**Noname manuscript No.**
(will be inserted by the editor)

# Periodicity of Real-Time Schedules for Dependent Periodic Tasks on Identical Multiprocessor Platforms

**Joël Goossens · Emmanuel Grolleau · Liliana Cucu-Grosjean**

the date of receipt and acceptance should be inserted later

**Abstract** This paper gives and proves correct a simulation interval for any schedule generated by a deterministic and memoryless scheduler (i.e., one where the scheduling decision is the same and unique for any two identical system states) for identical multiprocessor platforms. We first consider independent periodic tasks, then generalize the simulation interval to tasks sharing critical resources, and subject to precedence constraints or self-suspension. The simulation interval is based only on the periods, release times and deadlines, and is independent from any other parameters. It is proved large enough to cover any feasible schedule produced by any deterministic and memoryless scheduler on multiprocessor platforms, including non conservative schedulers. To the best of our knowledge, this simulation interval covers the largest class of tasks systems and scheduling algorithms on identical multiprocessor platforms ever studied. This simulation interval is used to derive a simulation algorithm using a linear space complexity. Finally, a generic exact schedulability test based on simulation is presented. This test can be applied only when sustainability is consistent with online variability of the tasks' parameters.

**Keywords** real-time scheduling, simulation intervals, multiprocessor

## 1 Introduction

### 1.1 Feasibility and simulation intervals

Real-time systems are widely used nowadays; their correctness is determined by functional and temporal requirements. Real-time scheduling theory focuses on the temporal validation of such systems. The temporal validation of a real-time system relies on a set of worst-case behaviors depending on the task model: each task is characterized by some temporal properties and constraints

that have to be met by the scheduling algorithm. Most task models are based on the model initially defined in [25], where a task is characterized by its worst-case execution time (WCET), and its release period. Every task is generating a potentially infinite sequence of jobs, each job is using up to its WCET amount of processor time. In the beginning of the paper, we consider schedules where every job is using exactly its WCET. This hypothesis is relaxed in Section 7. The temporal constraints of a task are represented by a relative deadline, and every job has to be completely executed between its release and its deadline.

One of the key problems in real-time system design is the *schedulability* problem [7, 13]: a task system is schedulable by a scheduling algorithm if, in any scenario, all the temporal constraints are met. A scenario represents a specific instance of the real-time system (actual release times, actual execution requirements, etc.) Regarding the schedulability of the system it is required to identify the worst-case scenario(s). For some task systems, the worst-case scenario is easy to determine. As an example, for independent synchronous (the first job of every task is released at the same time) task systems executed on a uniprocessor platform, the worst-case scenario is the initial instant, known as the critical instant. In this context, the worst-case response time[1] of the tasks is encountered in the first synchronous busy period[2]. This allows efficient exact[3] schedulability tests to exist, running in pseudo-polynomial time (e.g. [21, 22] for fixed-task priority scheduling like Deadline Monotonic or [6,20] for fixed-job priority scheduling like Earliest Deadline First [25]).

However, the critical instant does not correspond to the first synchronous busy period when the tasks are asynchronous, or for multiprocessor platforms. In these cases, a larger time interval has to be considered in order to reach a cycle in the schedule, and the schedulability problem is NP-hard in the strong sense [24]. The only known exact schedulability tests are simulation-based like in [16] and have to consider the whole schedule, using a simulation interval representing finitely the infinite schedule, before concluding about the schedulability of a system, or the worst-case response time of the tasks. Simulation-based schedulability tests can only be used when the context (scheduling algorithm, task system, and platform) is such that the simulation is $C-$sustainable. A schedulability test is $C-$sustainable if a system deemed schedulable when tasks are using their WCET is schedulable even if some tasks do not use up to their WCET [5]. In the sequel, every time a simulation is mentioned, every job of the tasks are assumed to use their WCET.

If the context is such that simulation is $C-$sustainable, we need a finite interval to conclude about schedulability. This interval can either be a simulation interval or a feasibility interval with the following definitions:

- *Simulation interval*: a safe time interval such that the schedule repeats in a cycle.

[1] duration between a job release and its completion

[2] period of continuous processor occupation starting at the critical instant, ignoring tasks of lower priority

[3] necessary and sufficient

– *Feasibility interval*: a finite interval $[a, b]$ such that if all the deadlines of jobs released in the interval are met, then the system is schedulable.

Knowing the length of the simulation interval is also required for capturing the whole behavior of a system when building a pre-run-time schedule [33], also called offline schedule. In this case, the online execution of the system is controlled by a dispatcher. The dispatcher is using the pre-run-time schedule to allocate the tasks to the processors. There is a rich literature addressing the problem of building pre-run-time schedules, see for example [4, 17, 29, 32]. In every case, a pre-run-time schedule has to correspond to a simulation interval, since the schedule has to be repeated infinitely. Our simulation interval will allow to bound the time interval to consider in the search space, or the number of jobs, when building a pre-run-time schedule.

Finally, when displaying a schedule, either for pedagogical purpose, or for characterizing some metrics, it is important to know how long a schedule should be built to capture the whole behavior of the system. Several schedulability analysis tools [30] are intensively based on simulation for illustration purpose.

In this paper, we give and prove correct a general simulation interval for schedules produced by deterministic and memoryless schedulers, for periodic, dependent, arbitrary deadlines tasks executed on identical multiprocessor platforms. Since it is addressing a wide class of scheduling algorithm, this simulation interval is safe, but not tight. Then we provide simulation algorithms, including one requiring a linear space complexity. Finally we compare our simulation interval to other simulation intervals, and we discuss how simulation can be used as an exact schedulability test.

### 1.2 Definitions and notations

In order to present the state-of-the-art about simulation intervals, some definitions are introduced.

A system Sys $= \{\tau_1, \ldots, \tau_n\}$ is a task set, where every task $\tau_i$ is defined by:

- $O_i \in \mathbb{N}$ the task offset, i.e., the release date of the first job $\tau_{i,1}$ of $\tau_i$,
- $C_i \in \mathbb{N}$ the Worst-Case Execution Time (WCET), i.e., the maximum amount of time required on a processor for a job of $\tau_i$ to be executed,
- $T_i \in \mathbb{N}$ the task period, the jobs are released at the instants $O_i + kT_i, k \in \mathbb{N}$,
- $D_i \in \mathbb{N}$ is the relative deadline and represents the timing constraint of a task: the $k^{\text{th}}, (k \in \mathbb{N})$ job $\tau_{i,k}$ of $\tau_i$ must be completely executed in the window $[O_i + k \cdot T_i, O_i + k \cdot T_i + D_i)$. If $\forall i \in \{1 \ldots n\}, D_i \leq T_i$, then the system has *constrained deadlines*, else if deadlines are equal to periods then the system has *implicit deadlines*. In this paper, we consider the most general case of *arbitrary deadlines* (i.e., deadlines and periods are unrelated).

In this paper, we do not assume any relation between $O_i$, $D_i$, and $T_i$ which are independent, arbitrary, integers. The following parameters can be deduced[4]:

- $a_{i,j} \doteq O_i + j \cdot T_i$ is by definition the release time of the job $\tau_{i,j}$,
- $d_{i,j} \doteq O_i + j \cdot T_i + D_i$ the absolute deadline of $\tau_{i,j}$,
- $H \doteq \mathrm{lcm}(T_1, \ldots, T_n)$ is the hyperperiod of the system, with lcm the least common multiple,
- $O^{\max} \doteq \max_{i=1\ldots n}(O_i)$ is the largest offset,
- $U \doteq \sum_{i=1}^{n} C_i/T_i$ is the processor utilization factor.

We consider that tasks are sequential, i.e., a task can execute at most upon one processor simultaneously, i.e. job/task parallelism is forbidden. Moreover, we assume a FIFO order for the execution of the jobs of the same task: the job $\tau_{i,j+1}$ cannot be started before the completion of the job $\tau_{i,j}$.

A task system is said *concrete* if $O_i$ is specified for every task at design time (see [19] for details). Tasks are said *independent* if the executions of jobs of different tasks are not related to each other, and if they do not suspend themselves (e.g., input/output operation). If two tasks $\tau_i$ and $\tau_j$ share a critical resource, then their critical sections (portion of code where they use the critical resource) shall mutually exclude each other (this may be ensured by synchronization tools as semaphores or monitors, or by the scheduler). If the executions of the jobs of $\tau_j$ cannot occur before some executions of the jobs of $\tau_i$, then we say that $\tau_i$ precedes $\tau_j$, noted $\tau_i \prec \tau_j$, and the system is said *precedence constrained*. In this paper, we consider precedence constraints between tasks sharing the same period. Therefore, if $\tau_i \prec \tau_j$ then for any positive integer $k$, job $\tau_{i,k}$ must be completed before starting the job $\tau_{j,k}$.

We consider that the temporal parameters are integer numbers, called time units, which are multiples of the processor clock ticks, and that a scheduling decision can occur at most once at the beginning of every time unit. A scheduler is a decision algorithm which is deciding at every time unit, considering the state of the system, which task is executed on which processor. Note that depending on the scheduling algorithm, the scheduling decision may occur less frequently than at every time unit (e.g., in a fixed-task priority scheduling[5] algorithm, a scheduling decision has to take place only when a new job is released or when a job is completed).

In this paper, we consider *identical* multiprocessor platforms, and we assume that preemption and migration durations are negligible. We assume scheduling points to occur only at integer time units. We consider deterministic and memoryless schedulers, which are those schedulers for which the scheduling decision depends only on the state of the system (see Definition 1) at the current time unit. Therefore, for those schedulers, let $t$ be an integer time unit, the scheduler is taking a decision according to the state $S$ of the system at this time. For the sake of the following proofs, we also define the notion of pre-state $\hat{S}$, which is the state of the system at time $t^-$, occurring at time $t$ but

[4] where $\doteq$ means "equals by definition"

[5] see Definition 4

before releasing the new jobs, and before the scheduling decision. In the following definition, every concept (state, pre-state, local clock, remaining work) is function of the time, but for simplicity, we omit the time in the notations.

**Definition 1 (State and pre-state of a system)** The state of a system of $n$ tasks can be defined as a $(2n)$-tuple $S \doteq \langle C_{\text{rem}_1}, \ldots, C_{\text{rem}_n}, \Omega_1, \ldots, \Omega_n \rangle$, where:

- $\Omega_i$ is the local clock of $\tau_i$, undefined before $O_i$, initialized at 0 at the time $O_i$, being reset at every new request of the task. Formally, at time $t \geq O_i$, $\Omega_i \doteq (t - O_i) \mod T_i$,
- while $C_{\text{rem}_i}$ is the remaining work to process for $\tau_i$.

The pre-state of a system of $n$ tasks can be defined as a $(2n)$-tuple $\hat{S} \doteq \left\langle \hat{C}_{\text{rem}_1}, \ldots, \hat{C}_{\text{rem}_n}, \Omega_1, \ldots, \Omega_n \right\rangle$, where:

- $\Omega_i$ is the same local clock as in the state $S$ of the system,
- $\hat{C}_{\text{rem}_i}$ is the remaining work to process for $\tau_i$ not taking the releases at the considered instant into account.

We can formalize the remaining work in state and pre-state, for any $t \geq O_i$ as follows:

$$\begin{aligned}
\hat{C}_{\text{rem}_i}(t) &\doteq 0, \forall t \leq O_i \\
C_{\text{rem}_i}(t) &\doteq \hat{C}_{\text{rem}_i}(t) + C_i \text{ if } \Omega_i = 0 \\
&\quad\; \hat{C}_{\text{rem}_i}(t) \text{ otherwise} \\
\hat{C}_{\text{rem}_i}(t+1) &\doteq C_{\text{rem}_i}(t) - 1 \text{ if } \tau_i \text{ executed on } [t, t+1) \\
&\quad\; C_{\text{rem}_i}(t) \text{ otherwise}
\end{aligned}$$

Since deadlines are arbitrary, at some time instant several jobs of the same task can be pending: in this case, its remaining work can be greater than its WCET. A task is *ready* at time $t$ if its remaining work to process is not zero. In the sequel, the *total remaining work* of a system is referring to the sum of the individual remaining work of the tasks.

**Definition 2 (Scheduling decision)** A *scheduling decision* on $m$ identical processors at time $t$ is a subset of cardinality $\leq m$ of ready tasks. Every task in the subset is executed on a processor in the time interval $[t, t+1)$.

**Definition 3 (Deterministic and memoryless scheduler)** A scheduler is *deterministic and memoryless* if, and only if, the scheduling decision at time $t$ is *unique* and *univocally* defined by the state of the system (as defined in Definition 1) at time $t$.

**Definition 4 (Fixed-task priority scheduler)** In a *fixed-task priority* (FTP) scheduler, every task is assigned a fixed priority. The scheduling decision is selecting the tasks to be executed based on their priority.

The most popular fixed-task priority schedulers are Rate Monotonic [25] and Deadline Monotonic [24].

**Definition 5 (Fixed-job priority scheduler)** In a *fixed-job priority* (FJP) scheduler, every job is assigned a fixed priority. The scheduling decision is selecting the jobs to be executed based on their priority.

The most common fixed-job priority scheduler is Earliest Deadline First (EDF) [25] where the priority of a job is related to the date of its absolute deadline: the closer the deadline, the higher the priority.

Popular real-time schedulers are deterministic and memoryless as long as the tie-breaker (rule used when two jobs have the same priority) is deterministic and memoryless (e.g., using the task index).

A simulation interval is defined such that an infinite feasible schedule can be expressed on a finite time interval.

**Definition 6 (Feasible schedule)** Let Sys be a task system where tasks are defined by a first release time $O_i$, activated at a period $T_i$ and having a relative deadline $D_i$. If the tasks are independent, an infinite feasible schedule $\sigma$ is such that every job of every task $\tau_i$ is executed and completed in its time window. We denote $s_\sigma(\tau_{i,j})$ (resp. $e_\sigma(\tau_{i,j})$) the starting date (resp. ending date) of the $j^{th}$ job of $\tau_i$ in the schedule $\sigma$. Every job is executed and completed in its time window $[a_{i,j}, d_{i,j}]$ if and only if it satisfies $s_\sigma(\tau_{i,j}) \geq a_{i,j}$ and $e_\sigma(\tau_{i,j}) \leq d_{i,j}$.

**Definition 7 (Feasible schedule on a simulation interval)** A simulation interval of a feasible schedule $\sigma$, generated by a deterministic and memoryless scheduler, is restricted to the interval $[0, b]$, where $b$ is the simulation duration, and is such that at least two states reached in the simulation interval are identical.

## 1.3 State of the art

### *1.3.1 Uniprocessor simulation intervals*

Note that the necessary condition $U \leq 1$ has to hold in the following results.

- The seminal work of [23] shows that $[0, O^{\max}+2H)$ is an upper bound of the simulation interval for fixed-task priority schedulers, and independent task systems with constrained deadlines (i.e., $D_i \leq T_i$). The *transient phase* of the schedule is included in the time window $[0, O^{\max} + H)$, while its *steady phase* (i.e., cyclic part) is given by the schedule in the time window $[O^{\max} + H, O^{\max} + 2H)$.
- It is shown in [16] that, with arbitrary deadlines, $[0, O^{\max} + 2H)$ is still giving an upper bound of the simulation interval for Earliest Deadline First, and fixed-task priority scheduling algorithms.
- The most general result concerning task systems with constrained deadlines is given in [8]. It shows how to determine the exact[6] simulation interval for most online and offline scheduling algorithms. The steady phase of a

[6] here exact means that a shorter interval would not be a simulation interval

schedule starts exactly at the date $\theta_c$, date following the last acyclic idle time, thus the simulation interval is given by $[0, \theta_c + H)$. The date of the last acyclic idle time is $0 \leq \theta_c \leq O^{\max} + H$. This exact bound shows that the first time window of length $H$ with exactly $H(1-U)$ idle times within the time interval $[0, O^{\max} + 2H)$ is the steady phase of the schedule. This result has been extended to non-preemptible tasks, precedence constraints, and resource sharing. It has been extended to multi-threaded tasks in [2].
- An upper bound to the simulation interval is $[0, s_n + H)$ [15] for fixed-task priority scheduling algorithms, for independent tasks with constrained deadlines, where $s_n$ is calculated iteratively on the system, giving the tasks ordered by priority level:

$$\begin{aligned} s_1 &\doteq O_1 \\ s_i &\doteq \max(O_i, O_i + \left\lceil \frac{s_{i-1} - O_i}{T_i} \right\rceil T_i) \end{aligned} \tag{1}$$

We can notice that the feasibility or simulation interval problem for arbitrary deadlines systems is still an open problem in the case of any algorithm other than EDF or fixed-task priority: this paper will fill this gap with an upper bound.

#### *1.3.2 Multiprocessor results*

Two main families of multiprocessor schedulers are usually considered: global schedulers consider one ready queue for the whole set of processors, while partitioned schedulers consider a scheduler and a ready queue per processor. Global schedulers thus allow job migration.

The periodic behavior of schedulers has been studied in the context of global scheduling on multiprocessor platforms for specific scheduling algorithms. For partitioned scheduling, as long as there is no migration, the simulation duration problem consists in studying the simulation duration on each processor: this is thus related to the uniprocessor problem. In the sequel, we consider the problem of the periodic behavior of *global schedulers*.

Every known result concerning global scheduling is provided for independent task systems, except for [4] considering precedence constraints. There are several periodicity results in [10] concerning *constrained deadline* systems, on uniform multiprocessor systems, that can be applied to the identical multiprocessor platforms. If the tasks are synchronous, then any feasible schedule generated by a deterministic and memoryless scheduler has a periodic behavior on the interval $[0, H)$, under the assumption that each job of the same task has the same execution time. For asynchronous tasks systems, $[0, s_n + H)$ is a simulation interval of any feasible schedule generated by a global fixed-task priority scheduler, using the same $s_n$ as in Equation 1.

The case of arbitrary deadlines systems has been studied in [11] for identical multiprocessor platforms. It is shown that any feasible schedule generated by a deterministic and memoryless scheduler is finally periodic. Moreover, for

**Table 1** Main results concerning simulation duration

| Processor | Deadlines | Dependency | Scheduling algorithm | Simulation interval | Reference |
|---|---|---|---|---|---|
| 1 | $D_i \leq T_i$ | independent | fixed-task priority | $[0, O^{\max} + 2H)$ | [23] |
| 1 | arbitrary | independent | fixed-job priority | $[0, O^{\max} + 2H)$ | [16] |
| 1 | $D_i \leq T_i$ | independent | fixed-job priority | $[0, S_n + H)$ | [15] |
| 1 | $D_i \leq T_i$ | mutual exclusion,simple precedence | any work-conserving (with idle task) | $[0, \theta_c + H)$ | [2,8] |
| uniform | $D_i \leq T_i$ | independent | global fixed-task priority | $[0, S_n + H)$ | [10] |
| unrelated | $D_i \leq T_i$ | independent | global fixed-task priority | $[0, S_n + H)$ | [12] |
| identical | arbitrary | independent | global fixed-task priority | $[0, \hat{S}_n + H)$ | [11] |
| identical | $D_i \leq T_i$ | independent | any | $[0, O^{\max} + H \prod_{i=1}^{n}(C_i + 1))$ | [4,26] |
| identical | $D_i \leq T_i$ | simple precedence | any | $[0, O^{\max} + H \prod_{i=1}^{n}(C_i + 1))$ | [4] |
| identical | arbitrary | structural constraint | any | $[0, H \prod_{i=1}^{n}((O_i + D_i - T_i)_0 + 1))$ | this research |

a feasible schedule generated by a fixed-task priority scheduler, $[0, H)$ is a simulation interval for synchronous systems, while $[0, \hat{s}_n + H]$ is a simulation interval for asynchronous systems, with, assuming task indexes ordered from high to low priority:

$$\begin{aligned} \hat{s}_1 &\doteq O_1 \\ \hat{s}_i &\doteq \max\left(O_i, O_i + \left\lceil \frac{\hat{s}_{i-1} - O_i}{T_i} \right\rceil T_i\right) + H_i \end{aligned} \tag{2}$$

with $H_i \doteq \mathrm{lcm}_{j=1\dots i}(T_j)$. This result has been extended to the case of unrelated multiprocessor platforms in [12].

For identical multiprocessor platforms, constrained deadline systems of asynchronous tasks subject to simple precedence constraints, Baro et al. proved that $[0, O^{\max} + H \prod_{i=1}^{n}(C_i + 1))$ is a simulation interval of any feasible schedule generated by an *offline* scheduler [4]. The same interval is used and tuned for fixed-job priority schedulers and independent tasks in [26].

We summarize results and contexts of the state of the art concerning simulation intervals in Table 1.

*This research* This paper is the first result concerning the simulation interval applicable to a large context. It deals with identical multiprocessor platforms, any deterministic and memoryless scheduler, asynchronous periodic tasks with *arbitrary deadlines*, subject to a large class of *structural constraints* (including precedence constraints, mutual exclusions, self-suspensions, preemptive or non-preemptive tasks, see Section 5). Most results concerning multiprocessor platforms currently known in the literature consider independent and preemptive periodic tasks scheduled by specific schedulers (to the best of our knowledge, the global versions of fixed-task priority and Earliest Deadline First). Moreover, we propose an interesting intermediate result, Lemma 1, showing that, for the cyclicity problem, the synchronous case can be used as a worst-case scenario.

### 1.4 Organization of the paper

In Section 2 we present a simple motivating example showing that for synchronous task systems on multiprocessor platforms, the first hyperperiod cannot be considered as a simulation interval for arbitrary deadline systems. In Section 3, we show that the set of feasible schedules for asynchronous task systems is included in the set of feasible schedules for synchronous arbitrary deadlines systems. This is allowing us to easily prove our general result which is Theorem 1. We derive several simulation algorithms, including a linear space complexity algorithm, called zero-memory simulation in Section 4. In Section 5, the simulation duration bound is shown correct also for a large set of tasks dependencies. Then, we compare our simulation interval to other simulation intervals in Section 6. Finally, we discuss usability of simulation as an exact feasibility test for scheduling algorithms in Section 7.

## 2 Motivational example

Let $\mathrm{Sys}_1$ be a system containing three synchronous tasks executed on two processors: $\tau_1$, characterized by $O_1 = 0$, $C_1 = 1$, $T_1 = D_1 = 2$, $\tau_2$, with the same parameters as $\tau_1$, and $\tau_3$ with $O_3 = 0$, $C_3 = 3$, $T_3 = 4$ and $D_3 = 7$. Note that the processor utilization factor of $\mathrm{Sys}_1$ is $U_{\mathrm{Sys}_1} = 7/4$, the hyperperiod is $H = 4$, and the task $\tau_3$ has a deadline greater than its period. The global-EDF schedule of the system $\mathrm{Sys}_1$, while each job of the same task has the same execution time, is shown in Figure 1. We can notice that during the two first hyperperiods (i.e., in the time interval [0,8)), there are two idle slots per hyperperiod. Giving the number of processors $m = 2$ and the utilization factor, the processor executes less workload than the requested workload. For $\mathrm{Sys}_1$, $U = 7/4$ and $m = 2$, in order for the system to execute as much workload as the requested workload, there must be exactly $k$ idle slots in a schedule of length $kH$. We can observe that the states of the system at date 8 and at date 12 correspond to an instant where all previous work is finished by $\tau_1$ and

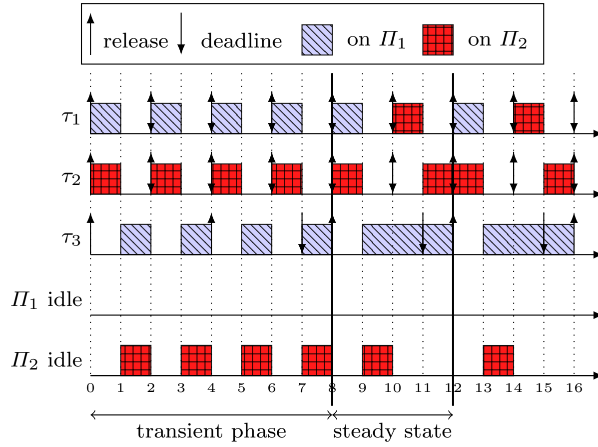


**Fig. 1** Global-EDF schedule of $\mathrm{Sys}_1$ on two processors

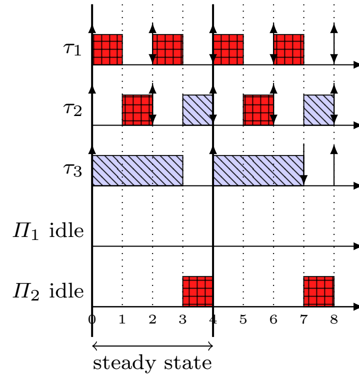


**Fig. 2** LRPTF schedule of $\mathrm{Sys}_1$ on two processors

$\tau_2$, while $\tau_3$ is backlogged by 2 units of execution, hence, the steady state of the schedule is given by the time interval $[8, 12)$, while the transient phase, despite the fact that the tasks are synchronous, lasts during 8 time units. In the steady state of duration $H$, there is, as expected, one idle slot.

If we consider the Longest Remaining Processing Time First (LRPTF) [28] scheduling algorithm, the schedule of the same system has an *empty transient phase* (see Figure 2).

We can observe that, for the considered task system, global-EDF inserts more idle slots than LRPTF because the job of $\tau_3$ cannot be parallelized, while LRPTF is equalizing the remaining work and reduces the idle slots occurring because of jobs non-parallelization.

Deadline Monotonic (the shorter the relative deadline, the higher the priority [24]) priority assignment of the system $\mathrm{Sys}_1$ is infeasible (see Figure 3), and never enters in a cycle where the right amount of idle slots is present. Since

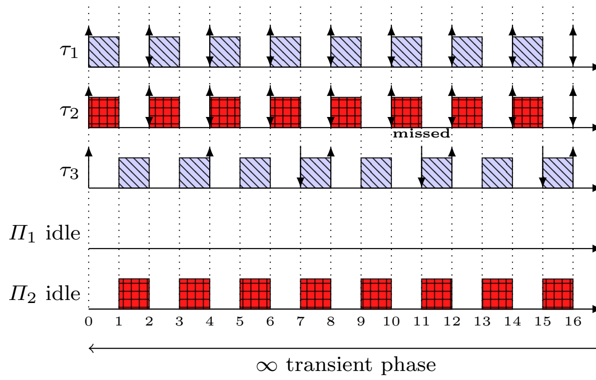


**Fig. 3** Deadline Monotonic schedule of $\text{Sys}_1$ on two processors

there are always two idle slots instead of one per hyperperiod, the lateness of the task $\tau_3$ is increasing with every hyperperiod. The transient phase of the schedule never ends, and in the third hyperperiod of the system, at the time instant 11, $\tau_3$ misses its deadline.

Our example $\text{Sys}_1$ illustrates that several scheduling algorithms which are work-conserving in the uniprocessor case insert different idle slots in the multiprocessor case and do not have the same simulation interval. It is showing also that in the case of synchronous systems with arbitrary deadlines, the time window $[0, H)$ cannot be used as a simulation interval.

## 3 General periodicity result

We first consider independent tasks systems. Moreover we assume in this section that each job of the same task has the same execution time. These two restrictions are relaxed in Section 5.

The time window $[0, H)$ cannot be used as a simulation interval because some tasks with a deadline greater than their period are allowed to be backlogged at the end of the hyperperiod. The factor allowing a task $\tau_i$ to be backlogged is the fact that at least one of its jobs has a release date in $[0, H)$ but its deadline in a subsequent hyperperiod. This is possible only in two cases:

1. the release time $O_i$ of $\tau_i$ is greater than 0, assuming that 0 is corresponding to the first release in the system,
2. the relative deadline $D_i$ of $\tau_i$ is greater than its period $T_i$.

The proof used to obtain an upper bound on the simulation duration is as follows: we are looking for some point in a schedule where the system is behaving cyclically, in other words, two points in time where the same state is encountered. We know, by definition of the states, that the local clocks must be identical, therefore, these two points are an integer number of hyperperiods apart. The difficulty comes from the fact that tasks are asynchronous: we

cannot focus on a specific point in time to look for the cycle. If tasks were synchronous, then we could focus on the hyperperiods, one after the other, looking for backlogged tasks. In order to do so, we show in the sequel that we can study, without loss of generality, only *synchronous* task systems with arbitrary deadlines, and that any result holding for this case is holding also (modulo a transformation of the release times and relative deadlines) to the asynchronous case.

**Definition 8 (Set of feasible schedules)** We define the function $\mathcal{F}$ such that $\mathcal{F}(S)$ is the set of all feasible schedules obtained by deterministic and memoryless schedulers for task system $S$.

**Lemma 1** *Let $S$ be a set of independent tasks with $\forall i \in 1, \ldots, n, O_i \geq 0$. We denote $O_i$ the offset of the task $\tau_i$ and $D_i$ its relative deadline. Let $S'$ be the same system, except for the release dates given by $O'_i = 0$ and the relative deadlines $D'_i = D_i + O_i$. The set of feasible schedules of $S$ is included in the set of feasible schedules of $S'$, i.e., $\mathcal{F}(S) \subseteq \mathcal{F}(S')$.*

*Proof* Let $\sigma \in \mathcal{F}(S)$ be a feasible schedule for $S$, since from Definition 6, for any job $\tau_{i,j}$, $s_\sigma(\tau_{i,j}) \geq O_i + jT_i \geq 0 + jT_i$ and $e_\sigma(\tau_{i,j}) \leq O_i + jT_i + D_i = 0 + jT_i + D'_i$, hence $\sigma \in \mathcal{F}(S')$, proving the lemma. □

The underlying idea behind Lemma 1 is that the time window allocated to every job in $S'$ is including the time window allocated to every job in $S$. Since we are interested in any deterministic and memoryless scheduling algorithm, we see that focusing only on synchronous task systems with an arbitrary deadline cannot reduce the possibilities for a scheduling algorithm to delay its steady phase. For this reason, an upper bound on the case where deadlines are arbitrary is also an upper bound for asynchronous task systems.

**Lemma 2** *For synchronous task systems, if two pre-states are identical, then the scheduling decision of a deterministic and memoryless scheduler is the same.*

*Proof* Following the definition of deterministic and memoryless schedulers, if two states are identical, then the scheduling decision is the same. This lemma states that it is sufficient to consider the pre-state in the case of synchronous systems. Indeed, if two pre-states are identical, their local clocks are the same (and so do the clocks of the corresponding states). Considering $t$ and $t'$ the respective time instants where $\hat{S}$ and $\hat{S}'$ occur, we have $t' = t + kH, k \in \mathbb{N}$, which are the only possible solutions such that every local clock, all starting at the instant 0 (the system is synchronous), are the same. If the values of the remaining work are the same in two pre-states $\hat{S}$ and $\hat{S}'$, then the values are also the same for the corresponding states $S$ and $S'$ because giving Definition 1, we have $C_{\text{rem}_i}(t) = \hat{C}_{\text{rem}_i}(t) + C_i$ if $\Omega_i = 0$, and $C'_{\text{rem}_i}(t) = \hat{C}'_{\text{rem}_i}(t') + C_i$ if $\Omega'_i = 0$. Since $\Omega_i = \Omega'_i$, and $\hat{C}_{\text{rem}_i}(t) = \hat{C}'_{\text{rem}_i}(t')$, then $C_{\text{rem}_i}(t) = C'_{\text{rem}_i}(t')$, and $S = S'$. □

It follows from Lemma 2 that we can only focus on the pre-states to prove the periodicity of synchronous task systems.

**Lemma 3** *Any feasible schedule of a synchronous independent task system generated by a deterministic and memoryless scheduler reaches a cycle at or prior to* $\left(\prod_{i=1}^{n}((D_i - T_i)_0 + 1)\right) H$*, where* $(a)_0 \doteq \max(a, 0)$.

*Proof* Note that the pre-state at the time 0 is given by $\hat{S}_0^0$ in Figure 4. In this figure, since only hyperperiods are considered, the pre-states can be represented only by the values of $\hat{C}_{\text{rem}_i}$, every local clock being null. In order to prove the lemma, we will show that the number of distinct pre-states for every hyperperiod $kH, k \in \mathbb{N}^+$, in any feasible schedule, is bounded above by $\prod_{i=1}^{n}((D_i - T_i)_0 + 1)$.

- Constrained deadlines case: if every task has a constrained deadline (i.e., $D_i \leq T_i$), then if the pre-state reached at the date $H$ is such that there is an $i \in 1..n$ such that $\hat{C}_{\text{rem}_i}(H) > 0$, then the schedule is infeasible. Indeed, every job started during the first hyperperiod has to be finished before the end of this hyperperiod. As a result there is only one possible pre-state at the date $H$ for any feasible schedule, which is identical to the initial state. Hence, any feasible schedule has a steady phase given by the interval $[0, H)$, showing the lemma for this case.
- Case of only one task, $\tau_i$, having a deadline greater than its period. We first give the proof for $D_i = T_i + 1$. In any feasible schedule there is at most one time unit of the $(H/T_i)^{th}$ job of $\tau_i$ backlogged at the time instant $H$, otherwise the system cannot be feasible, while every other job released in the first hyperperiod has to be finished since $\forall j \neq i, D_j \leq T_j$. Therefore the only possible pre-states of the system in a feasible schedule at the date $H$ can be defined by $\hat{S}_0^H$ and $\hat{S}_1^H$ in Figure 4. Note that $\hat{S}_0^H$ is the same as $\hat{S}_0^0$, and so if the schedule reaches this state, then it is behaving cyclically from this point: the schedule $[0, H)$ will be repeated infinitely. If the system is in $\hat{S}_1^H$, then consider the schedule at the time $2H$: there again, only two possible pre-states can be part of a feasible schedule, $\hat{S}_0^{2H} = \hat{S}_0^0$ and $\hat{S}_1^{2H} = \hat{S}_1^H$. If the system is in the pre-state $\hat{S}_0^{2H}$ then the schedule behaves cyclically over the interval $[0, 2H)$; else the schedule has a transient phase on $[0, H)$ (from pre-state $\hat{S}_0^0$ to $\hat{S}_1^H$) followed by a steady phase on $[H, 2H)$ (from $\hat{S}_1^H$ to $\hat{S}_1^{2H}$). The maximal simulation duration is hence $2H$, proving the lemma for one task having $D_i = T_i + 1$.
  Now suppose that $D_i = T_i + k$ with $k$ an arbitrary finite positive integer. If we name $\hat{S}_j^{pH}$ any reachable pre-state in a feasible schedule where $0 \leq j \leq k$ gives the remaining work to process for $\tau_i$ at the date $pH$, with $p$ a positive integer, it is obvious that there are only $k+1$ possible different pre-states. As a consequence, the possible cyclic behaviors of any feasible schedule are bounded by $(k+1)H$. Any combination of a transient phase lasting over $[0, qH)$ followed by a steady phase over $[qH, rH)$ with $0 \leq q < k$, $r \geq q+1$ and $r \leq k$ can be a feasible memoryless and deterministic schedule.
- If several tasks have a deadline greater than their period, then we could represent the pre-states that may be reached by a feasible schedule each

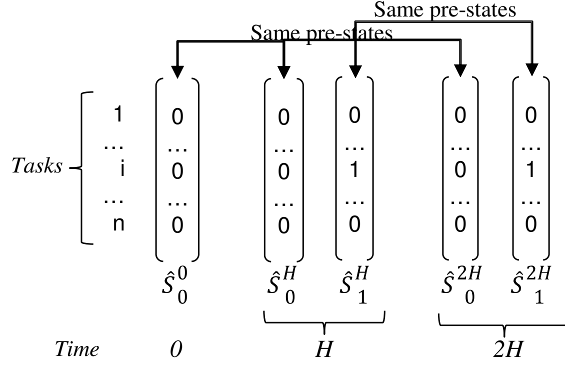


**Fig. 4** States that can be reached in a feasible schedule at times 0, $H$ and $2H$ for $D_i - T_i = 1$

hyperperiod $H$ as a $n$-dimensional matrix given by the Cartesian product of pre-states where each task can be delayed by an amount between 0 and $(D_i - T_i)_0$. The number of elements of this matrix is therefore $\prod_{i=1}^{n}((D_i - T_i)_0 + 1)$. As a result, it is impossible for a feasible schedule not to have reached two identical pre-states after $\left(\prod_{i=1}^{n}((D_i - T_i)_0 + 1)\right) H$ time units.

□

Now we have the material to provide and prove correct our main result.

**Theorem 1** *Any feasible schedule of an asynchronous independent tasks system generated by a deterministic and memoryless scheduler reaches a cycle at or prior to* $\left(\prod_{i=1}^{n}((O_i + D_i - T_i)_0 + 1)\right) H$.

*Proof* We know from Lemma 1 that any feasible schedule for an asynchronous system $S$ is a feasible schedule for a synchronous system $S'$ such that $O'_i = 0$ and $D'_i = O_i + D_i$, i.e., $\mathcal{F}(S) \subseteq \mathcal{F}(S')$. From Lemma 3, any feasible schedule of $S'$ reaches a cycle at or prior to $\left(\prod_{i=1}^{n}((D'_i - T_i)_0 + 1)\right) H$, since $\mathcal{F}(S) \subseteq \mathcal{F}(S')$, then any feasible schedule of $S$ reaches a cycle at or prior to $\left(\prod_{i=1}^{n}((D'_i - T_i)_0 + 1)\right) H$. Substituting $D'_i$ by $O_i + D_i$ we obtain the theorem.

□

## 4 Simulation algorithms

In this section, we propose a generic method to build a feasible schedule for sets of independent asynchronous tasks, scheduled on identical processors by a deterministic and memoryless scheduling algorithm.

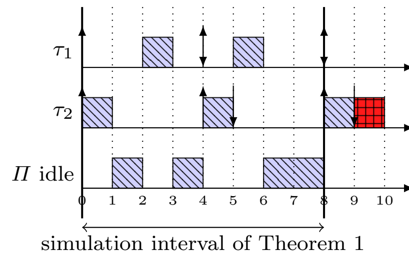


**Fig. 5** A non-feasible deterministic and memoryless schedule for $\mathrm{Sys}_2$ on one processor

## 4.1 Motivation

Theorem 1 states that any feasible schedule reaches a cycle at or prior to the given bound. Nevertheless, a more important question is if a given schedule, meeting the timing constraints on the simulation interval, is indeed a feasible schedule. As an example, let us consider a task system $\mathrm{Sys}_2$ of two tasks $\tau_1$ and $\tau_2$ scheduled by a deterministic and memoryless scheduler on a single processor. Both tasks are simultaneous $O_1 = O_2 = 0$, and share the same period $T_1 = T_2 = 4$. Their worst-case execution times are $C_1 = 1$, and $C_2 = 2$. $\tau_1$ has an implicit deadline $D_1 = T_1 = 4$, while $\tau_2$ has a greater deadline than its period: $D_2 = 5$. Using Theorem 1, we know that any feasible schedule produced by a deterministic and memoryless scheduler for $\mathrm{Sys}_2$ reaches a cycle at, or prior to, $\left(\prod_{i=1}^{n} ((O_i + D_i - T_i)_0 + 1)\right) H = (1 \times 2) \times 4 = 8$. We consider Figure 5, giving a schedule produced on the interval $[0, 10]$ by a deterministic and memoryless scheduler. On the simulation interval $[0, 8]$, no deadline is missed. It would nevertheless be a mistake to conclude that the schedule is feasible when reaching the time 8: the next deadline of $\tau_2$, at time 9, cannot be met. Indeed, the backlogged work for $\tau_2$ is 2 time units at time 8, while the corresponding deadline is only one time unit later, at time 9.

It is important to stress the fact that Theorem 1 is giving a simulation interval for feasible schedules, but does not state that a schedule reaching the simulation interval while not missing a deadline is feasible. This statement would be wrong, like illustrated in Figure 5.

## 4.2 Simulation using exponential memory

A trivial way to check for feasibility of a schedule under construction is directly based on Definition 7. In this case, during a simulation, every state is stored during the construction of the schedule. For every reached state at a time $t$, if any deadline is missed, then the schedule is not feasible. If no deadline is missed, we have to check in history if the current state has already been reached. In order to avoid checking useless points in history, we can use the fact that two states can be identical only if their clocks are an integer amount of hyperperiods apart. This technique is highly memory consuming, since every

state of the schedule has to be stored, but the benefit is that the simulation interval is tight. Theorem 1 is giving an upper bound on the interval that has to be studied.

In order to limit the amount of states to store, we can use Theorem 1: in this case, we only have to store one state per hyperperiod. In this case, for any time $t$, as long as no deadline is missed, if $t$ is not a multiple of the hyperperiod, then we carry on the next time unit. If $t$ is a multiple of the hyperperiod, then we store it and check in the previously stored states if any state is identical to the current state. Simulation is then stopped and the schedule claimed feasible. The benefits are that we store very few states compared to storing every state, and that we do not have to store the local clocks. The drawback of this algorithm compared to the previous one is that the simulation interval is not tight, since cycle detection is only checked at the end of each hyperperiod, but not between two subsequent hyperperiods. The major drawback is that it is also exponential in space, since the maximum number of states to store is given by $\left(\prod_{i=1}^{n}\left((O_i + D_i - T_i)_0 + 1\right)\right)$ (see Theorem 1).

### 4.3 Zero-memory simulation

The two previous ways to build a simulation presented in the previous section are both storing an exponential amount of states, even if the second one is smaller by far than the first one. We show in this section that we can build a simulation without storing any state of the history.

**Lemma 4 (Non-negative laxity condition)** *For any hyperperiod $kH, k \in \mathbb{N}$, a necessary feasibility condition, called non-negative laxity condition, is that in the pre-state of a schedule, $\forall i \in 1 \ldots n, \hat{C}_{\text{rem}_i} \leq (O_i + D_i - T_i)_0$. There are only $\left(\prod_{i=1}^{n}\left((O_i + D_i - T_i)_0 + 1\right)\right)$ pre-states meeting the non-negative laxity condition.*

*Proof* We remind that the remaining work to process $\hat{C}_{\text{rem}_i}$ in a pre-state corresponds to the work to process for jobs released before the considered instant, and that $(O_i + D_i - T_i)_0$ represents the largest time interval between the end of a hyperperiod and the deadline of any job released in this hyperperiod. When building a schedule, if we encounter at the date $kH$ a pre-state such that $\exists i \in 1 \ldots n, \hat{C}_{\text{rem}_i} > (O_i + D_i - T_i)_0$ then the deadline of $\tau_i$ at time $kH + (O_i + D_i - T_i)_0$ will be missed since there is more remaining work than time left until the deadline following or at the hyperperiod $kH$. The enumeration of the possible states meeting this condition is given by the Cartesian product of the possible values of $\hat{C}_{\text{rem}_i} \in 0 \ldots (O_i + D_i - T_i)_0$, giving at most $\left(\prod_{i=1}^{n}\left((O_i + D_i - T_i)_0 + 1\right)\right)$ pre-states. □

If we look back at Figure 5, we see that at the time instant 8, $\tau_2$ has a negative laxity since $\hat{C}_{\text{rem}_2} = 2 > (O_2 + D_2 - T_2)_0 = 1$. We show in the next theorem that the non-negative laxity condition is not only necessary but also sufficient.

**Theorem 2 (Zero-memory schedule)** *Let a schedule of an asynchronous independent task system, be generated by a deterministic and memoryless scheduler on the time interval* $[0 \ldots (\prod_{i=1}^{n}((O_i + D_i - T_i)_0 + 1)) H]$*, then the (infinite) schedule is feasible if, and only if, all the following conditions are satisfied:*

- *no deadline is missed in the interval,*
- *for any integer* $k \in 0, \ldots, (\prod_{i=1}^{n}((O_i + D_i - T_i)_0 + 1))$*, there is a non-negative laxity,*

*Proof* The **only if** part is trivial and a direct application of Lemma 4.

For the **if** part, we know from the first item that no deadline is missed in the built interval, nevertheless, we have to show that no deadline will be missed after the end of the interval. The non-negative laxity condition is met at every hyperperiod $k \in \{0, 1, \ldots, (\prod_{i=1}^{n}((O_i + D_i - T_i)_0 + 1))\}$, but there are only $(\prod_{i=1}^{n}((O_i + D_i - T_i)_0 + 1))$ possible pre-states meeting the non-negative laxity condition (see Lemma 4).

Since we encountered $(\prod_{i=1}^{n}((O_i + D_i - T_i)_0 + 1)) + 1$ pre-states, at least two are identical, meaning that the schedule behaves cyclically. Since no deadline has been missed in the interval, and that a cyclic behaviour has been reached, then no deadline will ever be missed. □

The direct application of Theorem 2 is a memory efficient simulation algorithm, not storing any state, hence the name zero-memory schedule. At every hyperperiod, we check if the non-negative laxity condition (Lemma 4) is met. When reaching the simulation upper bound given by Theorem 1, if the non-negative laxity condition was never violated in the previous hyperperiods, then the schedule is feasible. The drawback of this method is that the upper bound of the simulation interval is always reached.

### 4.4 Optimization of the simulation interval

The non-negative laxity condition of Lemma 4 is considering every task independently. Consider three tasks $\tau_i, i = \{1, 2, 3\}$, executed on two processors such that $(O_i + D_i - T_i)_0 = 1$, and that their backlogged work at the hyperperiod is one time unit. Then we cannot execute them in time, since only two of them at most can be executed prior to their next deadline. Therefore, even if the pre-states having a remaining execution time of one for these three tasks are meeting the non-negative laxity condition, all of them are leading to a deadline miss. We could therefore use a tighter necessary schedulability test for the non-negative laxity condition, either based on the system, or based on the scheduling algorithm (e.g. global fixed-priority).

An easy way to do that is to replace the trivial non-negative laxity condition by a uniprocessor test on a processor of speed $m$. Any necessary feasibility condition on a $m-$speed processor is also a necessary feasibility condition on $m$ processors of unitary speed.

As an example, we could use a demand bound function, or a simplified version of a demand bound function checking only if the next deadline of the tasks could be met on a $m-$speed processor. This step is straightforward, the only difficulty is to enumerate the number of reachable pre-states which are satisfying this new necessary condition in order to obtain a less pessimistic upper bound on the amount of hyperperiods to consider.

## 5 Generalization to dependent tasks

The main cyclicity result of Theorem 1 is based on the following intermediate results:

- for synchronous task systems, the amount of possible pre-states at each hyperperiod of the system is limited by the Cartesian product of the possible remaining execution time of the tasks allowed to be backlogged at the end of a hyperperiod (Lemma 3);
- the set of feasible schedules for asynchronous task systems is included in the set of feasible schedules of synchronous task systems where the absolute deadlines are preserved (Lemma 1).

In order to generalize our simulation interval to a large set of systems, we consider structural constraints.

**Definition 9 (structural constraints)** A structural constraint is a relation between jobs or sub-jobs, forbidding some execution orders, preemptions, or insuring a minimal delay between the end of a job (or sub-job) and the start of another one.

A set of tasks subject to structural constraints is called a dependent tasks system. This large definition covers mutual exclusions, precedence constraints between jobs, as well as tasks suspension. We do not need here to give further details, since we can only see a structural constraint as forbidding some schedules, while any feasible schedule meeting the structural constraints also has to be feasible regarding the temporal constraints.

We consider here linearized tasks models, where the control flow graph of a task is reduced to a single line of execution, capturing the longest duration, and structural constraints. Such a linearized model is classic in the literature, and used for example in [32]. A concrete example of how to obtain a linearized task from a control flow graph is detailed in [27] in the context of the Spring C compiler.

**Lemma 5** *Let $S$ be a dependent tasks system. Let $S'$ be the same set of tasks, considered as independent.* $\mathcal{F}(S) \subseteq \mathcal{F}(S')$.

*Proof* Any feasible schedule of $S$ also has to meet the temporal constraints that the corresponding independent task system has to meet. Therefore any feasible schedule of $S$ is also included in $\mathcal{F}(S')$. □

**Theorem 3** *Any feasible schedule of an asynchronous dependent tasks system generated by a deterministic and memoryless scheduler reaches a cycle at or prior to* $\left(\prod_{i=1}^{n}((O_i + D_i - T_i)_0 + 1)\right) H$.

*Proof* We know from Lemma 5 that any feasible schedule for a dependent system $S$ is also a feasible schedule for the corresponding independent task system $S'$, i.e.,$\mathcal{F}(S) \subseteq \mathcal{F}(S')$. Since from Theorem 1, the result holds for any independent tasks system $S'$, then the result also holds for any dependent task system $S$. □

It is important to recall that the main results, Theorem 1 and Theorem 3, concern the cyclicity of the schedule generated for the tasks model. The possible behaviors of the real system are close to infinite, and, assuming an arbitrary small processor cycle, a system with a task $\tau_i$ having an arbitrary deadline or a non null offset could, in theory, exhibit, when scheduled online, a close to infinite behavior without any cycle without missing a deadline. For example, after one hyperperiod, the remaining execution time could be $\delta_i \leq D_i - T_i + O_i$, then $\delta_i - \epsilon$ at the next hyperperiod, with an $\epsilon$ as small as the processor cycle, and so on. Theorem 1 and Theorem 3 are therefore limited to the simulation duration of the model of the tasks. This result is nevertheless interesting if the simulation of the tasks model exhibits the worst-case behavior of any possible online execution of the system. It is the case only if the context is C-sustainable. This property, and the usefulness of our contribution for schedulability analysis is discussed in Section 7.

## 6 Discussion, comparison with other simulation intervals

In this section, we consider the star operator as the repetition of its preceding interval, and $[a, b)[c, d)^*$ represents the schedule over the interval $[a, b)$ followed by the schedule over $[c, d)$ repeated cyclically.

*Application of the main result* If we use Theorem 1 on $\text{Sys}_1$ (see Section 2), we obtain an upper bound of $(3+1)\times1\times1\times H = 4H$ for the simulation interval. We see that, for LRPTF in Figure 2 we have an infinite feasible schedule $[0, H)^*$, while global-EDF in Figure 1 gives a feasible schedule $[0, 2H)[2H, 3H)^*$. The states reached by global-EDF at each hyperperiod are $(0, 0, 0, 0, 0, 0)$ at the origin, $(0, 0, 1, 0, 0, 0)$ at the time $H$, $(0, 0, 2, 0, 0, 0)$ at $2H$, $(0, 0, 2, 0, 0, 0)$ at $3H$. We can, as an example, build a feasible schedule lasting $[0, 4H)^*$ starting at the state $(0, 0, 0, 0, 0, 0)$, and then passing by the states $(0, 0, 1, 0, 0, 0)$ at $H$, $(0, 0, 2, 0, 0, 0)$ at $2H$, $(0, 0, 3, 0, 0, 0)$ at $3H$, and $(0, 0, 3, 0, 0, 0)$ at $4H$, as illustrated in Figure 7. The scheduling algorithm used to generate such a schedule is not corresponding to any popular scheduling algorithm, but we can imagine a deterministic and memoryless scheduling algorithm, giving this schedule, defined by an array indexed by a state of a system giving for any possible state a scheduling decision.

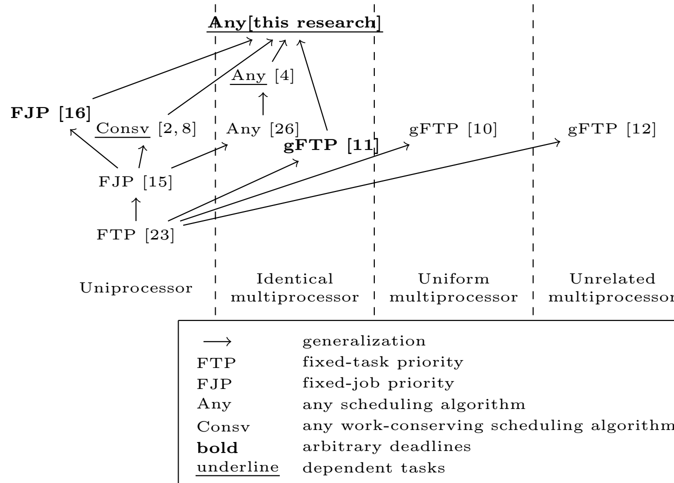


**Fig. 6** Classification of the main results concerning simulation duration

*Comparison with other existing bounds* In Table 1, the main results concerning periodicity are summarized. All the results assume a deterministic and memoryless scheduling algorithm. Fig 6 is giving a classification of these results based on a generalization relationship: we see that, except for [10] and [12], our simulation interval can be applied to any context where the other simulation intervals hold.

We can note that for identical processors, this research considers the widest area of application: arbitrary deadlines, the widest class of structural constraints ever considered, and any deterministic and memoryless algorithm (including any popular algorithm like fixed-task or fixed-job priority based schedulers, as well as offline methods a.k.a. time-triggered scheduling). This generality has a cost on the exactness of the upper bound which may perform less well than a more specific simulation interval bound. Nevertheless, we show in this section that it is incomparable to other simulation interval bounds for multiprocessor systems, where by incomparable we mean that for some task systems, our upper bound behaves better (i.e., is lower) than the other upper bounds, while for other systems the other upper bounds behave better than ours.

In order to compare our bound to the bound provided for the case of fixed-task priority schedulers in [11], we consider a simple system of two tasks $\tau_1$ and $\tau_2$ with the same period $T_1 = T_2 = 8$, and offsets and deadlines given by $O_1 = 1, D_1 = 7, O_2 = 0, D_2 = 8$, and we consider a fixed-task priority scheduler assigning a higher priority to $\tau_1$ than to $\tau_2$. Theorem 1 gives a simulation interval $[0, 8)$, while the bound given in [11] (see Equation 2) gives $[0, 24)$. In this case, since the deadlines are lower than the periods, we could also use the upper bound given in [10] (see Equation 1), and obtain the simulation interval $[0, 16)$.

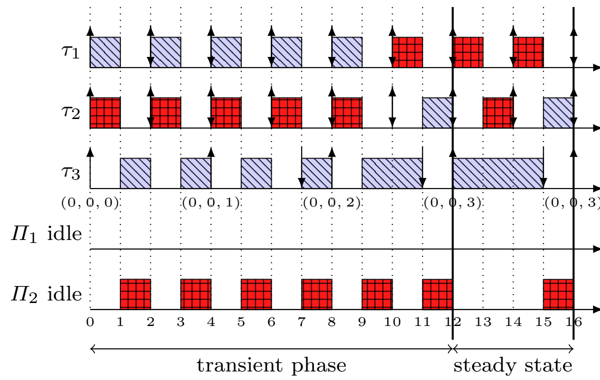


**Fig. 7** Schedule lasting $4H$ generated by a deterministic and memoryless scheduler for $\mathrm{Sys}_1$ on two processors

If we consider a different system with $O_1 = 1, D_1 = 7, T_1 = 12, O_2 = 0, D_2 = 9, T_2 = 8$, then the simulation intervals are $[0, 96)$ for Theorem 1, and still $[0, 24)$ for [11], and cannot be calculated with [10], because $D_1 > T_1$. We can see that the bounds are not comparable, therefore, in the case where several upper bounds could be applied, the minimal value giving a simulation interval upper bound should be chosen.

Note that if the tasks were involving any structural constraint as mutual exclusions, precedence constraints, suspension delays, or non preemptive tasks, Theorem 1 would still hold, while the other periodicity results concerning multiprocessor systems are not applicable.

*Tightness* Our bound is safe, but not tight, as illustrated in the following example. Let a system be composed of two synchronous independent tasks $\tau_1$ and $\tau_2$, executed on a single processor, such that $D_1 = T_1 + 1$ and $D_2 = T_2 + 1$. Theorem 1 is giving an upper bound of $4H$ for the cycle, because the pre-states that can be reached at each hyperperiod are given by $(0, 0, 0, 0)$, $(0, 1, 0, 0)$, $(1, 0, 0, 0)$ and $(1, 1, 0, 0)$. But clearly, if both tasks have a remaining processing time of one time unit, with zero laxity (both deadlines happen one time unit after the considered hyperperiod), then the schedule cannot be feasible. As a consequence, in this case, the longest feasible schedule without reaching twice the same state is constrained to the time interval $[0, 3H)$. In general, a test like a demand bound function could be used to check if the states obtained by the Cartesian product of the possible lateness of the tasks can lead to a feasible schedule or not in order to reduce the bound (see Section 4.4).

## 7 Using simulation as a schedulability test

Previous sections considered feasibility of a schedule. This section considers the (online) schedulability of a system giving a scheduler, and also discusses implementation issues of pre-run-time schedules.

Checking for schedulability of a task system by a scheduling algorithm has been addressed in many ways in the literature. Several methods, based on the demand for optimal schedulers, or the request for fixed-task priority schedulers, are exact for fully preemptive task systems executed on a single processor as long as a critical instant can occur. Nevertheless, as soon as there is no critical instant, the schedulability problem is co-NP-hard in the strong sense [23]. Simulation based tests have been proposed as exact tests for this type of systems. Simulation based tests have an exponential complexity, because every simulation interval includes at least a hyperperiod, which is exponential. As a consequence, simulation should be used as a schedulability test only for classes of schedulability problems which are NP-hard or co-NP-hard in the strong sense.

Moreover, simulations usually consider fixed parameters: the offset, the execution time, as well as the period and deadline are fixed. This is untrue in general concerning the execution time: the online execution of a system does not exhibit (unless forced) the WCET of the task for every job. As a result, using simulation for schedulability analysis has to carefully consider if considering that every job consumes the WCET of the task is always the worst-case scenario. This property is named C-sustainability.

Contexts which are not C-sustainable are subject to scheduling anomalies. A scheduling anomaly occurs when reducing resource consumption (i.e., reducing the execution time) can increase the worst-case response time of a task. For example, as soon as tasks are not fully preemptible (non preemptible tasks, mutual exclusion), it is easy to exhibit scheduling anomalies. Scheduling anomalies can also occur in most cases of structural constraints. The sustainability concept has been extended to most tasks parameters [5]. Sustainability addresses a context which involves three aspects: the constraints on the tasks (independent versus structural constraints subject to scheduling anomalies), the schedulability or feasibility test, the addressed online execution of the tasks. C-sustainability represents the property for a positive schedulability or feasibility test to stay true for any possible variation (typically in the interval $[0..C_i]$) of the actual execution time of any task $\tau_i$ taking the structural constraints into account and considering the way the system will be executed online.

When the variation of the online parameters of the system does not offer sustainability, some tests with a highly exponential computational and space complexity have been proposed to address the schedulability problem. For example, schedulability of sporadic tasks systems on identical multiprocessor platforms is addressed in [3], which is explicitely storing and exploring the search space of every possible simulation based on every possible release date of every job. The storage of every state is required in order to find a cycle by comparing every new state to the previously built states. Other methods, using e.g., timed automata to model the task system, let a model checker build the search space and find the cyclic points, like [9, 18, 31]. The same kind of exhaustive methods have also been used in the uniprocessor case, when scheduling anomalies can occur, like when tasks can self-suspend [1]. These

methods are not only highly exponential in time, but also in space, since they store every state of every possible behavior of the system.

The simulation interval can be used for schedulability analysis only for scheduling algorithms and contexts which are sustainable regarding the online variation of the parameters. For example, most popular scheduling algorithms are C-sustainable when tasks are independent.

For C-sustainable contexts, if the tasks are strictly periodic, when considering tasks executed with their WCET as execution time, period, release date, and deadline, obtaining a feasible schedule by simulation is an exact test proving that the system is schedulable. Using our zero-memory simulation algorithm requires only to store one state at any time, has a space complexity of $O(n)$, but an exponential computational complexity.

On the opposite, if the tasks are sporadic, most multiprocessor scheduling algorithms are not T-sustainable, therefore, a simulation cannot be used as a schedulability test, and in the best of our knowledge, only exhaustive methods can be used, at the cost of an exponential space complexity, and a highly exponential computational complexity.

Finally, for dependent tasks systems, when scheduling anomalies are possible, e.g. tasks subject to mutual exclusion, simulation is not C-sustainable when considering an implementation on an online scheduler. Nevertheless, in this context, a static scheduler (a dispatcher using a pre-run-time schedule to allocate the processing resources) can be used to execute infinitely a feasible schedule [33]. In every case, this static scheduler, and the task model, have to be carefully designed to ensure that no scheduling anomaly can occur. For example, if in the model, a task is supposed to enter in a critical section after 2 time units, the real execution of the task may reach this critical section earlier than expected, and if the task was allowed to continue its execution in the time window planned in the pre-run-time schedule, it could create a scheduling anomaly. A possible way to prevent scheduling anomalies in static scheduling is to split tasks around the synchronization points (e.g. critical sections) into sub-tasks. Precedence constraints are then added between the sub-tasks to enforce the sequential behavior of the original task. When executing the pre-run-time schedule, the scheduler has to ensure precedence constraints, and that a (sub-)task does not start too early like in [14] in order to enforce C-sustainability. Building a pre-run-time schedule to be executed by a static scheduler can thus be done using the contribution of the paper, but this scheduler has to be carefully designed to avoid any scheduling anomaly.

## 8 Conclusion

The problem tackled in this paper is the periodicity problem for feasible schedules produced by any deterministic and memoryless scheduler, in uniprocessor and multiprocessor cases, for any structural constraints (mutual exclusions, precedence constraints, self-suspension, non-preemptive tasks, etc.). The result concerning the periodicity of schedules is, to the best of our knowledge,

the most general result ever proposed in the context of uniprocessor scheduling as well as in the context of identical multiprocessor systems, since it concerns any deterministic and memoryless scheduler, arbitrary deadlines, and dependent task systems.

We prove in Lemma 1 how to reduce the general asynchronous and arbitrary deadlines problem to a synchronous and arbitrary deadlines problem. This intermediate result has a major impact on the relative simplicity of the proof of the main theorem. Then we have shown that the cycle is reached for any feasible schedule at most at the time $\left(\prod_{i=1}^{n}\left((O_i + D_i - T_i)_0 + 1\right)\right) H$. This result might be improved if we take into account the local feasibility of the tasks, but we believe that the applicability of the upper bound would be weakened by the difficulty to handle it in this extended form.

We stress the fact, using an example, that reaching the simulation bound without missing a deadline does not prove that the schedule is feasible. We introduce the system laxity as a means to check for feasibility of a schedule under construction when reaching the simulation bound. We then derive several simulation algorithms, including a zero-memory simulation algorithm, allowing not to store any past state.

Finally, we discuss how simulation can be used for schedulability analysis of strictly periodic task systems, in the context of C-sustainable algorithms. This exact simulation based schedulability test can be applied to any deterministic and memoryless scheduler, at the cost of an exponential computational complexity, but a $O(n)$ space complexity.

We also want to stress the fact that our result is an upper bound for any deterministic and memoryless scheduler, therefore it may be improved for specific scheduling algorithms. As an example, specific bounds concerning fixed-task priority schedulers like in [11], can in some specific contexts be lower than ours. In other contexts our bound can be lower. The best known bound would then to be considered, for such a specific case (fixed-task priority, independent tasks), as the minimal value of the two upper bounds.

In the future, we plan to extend this result to uniform and unrelated multiprocessor platforms. We also plan to improve existing bounds for specific scheduling algorithms using our intermediate result, the Lemma 1.

## References


1. Abdeddaïm, Y., Masson, D.: The scheduling problem of self-suspending periodic real-time tasks. In: Proceedings of the 20th International Conference on Real-Time and Network Systems, RTNS '12, pp. 211–220. ACM, New York, NY, USA (2012). DOI 10.1145/2392987.2393014. URL http://doi.acm.org/10.1145/2392987.2393014
2. Bado, B., George, L., Courbin, P., Goossens, J.: A semi-partitioned approach for parallel real-time scheduling. In: Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS), pp. 151–160. ACM (2012)
3. Baker, T.P., Cirinei, M.: Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In: Proceedings of the 11th international conference on Principles of distributed systems, pp. 62–75. Springer-Verlag (2007)

4. Baro, J., Boniol, F., Cordovilla, M., Noulard, E., Pagetti, C.: Off-line (optimal) multiprocessor scheduling of dependent periodic tasks. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC), pp. 1815–1820. ACM (2012)
5. Baruah, S., Burns, A.: Sustainable scheduling analysis. In: Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International, pp. 159–168. IEEE (2006)
6. Baruah, S.K., Howell, R.R., Rosier, L.: Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Systems **2**, 301–324 (1990)
7. Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., Baruah, S.: A categorization of real-time multiprocessor scheduling problems and algorithms. Handbook of Scheduling (2005)
8. Choquet-Geniet, A., Grolleau, E.: Minimal schedulability interval for real time systems of periodic tasks with offsets. Theoretical of Computer Sciences **310**, 117–134 (2004)
9. Cordovilla, M., Boniol, F., Noulard, E., Pagetti, C.: Multiprocessor schedulability analyser. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, pp. 735–741. ACM, New York, NY, USA (2011). DOI 10.1145/1982185.1982345. URL http://doi.acm.org/10.1145/1982185.1982345
10. Cucu, L., Goossens, J.: Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors. In: Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 397–404 (2006)
11. Cucu, L., Goossens, J.: Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems. In: Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE), pp. 1635–1640 (2007)
12. Cucu-Grosjean, L., Goossens, J.: Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. Journal of Systems Architecture - Embedded Systems Design **57**(5), 561–569 (2011)
13. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys **43**(4), 35 (2011)
14. Fohler, G.: Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In: Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE, pp. 152–161 (1995). DOI 10.1109/REAL.1995.495205
15. Goossens, J., Devillers, R.: The non-optimality of the monotonic assignments for hard real-time offset free systems. Real-Time Systems: The International Journal of Time-Critical Computing **13**(2), 107–126 (1997)
16. Goossens, J., Devillers, R.: Feasibility intervals for the deadline driven scheduler with arbitrary deadlines. In: Proceedings of the 6th IEEE International Conference on Real-time Computing Systems and Applications, pp. 54–61 (1999)
17. Grolleau, E., Choquet-Geniet, A.: Off-line computation of real-time schedules using petri nets. Discrete Event Dynamic Systems **12**(3), 311–333 (2002)
18. Guan, N., Gu, Z., Lv, M., Deng, Q., Yu, G.: Schedulability analysis of global fixed-priority or edf multiprocessor scheduling with symbolic model-checking. In: Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on, pp. 556–560 (2008). DOI 10.1109/ISORC.2008.74
19. Jeffay, K., Stanat, D.F., Martel, C.U.: On non-preemptive scheduling of period and sporadic tasks. In: Real-Time Systems Symposium, pp. 129–139. IEEE (1991)
20. Jeffay, K., Stone, D.L.: Accounting for interrupt handling costs in dynamic priority task systems. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), pp. 212–221 (1993)
21. Joseph, M., Pandya, P.: Finding response times in real-time system. The Computer Journal **29**(5), 390–395 (1986)
22. Lehoczky, J.P.: Fixed priority scheduling of periodic task sets with arbitrary deadlines. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), pp. 201–213 (1990)
23. Leung, J., Merrill, J.: A note on preemptive scheduling of periodic, real-time tasks. Information Processing Letters **11**(3), 115–118 (1980)
24. Leung, J., Whitehead, J.: On the complexity of fixed-priority scheduling of periodic real-time tasks. Performance Evaluation(2) pp. 237–250 (1982)
25. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20**(1), 46–61 (1973)

26. Nélis, V., Yomsi, P.M., Goossens, J.: Feasibility intervals for homogeneous multicores, asynchronous periodic tasks, and fjp schedulers. In: Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS), pp. 277–286. ACM (2013)
27. Niehaus, D.: Program representation and execution in real-time multiprocessor systems (1994)
28. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems, 3rd edn. Springer Publishing Company, Incorporated (2008)
29. Pop, P., Eles, P., Peng, Z., Pop, T.: Analysis and optimization of distributed real-time embedded systems. ACM Transactions on Design Automation of Electronic Systems (TODAES) **11**(3), 593–625 (2006)
30. Singhoff, F., Plantec, A., Dissaux, P.: Can we increase the usability of real time scheduling theory? the cheddar project. In: Reliable Software Technologies–Ada-Europe 2008, pp. 240–253. Springer (2008)
31. Sun, Y., Lipari, G.: A weak simulation relation for real-time schedulability analysis of global fixed priority scheduling using linear hybrid automata. In: Proceedings of the 22Nd International Conference on Real-Time Networks and Systems, RTNS '14, pp. 35:35–35:44. ACM, New York, NY, USA (2014). DOI 10.1145/2659787.2659814. URL http://doi.acm.org/10.1145/2659787.2659814
32. Xu, J., Parnas, D.L.: Scheduling processes with release times, deadlines, precedence and exclusion relations. Software Engineering, IEEE Transactions on **16**(3), 360–369 (1990)
33. Xu, J., Parnas, D.L.: Priority scheduling versus pre-run-time scheduling. Real-Time Systems **18**(1), 7–23 (2000)