Exact Schedulability Analysis for Limited-Preemptive Parallel Applications Using Timed Automata in UPPAAL

Jonas Hansen
DEIS
Aalborg Universitet
Aalborg, Denmark

Srinidhi Srinivasan
IRIS
Eindhoven University of Technology
Eindhoven, The Netherlands

Geoffrey Nelissen
IRIS
Eindhoven University of Technology
Eindhoven, The Netherlands

Kim G. Larsen
DEIS
Aalborg Universitet
Aalborg, Denmark

***Abstract*—We study the problem of verifying schedulability and ascertaining response time bounds of limited-preemptive parallel applications with uncertainty, scheduled on multi-core platforms.**

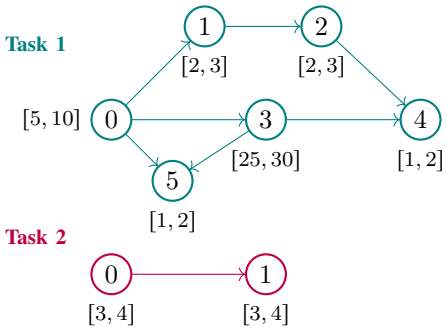
While sufficient techniques exist for analysing schedulability and response time of parallel applications under fixed-priority scheduling, their accuracy remains uncertain due to the lack of a scalable and exact analysis that can serve as a ground-truth to measure the pessimism of existing sufficient analyses.

In this paper, we address this gap using formal methods. We use Timed Automata and the powerful UPPAAL verification engine to develop a generic approach to model parallel applications and provide a scalable and exact schedulability and response time analysis. This work establishes a benchmark for evaluating the accuracy of both existing and future sufficient analysis techniques. Furthermore, our solution is easily extendable to more complex task models thanks to its flexible model architecture.

I. Introduction

Parallel programming enables more efficient use of computing resources on multicore platforms, thus allowing for a reduction of the overall response time of applications. It is therefore also being increasingly employed in real-time systems whose tasks have to respect strict deadlines [1]. However, parallelising real-time applications makes the verification of their timing requirements more challenging. Designing a schedulability or response time analysis that is both accurate and scalable for real-time parallel applications scheduled on multi-core platforms is an open research line [2]–[10].

In this work, we focus on the analysis of systems made of a set of periodic limited-preemptive parallel DAG tasks executing with a work-conserving global fixed-priority (FP) scheduling algorithm under constrained deadlines. With work-conserving global FP scheduling, each task is assigned a fixed priority and a deadline. Whenever a core is free, the highest priority task with pending work is executed on that core. Parallel DAG tasks are divided into a set of execution segments that can be executed in parallel on different cores. Precedence constraints between the execution of those segments are represented by a directed acyclic graph (DAG). Tasks are limited preemptive in the sense that segments execute non-preemptively, i.e., their execution cannot be interrupted even by higher-priority tasks, but tasks can be preempted between the execution of two different segments, i.e., at so-called preemption points. Limited-preemptive is considered to be a good alternative to both non-preemptive and fully preemptive scheduling [11] as carefully chosen preemption points allow to reduce the cost of preemptions in comparison to fully-preemptive scheduling, and, thanks to allowing preemption between segments, it effectively limits the blocking time caused by non-preemptive scheduling.



Task ID	Period	Deadline	Priority
Task 1	350	350	Low
Task 2	60	60	High

Fig. 1: Example of two LPPA tasks defined by their DAG structure and task parameters.

Fig. 1 shows an example of a limited-preemptive parallel application (LPPA) made of two tasks. Each task is modelled with a DAG. Each vertex of the DAG denotes a task segment and each directed edge represents a precedence constraint between two segments. Each segment is defined by a best- and worst-case execution time bound, that is, it may execute non-preemptively for any time duration within the interval defined by those two values. Each edge in the DAG give rise to a preemption point.

The schedulability and response-time analysis (RTA) of limited preemptive *sequential* tasks, i.e., tasks modelled by a DAG with a single path, has been extensively studied over the years. For instance, [11]–[13] propose RTAs for different variations of sporadic limited-preemptive tasks on single-core platforms. Still focusing on a single core, [14] presents an exact analysis for periodic sequential limited-preemptive tasks using Timed Automata (TA) [15] and UPPAAL [16]. [17] proposes an UPPAAL-based analysis for periodic sequential limited-

978-3-9826741-0-0/DATE25/© 2025 EDAA

preemptive self-suspending tasks on multicore platforms. For *parallel* limited-preemptive tasks, only a few analyses have been proposed. [4] presents a sufficient RTA for sporadic partitioned LPPA and [3] proposes a reachability-based analysis for periodic LPPAs that are globally scheduled on multicore.

To the best of our knowledge, no exact analysis currently exists for LPPA. This gap in the state-of-the-art is largely due to the difficulty of capturing the impact of precedence constraints on response time, and to account for scheduling anomalies that come with limited-preemptiveness. The absence of an exact analysis means that the accuracy of existing sufficient analyses remains uncertain as there is no ground truth to compare against.

To address this gap, we develop a generic modelling approach using TA and utilise the UPPAAL verification engine to verify schedulability and to ascertain response time bounds for LPPA instances. Our approach is based on four key observations, developed specifically to improve the scalability of UPPAAL models of LPPAs by minimizing the number of clocks and locations in the TAs, and by using efficient data structures.

II. Key Observations

Generally, automata based frameworks syntactically describe finite state machines defined over a number of discrete locations with directed transitions connecting them. TAs extend the semantics of automata with continuous dynamics using clocks. Clocks are continuous variables capable of guarding discrete transitions from firing using constraints on their current values. The value of a clock increases at a constant rate and can be reset when firing transitions. Additionally, UPPAAL allows us to define and manipulate integer typed data structures through C-like function calls whenever transitions are fired. The semantics of TAs are commonly defined by Timed Labelled Transition Systems in which model state is defined by the value of all clocks and by the currently occupied discrete locations. The value of data structures are also part of the global state in UPPAAL models. Furthermore, UPPAAL allows multiple parallel models that can communicate both through broadcasting and hand shake synchronisation. To initiate a handshake synchronisation, a model fires an output over a channel. This output must synchronise with another model using an input over the same channel. Different from handshakes, broadcasting does not require to have a receiver ready to receive input.

Most often, improving scalability for UPPAAL models comes down to reducing the amount of clocks used, reducing the number of discrete locations defined in the model and utilising memory-efficient data structures whenever possible. In this work, we make four key observations, which we use to improve the scalability of our modelling approach.

Obs. 1. Our first observation is related to the deterministic nature of periodic tasks with constrained deadlines. Releases of jobs are typically done using one clock per task that keeps track of the time until the next job release for that task. That same clock is also usually used to check the relative distance until the job deadline is reached. However, we observe that since the distance between consecutive job releases and deadlines is fixed for all tasks, a single clock can be used to keep track of all the job releases and deadlines of all tasks. This can be done by encoding the delta between consecutive events (i.e., job releases and deadlines) in the UPPAAL model.

Obs. 2. Our second and arguably most significant observation is that, in addition to the single clock that keeps track of job releases and deadlines, the number of clocks needed to model scheduling on a multicore platform is upper bounded by the number of processors in the platform. The typical approach is to define clocks for each task that keep track of when, and for how long their segments execute on processors. The problem with this approach is that the number of clocks is then bounded by the maximum number of segments that can be concurrently pending, which may be significantly larger than the number of processors in the platform. UPPAAL must then decide which subset of clocks defines the evolution of a system state when exploring the state space. In our model instead, clocks are not defined at the level of tasks. They are defined at the level of processors. We define one clock per processor. Then, each clock keeps track of the duration for which the last segment dispatch on the core executed for. This significantly reduces the number of clocks that must be maintained by UPPAAL.

Obs. 3. The set of segments that have all or a subset of their precedence constraints fulfilled is part of what defines the system state as it decides which segments are ready to be dispatched on the processors. A naive implementation would keep track of the status of all segments of all tasks as part of the model state. This is both memory and computationally inefficient. Indeed, whenever a new segment of a task would have to be dispatched on a processor, the model would have to check all segments to know if they are eligible. This would drastically increase the complexity of analysing task models. Instead, we observe that the number of segments of a task that can be simultaneously ready (i.e., those that have not started to execute yet and have all their predecessors completed) is bounded by the maximum parallelism of the task. Similarly, the number of segments that have only a subset of their predecessors completed is also bounded by the structure of the DAG of each task. These two bounds can easily be computed with a single pass through the DAG of each task. The knowledge of those two bounds allows us to use minimal sized lists that keep track of segments with all or part of the precedence constraints completed as part of the model's state.

Obs. 4. Using predominantly asynchronous components by developing our models around broadcast communication utilising channel priorities minimises the number of discrete locations present in the models. As we will see in III, this allows us to model urgency more concisely and precisely, while also lowering exploration overhead in general for the verification engine. This is important because the verification engine of UPPAAL often cannot flatten static sequences of discrete transitions. By utilising less discrete locations we can potentially improve its state-space exploration.

III. UPPAAL Model

In this section, we introduce our modelling approach for LPPA. For the sake of brevity, we only present our model

for limited-preemptive *parallel* applications globally scheduled on multicore platforms. However, a variation of that model optimised for *sequential* tasks was also developed but is not covered in this paper. Both are available in [18].

This section first describes the UPPAAL model and then explains how to use that model to verify schedulability and calculate tasks' response time bounds. We expect the reader to be familiar with basic UPPAAL syntax.

Our model is divided into two distinct components modelling the **scheduler** and **tasks**, respectively. We first discuss the input parameters of the model, Then, for both components, we outline their functional responsibilities and detail how this manifests into its intended behaviour in relation to LPPA under FP scheduling. Doing so, we also explain how the observations discussed in II are exploited in the model.

A. Model Inputs

We begin by explaining central data types, parameters and key data structures used by the model.

The central entities in our model are tasks. A task is defined by its period, release offset, deadline, priority and a set of segments. Each segment is defined by its best- and worst-case execution time `c_min` and `c_max`, respectively. Those properties are encoded using the `task_t` data type defined as: `struct{ time_t period; time_t deadline; time_t offset; int priority; struct{ time_t c_min; time_t c_max;} segments[K];}`

where `time_t` corresponds to the set of natural numbers (encoded on 32-bit integers in UPPAAL) and $K \in \mathbb{N}$ is the maximum number of segments for any task.

Definition 1 (Input Parameters): The model takes the following tuple as input parameters that define the number of tasks, processors and constraints on task execution: $(N, M, K, PER, I, O, B, R, tasks, succs, preds, inits)$, where N is the number of tasks, M is the number of processors, K is the maximum number of segments in a single task, PER : `int[N]` is a list of task periods, I is the maximum amount of predecessors for any segment, O is the maximum number of successors for any segment. To capitalize on Obs. 3 from Sec. II, the model also takes the input parameters B and R where B is the maximum number of segments of a task with part of their precedence constraints fulfilled, and R is the maximum number of ready segments of a task. Finally, $tasks$: `task_t[N]` is a list of N tasks, $succs$: `seg_t[N][K][O]` and $preds$: `seg_t[N][K][I]` are matrices encoding successors and predecessors, respectively, of each segment in the DAG of each task, and $inits$: `seg_t[N][K]` is a sparse matrix denoting which segments are initial segments (i.e., segments without predecessors) in the DAGs of each task, with `seg_t` encoding segments.

B. Model State

In this section, we define the clocks and data structures used by the model to keep track of its internal state.

From Obs. 1 and 2 in Sec. II, we know that we need no more than $M+1$ clocks to model timing constraints. We define `tp : clock` as the global clock that keeps track of the time

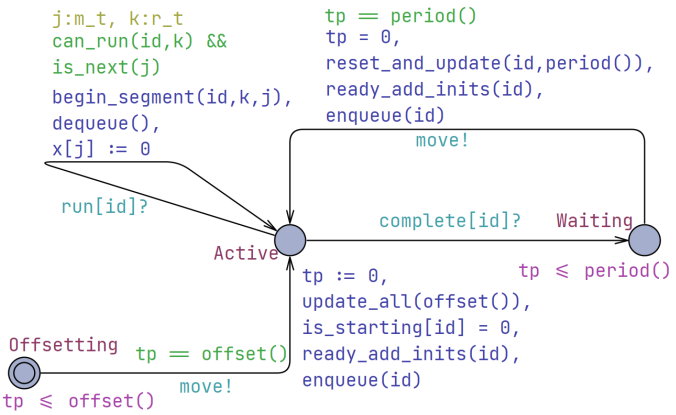


Fig. 2: Task model. The functions `offset()` and `period()` return the time until the next task release relative to the last reset of the clock `tp`. Note, `m_t:int[0,M-1]` and `r_t:int[0,R-1]`.

until the next event in the system (i.e., job release or deadline). A list of M clocks `x : clock[M]` is then used to keep track of execution times on each independent processor.

Additionally, the model controls its internal state by manipulating the data structures `ready : seg_t[N][R]`, `running : bound_t[M]`, `blocked : seg_t[N][B]`, and `inc_preds : seg_t[N][K][I]`, where `running` contains the segments that are currently running, `ready` contains the segments that are ready to start executing, `blocked` contains the segments with some but not all their predecessors completed, and `inc_preds` keeps track of all precedence constraints that are not fulfilled yet, with `bound_t` encoding closed intervals.

To improve efficiency in finding the highest-priority segment that is ready to be executed, we also define $queue$: `id_t[N]` as a queue containing the id of every task with a segment in `ready` in descending priority order (with `id_t: int[0,N-1]` being the task index type).

C. Task Model

The task component consists of N instances of the task model shown in Fig. 2, each of which is instantiated with a unique identifier `id`.

A task can be in three discrete modes, `Offsetting`, `Active` or `Waiting`. A task is `offsetting` until its first segment is released for the first time. The exact timing of the transition is controlled by the invariant `tp ⩽ offset()` and guard `tp == offset()` involving the global clock `tp`. A task becomes `active` from the moment it is first enqueued into the priority queue `queue` until its last segment completes its execution. Note that only `active` tasks may miss their deadlines. A task that completes the execution of all of its segments starts `waiting` until its next period, at which point it enters the priority queue and becomes `active` again. A task cannot be `waiting` for longer than its period and neither can it stop `waiting` before the period has run its full duration.

Transitions between modes is controlled by the broadcast channel `move` used by all tasks and two handshake channels

run and complete activated by the scheduler presented in the next section.

The move broadcast channel is how we keep our models mostly asynchronous, not requiring explicit synchronisations between tasks, which allows us to reduce the number of locations and accelerate state space exploration as noted in Obs. 4 in Sec. II. The move and completed channels are given higher priorities than the run channel. This effectively ensures that, whenever a task may enter or leave the Active state, that transition is urgently taken before the scheduler may pick a new segment to run on one of the processors. This ensures that the set of ready segments and the priority queue are properly updated by the functions enqueue(id) and ready_add_inits(id), before the scheduler checks which segment should be run next.

The scheduler model presented in Sec. III-D is responsible for selecting which ready tasks must start executing segments. It communicates those decisions using the run channel. Whenever the scheduler outputs a run[id]!, it indicates that a segment of task id must start to execute. Then, the task that receives the inputs on run[id]? picks non-deterministically one of the ready segments in ready[id] (using the function can_run(id,k) that checks if there is a ready segment at position k in ready[id] with k being any value in the range $[0, R)$). It then assigns one of the M clocks associated with a processor on which nothing is running (using the function is_next(j) that checks if the j[th] clock is free). Once a ready segment and a clock are selected, the function begin_segment(id,k,j) adds the segment to running[j], and removes it from ready[id]. Finally, dequeue() removes the task id from the priority queue queue if no other segments of this task are in ready[id].

D. Scheduler

The scheduler component model is shown in Fig. 3, consisting of three discrete modes. Two of those modes, including the initial one, are committed (indicated by the C annotation in the location). Committed modes prevent time from passing when the scheduler is in one of them. Committed modes also enforce that the next transition fired in the model must originate from a committed mode. Since the initial location is committed, the first transition to fire in the model is always the one that initialises all lists in the model using the function init_lists(). Once all lists have been initialised, the scheduler becomes Idle.

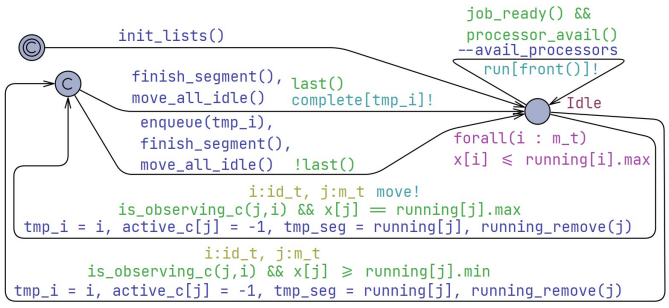


Fig. 3: The scheduler model. the update active_c[j] = −1 releases x[j] and makes it available to be allocated to the highest priority task. Note, tmp_seg and tmp_i are temporary variables.

The scheduler remains Idle when no job is ready, i.e., if ready is empty, or all processors are currently executing segments, i.e., all clocks in x are observing a segment execution. The invariant in Idle ensures that no segment is allowed to run longer than its maximum execution time. If a processor is not executing any segment and the corresponding clock x[i] is not allocated, the model prevents a deadlock from happening by ensuring that running[i].max returns a value that will certainly never be reached by the clock x[i].

Whenever a job is ready and a processor is free, the scheduler selects the ID of the highest priority task in the priority queue queue using the front() function and urgently outputs a run action for that task, thereby ensuring that the work-conserving policy is satisfied. run is an urgent channel, meaning that whenever enabled, time cannot pass, while also cascading urgency to any other enabled channels of higher priority, forcing those higher-priority enabled channels to fire first.

Whenever a segment has reached its upper execution bound, i.e., when there is a clock x[j] that observes the execution of a segment such that x[j]==running[j].max, the scheduler transitions away from Idle. It then sets the clock x[j] as being inactive (by setting active_c[j] = −1) and removes the segment associated with clock x[j] from the list of running segments (using running_remove(j)). Similarly, the scheduler can non-deterministically choose to cease the execution of a segment if its minimum execution bound is reached, i.e., x[j]⩾running[j].min. In either case, the scheduler will stop being idle and move to a committed mode from where it immediately transitions back to Idle after recording that the segment execution has completed by updating the content of the ready, blocked and inc_precs data structures using the functions finish_segment() and move_all_idle(). Whenever this occurs, either of two things can happen; If the segment that just completed is not the last segment of the task, the task is enqueued back into the priority queue. If the segment was the last uncompleted segment of the task then the scheduler model fires a complete[id]! output for which task id synchronises with and begins waiting.

Critically, the scheduler will always make sure that ready is up-to-date before the highest priority task is told to run a task. Because run is the lowest priority channel, whenever job_ready() and processor_avail() are satisfied, the run transition does not become enabled until all enabled move! outputs have fired, ensuring that all potentially ready segments are indeed considered ready by the scheduler. In other words, the scheduler will always force segments that are currently at their upper execution bound to cease before selecting the next segment to execute.

E. Schedulability and Response-Time Analysis

We now discuss how our model can be used to check schedulability and derive response time bounds for every task.

To verify that a model instance is schedulable, we ask UPPAAL to check whether the system may reach a state where a task is in the `active` mode and the clock `tp` is strictly greater than the relative deadline of said task. This is done using the following query in UPPAAL: `E<> exists (i : id_t) Task(i).Active && tp > Task(i).deadline()`

If the property is satisfied, then the system is unschedulable. Taking the contra-positive, the system is schedulable if and only if the property is not satisfied. Note that finding a single counter-example is enough to prove that a system is unschedulable. This may however require full exploration of the full state space. UPPAAL may be configured to efficiently find counter-examples (when one exists) using depth-first search instead of breadth-first search. On the other hand, breadth-first search should be preferred when trying to prove that a system is schedulable since it then explores the full state space faster.

To derive response time bounds, we define an additional clock `response : clock`, declared locally in each task. The clock `response` is reset whenever a task transitions from the `offsetting` or `waiting` mode to enter the `active` mode. To obtain the worst-case response time for a task i, we ask UPPAAL to find the largest value attainable by clock `response` when task i is in the `active` mode. This is done using the following query in UPPAAL: `sup{Task(i).Active}: Task(i).response`

This is correct because tasks will urgently start to wait whenever a job finishes. To obtain best-case response time for task i, we ask UPPAAL to find the smallest value of `response` when task i is `waiting`: `inf{Task(i).Waiting}: Task(i).response`

Again, this is correct because task i will urgently start `waiting` whenever its current job is done. Note, clock `response` does not impact schedulability analysis because it does not occur in invariants or guards and therefore is not part of the model state.

IV. Experimental Results

Since, to the best of our knowledge, we present the first exact schedulability analysis based on TA for LPPA, our primary objective is to demonstrate that it can be used as a schedulability test for simple problem instances, but more interestingly we highlight its effectiveness as a benchmarking tool to evaluate the accuracy of other sufficient analysis techniques even for more complex problem instances. To convey this, we aim to answer the following questions: How well does our analysis scale with the number of tasks, segments, cores and increasing parallelism within a task? How efficient is our method in terms of runtime when used to ascertain that systems deemed unschedulable by other sufficient analysis techniques are indeed unschedulable? All experiments are run on an Intel Core Ultra 7 processor clocked at 4.80 Ghz with 64.0 GiB of RAM.

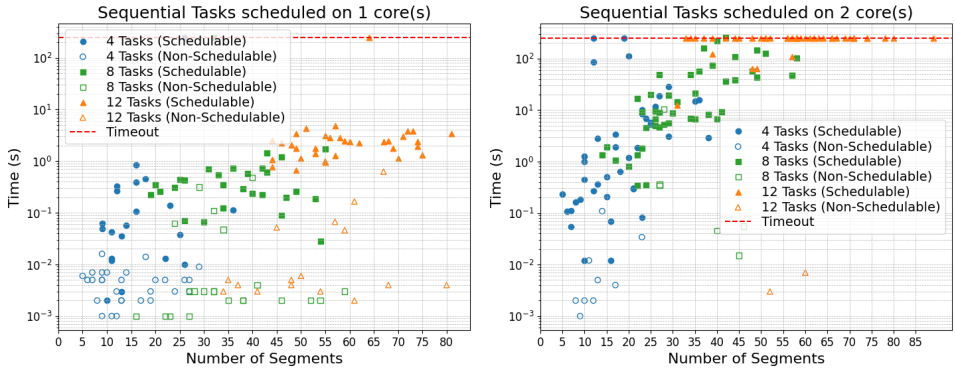
Scalability We tested the scalability of our analysis by analysing limited-preemptive sequential tasks scheduled on a single processor. Limited-preemptive tasks were generated by breaking non-preemptive tasks into a sequence of smaller segments with precedence constraints between them. The segments of each task were generated by iteratively breaking down each segment into two smaller segments with a probability of 0.8 until no segment is broken anymore or the task contains 20 segments. Whenever a segment is broken into two smaller ones, its best- and worst-case execution time is divided randomly between the resulting segments. In this experiment, we generated 50 random task sets with a total utilisation of 30%, aiming to increase the likelihood of generating schedulable task sets. We varied the number of tasks N within $\{4, 8, 12\}$, and set the number of processors M to either 1 or 2.

In Figs. 4a and 4b, we provide the runtime analysis for various configurations of tasks and cores, to show the impact that the number of segments, tasks and cores has on the runtime of the analysis. For these experiments, we assigned a very strict timeout of 250 seconds. Even within this short time frame, the single core analysis successfully processed almost all problem instances without timing out. As expected, increasing the number of tasks increases the analysis runtime. Increasing the number of segments in each task, however, has limited impact on the analysis runtime. Predictably, when analysing systems with 2 cores, we observed a significant increase in the analysis runtime. We also found that task sets deemed unschedulable had lower runtimes compared to schedulable ones, which aligns with the expected outcome since the analysis may stop as soon as a counter-example is found and it does not need to explore the whole state space. Proving schedulability on the other hand requires a full exploration of the state space. Differently from the single-core case, the analysis runtime does not only increase with the number of tasks but also increases with the number of segments in tasks, simply because blocking occurs more frequently.

Efficiency in generating counterexamples To determine how efficient our solution is in being used as a benchmarking tool for sufficient analyses, we considered the Schedule Abstraction Graph (SAG) analysis tool available at [19]. We chose SAG because it is, as far we know, the best sufficient schedulability test for periodic LPPA globally scheduled on multicore platforms [2], [3]. For each task set for which the SAG returned that the task set was potentially unschedulable, we ran our tool to try and find a counter-example to schedulability. If a counter-example could be found with our new analysis, it confirms that the SAG was correct in not deeming the task set schedulable.

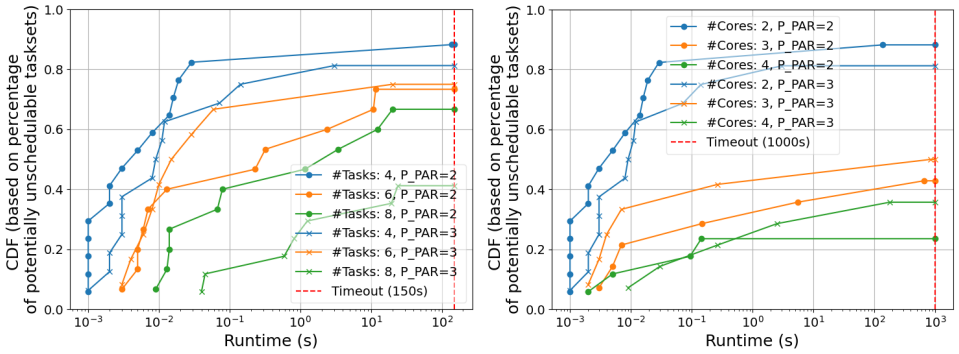
For this experiment we generated parallel tasks using the method described in [4] and [20], using a node forking probability of 0.3, a maximum of 2 nested forks, and the maximum number of parallel segments originating from a fork, denoted as P_{PAR}, set to either 2 or 3 depending on the experiment. We generated 20 random sets of limited-preemptive parallel tasks for each variation of the experiment, all with a total utilisation of 30%. The number of tasks N ranged within $\{4, 6, 8\}$, and the number of processors M ranged within $\{1, 2, 3, 4\}$.

These task sets were analysed by the SAG to identify those that it could not deem schedulable. Each of these task sets were then given to our UPPAAL model. Using depth-first search approach we then asked it to try and find counterexamples. If a counterexample was found within the time limits 150 or 1000 seconds (depending on the experiment), re-enforcing the SAG



(a) Runtime of the analysis for limited-preemptive sequential tasks on a single core.

(b) Runtime of the analysis for limited-preemptive sequential tasks on two cores



(c) CDF of task sets for which a counter-example was found while varying the number of tasks

(d) CDF of task sets for which a counter-example was found while varying the number of cores

Fig. 4: Experimental Results

result, we considered it efficient in terms of runtime.

The single-core version of these experiments demonstrated that, for all task sets that were deemed potentially unschedulable by the SAG, our analysis was able to find a counterexample within 150 seconds. This shows the efficiency of our analysis and the accuracy of the SAG on this small test run.

Figs. 4c and 4d presents a cumulative density function (CDF) that shows the cumulative percentage of task sets for which a counter-example was found with a runtime no more than the corresponding timeout shown on the x-axis. When varying the number of tasks, we observe that an increase in the number of tasks leads to longer runtimes when looking for counter-examples. Consequently, as the number of tasks increases, the percentage of task sets for which we are able to find a counter-example within a given timeout decreases. Similarly, when varying the number of cores, we observe that as the number of cores increases, the percentage of task sets for which a counter-example is found within a given timeout also decreases. Note, both CDFs are based on the total number of potentially unschedulable task sets as reported by SAG, it is entirely possible that some of these are in fact schedulable which would make it impossible to find a counter example.

Case study. [14] proposes an exact schedulability analysis for limited-preemptive periodic tasks using UPPAAL that we would like to test against given the similarities with our work. However, key differences and limitations to the [14] analysis is that it is only able to analyze single-core platforms and cannot handle DAG tasks. In their paper, they test their analysis on a case study extracted from the WATERS 2017 industrial challenge [21], consisting of seven limited-preemptive sequential tasks consisting of a total of 710 segments. We ran the exact model provided by the authors of [14] and compared it to ours. The average runtime and memory usage for verifying the use-case schedulability using the model presented in [14] is $96,7$ seconds requiring $2741,918$ KB of memory. Our model checks the schedulability with an average runtime of $46,1$ seconds with a memory usage of $770,997$ KB. This shows the effectiveness of the key observations we made in Sec. II to develop our model.

V. Conclusion

We have successfully developed an exact schedulability analysis framework for LPPAs and shown how it scales in terms of tasks, segments, cores and increasing parallelism. Under relatively small time frames it is capable of finding counter examples, even for parallel tasks running on multi core platforms. Our results show that our solution looks very promising as a benchmark for evaluating the accuracy of other sufficient analysis techniques for LPPAs. Our model is generic, allowing for future enhancements. For example, it can be extended to handle self-suspending tasks, event-driven delay-induced tasks, and tasks with sporadic release patterns. Furthermore, the model is adaptable to arbitrary deadlines beyond the current assumption of constrained deadlines.

REFERENCES

[1] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "A comprehensive survey of industry practice in real-time systems," *Real-Time Systems*, vol. 58, no. 3, pp. 358–398, 2022.

[2] M. Verucchi, I. S. Olmedo, and M. Bertogna, "A survey on real-time dag scheduling, revisiting the global-partitioned infinity war," *Real-Time Systems*, vol. 59, no. 3, pp. 479–530, 2023.

[3] M. Nasri, G. Nelissen, and B. B. Brandenburg, "Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019, pp. 21–1.

[4] D. Casini, A. Biondi, G. Nelissen, and G. C. Buttazzo, "Partitioned fixed-priority scheduling of parallel tasks without preemptions," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2018, pp. 421–433.

[5] M. A. Serrano, A. Melani, S. Kehr, M. Bertogna, and E. Quiñones, "An analysis of lazy and eager limited preemption approaches under dag-based global fixed priority scheduling," in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2017, pp. 193–202.

[6] J. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic dag tasks for global fp scheduling," in *Proceedings of the 25th international conference on real-time networks and systems*, 2017, pp. 28–37.

[7] ——, "Schedulability analysis of dag tasks with arbitrary deadlines under global fixed-priority scheduling," *Real-Time Systems*, vol. 55, pp. 387–432, 2019.

[8] Q. He, N. Guan, Z. Guo *et al.*, "Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2283–2295, 2019.

[9] F. Guan, J. Qiao, and Y. Han, "Dag-fluid: A real-time scheduling algorithm for dags," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 471–482, 2020.

[10] F. Aromolo, A. Biondi, G. Nelissen, and G. Buttazzo, "Event-driven delay-induced tasks: Model, analysis, and applications," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 53–65.

[11] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2012.

[12] R. J. Bril, J. J. Lukkien, and W. F. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems*, vol. 42, pp. 63–119, 2009.

[13] R. J. Bril, M. M. van den Heuvel, U. Keskin, and J. J. Lukkien, "Generalized fixed-priority scheduling with limited preemptions," in *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 2012, pp. 209–220.

[14] M. Foughali, P.-E. Hladik, and A. Zuepke, "Compositional verification of embedded real-time systems," *Journal of Systems Architecture*, vol. 142, p. 102928, 2023.

[15] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397594900108

[16] G. Behrmann, A. David, and K. G. Larsen, *A Tutorial on Uppaal*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236. [Online]. Available: https://doi.org/10.1007/978-3-540-30080-9_7

[17] B. Yalcinkaya, M. Nasri, and B. B. Brandenburg, "An exact schedulability test for non-preemptive self-suspending real-time tasks," in *DATE*. IEEE, 2019, pp. 1228–1233.

[18] "Uppaal models for lppa - date 2025," 2025. [Online]. Available: https://github.com/nidhi2396/UPPAAL_models_for_LPPA-DATE25

[19] "Schedule-Abstraction Graph framework GitHub organization." [Online]. Available: https://github.com/SAG-org

[20] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *2015 27th Euromicro Conference on Real-Time Systems*, 2015, pp. 211–221.

[21] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein, "Waters industrial challenge 2017," in *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2017.