\forall j \ne i \, (D_i < C_j)$, *or*
- $\exists j < i \, (C_j \le D_i < 2 \cdot C_j \land C_i > D_j - C_j)$.

Algorithm 1 is based on this theorem. It returns "NO" if an input real-time system is unsafe (infeasible) and "YES" if this system is schedulable. Obviously, the time complexity of this algorithm is $\mathcal{O}(n^2)$, which makes our schedulability test using this algorithm faster than any state-of-the-art exact schedulability test for a real-time system with the non-preemptive global fixed-priority scheduler running on a multiprocessor platform.

Algorithm 1 The base algorithm for the exact schedulability test

```
Input: real-time system $T$ with $n$ tasks and $n-1$ processors.
Output: YES/NO
1: for $i = 1$ to $n$ do
2:     if $\forall j \ne i \, (D_i < C_j)$ then return NO
3:     end if
4:     if $\exists j < i \, (C_j \le D_i < 2 \cdot C_j \land C_i > D_j - C_j)$ then return NO
5:     end if
6: end for
7: return YES
```

It is easy to see that if we add several tasks to the system T, the above unsafety criteria remain the same but the safety criteria (pictures with bright green lines) do not work in the new settings, since they are based on the absence of tasks for the full load of processors, and in the presence of additional tasks, this assumption is violated. In this case, Algorithm 1 can be redefined as Algorithm 2 in order to perform a pessimistic sufficient infeasibility test for a real-time system. Algorithm 2 returns "NO" if an input real-time system is unsafe (infeasible) and "UNKNOWN" if the unsafety criterion is false. It recognizes real-time systems that are unschedulable even if the number of processors is just one less than the number of tasks.

Algorithm 2 The algorithm for the sufficient infeasibility test

```
Input: real-time system $T$ with $n$ tasks and $m$ processors ($m < n - 1$).
Output: NO/UNKNOWN
1: for $i = 1$ to $n$ do
2:     if $\forall j \neq i \; (D_i < C_j)$ then return NO
3:     end if
4:     if $\exists j < i \; (C_j \leq D_i < 2 \cdot C_j \wedge C_i > D_j - C_j)$ then return NO
5:     end if
6: end for
7: return UNKNOWN
```

Note that both algorithms have quadratic time complexity with respect to the number of tasks, since they contain two nested loops that iterate and pairwise compare all the tasks of the input system in turn. The outer loop is specified by the "for" statement, and the two successive inner loops are specified by the universal and existential quantifiers. Our algorithms do not use backward reachability as such. Backward reachability is necessary for analyzing cases 1–14 of the bad state reachability. The combinatorial nature of backward reachability in our work disappears due to the condition of the number of tasks and processors $n = m + 1$: under this restriction, the backward paths from bad states are uniquely defined by inequalities on the values of the time parameters of the system's tasks. These inequalities are provided for each of the cases 1–14. These cases generalize into the inequalities of Theorem 1, which are used in Algorithms 1 and 2 and are computed in constant time.

3.3. Using the Schedulability Test for Checking Systems with an Arbitrary Number of Processors

So far, we have assumed that our scheduler uses dynamic scheduling: in a multiprocessor system, such a scheduler determines the next (according to its priority) ready task to be executed by any of the freed processors. In static scheduling on a multiprocessor system, only one of many processors is assigned a set of tasks in advance, and the scheduler schedules each task for its assigned processor. We propose *semi-dynamic scheduling*: the set of tasks is divided into groups in advance, each of which is assigned not one processor, as in static scheduling, but several ones. This technique allows us to effectively use our schedulability test proposed in the previous section. The idea is to partition the task set into *1-test groups* that include (1) $n_i + 1$ tasks to be executed on n_i processors, and (2) the remaining groups of n_j tasks to be executed on m_j processors, such that the numbers n_j and m_j are small enough for exact schedulability testing these groups, using known methods such as [9,17] or using the SPIN verifier [14]. Such partitioning depends on the ratio of the number of tasks and processors. In general, this partition problem can be represented as a version of the well-known bin-packing problem, widely used in the analysis of real-time systems [3,30]. The bins here are multiprocessor clusters, and items are tasks. This version is quite complex, since the size and number of bins depend on both the size and number of items.

We propose the following algorithm that constructs 1-test groups for a desired partition of a real-time system T, taking into account the task utilization C_i/D_i and partially tests T. Let procedure $Test1(T)$ use Algorithm 1 to return a schedulability verdict for a real-time system T, and procedure $sort(T)$ sort the tasks of real-time system T by their utilization value from highest to lowest. The following procedures preserve this sorting:

$first(T,k)$ returns the first k tasks in T, $head(T)$ returns the first task in T removing it from T, $remove\_first(T,k)$ removes the first k tasks from T, and $replace(T,k,x)$ replaces task k in T with task x. Our Algorithm 3 takes a real-time system running on a multiprocessor platform as the input and returns a set of 1-test groups for this system or a result FAIL, which means that the algorithm cannot find a partition with schedulable groups of the required size. The algorithm first sorts the tasks of the real-time system T by their utilization from largest to smallest. It then successively divides the number of processors by two (line 4), testing the schedulability of the corresponding number of the first most-expensive tasks using Algorithm 3 (line 8). If it finds that this set of tasks is not schedulable, it replaces these tasks with less expensive ones in line 13, until it either makes a schedulable set T' (line 17) or fails to do so (line 15).

Algorithm 3 The algorithm for constructing 1-test groups in semi-dynamic scheduling

```
Input: real-time system $T$ with $n$ tasks and $m$ processors ($m < n$).
Output: a set of 1-test groups $G^T$ or FAIL
 1: sort($T$)
 2: $G_1^T \leftarrow \emptyset$
 3: while $m > 1$ or $|T| = m + 1$ do
 4:     $m \leftarrow \lfloor m/2 \rfloor$
 5:     $k \leftarrow m + 1$
 6:     $T' \leftarrow$ first($T$, $k$)
 7:     $T'' \leftarrow T \setminus T'$
 8:     while $\neg$Test1($T'$) and $k > 0$ do
 9:         if $T'' = \emptyset$ then
10:             $T'' \leftarrow$ remove_first($T \setminus T'$, $m + 2 - k$)
11:             $k \leftarrow k - 1$
12:         end if
13:         replace($T'$, $k$, head($T''$))
14:     end while
15:     if $k = 0$ then return FAIL
16:     end if
17:     $G_1^T \leftarrow G_1^T \cup \{T'\}$
18:     $T \leftarrow T \setminus T'$
19: end while
20: $G^T \leftarrow G_1^T \cup \{T\}$
21: return $G^T$
```

Note that this algorithm does not search for an optimal 1-test partition or even a feasible one in the sense that if our algorithm fails to find a partition, it may still exist. However, it can be used as a fast preprocessing tool before using more sophisticated and accurate methods. In the future, we plan to develop more efficient partitioning algorithms for finding 1-test groups.

4. Experiments

In this section, we describe two experiments. The first experiment refers to the schedulability test, and the second refers to the infeasibility test. For the experiments, we develop a framework [31], which is written in SBCL [32], a dialect of the Common Lisp language. We use a laptop with Intel(R) Core(TM) i7-10510U CPU @ 1.80 GHz 2.30 GHz and RAM 16.0 GB.

The first experiment compares our Algorithm 1 with state-of-the-art algorithms in this field. For our experiments, we chose algorithms developed for the same class of real-time systems as in our paper, namely non-preemptible tasks with the GFP scheduler. For our comparison, we selected the most successful algorithms for schedulability testing:

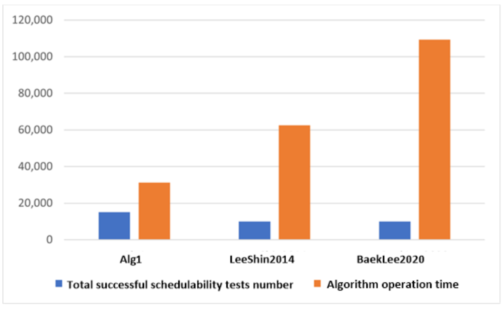
LeeShin2014 from [6] and BaekLee2020 from [8]. Their higher performance compared to other similar algorithms [7,25–27] is confirmed by the experiments presented in the corresponding sections of papers [6,8]. We run experiments on all datasets for every mentioned algorithm and compare their performance with the performance of our algorithm.

We refer to our algorithm and the compared algorithms as ALg1, LeeShin2014, and BaekLee2020, respectively. The experiment uses a dataset consisting of 40,000 task sets. Our experiments satisfy the following requirements:

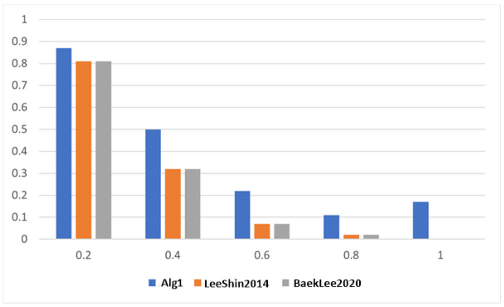
1. A period of every task in every set of the dataset is equal to its deadline: $T_i = D_i$;
2. Every set of tasks in every set of the dataset is deadline-monotonic, i.e., tasks with a lower deadline have a higher priority;
3. The number of tasks is one more than the number of processors: $n = m + 1$.

Here, Items 1 and 2 are the constraints that are used in [6,8] to compare the LeeShin2014 and BaekLee2020 algorithms with previously developed algorithms. Item 3 is the specifics of our algorithm. We construct this dataset using the UUnifast algorithm [33,34], which provides a uniform distribution of sets of tasks within a particular utilization. The paper [8] also takes this algorithm for the experiments. Then, we sort the UUnifast resulting sets of tasks by deadlines to ensure Item 2 of the requirements.

The results of the experiment are shown in Tables 1 and 2 and in Figure 2, respectively. Table 1 and Figure 2a shows the total number of successful tests and their operation time. The algorithm operation time is measured in internal time units of the Lisp machine. The precise meaning of this quantity is implementation-defined; it may measure real time, run time, CPU cycles, or some other quantity. One internal time unit equals $1/n$ of a second for some implementation-defined integer value of n specified in the variable `internal-time-units-per-second`. To show the distribution of successful schedulability tests on utilizations, schedulability indicators in Table 2 and Figure 2b are divided into five utilization classes, characterized by the representatives 0.2, 0.4, 0.6, 0.8, 1.0. The values of the columns with algorithm names contain the ratios of successful schedulability tests to the total number of tests (sets of tasks).



(a)



(b)

Figure 2. Performance of schedulability tests `Alg1`, `LeeShin2014`, and `BaekLee2020`: (**a**) an algorithm operation time, and (**b**) an acceptance ratio.

Table 1. Performance of our schedulability test 1, schedulability test `LeeShin2014`, and schedulability test `BaekLee2020` in terms of algorithm operation time.

Parameters	Alg1	LeeShin2014	BaekLee2020
Total successful schedulability tests number	15,063	10,035	10,035
Algorithm operation time	31,250	62,500	109,375

Thus, our algorithm shows better results in each utilization class compared to the algorithms LeeShin2014 and BaekLee2020. Our algorithm is also faster than these algorithms.

Note that with the constraint $n = m + 1$, the algorithms LeeShin2014 and BaekLee2020 showed the same result on this dataset (both in absolute and relative numbers). Also note that 0.0 for the 1.0 utilization class in the LeeShin2014 and BaekLee2020 algorithms does not mean that these algorithms do not have successful tests. This only means that the number of such tests is negligible (in this particular case, 3 successful tests out of 3271 sets of tasks that fall into this class). It is also worth noting the small growth in the number of successful tests in class 1.0. This artifact is associated with a relatively small total number of tests in this class, due to the monotonization of the dataset.

Table 2. Performance of our schedulability test 1, schedulability test `LeeShin2014`, and schedulability test `BaekLee2020` in terms of acceptance ratio.

Utilisation	Alg1	LeeShin2014	BaekLee2020
$[0.0, 0.2)$	0.87	0.81	0.81
$[0.2, 0.4)$	0.5	0.32	0.32
$[0.4, 0.6)$	0.22	0.07	0.07
$[0.6, 0.8)$	0.11	0.02	0.02
$[0.8, 1)$	0.17	0	0

The second experiment illustrates the performance of our Algorithm 2 on a dataset also generated by the UUnifast algorithm [33]. To the best of our knowledge, there are no algorithms that perform the impossibility test. The only similar algorithm is the algorithm in [28]. However, we cannot compare it with our Algorithm 2 in terms of coverage, since our algorithm starts with an already fixed priority, whereas this algorithm creates a priority during the infeasibility test.

The results of the second experiment are shown in Tables 3 and 4 and in Figure 3. Unlike the first experiment, in this case, we check infeasibility and remove restrictions on the dataset such as $n = m + 1$ and the monotony of deadlines. Just like in the first experiment, we show the distribution of successful tests. In this case, the values of the column with the algorithm name contain the ratio of successful infeasibility tests to the total number of tests. Figure 3 shows an increase in the number of successful tests with increased utilization. Table 3 shows the total number of successful tests and their operation time.

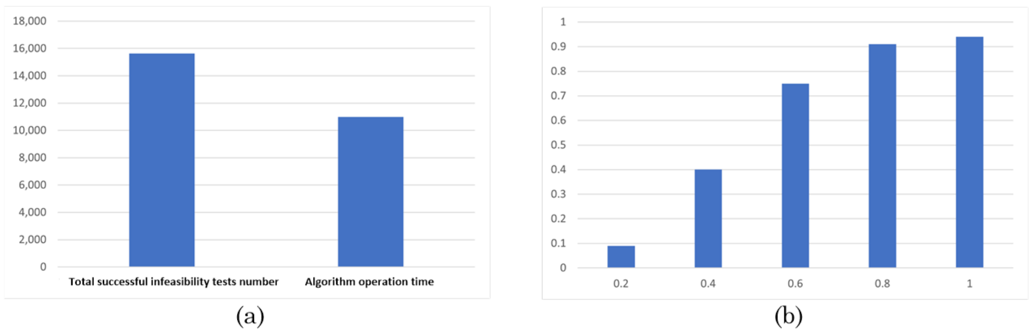


Figure 3. Performance of our infeasibility test 2: (**a**) an algorithm operation time, and (**b**) an acceptance ratio.

Table 3. Performance of our Infeasibility Test 2 in terms of algorithm operation time.

Parameters	Alg2
Total successful infeasibility tests number	15,625
Algorithm operation time	10,995

Table 4. Performance of our infeasibility test 2 in terms of acceptance ratio.

Utilisation	Infeasibility Ratio
$[0.0, 0.2)$	0.09
$[0.2, 0.4)$	0.4
$[0.4, 0.6)$	0.75
$[0.6, 0.8)$	0.91
$[0.8, 0.1)$	0.94

As a side result of the experiments, a software infrastructure was obtained for conducting such experiments in the form of a set of Lisp functions, including the functions of generating datasets, serializing datasets, running algorithms on datasets, evaluating the performance of algorithms, serializing the results of algorithms, etc. The results of the experiments (including datasets) are available at the link [31].

Note, that the datasets are generated by the UUnifast algorithm [33], which provides a uniform distribution of task time characteristics. All algorithms are not statistical, but strict algorithms based on analytical computations. As a rule, when comparing the performance of schedulability test algorithms, task set coverage criteria and time/resource characteristics are used, as shown by the papers we cite. Statistical criteria such as error bars and confidence intervals in schedulability test algorithms can be interpreted as the accuracy of schedulability detection, i.e., algorithm performance (using real-time systems terminology). Unfortunately, an exact schedulability test in general is a very resource-intensive task and can only be used for small systems with no more than 8 processors and 12 tasks, as shown in [17]. Therefore, obtaining error bars and confidence intervals in practice is impossible for the sizes of the datasets that we used in our experiments.

5. Related Work and Discussion

One of the first sufficient schedulability tests for non-preemptive fixed-priority multiprocessor scheduling was described in Guan's paper in el. published in 2008 [25]. The authors bounded the number of tasks doing carry-in jobs in the "problem window" to obtain this test. The definition of carry-in job is given as follows: a carry-in job is released before the interval of interest, but its deadline is within the interval. To bound the number of carry-in jobs, the authors consider jobs that execute as late as possible. This test is later improved to quadratic time complexity [26].

Lee et al. propose a sufficient schedulability test for the same case without the carry-in limitation [6]. In this test, they analyze job interference. Interference is the amount of execution of other tasks that interferes with the execution of the task of interest. The proposed test is based on using the upper bound of interference.

Davis et al. consider a superset of fixed-priority preemptive multiprocessor scheduling and fixed-priority non-preemptive multiprocessor scheduling [27]. They call this superset Global Fixed Priority Scheduling with Deferred Preemption (gFPDS). The authors propose two types of sufficient schedulability test for this case: with and without the carry-in limitation technique. These tests are interesting as generalizations of the two cases: preemptive and non-preemptive scheduling.

In 2017, Lee proposed a test for non-preemptive fixed-priority multiprocessor schedulability based on an improved carry-in limitation technique [7]. This improvement, related to previous research, is to consider the critical instant for a task. The critical instant is the point in time at which a job request results in the longest task response time.

The first exact schedulability test for the special case of fixed-priority non-preemptive multiprocessor scheduling is provided by Yalcinkaya et al. in 2019 [17]. This special case considers self-suspensible tasks with fixed preemption points. The authors reduce schedulability analysis to the reachability problem in synchronized automata (TA). They use the UPPAAL model checker [35] to solve this reachability problem.

Baek and Lee, in 2020 [8], improved the interference upper bound proposed in 2014 [6]. This improvement is based on the analysis of tasks with higher priority than the task of interest.

In [28], Chwa and Li propose the first infeasibility test for fixed-priority non-preemptive multiprocessor scheduling. This approach finds a task priority that makes a set of tasks schedulable. If this algorithm cannot create such a priority for a particular set of tasks, the set is infeasible. The main feature of the proposed approach is the checking of a special constraint on the infeasibility of a task. This constraint is based on an analysis of the bounds of the sum of the total number of executions of higher priority tasks in the problem window for the task of interest. Using this constraint eliminates the need to consider all combinations of priority assignments, since this constraint depends only on the higher-priority tasks. Thus, this approach is based on the constraint testing using a decreasing order of priorities.

The main advantage of our work compared to the above is the quadratic complexity of the exact test for multiprocessor scheduling (Algorithm 1). If a tasks' parameters are unstable, this test can be used without modification at a higher planning level as an online acceptance test to determine the schedulability of the system with changed characteristics, which, however, remains within the specified task-to-processor ratio.

On the one hand, our exact test is directly applicable only to systems with a certain ratio of the number of processors and tasks. Actually, a specific combination of the number of tasks and processors $n = m + 1$ is quite rare in practice, although there is an example of such a combination: map-reduce data processing, where a map is executed in m parallel threads and there is no separate processor for reduce.

But on the other hand, we additionally propose Algorithm 3, which makes our exact test applicable to systems with any tasks-to-processors ratios. This algorithm is the first proof-of-the-concept version for partitioning the set of tasks and processors into groups to which our test can be applied. This partitioning is a variant of the bin-packing problem, widely used in the analysis of real-time systems. Improving this algorithm for more optimal partitions requires a particular intensive study which is out of the scope of this paper. We leave it for our future research. We also suppose that increasing the number of tasks even by 1 ($n = m + 2$) immediately makes the exact schedulability test for NP-GFP systems NP-hard. Similar cases of jump complexity increase are the travelling salesman problem for $n = 3$ and $n = 4$ or 2-SAT and 3-SAT problems. It is imperative that this fact should be proved, but this is out of the scope of our paper. We also leave this proof for the future.

As for the infeasibility testing Algorithm 2, the modification of our schedulability test algorithm can be used for arbitrary task-to-processor ratios, and due to its quadratic complexity, our infeasibility algorithm is rather fast. Unfortunately, we cannot compare our algorithm with the infeasibility test from [28], since that test tries to determine the priority that makes a task set schedulable, and says that the task set is infeasible if no such priority exists, while our infeasibility test deals with a given priority.

6. Conclusions

In our paper, we propose representations of real-time systems as Kripke models and the use of such representations for a new method of schedulability analysis using backward reachability. This method allows us to justify an algorithm for an exact schedulability test that has quadratic complexity for a special class of real-time systems in which there is one more task than the processors. This time complexity is significantly better than the complexity of exact general analysis for such systems. Based on this method, we also develop an algorithm for testing the infeasibility of real-time systems with arbitrary processor-to-task ratios, which also has quadratic time complexity. Comparative experiments show the competitiveness of our approach, since it detects the infeasibility of real-time systems in significantly less time and with slightly worse accuracy. To extend the scope of our exact schedulability test, we propose an algorithm for splitting a set of tasks into groups of k for execution on $k-1$ processors, which simultaneously performs the schedulability test of the grouped tasks.

Our future plans include developing more efficient algorithms for partitioning the system's tasks into execution clusters on a suitable set of processors to improve the applicability of our exact schedulability test to real-time systems. We also plan to adapt our backward reachability method to iteratively decrease the number of processors, or to prove that any decrease in this number leads to a sharp increase in the number and complexity of inequalities used by the algorithm of our method. In addition, we will study the applicability of our method to preemptive tasks and other scheduling priorities.

Author Contributions: Conceptualization, N.G.; methodology, N.G.; software, I.A.; validation, N.G. and I.A.; formal analysis, N.G., I.A., and D.K.; investigation, D.K.; resources, D.K.; data curation, I.A.; writing—original draft preparation, N.G.; writing—review and editing, N.G., I.A., and D.K.; visualization, I.A. and D.K.; project administration, N.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a grant for research centers, provided by the Analytical Center for the Government of the Russian Federation in accordance with the subsidy agreement (agreement identifier 000000D730324P540002) and the agreement with the Novosibirsk State University, dated 27 December 2023 No. 70-2023-001318.

Data Availability Statement: All the data, algorithms, and results of the research presented in this paper are publicly available in the repository https://github.com/anureev/Schedulability-Testing-Framework-and-Experiments under MIT license (accessed on 16 January 2025).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Negrean, M.; Ernst, R. Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources. In Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), Karlsruhe, Germany, 20–22 June 2012; pp. 191–200.
2. Sun, B.; Kloda, T.; Chen, J.; Lu, C.; Caccamo, M. Schedulability Analysis of Non-preemptive Sporadic Gang Tasks on Hardware Accelerators. In Proceedings of the 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS), San Antonio, TX, USA, 9–12 May 2023; pp. 147–160.
3. Liu, J.W.S. *Real-Time Systems*; Prentice Hall: Hoboken, NJ, USA; 2001; 610p.
4. Bonifaci, V.; Marchetti-Spaccamela, A. Feasibility Analysis of Sporadic Real-Time Multiprocessor Task Systems. *Algorithmica* **2010**, *63*, 763 – 780. [CrossRef]
5. Brandenburg, B.B.; Gül, M. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In Proceedings of the 2016 IEEE Real-Time Systems Symposium (RTSS), Porto, Portugal, 29 November–2 December 2016; pp. 99–110. [CrossRef]
6. Lee, J.; Shin, K.G. Improvement of real-time multi-coreschedulability with forced non-preemption. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *25*, 1233–1243. [CrossRef]
7. Lee, J. Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling. *IEEE Trans. Comput.* **2017**, *66*, 1816–1823. [CrossRef]
8. Baek, H.; Lee, J. Improved schedulability test for non-preemptive fixed-priority scheduling on multiprocessors. *IEEE Embed. Syst. Lett.* **2020**, *12*, 129–132. [CrossRef]

9. Burmyakov, A.; Bini, E.; Lee, C.G. Towards a Tractable Exact Test for Global Multiprocessor Fixed Priority Scheduling. *IEEE Trans. Comput.* **2022**, *71*, 2955–2967. [CrossRef]
10. Zhou, Q.; Li, G.; Zhou, C.; Li, J. Limited Busy Periods in Response Time Analysis for Tasks Under Global EDF Scheduling. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *40*, 232–245. [CrossRef]
11. Burmyakov, A.; Bini, E.; Tovar, E. An exact schedulability test for global FP using state space pruning. In Proceedings of the 23rd International Conference on Real Time and Networks Systems, Lille, France, 4–6 November 2015; RTNS '15; Association for Computing Machinery: New York, NY, USA, 2015; pp. 225–234. [CrossRef]
12. Ranjha, S.; Gohari, P.; Nelissen, G.; Nasri, M. Partial-order reduction in reachability-based response-time analyses of limited-preemptive DAG tasks. *Real-Time Syst.* **2023**, *59*, 201–255. [CrossRef]
13. Gohari, P.; Voeten, J.; Nasri, M. Reachability-Based Response-Time Analysis of Preemptive Tasks Under Global Scheduling. In *Leibniz International Proceedings in Informatics (LIPIcs), Proceedings of the 36th Euromicro Conference on Real-Time Systems (ECRTS 2024), Lille, France, 9–12 July 2024*; Pellizzoni, R., Ed.; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Dagstuhl, Germany, 2024; Volume 298, pp. 3:1–3:24. [CrossRef]
14. Garanina, N.O. An Exact Schedulability Test for Real-Time Systems with Abstract Scheduler on Multiprocessor Platforms. *Model. Anal. Inf. Syst.* **2024**, *31*, 474–494. (In Russian) [CrossRef]
15. Cheng, A.M.K. *Real-Time Systems: Scheduling, Analysis, and Verification*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2002.
16. Guan, N.; Gu, Z.; Deng, Q.; Gao, S.; Yu, G. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. In *Proceedings of the Software Technologies for Embedded and Ubiquitous Systems*; Obermaisser, R., Nah, Y., Puschner, P., Rammig, F.J., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4761, pp. 263–272.
17. Yalcinkaya, B.; Nasri, M.; Brandenburg, B.B. An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks. In Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 25–29 March 2019; pp. 1228–1233. [CrossRef]
18. Staroletov, S.M. A Formal Model of a Partitioned Real-Time Operating System in Promela. *Proc. Inst. Syst. Program. RAS* **2020**, *32*, 49–66. [CrossRef] [PubMed]
19. SPIN. Promela Grammar. Available online: http://spinroot.com/spin/Man/grammar.html (accessed on 16 January 2025).
20. Holzmann, G.J. *The Spin Model Checker, Primer and Reference Manual*; Addison-Wesley Professional: Boston, MA, USA, 2003.
21. Sukvanich, P.; Thongtak, A.; Vatanawood, W. Formalizing Real-Time Embedded System into Promela. *MATEC Web Conf.* **2015**, *35*, 03003. [CrossRef]
22. Geeraerts, G.; Goossens, J.; Nguyen, T.V.A. A Backward Algorithm for the Multiprocessor Online Feasibility of Sporadic Tasks. In Proceedings of the 2017 17th International Conference on Application of Concurrency to System Design (ACSD), Zaragoza, Spain, 25–30 June 2017; pp. 116–125. [CrossRef]
23. Garanina, N. The Promela-Model for Real Time Systems with Four Schedulers. Available online: https://github.com/GaraninaN/RealTimeSystems/blob/main/rts.pml (accessed on 16 January 2025).
24. Chen, T.; Cai, Y.; Chen, L.; Xu, X. Trajectory and Velocity Planning Method of Emergency Rescue Vehicle Based on Segmented Three-Dimensional Quartic Bezier Curve. *IEEE Trans. Intell. Transp. Syst.* **2023**, *24*, 3461–3475. [CrossRef]
25. Guan, N.; Yi, W.; Gu, Z.; Deng, Q.; Yu, G. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In Proceedings of the 2008 Real-Time Systems Symposium, IEEE, Barcelona, Spain, 30 November–3 December 2008; pp. 137–146.
26. Guan, N.; Yi, W.; Deng, Q.; Gu, Z.; Yu, G. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *J. Syst. Archit.* **2011**, *57*, 536–546. [CrossRef]
27. Davis, R.I.; Burns, A.; Marinho, J.; Nelis, V.; Petters, S.M.; Bertogna, M. Global and partitioned multiprocessor fixed priority scheduling with deferred preemption. *ACM Trans. Embed. Comput. Syst. (TECS)* **2015**, *14*, 1–28. [CrossRef]
28. Chwa, H.S.; Lee, J. Infeasibility Test for Fixed-Priority Scheduling on Multiprocessor Platforms. *IEEE Embed. Syst. Lett.* **2022**, *14*, 55–58. [CrossRef]
29. Clarke, E.M.; Henzinger, T.A.; Veith, H.; Bloem, R. *Handbook of Model Checking*; Springer: Berlin/Heidelberg, Germany, 2018; Volume 10.
30. Coffman., E.G., Jr.; Csirik, J.; Galambos, G.; Martello, S.; Vigo, D. Bin Packing Approximation Algorithms: Survey and Classification. In *Handbook of Combinatorial Optimization*; Pardalos, P.M., Du, D.Z., Graham, R.L., Eds.; Springer: New York, NY, USA, 2013; pp. 455–531. [CrossRef]
31. Anureev, I. Polynomial Exact Schedulability and Infeasibility Test for Fixed-Priority Scheduling on Multiprocessor Platforms: Experiments. Available online: https://github.com/anureev/Schedulability-Testing-Framework-and-Experiments (accessed on 16 January 2025).
32. Rhodes, C. Sbcl: A sanely-bootstrappable common lisp. In Proceedings of the Workshop on Self-Sustaining Systems, Potsdam, Germany, 15–16 May; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5146, pp. 74–86.

33. Bini, E.; Buttazzo, G.C. Measuring the performance of schedulability tests. *Real-Time Syst.* **2005**, *30*, 129–154. [CrossRef]
34. Davis, R.I.; Burns, A. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In Proceedings of the 2009 30th IEEE Real-Time Systems Symposium, Washington, DC, USA, 1–4 December 2009; pp. 398–409.
35. Model Checking Tool UPPAAL. Available online: http://www.uppaal.com (accessed on 16 January 2025).