Online Admission Test for Real-Time Tasks with Arrival Curves for Server Platforms

Nasim Samimi, Mitra Nasri, Twan Basten, Marc Geilen
Eindhoven University of Technology
Eindhoven, Netherlands

Abstract

Centralised servers can provide on-demand resources to edge devices for offloading workloads. Resource-constrained computing nodes send requests to execute a task on the server for higher quality or faster execution. These requests are best characterised by an arrival curve. A server may become overloaded if too many requests come at once. For safety-critical applications, therefore, an admission test is required to ensure that the admitted requests meet their timing requirements.

In this work, we present an online admission test to decide whether an incoming request can meet its timing requirements on a server without jeopardising the timing requirements of already admitted requests and itself, considering potential future requests to higher-priority tasks. The server executes tasks using a non-preemptive global fixed-priority scheduling policy. Our admission test extracts arrival times of future higher-priority jobs from their arrival curves and past observations and uses these, together with a reachability-based response-time analysis, to obtain a safe bound on the worst-case response time of the incoming request.

Our empirical evaluations show that our admission test is effective, admitting more than 95% of the incoming jobs to a 4- or 8-core server, when there are 20 tasks in the system. Comparing our admission test with an exact task-level schedulability test for tasks with arrival curves shows that the number of rejected admissible requests is small. The runtime of our test is practical for online analysis (typically below 6 microseconds).

CCS Concepts

• **Computer systems organization → Real-time systems**.

Keywords

Admission test, online response-time analysis, arrival curves, computation offloading, real-time systems

ACM Reference Format:
Nasim Samimi, Mitra Nasri, Twan Basten, Marc Geilen. 2024. Online Admission Test for Real-Time Tasks with Arrival Curves for Server Platforms. In *The 32nd International Conference on Real-Time Networks and Systems (RTNS 2024), November 7–8, 2024, Porto,Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3696355.3696359

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RTNS 2024, November 7–8, 2024, Porto, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1724-6/24/11
https://doi.org/10.1145/3696355.3696359

1 Introduction

Server-based platforms are frequently used to offer centralised resources for offloading data processing, application hosting, and service delivery tasks from resource-constrained computing nodes. The advent of cloud computing is an example of how server-based systems have expanded and evolved to meet on-demand access to distributed computing resources [34]. Examples of server-based applications include object recognition in self-driving cars [12, 15] and unmanned aerial vehicles [29], where local computing nodes offload their computation to a server, when possible.

Server-based architectures prevalent in real-time systems [8, 26] have several important characteristics. (i) Real-time tasks (applications) are offloaded from resource-constrained computing nodes (e.g., micro-controllers) to the server, where they can benefit from executing on a multicore platform and therefore, finish faster. (ii) The requests (also called jobs) that arrive to the server are often not strictly periodic (due to communication overheads between the computing node and the server). They are better characterised by *arrival curves* [48], which allow to model a wide class of arrival patterns including sporadic, bursty, and periodic arrivals with release jitter). (iii) The server may or may not be able to meet the timing requirements of each incoming request, indicating the need for a job-level admission test that dynamically decides at runtime to either accept or reject requests/jobs. Rejected jobs can then be executed on the local computing nodes, albeit with a potentially lower quality of service.

Our goal is to develop a server-based admission test that satisfies the requirements for real-time applications offloading workloads to a server (see Fig. 1). Such a test must be fast, to be performed at runtime, and ensure the timing constraints of the admitted workload. For sustained efficacy in the system, the admission test must anticipate potential future arrivals of higher-priority requests before admitting a lower-priority one.

Given that a server is typically a powerful computing machine with fast CPU cores, it is more efficient if it executes workloads non-preemptively. This not only reduces the actual execution times of applications by avoiding preemption and migration overheads but also simplifies the derivation of the *worst-case execution time* (WCET), both for *static-* and *measurement-based timing analysis* methods [42, 43]. Hence, we assume a non-preemptive execution model for the server.

We choose to use global scheduling since it offers flexibility and higher resource utilisation compared to partitioned scheduling. Also, it allows for better load balancing when execution times of jobs vary. This choice further enhances predictability of the setup for hard real-time applications.

Problem definition and challenges. Developing an *online admission test* for multicore servers that use global fixed-priority

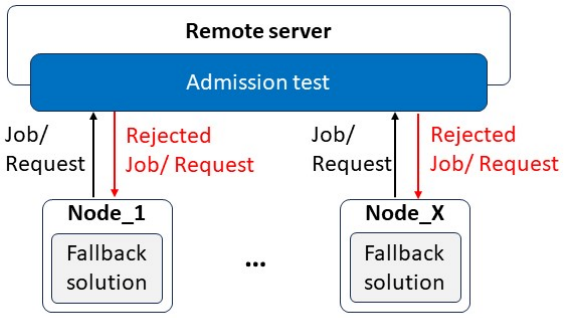

Figure 1: An admission test in a server-based platform

non-preemptive scheduling poses two key challenges: (i) *extracting interference patterns of possible future higher-priority requests exploiting knowledge of recent past requests*, and (ii) *designing an effective, runtime response-time analysis that uses these patterns and can be used to guarantee that every admitted request meets its deadline*. The former includes predicting future request arrivals given the observations of jobs that have recently arrived, and understanding their impact on schedulability of the current requests. The latter poses constraints on the analysis time.

Related work. Admission control and resource management have been studied in the context of cloud computing, for example, for the latency reduction of IoT applications in fog-cloud computing [46], optimal resource allocation [23, 24], or energy efficient offloading of soft and weakly hard (firm) real-time applications (while guaranteeing schedulability of compute-intensive tasks that are offloaded to the cloud) [11]. However, to our knowledge, there is no admission test to decide if incoming requests with hard or soft deadlines can be admitted on the cloud without missing their deadlines.

The literature offers various types of admission tests: (i) task-level admission tests determine whether a new task can be added to a system without causing other tasks to miss their deadlines [5, 9, 10, 26], and (ii) admission tests for aperiodic jobs with soft deadlines determine if an aperiodic job can be admitted without jeopardising deadlines of hard real-time periodic or sporadic tasks [30, 31]. However, these works neither propose a job-level online admission test (or a response-time analysis) for jobs with arrival curves nor they consider multicore platforms and global scheduling policies.

To guarantee that admitted jobs meet their deadlines, the admission test performs a response-time analysis (RTA) at runtime. The literature has extensively studied the response-time analysis problem for periodic and sporadic tasks or tasks with arrival curves [4, 6, 9, 16, 17, 19, 32, 33, 36, 48, 50]. These analyses often are designed to be used offline, and they assume that all jobs of a task must be able to meet their deadline when executing on the server, while in our problem, the server may discard jobs if it cannot guarantee their timing requirements. Moreover, when it comes to analysing global scheduling policies on multicore platforms, the existing RTA are often pessimistic (e.g., as shown by [7, 36]).

Nasri et al. [36, 37] introduce the *schedule-abstraction technique*, a highly accurate schedulability analysis for non-preemptive job sets (or periodic tasks) scheduled by a global job-level-fixed-priority (JLFP) scheduling policy upon homogeneous multicore platforms. This analysis does not support tasks with arbitrary arrival curves or jobs whose arrival time is now known *a priori*. Given the accuracy of their analysis, we decided to adapt their technique to match with our workload model and use it in our admission test.

As explained earlier, the admission test must take into account the arrival of future jobs, namely, infer them from their arrival curves. Outside the context of admission tests, Kuenzli et al. [25] proposed a method to generate event traces from arrival curves where the number of events in an interval is bounded by the arrival curves rather than deriving the arrival times of future jobs given a history of past arrivals.

Another potentially relevant literature on arrival curves is the work on conformance tests (a.k.a. admission control) where the goal is to reject job arrivals that do not conform to a given arrival curve [2, 18, 20–22, 27, 39, 40]. These admission control tests are usually based on the l-repetitive function and dynamic counters, where the former establishes a lower bound on inter-arrival times between jobs and limits the maximum number of admitted jobs in a specific interval while the latter tracks the number of jobs arriving during specific intervals of time. While these solutions could be used in extracting the constraints on the arrival times of the jobs, they are limited to arrival curves that are segmentally repetitive and sub-additive, which means the minimum distance between consecutive events is periodic and bounded. Even though these techniques cannot be immediately used in our problem, we draw inspiration from them to extract arrival times of future jobs considering the history of the previous job arrivals.

Contributions. To address the aforementioned challenges, we propose an online admission test. This test admits an incoming request (job) of a hard real-time task characterised by an arrival curve. It allows the execution of the job on the server only if its admission would not violate the timing requirements of already admitted jobs and the job itself. The test considers all higher-priority jobs that may arrive in the future, get admitted, and therefore, interfere with the job under analysis. We assume a server with a homogeneous multicore platform that schedules jobs using global non-preemptive fixed-priority scheduling. This paper makes the following contributions:

- We propose a method to extract the interference of potential future higher-priority jobs whose arrival times follow arrival curves, taking into account past arrivals.
- We introduce an online response-time analysis that adapts and extends the schedule-abstraction technique of Nasri et al. [36, 37] so that it can be used as an admission test using the current state of the system (i.e., queued jobs and core availabilities) for an individual job whose arrival time follows an arrival curve.

Together, these contributions improve the analysis and management of real-time systems in server-based platforms. Via empirical evaluations, we show that the runtime of our admission test is practical (typically between 2 to 6 microseconds). Moreover, it rejects only up to 10% of the jobs, even when the task set utilisation is near 100%. Our test performs better (i.e., is more successful) when there are more cores and tasks in the system. For example, when there

are more than 20 tasks in the system, it admits at least 96% of the incoming requests (jobs) on a 4-core platform.

2 System model

2.0.1 Architecture of the system. We assume a server-based architecture where computing nodes (with limited computational resources) are connected via reliable network connections to a server that is equipped with a multicore platform and can run a predefined set of applications upon request (see Fig. 1). The computing nodes send requests to those applications on the server through a reliable network. The server notifies the node about whether it has admitted the request or not. If the request is rejected, the node has to execute the application locally, potentially with degraded quality. The fallback scenarios on the computing nodes are not considered in this work.

On the server, an application is called a *task* and incoming requests to that application are called *jobs*. Jobs are assumed to be independent, single-threaded, and execute non-preemptively. A job does not occupy more than one core at a time but multiple jobs of a task may run concurrently on different cores. When a job is admitted, it enters a queue and waits until the underlying global fixed-priority scheduling policy dispatches the job on the first available core. We assume cores are homogeneous and tasks have a fixed priority.

2.0.2 Resource model. We assume a homogeneous server with m identical cores.

2.0.3 Workload model. We assume a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n tasks the jobs of which can be executed by the server upon request. A task τ_i is a tuple $(\alpha_i^-, \alpha_i^+, C_i^{min}, C_i^{max}, D_i, p_i)$, where α_i^- and α_i^+ are the lower and upper arrival curves of the requests that may be generated by the computing nodes of the system to that task, respectively. C_i^{min} and C_i^{max} are the best-case (BCET) and worst-case (WCET) execution times, D_i is the relative deadline, and p_i is the priority of the task, where a smaller p_i value indicates a higher priority. The WCET includes conservative margins for hard real-time guarantees.

2.0.4 Arrival model. Jobs of a task τ_i arrive to the server following *a priori* known arrival curves modelled by (α_i^-, α_i^+), where α^- and α^+ are functions in $\mathbb{R}^{\geq 0} \rightarrow \mathbb{N}^0$ that represent the minimum and the maximum number of jobs that may arrive in any interval of a given length, respectively.

2.0.5 Job model. A request (initiated by a node) to a task τ_i is called a job. For a job J we define $\sigma(J)$, $d(J)$, $C^{min}(J)$, $C^{max}(J)$, and $p(J)$ as the arrival time, absolute deadline, BCET, WCET, and the priority of J, respectively. The BCET, the WCET, and p are inherited from the corresponding task.

2.0.6 Problem description. In order to ensure that incoming jobs that are admitted meet their deadlines on the server, we have developed an online admission test. This test guarantees that a request will be admitted only if it can be completed before its deadline (taking into account other higher-priority jobs that have been admitted or may arrive in the future). This is achieved by calculating an upper bound on the worst-case response time (WCRT) of the job if it is admitted to the server upon its arrival.

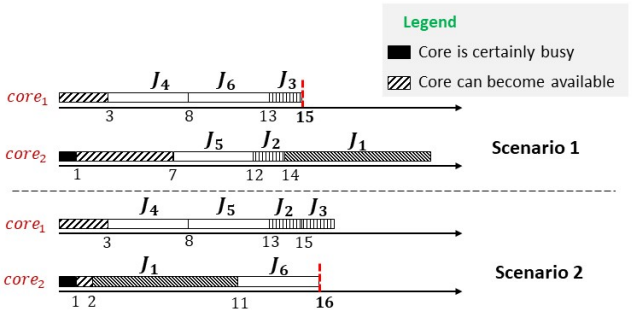

Figure 2: Fixed-priority scheduling anomalies

3 response-time analysis

In this section, we explain how to derive an upper bound on the WCRT of a newly arrived job J. Assume that when job J arrives in the server, there may be some jobs pending in the queue, some jobs running on the cores, and some higher-priority jobs arriving afterwards. For simplicity, we refer to the arrival time and absolute deadline of J as σ and d, respectively.

3.1 Anomalies in non-preemptive scheduling

In the following example, we show that due to the existence of anomalies in non-preemptive fixed-priority scheduling, one cannot identify the scenario where J has the WCRT by assuming the WCET for the high-priority jobs or assuming that cores will be available as late as possible.

Example 1. *Consider a system with the set $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ of tasks with $C_i^{min} = C_i^{max}, 1 \leq i \leq 4$, as 5, 2, 9, and 4, respectively. Tasks have arbitrary arrival curves as $\alpha_i^+, \alpha_i^-, 1 \leq i \leq 4$, and $p_1 < p_2 < p_3 < p_4$.*

Assume that the queue is empty at time 0 when job J arrives in the system. $core_1$ is running a job with BCET and WCET of zero and 3, respectively. Hence, it may become available at any point and it is certainly available at time 3.

$core_2$ is running a job with BCET and WCET of 1 and 7, respectively. It may become available at any point between time 1 to 7, and it is certainly available at time 7.

Since no core is available when J arrives, it enters the queue. In this example, other jobs with higher priority than J might enter the system while J is pending in the queue. Assume that within the time interval of 0 to 3, the number of job arrivals for τ_1, τ_2, τ_3 is as $\alpha_1^+(\Delta_1) = 3, \alpha_2^+(\Delta_1) = 2$, and $\alpha_3^+(\Delta_1) = 1$. Assume that J_1 arrives in the system at time 2, and the rest arrive at time 3.

If jobs execute for their WCET and the cores become available as late as possible (scenario 1), the response time of J will be 15 as shown in Fig 2. However, we show that a larger response time happens when $core_2$ becomes available at $t = 2$ instead of $t = 7$ (i.e., five units of time earlier than the previous case). When $core_2$ becomes available, the highest-priority job in the queue is J_1. Hence J_1 is scheduled on $core_2$. In this case, the response time of J is 16 as shown in Fig. 2 (scenario 2).

This example shows that considering the WCET and the maximum number of interfering jobs, may not necessarily lead to the

WCRT of the job under analysis due to scheduling anomalies. Therefore, a WCRT analysis should ensure that all execution scenarios have been considered, namely, scenarios with any possible execution time, arrival time, core availability time, and any number of job arrivals of future high-priority jobs. Moreover, this search should be done efficiently; otherwise, the method would not be useful as an online admission test. The next section introduces a recent accurate WCRT analysis technique for non-preemptive scheduling, called the schedule-abstraction technique, which is the basis of our work.

3.2 Schedule-abstraction graph technique

The *schedule-abstraction technique* (a.k.a. schedule-abstraction graph (SAG)) [13, 14, 35–38, 41, 44, 45, 47] is a reachability-based schedulability analysis that has been applied to a wide class of scheduling problems from single-core [35, 45] to multicore platforms supporting global job-level fixed priority (JLFP) scheduling policies such as fixed-priority scheduling and EDF [36, 37]. SAG explores the space of possible scheduling decisions that can be taken to schedule a given job set (in which jobs have bounded release jitter and execution time variation) by the given scheduling policy. It outputs a lower and an upper bound on the response time of each of the jobs in the job set. Despite taking into account all uncertainties in release and execution times, SAG is very efficient in analysing periodic tasks due to its efficient state-space reduction techniques such as state merging, symmetry reduction, abstract interpretation, etc[1]. Moreover, SAG is independent of the granularity of the timing parameters of the system as it abstracts schedules into job orderings.

Currently, SAG either analyses job sets or periodic tasks with known arrival times and bounded release jitter. On the other hand, the admission test addresses the problem of determining whether a job J can meet its deadline under any scenario. We need to find the WCRT of job J taking into account all other jobs that can impact its response time. To accomplish this, we introduce a set of jobs that can impact the WCRT of job J. We then provide this new job set to SAG to analyse and determine the WCRT of job J.

3.3 Key factors impacting the WCRT of J

The *WCRT* of J may be impacted by:

(i) jobs that have already been admitted but not yet dispatched on the cores (jobs in the queue), denoted by $\mathcal{J}^Q$,
(ii) higher-priority jobs that may arrive after the arrival of J while J is waiting in the queue denoted by $\mathcal{J}^\Psi$, and
(iii) expected times by which cores are *possibly* or *certainly* available, represented by a set of intervals denoted by A as defined below.

We keep track of the availability of the cores using a set of intervals. Each interval spans from the earliest time that a core may become available to the latest time after which the core is certainly available to process other jobs. Using the fact that cores are homogeneous, we create a set of availability intervals that combine the availability of multiple cores as $A = \{A_1, A_2, \ldots, A_m\}$, where $A_i = [A_i^{min}, A_i^{max}]$

[1]The SAG framework is expanding rapidly, the latest version of the framework, v3.0, released in September 2024, appears to be multiple orders of magnitude faster than the ones before [1].

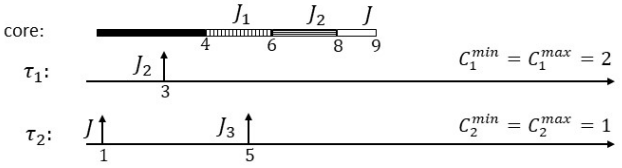

Figure 3: An example showing the status of a single-core server at the arrival of a job J, where $\mathcal{J}^Q = \{J_1\}$, $\mathcal{J}^\Psi = \{J_2\}$, and $A_1^{min} = A_1^{max} = 4$.

and A_i^{min} and A_i^{max} represent the earliest and latest time of availability of at least i cores with $i \in \{1, \ldots, m\}$. These intervals are obtained by sorting the earliest and latest availability times of individual cores. For example, A_1^{min} is the earliest time that at least one core is available. The earliest time that two cores become available is shown by A_2^{min}. Whenever a core becomes available, the scheduler dispatches the highest-priority job from the queue onto the available core. The core availability set is thus updated after a dispatch event.

Example 2. *Fig. 3 illustrates a system with $m = 1$ and $\Gamma = \{\tau_1, \tau_2\}$, where τ_1 has a higher priority than τ_2 (i.e., $p_1 < p_2$). J_1, J_2 from τ_1 and J, J_3 from τ_2 arrive. When J arrives, J_1 is in the queue ($\mathcal{J}^Q = \{J_1\}$), the core is busy, and $A_1^{min} = A_1^{max} = 4$. When the core becomes available, at time 4, J_1 is scheduled before J, because it has a higher priority and is already in the system. While J is waiting for the core to become available, J_2 and J_3 arrive at times 3 and 5, respectively. Since J_2 has a higher priority than J, it is dispatched before J, at time 6 after J_1 finishes. Finally, J becomes the highest-priority job waiting in the queue at time 6 after J_2 is dispatched. J will then be dispatched at time 8.*

While A and $\mathcal{J}^Q$ are known at the arrival time of J, the set of higher-priority jobs that may impact the response time of J still need to be derived.

3.4 Challenges of deriving a tight bound on the *WCRT* of a task when using an admission test

Designing an exact admission test for a task set that has a non-periodic arrival pattern is challenging because the jobs that have not yet arrived in the system (i.e., future jobs) will also be subject to the outcome of their admission test at the time of their arrival. In other words, predicting which of the future jobs will, in the end, be admitted to the system, and therefore have a chance to interfere with the job J (that is currently under analysis) requires clairvoyance, i.e., an oracle would be required to build an exact admission test. The example below illustrates the challenge.

Example 3. *Fig. 4 illustrates job arrivals of three tasks: $\{\tau_1, \tau_2, \tau_3\}$, where $p_1 < p_2 < p_3$ and $m = 1$. The queue is empty and the core may become available from $t = 6$ and is certainly available by $t = 8$. In this example, J is the job under analysis, which arrives at time 1. The first step of the analysis is to obtain the higher-priority job set (jobs that may arrive after job J and may interfere with it if admitted to the system). Let's assume that two other high-priority jobs, J_2 and*

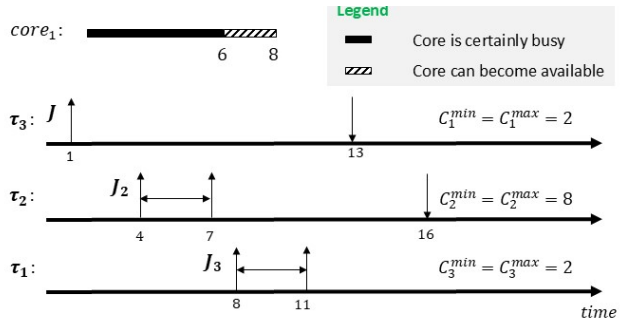

Figure 4: An example of variation in the impact of the future higher priority jobs due to uncertainty in arrival times of future jobs. J is the job under analysis. Here, J_3 may arrive at any time in the interval $[8, 11]$.

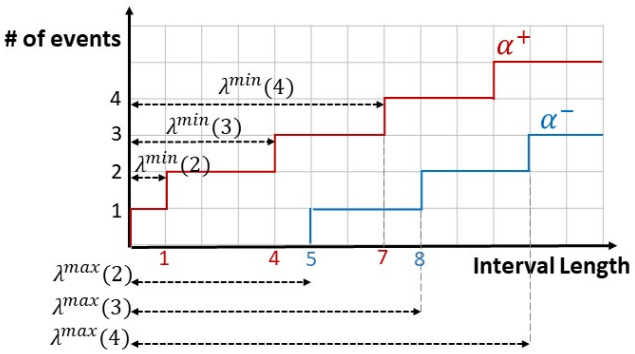

Figure 5: Arrival curve constraints

J_3, will soon arrive in the system. If J_2 is admitted to the system at $t = 4$ and the core becomes available earlier than $t = 8$, it is scheduled on the core and consequently, J will miss its deadline. If J_2 arrives at $t = 7$ and the core became available earlier, J is already scheduled on the core and meets the deadline.

In another scenario, if J_2 arrives at any time between 4 to 7, the core becomes available at $t = 8$ and J_3 arrives at $t = 8$, then at time 8, it will be J_3 that will be dispatched on the core. Since, in that scenario, the core becomes available only at $t = 10$, J_2 will miss its deadline. In this case, J_2 must be rejected by the admission test because there exists a scenario (of the arrival of higher-priority jobs) in which J_2 may miss its deadline. If the arrival of J_2 could be known in advance, then there would be scenarios in which J could be admitted to the system as it could be known to meet its deadline.

3.5 Generating the interfering job set

To address the challenges caused by the complexity of deriving $\mathcal{J}^\Psi$ for a given job J, we assume that $\mathcal{J}^\Psi$ is the set of *all the higher-priority jobs* that arrive in the interval $[\sigma, d]$ between arrival and deadline. In other words, we assume that all higher-priority jobs may be admitted by the admission test in the future. We only need to look for higher-priority jobs that could potentially arrive before the deadline d of job J, because the test would admit J only if it is certain that it will complete before d. Therefore, arrivals after d would only interfere with J if it has already missed its deadline, in which case the outcome of the test is to reject J.

Due to scheduling anomalies of non-preemptive scheduling, maximising the workload (number of jobs) in $\mathcal{J}^\Psi$ may not necessarily result in the worst-case start time for job J. Therefore, we include scenarios in the analysis where none, some, or all of the jobs in this set are rejected. These scenarios can be conservatively covered by considering that the best-case execution time of each job in $\mathcal{J}^\Psi$ is zero and assuming they are always admitted. This allows us to imitate the case where they are not admitted or do not arrive in the interval $[\sigma, d]$ with the scenario that the execution time equals 0.

The response time analysis starts with generating the set $\mathcal{J}^\Psi$ of jobs. We use arrival curves of tasks to determine the arrival intervals of jobs in $\mathcal{J}^\Psi$. First, we derive constraints on the intervals between the incoming jobs, then, employ them to derive the arrival intervals.

3.5.1 Deriving constraints from arrival curves. In this section, we derive constraints that bound the minimum and the maximum time interval of any j *consecutive jobs* of a task characterised by an arrival curve α. These intervals are inferred from the maximum and the minimum arrival curves represented by α^+ and α^-, respectively (see Fig. 5).

Example 4. *Fig. 5 depicts the arrival curve of a periodic task with a period of* 3 *and release jitter of* 1. *α^+ implies that the shortest interval of time during which two consecutive jobs of this task* may arrive *to the system is* 1. *For example, if the first job arrives at time 10, the earliest time that the second one can arrive is at time 11. This can be derived from the α^+ curve, where the shortest Δ for which $\alpha^+(\Delta) = 2$ is 1. We denote this shortest interval between any two jobs by $\lambda^{min}(2)$, i.e., $\lambda^{min}(2) = 1$. Similarly, the shortest interval of time during which* 3 *consecutive jobs of the task* may arrive *to the server is* 4, *i.e., $\lambda^{min}(3) = 4$.*

In the same way, we can drive constraints from α^- to find the longest interval of time between the arrival of the first and last job in a set of x consecutive jobs. In our example, the longest such interval for 2 consecutive jobs is 5 because, for any arbitrary job that arrives at time t, the next job will definitely arrive by time $t + 5$ according to the definition of the α^- curve. We denote this quantity by $\lambda^{max}(2) = 5$. Similarly, the longest interval between the arrival of 3 consecutive jobs is $\lambda^{max}(3) = 8$. For example, if an arbitrary job arrives at time t, the next two jobs will definitely arrive before time $t + 8$.

For simplicity, we call $\lambda(j) = [\lambda^{min}(j), \lambda^{max}(j)]$ (for $1 \leq j \leq \kappa$) the *arrival constraints* of j consecutive jobs of a task, where $\lambda^{min}(j)$ and $\lambda^{max}(j)$ represent the shortest and longest intervals between the arrival of the first and the last of the j consecutive jobs to the server, respectively. Given that arrival curves are infinite, it is not practical to derive and store an infinite number of arrival constraints. In Sec. 3.5.3, we derive a finite number (κ) of these constraints for each task.

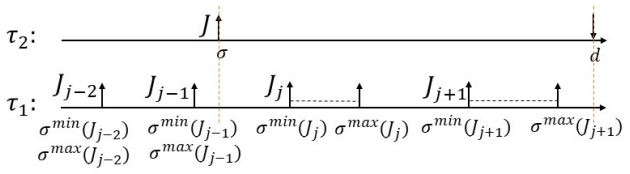

Figure 6: Arrival times of jobs of task τ, before and after arrival of J

Using the definition of arrival curves, we derive the arrival constraints between j consecutive jobs as follows:

$$\lambda^{min}(j) = \inf\{\delta \in \mathbb{R}^{\geq 0} \mid \alpha^{+}(\delta) \geq j\}, \quad (1)$$
$$\lambda^{max}(j) = \inf\{\delta \in \mathbb{R}^{\geq 0} \mid \alpha^{-}(\delta) \geq j-1\}.$$

Lemma 1. *For any l arbitrary consecutive jobs of a task with an arrival curve α, the interval between the arrival of the first and the last job in this set is not smaller than $\lambda^{min}(l)$ and not larger than $\lambda^{max}(l)$.*

Proof. Follows directly from the definition of arrival curves. □

3.5.2 Generating arrival intervals using constraints. In compliance with the arrival constraints, we use arrival times of the jobs of a task τ that have arrived before or have known arrival times, to derive the possible arrival times of future jobs of the same task. Alg. 1 explains how to derive these arrival times for task τ_i in an iterative process (shown in Fig. 6) starting from the first future job J_j. The algorithm starts by deriving the earliest and latest arrival times of a future job J_j. We represent these instants by $[\sigma_i^{min}(J_j), \sigma_i^{max}(J_j)]$.

We keep track of κ_i consecutive past jobs of a task (i.e., jobs that have already arrived) and denote them by J_{j-1} to $J_{j-\kappa_i}$, where J_{j-1} has the latest and $J_{j-\kappa_i}$ has the earliest arrival time. Using this information at runtime, we obtain the earliest and the latest arrival time of a future job J_j as:

$$\sigma_i^{min}(J_j) = \max\{\sigma_i^{min}(J_{j-k}) + \lambda_i^{min}(k) \mid 1 \leq k \leq \kappa_i\}, \quad (2)$$

$$\sigma_i^{max}(J_j) = \min\{\sigma_i^{max}(J_{j-k}) + \lambda_i^{max}(k) \mid 1 \leq k \leq \kappa_i\}, \quad (3)$$

where $\sigma_i^{min}(J_{j-k})$ is the earliest and $\sigma_i^{max}(J_{j-k})$ is the latest arrival time of job J_{j-k}. Note that the derived arrival interval must be the intersection of all possible arrival times according to the arrival constraints. Therefore, we use max and min functions to derive the earliest and latest arrival times, respectively. In Eq. (2) and (3), we assume that the number of already arrived jobs is at least as large as the total number of arrival constraints (κ_i). If it is smaller, κ_i should be replaced by the number of previously arrived jobs.

In Eq. (2) and (3), we used the arrival intervals of the jobs that arrive before J_j to derive the arrival interval of J_j, which has not arrived yet. Now, If J_j is the first job in the system (and hence no arrival times of previous jobs of the task are available), the first future job J_j can arrive at any time from the current time to the latest time that a job of task τ_i does not arrive in the system, which is $\lambda_i^{max}(2)$. Hence, Alg. 1 first checks the set of past arrivals (denoted by Λ_i). If the set of past arrivals is empty (Line 3), it uses the equation in Line 4 to derive the arrival interval of the first future job ($[\sigma_i^{min}(J_j), \sigma_i^{max}(J_j)]$) using $\lambda_i^{max}(2)$. Then, we initialise Λ_i^{Ψ}, in Line 5 using the result of Line 4.

Afterwards, from Line 8, Alg. 1 uses the set of past arrival times to derive the arrival interval of the next job. The derived interval is added to the set of known arrival intervals and iteratively used to derive the next arrival interval. Note that in Λ_i^{Ψ}, we use arrival intervals for the future jobs. We use the same format for Λ_i where we show the arrival times with intervals. However, since the arrival times of past jobs are known precisely, they are single points in time, not intervals. Therefore, for past jobs, we represent the arrival interval by using the exact arrival time as both the earliest and latest arrival time.

With the loop starting at Line 8, as long as the earliest arrival times of the future jobs are less than d, the loop continues. Inside the while loop, first, the set of previously arrived jobs (Λ_i) is updated with the new arrival information from Λ_i^{Ψ} in Line 9 to be used for the later future jobs. Then, Eq. (2) and (3) are used to derive the arrival interval of the future jobs (Line 10 and 11 for earliest ($\sigma_i^{min}(J_{j+k})$) and latest ($\sigma_i^{max}(J_{j+k})$) arrival times, respectively). The latest arrival times can be safely assumed to be less or equal to d (Line 12). We update Λ_i^{Ψ} in Line 13 using the results from the equations.

The procedure continues until the generated arrival intervals no longer overlap with the window of interest (i.e., the beginning of the intervals are larger than d). To achieve this, we update the condition of the loop in Line 14.

Theorem 1. *The arrival of any future job J_k of a task τ is not earlier than $\sigma^{min}(J_k)$ and not later than $\sigma^{max}(J_k)$ obtained from Eq. (2) and (3).*

Proof. We prove this statement by induction on the number of jobs between the arriving job J and the future job J_k.

Base of induction: Let J_k be the first future job of task τ after the arrival of J. We use contradiction to prove that the arrival of J_k is not earlier than $\sigma^{min}(J_k)$ and not later than $\sigma^{max}(J_k)$ obtained from Eq. (2) and (3), respectively.

Part 1: Assume that σ_k is the arrival time of job J_k, where $\sigma_k < \sigma^{min}(J_k)$. Then, according to Eq. (2)

$$\sigma_k < \max\{\sigma^{min}(J_{k-l}) + \lambda^{min}(l) \mid 1 \leq l \leq \kappa\}.$$

J_{k-l} is a past job such that $\sigma_k < \sigma^{min}(J_{k-l}) + \lambda^{min}(l)$. This means that the interval of $l+1$ consecutive jobs is shorter than $\lambda^{min}(l)$, which violates the minimum arrival constraints (since according to Lemma 1, $\lambda^{min}(l)$ is the shortest interval during which l consecutive jobs may arrive to the system). Hence, the arrival of J_k cannot be earlier than $\sigma^{min}(J_k)$.

Part 2: Assume that $\sigma_k > \sigma^{max}(J_k)$. Following a similar procedure as in Part 1, this assumption can be shown to contradict Lemma 1 and violate the lower arrival curve α^{-}. This shows that the arrival of J_k is not later than $\sigma^{max}(J_k)$.

By the induction hypothesis, we assume that Eq. (2) and (3) provide correct intervals for the arrivals of jobs up to the m^{th} job of τ after the arrival of job J. In the following, we prove that the arrival time of job $m+1$ is not earlier than $\sigma^{min}(J_{m+1})$ nor later than $\sigma^{max}(J_{m+1})$. We prove this statement by contradiction, considering t as the arrival time of J_{m+1}, where $t > \sigma^{max}(J_{m+1})$ (or $t < \sigma^{min}(J_{m+1})$). Similar to what is explained in the first step

of the induction, this assumption is in contradiction with Lemma 1 and definition of the arrival curves α^- (or α^+). □

Algorithm 1: Generating arrival intervals of a given task τ_i using constraints

Input :
$\lambda_i(l)$: The function for generating the constraints for l consecutive jobs of task τ_i,
J_j: first future job of task τ_i,
$\Lambda_i = \{(J_{j-l}, [\sigma_i(J_{j-l}), \sigma_i(J_{j-l})]), l \in [1, \kappa_i]\}$: the function that returns the arrival times of the previous jobs of τ_i,
σ, d: the arrival time and relative deadline of the target job.

Output: Λ_i^Ψ : the function that returns the arrival times of the future jobs of τ_i.

1 $k \leftarrow 0$
2 $\Lambda_i^\Psi \leftarrow \emptyset$
3 **if** $\Lambda_i == \emptyset$ **then**
4 $\quad [\sigma_i^{min}(J_j), \sigma_i^{max}(J_j)] \leftarrow [\sigma, \sigma + \lambda_i^{max}(2)]$
5 $\quad \Lambda_i^\Psi \leftarrow \{(J_j, [\sigma_i^{min}(J_j), \sigma_i^{max}(J_j)])\}$
6 $\quad k \leftarrow 1$
7 $a \leftarrow \sigma_i^{min}(J_j)$
8 **while** $a \leq d$ **do**
9 $\quad \Lambda_i \leftarrow \Lambda_i \cup \Lambda_i^\Psi$
10 $\quad$ Obtain $\sigma_i^{min}(J_{j+k})$ via Eq. (2)
11 $\quad$ Obtain $\sigma_i^{max}(J_{j+k})$ via Eq. (3)
12 $\quad \sigma_i^{max}(J_{j+k}) \leftarrow \min(d, \sigma_i^{max}(J_{j+k}))$
13 $\quad \Lambda_i^\Psi \leftarrow \Lambda_i^\Psi \cup \{(J_{j+k}, [\sigma_i^{min}(J_{j+k}), \sigma_i^{max}(J_{j+k})])\}$
14 $\quad a \leftarrow \sigma_i^{min}(J_{j+k})$
15 $\quad k \leftarrow k + 1$

In Alg. 1, we only consider the arrival times that happen before the deadline d and assume that $BCET = 0$ for all the jobs. We do this to ensure our analysis covers all scenarios, as expressed by the following theorem.

Theorem 2. *There is no potential scenario of the arrival time of a future higher-priority job of task τ_i that impacts the schedulability of J and is not included in Λ_i^Ψ.*

Proof. Assume that for the job (J), t is the arrival time of a higher-priority job J_k that will arrive in the future.

Part 1: Assuming that $t \leq d$, we prove that $t \in \Lambda_i^\Psi(J_k)$ obtained by Alg. 1. We prove this by contradiction, assuming that $t \notin \Lambda_i^\Psi(J_k)$, namely, $t < \sigma^{min}(J_k)$ or $t > \sigma^{max}(J_k)$, provided by Eq. (2) and (3). This violates Theorem 1.

Part 2: Assume that $t \geq d$. The arrival of jobs after d does not affect the outcome of the admission test for J. Because (i) if J finishes before its deadline d, jobs arriving after d won't interfere with it and (ii) if J cannot finish before its deadline, it is rejected by the admission test. To consider that in our analysis, we used $BCET = 0$ for all the higher-priority jobs which includes the states, where J_k arrives after d. Hence, the statement holds true. □

After generating Λ_i^Ψ for task τ_i, $\mathcal{J}^\Psi$ is obtained as:

$$\mathcal{J}^\Psi = \bigcup_{i \in [1,n]} \mathcal{J}_i^\Psi, \tag{4}$$

where $\mathcal{J}_i^\Psi$ is the set of all the higher-priority jobs of task τ_i that can arrive after the arrival of J and before its deadline. The arrival intervals of these jobs are provided by Λ_i^Ψ. In compliance with Theorem 2, there is no potential future higher-priority job that can impact the schedulability of J and is not included in $\mathcal{J}^\Psi$. The combination of Theorems 1 and 2 ensures the correctness of our derived higher-priority jobset.

3.5.3 Required number of arrival constraints for a task τ_i. As shown in Fig 6, our goal is to derive the arrival intervals of the future jobs within a specific interval (relative deadline of a newly arrived job J). The derived arrival constraints must be sufficient for the whole interval. That is, if the interval is equal to I and κ_i is the total number of arrival constraints of task τ_i, then $\lambda_i^{min}(\kappa_i)$ must be greater or equal to I. Moreover, I must be equal to the largest relative deadline of the tasks in a task set (D^{max}) to be valid for any newly arrived job of that task. Hence, κ_i is selected as the smallest number such that $D^{max} \leq \lambda_i^{min}(\kappa_i)$.

3.6 Adapting SAG for computing the WCRT of a job

Generating the set of future high-priority jobs (i.e., $\mathcal{J}^\Psi$) for the job under analysis, i.e., J, is the first step to calculating the response time of J. The next step is to derive an upper bound on the response time of J while being interfered with by the jobs in $\mathcal{J}^\Psi$ under a global non-preemptive fixed-priority scheduling policy.

SAG analyses the whole job set which is not required in our problem as we only need to know whether a specific job meets its deadline. Another issue is that SAG starts the analysis by considering that all the cores are available, whereas, in an admission test, when J arrives, we analyse the WCRT in a given system state, where some cores might be busy.

We modelled our proposed admission test using SAG. We do not change the analysis of the SAG. However SAG is modified to terminate analysis when the WCRT of J is ready, saving time compared to the original SAG. Also, core availabilities can be set at the beginning of the analysis. Once job J arrives, our admission test generates the higher-priority job set for J and then uses SAG to analyse the WCRT of J. Inputs of SAG are the set of higher-priority jobs, the set of queued jobs, current core-availability intervals, and J.

Alg. 2 presents our proposed admission test which uses both Alg. 1 and SAG. At the arrival of a job J to the server, arrival intervals of the higher-priority jobs are derived (Lines 1 to 3), then, $\mathcal{J}^\Psi$ is obtained and the super set $\mathcal{J}^\Psi \cup \mathcal{J}^Q \cup \{J\}$ are given to the modified SAG. If the WCRT of J is later than the deadline, the job is rejected, otherwise, J enters the queue.

For a newly arrived job J, given A as the availability intervals of the cores at the arrival time of J and $\mathcal{J}^Q \cup \mathcal{J}^\Psi \cup \{J\}$ as the set of queued jobs, set of higher-priority jobs, and the job J, under non-preemptive fixed-priority scheduling policy, if Alg. 2 confirms that J is schedulable, J meet its deadline.

Algorithm 2: Admission analysis for new job J

Input :

$[\alpha_1^-, \alpha_1^+], \ldots, [\alpha_n^-, \alpha_n^+]$: minimum and maximum arrival curves corresponding to Γ,

J: the job under analysis that has arrived at time σ and has a deadline d,

$\mathcal{J}^Q$: set of jobs in the queue

Output : Whether J can be admitted or not.

$\mathcal{J}^\Psi \leftarrow \emptyset$

for *each high-priority task* τ_i **do**

  $\mathcal{J}^\Psi \leftarrow \mathcal{J}^\Psi \cup \mathcal{J}_i^\Psi$ obtain the future jobs of τ_i within interval $[\sigma, d]$ from Alg. 1 and Eq (4)

schedulability of $J \leftarrow$ Call $SAG(J, \mathcal{J}^\Psi, J^Q)$

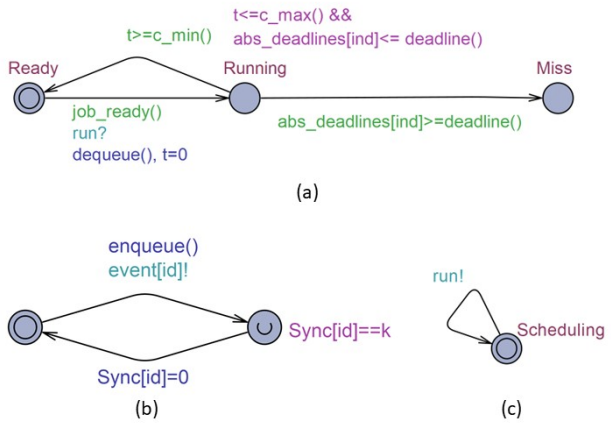

Figure 7: UPPAAL models: (a) TA for processor, (b) modified scheduler TA for emitting events, and (c) processor scheduler

4 experiments and evaluation

We conducted two experiments to answer the following questions: **(1)** How pessimistic is our admission test in scenarios where there is no need to reject any job? **(2)** Does the analysis have a practical runtime? **(3)** How generalisable is our admission test for different systems? To design our admission test, we used SAG v2.0, the version introduced in [37].

4.1 Experiment 1: comparing to task-level schedulability test

Since there is no online admission test for tasks with arrival curves that are scheduled by a global scheduling policy on multicore platforms, we cannot conduct an experiment to quantify the value added by our solution compared to existing methods. Yet, to answer the question on quantifying the potential pessimism of our admission test, we use timed automata [3] and the UPPAAL model checker to model and analyse an exact task-level schedulability analysis for tasks with arrival curves scheduled by global non-preemptive fixed-priority scheduling. Then, we compare our online admission test against this task-level schedulability test to see how many jobs of such a task set will be rejected by our admission test when the task set is schedulable, i.e., all the jobs can meet their deadlines).

Timed-automaton model. Our timed-automaton model, shown in Fig. 7, includes separate automata for tasks, scheduler, and processors. The tasks automaton is adapted from Lampka et al. [28]. This work uses three timed automata (TA), one to capture α^-, one for α^+, and one to synchronise between them. These TA can generate any set of job events according to the given arrival curves and put them in a queue. The scheduler TA generates scheduling emits scheduling events to the processor TA. The processor TA schedules a pending job from the queue if the processor is available (is in the Ready state and not running a job), and if it receives a scheduling event from the scheduler TA. Afterwards,The processor TA removes the highest priority job from the queue and goes to the Running state to execute the job.

The processor TA continues reading the jobs from the queue and executing them until the execution time is equal to the WCET and their deadline is met. If any job misses a deadline, the task set is reported unschedulable (the processor automaton reaches the Miss state); otherwise, it is schedulable (the processor TA goes to the Ready state).

Experiment setup and task set generation. We generated two types of task sets. **(1)** periodic task sets with utilisation $U \in \{0.4, 0.8, 1.2, 1.6, 2\}$, number of tasks in a set $n \in \{4, 6\}$, number of cores in the server $m \in \{1, 2\}$, and random jitter. Utilisation $U = 2$ for 2 cores means 100% of the capacity of the cores. **(2)** task sets with one non-recurrent task and periodic tasks with $U \in \{0.4, 0.6, 0.8, 1\}$, $n = 3$, $m = 1$, and random jitter. Task periods are chosen randomly in the range $[100, 300]$. Task priorities are chosen randomly in the range of the number of tasks in a set. Due to state-space explosion challenges in the UPPAAL analysis, we could not perform experiments on larger task sets and larger numbers of cores.

We generated 100 random task sets per system configuration. A long trace of jobs has been generated for each task set such that there are at least 1000 jobs per task in the job set.

Generating the arrival curves. We first generated a trace of 1000 random arrival times for non-recurrent arrival patterns. Then, we extracted the arrival curves by finding the minimum and the maximum number of jobs in certain intervals.

For periodic tasks, arrival curves are generated by a step function with the period as step width. Then, the steps are shifted to include the release jitter.

For burst arrivals, we generated the arrival curves using the method of [28]. This method first creates individual step functions with various step widths (starting from 100 to 150) and starting points (starting from 2 to 3) in such a way that the starting point of the function on the y-axis increases with the length of the step functions (i.e., it is sub-additive). The number of step functions is randomly chosen between 1 to 10. The minimum of these step functions generates the upper arrival curve for burst arrival. The reverse of this method is used for lower arrival curves.

To generate the WCET of tasks with arrival curves, we consider the upper arrival curve α^+. The utilisation of τ_i is determined by $U_i = (I_{\alpha_i^+} \times C_i^{max})/N_{\alpha_i^+}$, where N_{α^+} and I_{α^+} are the number of jobs and the interval over a repetitive segment in α^+, respectively (see

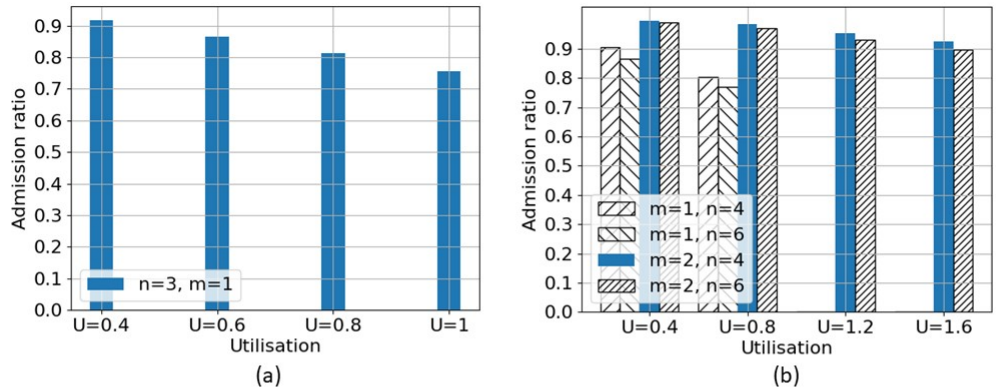

Figure 8: Experiment 1: admission ratio for schedulable task sets with (a) both periodic and non-recurrent and (b) only periodic arrival patterns.

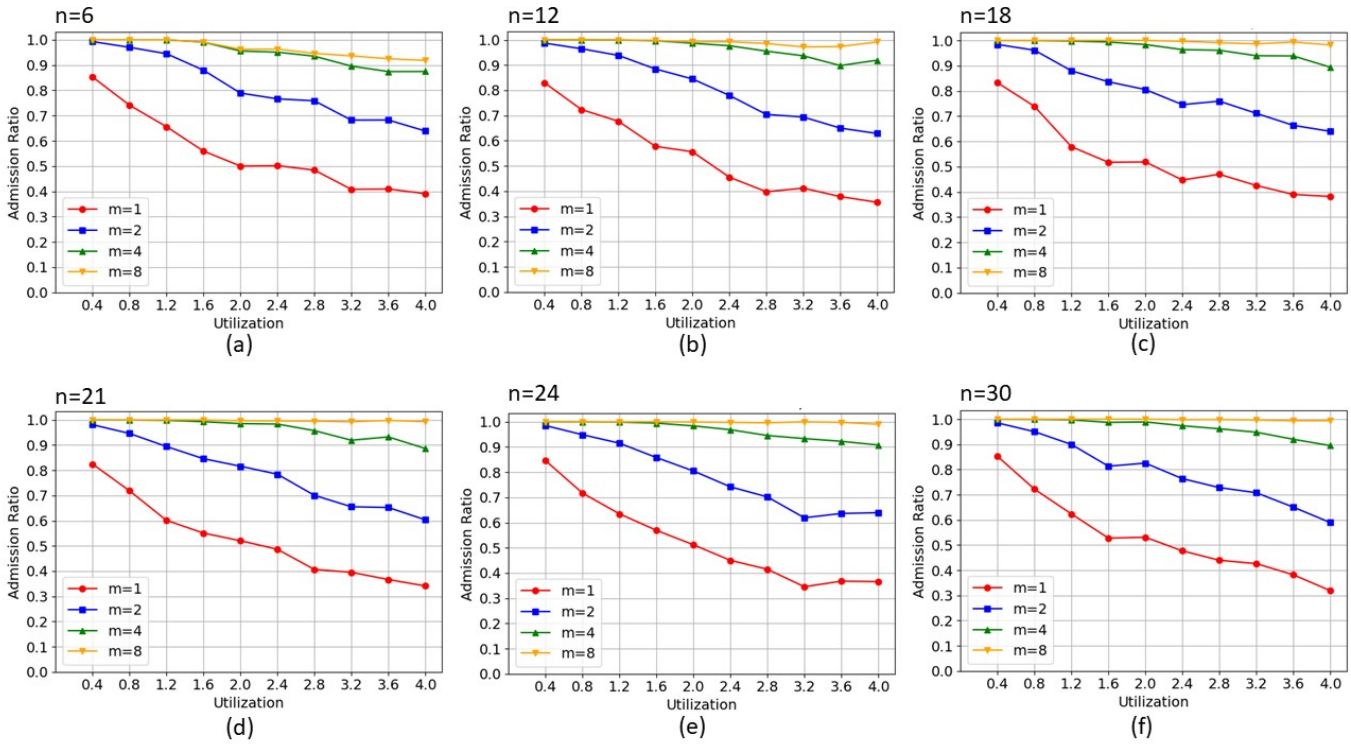

Figure 9: Experiment 2: admission ratio for different task sets as a function of utilisation for $m = 1$, $m = 2$, $m = 4$, and $m = 8$ when (a) $n = 6$, (b) $n = 12$, (c) $n = 18$, (d) $n = 21$, (e) $n = 24$, (f) $n = 30$.

[49], Section 7.2 for details of how arrival curves are represented). Then, we compute the WCET for a given utilisation and arrival curve.

Machine. We ran the experiments on an Intel machine i7-9750H (clocked at 2.6 GHz with 16GB of RAM).

Metrics. We report the *admission ratio*, i.e., the number of admitted jobs per total number of jobs in a single trace.

4.1.1 Pessimism of the admission test. As stated earlier, we measured the pessimism of our admission test by comparing it against task sets that are deemed *schedulable* according to the exact schedulability test in UPPAAL. Fig. 8 (a) shows the admission ratio of our admission test for schedulable task sets with both periodic and non-recurrent arrival patterns. Fig. 8 (b) shows the admission ratio for periodic task sets. Note that since there was no schedulable task set with utilisation over 100% on the uniprocessor platforms, we omitted the bars on the chart for utilisations above 100% when $m = 1$. As shown in these figures, our admission test has a small amount of pessimism, which increases as the task-set utilisation

increases. It is able to admit at least 75% of the jobs when the task set has 100% utilisation (see Fig. 8 (a) and (b)).

The pessimism originates from the over-approximations in generating the arrival intervals for the future higher-priority job set. For example, in the TA model of a periodic task, when a job is generated, no other job is generated until it reaches the period. In the admission test, we calculate only the arrival intervals of the jobs that can be overlapping (see Fig. 5). The SAG does not take into account whether tasks are periodic; instead, it searches for scheduling scenarios based on the arrival intervals. This approach allows for some arrival patterns that may never actually occur, such as the simultaneous arrival of two jobs from a periodic task with jitter. We made this choice to make the analysis faster. Moreover, we set a timeout of 200*s* for the admission test in SAG. If a job's admission test reaches the timeout, we report the job as unschedulable.

4.2 Experiment 2: runtime and generalisability

In this experiment, we focus on the benefits of our admission test. We consider task sets that may or may not be schedulable and show that only a small portion of the jobs generated from these task sets may be rejected by our admission test while the server can complete the rest of the jobs and meet their deadlines. We report both the admission ratio and runtime of our solution. Due to the large size of the task sets for this experiment, using the UPPAAL model is infeasible.

Experiment setup and task set generation. We generated random task sets with $U \in \{0.4, 0.8, 1.2, 16, 2, 2.4, 2.8, 3.2, 3.6, 4\}$, $n \in \{6, 9, 12, 15, 18, 21, 24, 27, 30\}$, $m \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, where U is the utilization, n is the number of tasks, and m is the number of cores. The task sets are a mix of periodic, burst and non-recurrent arrival patterns. Task periods are chosen randomly from the range $[100, 300]$; deadlines are assigned randomly between 1.5 to 2.5 times the WCET of the corresponding task. Task priorities are assigned randomly. Arrival curves were produced as explained in Sec. 4.1.

We generated 30 task sets per system configuration. Per task set, we generated a trace of jobs until at least 1000 jobs per task are included in the trace, similar to Sec. 4.1.

Machines. We ran the experiments on three Intel machines, an x5560 (clocked at 2.8GHz with 24GB of RAM) and two Core i9-9900K (at 3.6GHz, 64GB of RAM).

Metrics. We again report the *admission ratio*, i.e., the number of admitted jobs per total number of jobs in a trace.

4.2.1 Runtime of the admission test. We measured the runtime of the admission test for each individual job from job sets. We observed that the 98^{th} percentile of the runtime is between 2 to $6\mu s$, which validates the practical runtime of the test.

For jobs with a long relative deadline or low priority, the number of future higher-priority jobs may be large. Additionally, long deadlines result in larger, more conservative arrival intervals. This makes the analysis more time consuming. Consequently, the runtime of the admission test can sometimes become excessively long, which is unacceptable for an online admission policy. We therefore impose the 200*s* timeout for the admission test. If the analysis exceeds this limit, the corresponding job is rejected.

4.2.2 Impact of the utilisation. Fig. 9 depicts the average admission ratio for over 24, 000 job sets for different task sets. Each figure from (a) to (f) shows the admission ratio for a specific number of tasks in the task set. Each figure has 4 curves for $m = 1$ core, 2 cores, 4 cores, and 8 cores.

As shown in Fig. 9, the admission ratio decreases as the utilisation of the task set increases. However, both the rate of this decrease and the minimum ratio achieved depend on the number of cores (m). For example, in systems with 8 cores, nearly all jobs were admitted for any utilisation or any number of tasks that we considered in our experiment.

As explained in Sec. 4.2, we varied the utilisation of the task sets from 0.4 to 4. This means that none of the task sets with $U > 1$ were schedulable on 1 core. Yet, we wanted to evaluate how many jobs of such unschedulable task sets can still be safely admitted on the server. As can be seen, even when the (worst-case) workload is 4 times larger than the capacity of the system (i.e., 1 core), about 40% of the jobs can be safely admitted. This is because the jobs do not necessarily have their WCET at runtime. Hence, when a job finishes earlier than expected, the admission test uses the runtime information and admits more jobs. We also observe that the admission ratio increases with more cores. For example, when the workload is twice as large as the capacity of the system (i.e., $U = 2$ for $m = 1$ or $U = 4$ for $m = 2$), the admission ratio for a two-core platform is about 60% while for a single-core platform it is about 50%. The reason is that single-core platforms have more chance to be affected by blocking the low-priority tasks and hence the admission test becomes more conservative in rejecting low-priority jobs.

4.2.3 Impact of the number of cores. As shown in Fig. 9 (a) to (f), regardless of the number of tasks in the system or task set utilisation, the admission ratio increases when the system has more cores. This is due to the reduced impact of blocking as there are more cores upon which a job with a short deadline can be executed.

This experiment shows that for the setup we considered here, having 4 cores was enough to reach an admission ratio of 90% or above (regardless of the number of tasks or utilisation).

In Fig. 9 (d), for $n = 21$ the average admission ratios for $m = 1$, $m = 2$, $m = 4$, and $m = 8$ are 0.51, 0.78, 0.96, and 0.99, respectively. In Fig. 9 (e), for $n = 24$, the average admission ratios for $m = 1$, $m = 2$, $m = 4$, and $m = 8$ (across all utilisations) are 0.52, 0.78, 0.96, and 0.99, respectively. In Fig. 9 (f), for $n = 30$, the average admission ratios for $m = 1$, $m = 2$, $m = 4$, and $m = 8$ (across all utilisations) are 0.52, 0.78, 0.96, and 0.99, respectively. The admission ratio for large task sets ($n \geq 20$) increases following a similar pattern.

4.2.4 Impact of the number of tasks. As shown in Fig. 9 (a) to (f), increasing the number of tasks in a task set may improve the admission ratio. This phenomenon occurs because when there are fewer tasks in a set, each task's utilisation is relatively higher (and hence it can cause larger blocking) compared to when there are more tasks with the same total utilisation. When per-task-utilisation is large, there are larger slacks between the job's arrival and its deadline. However, this also means that there is more time for future higher-priority jobs to potentially interfere with the corresponding job. This can be observed in Fig. 9 (a) to (f), when we increase the

number of tasks in a task set and the admission ratio increases (for any number of cores).

5 Discussion and conclusion

In this work, we proposed an admission test for servers that execute real-time tasks with arrival curves under global non-preemptive fixed-priority scheduling. The test guarantees the deadline requirements of admitted jobs by performing an online WCRT analysis in which we derive the set of potential future jobs that could interfere with the current job. A job is admitted only if under any future arrival pattern of higher-priority jobs, the system respects its deadline.

We conducted empirical experiments to evaluate the runtime of our admission test and its generalisability across various systems with different task set utilizations, numbers of tasks, and processing cores. The results show that our admission test has a very small runtime (typically between 2 to 6 μs for systems with up to 9 cores and up to 30 tasks), and an average admission ratio for large task sets ($n \geq 20$) on 4 and 8 cores of at least 96% and 99%, respectively. In future work, we plan to extend our work to support dynamic resource scaling policies provided by cloud platforms, where the policy adjusts the number of cores at runtime. Additionally, we plan to consider network reliability and its overhead, scheduling with hard constant-bandwidth servers, and statistical distribution of execution time of the tasks. Moreover, we will provide guarantees for the acceptance ratio by considering weakly-hard constraints for tasks.

Acknowledgments

This work was supported by the EU ECSEL project TRANSACT (grant no. 101007260).

References

[1] [n. d.]. Schedule Abstraction Framework. Available at https://github.com/SAG-org.
[2] Syed Md Jakaria Abdullah, Kai Lampka, and Wang Yi. 2016. Improving performance by monitoring while maintaining worst-case guarantees. (2016), 257–260.
[3] Rajeev Alur and David Dill. 1992. The theory of timed automata. (1992), 45–73.
[4] Marko Bertogna and Michele Cirinei. 2007. Response-time analysis for globally scheduled symmetric multiprocessor platforms. *RTSS* (2007), 149–158. https://doi.org/10.1109/RTSS.2007.31
[5] Enrico Bini and Giorgio C. Buttazzo. 2004. Schedulability analysis of periodic fixed priority systems. *IEEE TC* 53 (2004), 1462–1473. Issue 11. https://doi.org/10.1109/TC.2004.103
[6] Reinder J. Bril, Johan J. Lukkien, and Wim F.J. Verhaegh. 2009. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *RTSJ* 42 (2009), 63–119. Issue 1-3. https://doi.org/10.1007/S11241-009-9071-Z/METRICS
[7] Artem Burmyakov, Enrico Bini, and Chang Gun Lee. 2022. Towards a Tractable Exact Test for Global Multiprocessor Fixed Priority Scheduling. *IEEE TC* 71 (2022), 2955–2967. Issue 11. https://doi.org/10.1109/TC.2022.3142540
[8] Tommaso Cucinotta, Antonio Mancina, Gaetano F. Anastasi, Giuseppe Lipari, Leonardo Mangeruca, Roberto Checcozzo, and Fulvio Rusinà. 2009. A real-time service-oriented architecture for industrial automation. *IEEE TII* 5 (2009), 267–277. Issue 3. https://doi.org/10.1109/TII.2009.2027013
[9] R. I. Davis and A. Burns. 2008. Response time upper bounds for fixed priority real-time systems. *RTSS* (2008), 407–418. https://doi.org/10.1109/RTSS.2008.18
[10] Robert I. Davis, Attila Zabos, and Alan Burns. 2008. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE TC* 57 (2008), 1261–1276. Issue 9. https://doi.org/10.1109/TC.2008.66
[11] Suzanne Elashri and Akramul Azim. 2020. Energy-efficient offloading of real-time tasks using cloud computing. *J. Clust. Comput.* 23 (2020), 3273–3288. Issue 4. https://doi.org/10.1007/S10586-020-03086-2
[12] Mario Gerla, Eun Kyu Lee, Giovanni Pau, and Uichin Lee. 2014. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. *WF-IoT* (2014), 241–246. https://doi.org/10.1109/WF-IOT.2014.6803166
[13] Pourya Gohari, Jeroen Voeten, and Mitra Nasri. 2023. Response-time Analysis of Fault-Tolerant Hard Real-Time Systems Under Global Scheduling. In *RTCSA*. IEEE, 263–264. https://doi.org/10.1109/RTCSA58653.2023.00039
[14] Pourya Gohari, Jeroen Voeten, and Mitra Nasri. 2024. Reachability-Based Response-Time Analysis of Preemptive Tasks Under Global Scheduling. In *ECRTS*. https://doi.org/10.4230/LIPIcs.ECRTS.2024.3
[15] Ken Goldberg and Ben Kehoe. 2013. *Cloud Robotics and Automation: A Survey of Related Work*. Technical Report. UCB/EECS. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-5.html
[16] Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. 2011. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *J. Syst. Archit.* 57 (2011), 536–546. Issue 5. https://doi.org/10.1016/J.SYSARC.2010.08.003
[17] Mario Günzel, Niklas Ueter, and Jian Jia Chen. 2021. Suspension-Aware Fixed-Priority Schedulability Test with Arbitrary Deadlines And Arrival Curves. *RTSS* (2021), 418–430. https://doi.org/10.1109/RTSS52674.2021.00045
[18] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. 2005. System level performance analysis - The SymTA/S approach. *IEE Proc. Comput. Digit. Tech.* 152 (2005), 148–166. Issue 2. https://doi.org/10.1049/IP-CDT:20045088
[19] Biao Hu, Kai Huang, Gang Chen, Long Cheng, Dongkun Han, and Alois Knoll. 2017. Schedulability Analysis Towards Arbitrarily Activated Tasks in Mixed-Criticality Systems. *J. Circuits Syst. Comput.* 26 (2017), 1750159. Issue 10. https://doi.org/10.1142/S0218126617501596
[20] Biao Hu, Kai Huang, Gang Chen, Long Cheng, and Alois Knoll. 2016. Adaptive Workload Management in Mixed-Criticality Systems. *ACM TECS* 16 (2016), 14. Issue 1. https://doi.org/10.1145/2950058
[21] Biao Hu, Kai Huang, Gang Chen, Long Cheng, and Alois Knoll. 2016. Evaluation and Improvements of Runtime Monitoring Methods for Real-Time Event Streams. *ACM TECS* 15 (2016). Issue 3. https://doi.org/10.1145/2890503
[22] Kai Huang, Luca Santinelli, Jian Jia Chen, Lothar Thiele, and Giorgio C. Buttazzo. 2011. Applying real-time interface and calculus for dynamic power management in hard real-time systems. *RTSJ* 47 (2011), 163–193. Issue 2. https://doi.org/10.1007/S11241-011-9115-Z/METRICS
[23] Kleopatra Konstanteli, Tommaso Cucinotta, Konstantinos Psychas, and Theodora Varvarigou. 2012. Admission control for elastic cloud services. *CLOUD* (2012), 41–48. https://doi.org/10.1109/CLOUD.2012.63
[24] Kleopatra Konstanteli, Tommaso Cucinotta, Konstantinos Psychas, and Theodora A. Varvarigou. 2014. Elastic admission control for federated cloud services. *IEEE TCC* 2 (2014), 348–361. Issue 3. https://doi.org/10.1109/TCC.2014.2325034
[25] Simon Kuenzli and Lothar Thiele. 2006. Generating Event Traces Based on Arrival Curves. *MMECCS Conf.* (2006), 1–18.
[26] Tei Wei Kuo, Li Pin Chang, Yu Hua Liu, and Kwei Jay Lin. 2003. Efficient online schedulability tests for real-time systems. *IEEE TSE* 29 (2003), 734–751. Issue 8. https://doi.org/10.1109/TSE.2003.1223647
[27] Kai Lampka, Kai Huang, and Jian Jia Chen. 2011. Dynamic counters and the efficient and effective online power management of embedded real-time systems. *ESWEEK - CODES+ISSS* (2011), 267–276. https://doi.org/10.1145/2039370.2039412
[28] Kai Lampka, Simon Perathoner, and Lothar Thiele. 2009. Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems. *EMSOFT* (2009), 107–116. https://doi.org/10.1145/1629335.1629351
[29] Jangwon Lee, Jingya Wang, David Crandall, Selma Sabanovic, and Geoffrey Fox. 2017. Real-time, cloud-based object detection for unmanned aerial vehicles. *IRC* (2017), 36–43. https://doi.org/10.1109/IRC.2017.77
[30] Chang Leng, Ying Qiao, Xiaobo Sharon Hu, and Hongan Wang. 2015. Utilization-based admission control for aperiodic tasks under EDF scheduling. *RTSJ* 51 (2015), 36–76. Issue 1. https://doi.org/10.1007/S11241-014-9216-6/TABLES/7
[31] Chang Leng, Ying Qiao, Xiaobo Sharon Hu, and Hongan Wang. 2020. Co-scheduling aperiodic real-time tasks with end-to-end firm and soft deadlines in two-stage systems. *RTSJ* 56 (2020), 391–451. Issue 4. https://doi.org/10.1007/S11241-020-09352-1/FIGURES/18
[32] Haoran Li, Chenyang Lu, and Chistopher Gill. 2019. Predicting latency distributions of aperiodic time-critical services. *RTSS* (2019), 30–42. https://doi.org/10.1109/RTSS46320.2019.00014
[33] Dorin Maxim and Liliana Cucu-Grosjean. 2013. Response time analysis for fixed-priority tasks with multiple probabilistic parameters. *RTSS* (2013), 224–235. https://doi.org/10.1109/RTSS.2013.30
[34] Peter Mell and Timothy Grance. 2011. The NIST Definition of Cloud Computing. (2011). https://doi.org/10.6028/NIST.SP.800-145
[35] Mitra Nasri and Bjorn B. Brandenburg. 2017. An Exact and Sustainable Analysis of Non-preemptive Scheduling. In *RTSS*. 12–23. https://doi.org/10.1109/RTSS.2017.00009
[36] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. 2018. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. *LIPIcs* 106 (2018), 9:1–9:23. https://doi.org/10.4230/LIPICS.ECRTS.2018.9

[37] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. 2019. Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling. In *ECRTS*. 21–1. https://doi.org/10.4230/LIPIcs.ECRTS.2019.21

[38] Geoffrey Nelissen, Joan Marcè i Igual, and Mitra Nasri. 2022. Response-time analysis for non-preemptive periodic moldable gang tasks. In *ECRTS*. 12:1–12:22. https://doi.org/10.4230/LIPIcs.ECRTS.2022.12

[39] Moritz Neukirchner, Philip Axer, Tobias Michaels, and Rolf Ernst. 2013. Monitoring of workload arrival functions for mixed-criticality systems. *RTSS* (2013), 88–96. https://doi.org/10.1109/RTSS.2013.17

[40] Moritz Neukirchner, Tobias Michaels, Philip Axer, Sophie Quinton, and Rolf Ernst. 2012. Monitoring arbitrary activation patterns in real-time systems. *RTSS* (2012), 293–302. https://doi.org/10.1109/RTSS.2012.80

[41] Suhail Nogd, Geoffrey Nelissen, Mitra Nasri, and Björn B Brandenburg. 2020. Response-time analysis for non-preemptive global scheduling with FIFO spin locks. In *RTSS*. IEEE, 115–127. https://doi.org/10.1109/RTSS49844.2020.00021

[42] Konstantinos Psychas and Javad Ghaderi. 2017. On Non-Preemptive VM Scheduling in the Cloud. *POMACS* 1 (2017), 1–29. Issue 2. https://doi.org/10.1145/3154493

[43] Harini Ramaprasad and Frank Mueller. 2008. Bounding worst-case response time for tasks with non-preemptive regions. *RTAS* (2008), 58–67. https://doi.org/10.1109/RTAS.2008.18

[44] Sayra Ranjha, Pourya Gohari, Geoffrey Nelissen, and Mitra Nasri. 2023. Partial-order reduction in reachability-based response-time analyses of limited-preemptive DAG tasks. *Real-Time Systems* 59, 2 (2023), 201–255. https://doi.org/10.1007/s11241-023-09398-x

[45] Sayra Ranjha, Geoffrey Nelissen, and Mitra Nasri. 2022. Partial-Order Reduction for Schedule-Abstraction-based Response-Time Analyses of Non-Preemptive Tasks. In *RTAS*. IEEE, 121–132. https://doi.org/10.1109/RTAS54340.2022.00018

[46] Eht E. Sham and Deo Prakash Vidyarthi. 2022. Admission control and resource provisioning in fog-integrated cloud using modified fuzzy inference system. *Journal of Supercomputing* 78 (2022), 15463–15503. Issue 13. https://doi.org/10.1007/S11227-022-04483-7/FIGURES/21

[47] Srinidhi Srinivasan, Geoffrey Nelissen, Reinder J. Bril, and Nirvana Meratnia. 2024. Analysis of TSN Time-Aware Shapers using Schedule Abstraction Graphs. In *ECRTS*. 16:1 – 16:24. https://doi.org/10.4230/LIPIcs.ECRTS.2024.16

[48] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. 2000. Real-time calculus for scheduling hard real-time systems. 4 (2000), 101–104.

[49] Ernesto Wandeler. 2006. *Modular performance analysis and interface-based design for embedded real-time systems.* https://tik-old.ee.ethz.ch/file/2b9491dbe13085961038e4a53f7792e6/diss_wandeler.pdf

[50] Beyazit Yalcinkaya, Mitra Nasri, and Björn B. Brandenburg. 2019. An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks. *DATE* (2019), 1228–1233. https://doi.org/10.23919/DATE.2019.8715111