Off-line Optimal Multiprocessor Scheduling of Dependent Periodic Tasks

Mikel Cordovilla*, Frédéric Boniol*, Eric Noulard*, Claire Pagetti*‡
*ONERA, Toulouse, France, Email: claire.pagetti@onera.fr
‡ENSEEIHT, Toulouse, France

***Abstract*—The paper focuses on the multithreaded execution of dependent periodic task sets on multiprocessor platforms. The problem consists in providing a schedule such that the tasks always satisfy their temporal and precedence constraints. Extensive work has been done in the last decades for defining efficient scheduling policies. Among them, optimal policies are particularly interesting because they are able to find a correct ordering if any exists. Unfortunately, on-line solutions with reduced vision of active tasks only cannot be optimal for asynchronous constrained deadline task sets on multiprocessor. On the contrary, off-line schedules have a complete knowledge of the task set and allow deterministic execution with a quite simple implementation.**

We propose a methodology for constructing an off-line sequencing based on the use of an exhaustive search of a repetitive pattern in an automaton of configurations. For this, we use the UPPAAL model checker. The approach is optimal since if no solution is found, then the task set is not feasible. Unfortunately, the problem is NP-complete but we obtain rather reasonable performances as illustrated in the benchmarks.

***Keywords*-Real-time, multiprocessor, off-line feasible schedule.**

I. INTRODUCTION

The evolution of real-time embedded systems leads to the increase of the number of features and the use of more powerful architectures. We consider homogeneous multicore architectures composed of $m \geq 2$ identical cores integrated on the same chip. The functional specification is given as a set of nodes (or tasks) that execute periodically and which are subject to extended precedences.

In critical embedded software, the code is as static as possible for determinism, predictable and purpose of verification. This is possible because all the characteristics are fully specified, and in particular, there is no uncertainty on the arrival times of any function. When the code is composed of several multiperiodic functions and the platform is monoprocessor, designers often choose to compute an off-line static sequencing which executes indefinitely. In the same way, we propose to define an off-line multiprocessor sequencing. The procedure defined in this paper is optimal in the sense that if it does not find a solution, then it means there is no way for sequencing and scheduling the functions.

A. Real-time scheduling

We consider periodic task set $\mathcal{S} = \{\tau_i\}_{i=1,\ldots,n}$ where the temporal behaviour of a task τ_i is defined by the four real-time attributes (T_i, C_i, O_i, D_i). T_i is the period of repetition, C_i is the worst case execution time estimation (wcet), O_i is the first arrival time and D_i is the relative deadline which is assumed to always be less or equal to the period (we say that the task is constrained deadline). The task set is called *synchronous* if all offsets are equal to 0. Where offsets are not 0 the task set is called *asynchronous*. Each repetition of the task is usually called *job* or *instance*. The *hyper-period* designates the least common multiple of all the task periods, i.e. $H = lcm_{1\leq i\leq n}(T_i)$. This model of task is a standard adaptation of the Liu and Layland model [1].

1) Reminder: Tasks execute on the shared platform and the way by which tasks access the processors is enforced by the *scheduler policy* (or strategy). A policy assigns the active tasks to the different processors using several criteria (for instance, according to the periods as Rate Monotonic policy). Typically, the scheduling criteria are transformed into the assignment of priority for each task and the m highest priority tasks access the m processors.

A policy is said *preemptive* if a task can be interrupted while executing by another task with a higher priority. Otherwise it is said *non-preemptive*. If a suspended instance may be resumed on a different processor, the policy accepts the (full) *migration*.

A scheduling is said *on-line* if execution decisions are taken at run-time, for instance when a job terminates or asks for the computing resources. An *off-line* schedule is computed statically off-line. The obtained order is stored in a table used at run-time to guide the execution.

A task set $\mathcal{S}$ is *feasible* for a given architecture if there exists a schedule that respects the temporal constraints of every task. A scheduling algorithm is *optimal* [2] with respect to an architecture if it can schedule all the feasible task sets.

We are interested in efficient preemptive policies with migration for real-time multiperiodic task set. Being optimal on multiprocessor requires to be *clairvoyant* [2] meaning that the scheduler must know a priori the future events, particularly, release times. Thus, on-line policies with only knowledge of currently active tasks (which is the case of most of the on-line policies) necessarily fail to be optimal. In the specific case of implicit deadline synchronous task sets ($O_i = 0$ and $D_i = T_i \ \forall i \leq n$), PFair [3] and its variants (e.g. PD^2 [4]) as well as LLREF [5] are known to be optimal on-line multiprocessors policies with preemptions and migrations. Indeed in that case, the verification that the task set is schedulable is quite simple: the idea is to check that the load on the processors does not exceed the processor's charge. Unfortunately, those policies are not optimal for more

general cases including synchronous constrained deadline task sets ($O_i = 0$ and $D_i \leq T_i$) for multiprocessor platforms (as shown in appendix A).

In our context, all the real-time attributes are known at design time so that they can be incorporated in the scheduler. Therefore, we focus on strategies using the complete knowledge of the task set, and off-line solutions belong to this category. The general problem of finding an optimal assignment of tasks to multiple processors is a bin-packing problem, which is NP-hard in the strong sense [6].This means that the best we can do to explore the solution space, apart from the use of heuristics, is to cut down some paths using necessary scheduling conditions.

2) Precedences constraints: Not only the tasks are defined by their real-time characteristics, but they can also be subject to *precedences*. We then speak of dependent task set. The precedences may come from data or message exchanges between tasks or may be the result of high level requirements. There are several formalisms for representing the precedences in real-time systems. In this paper, we consider *simple* and *extended* precedences [8].

Simple precedence describes a dependence between tasks with the same period. Precedence $\tau_i \rightarrow \tau_j$ expresses that τ_i must execute before τ_j. An extended precedence expresses precedence relation between instances of multiperiodic tasks.

3) Sequencing vs scheduling: there are three main interests in using off-line feasible schedules, also called sequencing. First, all the costs are assessable before run-time. In particular we know in advance the exact number of preemptions, migrations and all context switches. The scheduler is in fact a dispatcher which allocates the processors to the tasks at predefined dates and does this in a constant time.

Second, this notion of predefined instants where tasks are resumed or suspended avoids the use of semaphore or any other synchronisation mechanisms, in particular for managing the precedences. This greatly simplifies the implementation. Moreover, when dealing with precedences with on-line scheduling policies, the designer must be careful with the so-called *scheduling anomalies*. Indeed, it may that varying the execution times changes the (on-line) schedulability [11]. Off-line scheduling solves this issue because the sequencer will not start a task instance before the off-line pre-computed date, even if the previously running task did complete before its wcet.

Finally, implementing efficient and scalable schedulers is far from an easy task. The history of LINUX scheduler [9] (even if not for real-time) shows us that schedulers exhibiting $O(n)$ complexity are not scalable when the numbers of processors and tasks increase. When looking at [10], the different variants of scheduler can be complex to implement and require a complex data-structure for which a locking protocol must be carefully crafted in order to avoid contention. In multiprocessor SoC domain, authors of [12] advocate the use of predetermined schedule in order to minimize scheduler overhead and [13] even examine the use of hardware assisted schedulers. We think that, if the off-line schedule is tractable, it will bring very simple and efficient scheduler implementation for critical real-time systems.

B. Related works

Cucu and Goossens [14] have worked on the feasible interval for multiprocessor platform. In the monoprocessor case, the feasible window [15] is known to be $[0, H]$ in case of constrained deadline synchronous task set and $[0, max_i(O_i) + 2H]$ in case of constrained deadline asynchronous task set. In the multiprocessor case, Cucu and Goossens have proved the feasible window to be $[0, H]$ in case of constrained deadline synchronous system. They also note that it is possible to determine an optimal fixed priority assignement by checking all the possible priority orders in this interval.

Searching an off-line sequencing is quite standard for monoprocessor. Xu and Parnas [16], [17] have worked on *pre-run-time scheduling* which is equivalent to what we call to off-line feasible schedule. They propose an optimal scheduling method based on a *branch and bound* approach for synchronous dependent periodic task sets on mono and multiprocessor platforms with additional constraints such as mutual exclusion. Grolleau and Choquet-Geniet [18] have extended the optimal search of an off-line schedule for asynchronous dependent periodic task sets on monoprocessor platform. Their tool PENSMARTS constructs a schedule on the marking graph of a Petri net. Relaxing the synchronous hypothesis leads to the treatment of the cyclicity: when the tasks are synchronous, it is sufficient to reason on the hyper-period and unfold the tasks into a set of jobs. In asynchronous situation, it may that there is no time where all tasks terminated. In the monoprocessor case, the maximal interval is known and the authors improved the length of the search regarding the last spare time. Shepard and Gagné [19] extend the results of Xu and Parnas for multiprocessor but with no migration: it is a bin packing problem which consists in finding an adequate repartition of the tasks on the processors and then applying a monoprocessor sequencing. This approach rejects task sets that can be scheduled with migration [6].

In [20], the authors use priced timed automata for searching an optimal scheduling for a set of jobs related by precedence constraints. They apply the model checker UPPAAL [21] for the effective search. Their modelling relies on several automata and states which is not efficient.

We generalise the previous solutions since we consider asynchronous dependent periodic task sets on multiprocessor. But we are more restrictive on the task model because do not consider the mutually exclusive shared resource access.

C. Contribution

We first extend the result of [14] who showed that any schedule becomes repetitive at some point. We give a pessimistic quantitative bound. This ensures that an off-line solution can be found within this interval if it exists. Since our bound is very pessimistic, we do not search the whole interval but we rather use a model checker to find the first fixed point solution which is, by design, the first point to be

repetitive. We encode an execution as a path in an automaton of configurations. We then search among all the behaviours, a path which respects the temporal and precedence constraints and which is repetitive. This is described in section III. We also extend this approach to find a fixed priority solution, giving a possible implementation of the theoretical result of [14]. In the case of dependent task set, we avoid the scheduling anomalies [11], by assuming that the execution is not *work conservative* because the schedule will always enforce the wcet. This is a sound hypothesis for an off-line scheduler since it could be implemented as a simple run-time sequencer, executing a cyclic table of pre-computed task start dates. In the last section IV we describe several experiments and discuss on the usability of the approach. The process is shown in the figure 1. In this paper, we only describe the counter example generation from UPPAAL. We could then apply a constraint solving approach in order to improve the sequence, for instance for minimizing the number of preemption, of migrations, etc .... This is an on-going work.

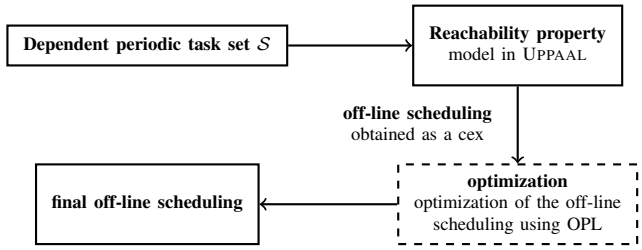


Fig. 1. Off-line scheduling construction

II. PRELIMINARY RESULTS

As said in introduction, we use a model checker to find an off-line static scheduling. For that purpose, it necessary to establish two preliminary results: (a) this problem can be considered as a search problem among a space of configurations (subsection II-A), and (b) this space configuration can be reduced to a finite space making the search problem decidable (subsection II-B).

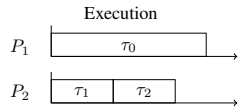
A. Problem encoding

From now on, we will represent the scheduling problem with a formal representation. For all time t, the state of the executing system is represented by $conf(t) = \langle conf(\tau_1, t), \ldots, conf(\tau_n, t)\rangle$ where for $i \in [1, n]$, $conf(\tau_i, t) = (T_i^c(t), C_i^c(t), O_i^c(t), D_i^c(t))$. $O_i^c(t) = max(O_i - t, 0)$ is the remaining time since the first awakening of the task τ_i. If $O_i^c(t) > 0$, all the other parameters are null. $O_i^c(t) = 0$ means that t is greater than the release time and the other parameters have a value. $T_i^c(t) = T_i + ((t - O_i) \ mod \ T_i)$ is the remaining time till the next activation of τ_i, $D_i^c(t) = max(0, T_i^c(t) - (T_i - D_i))$ is the remaining time till the next deadline and $C_i^c(t)$ is the remaining execution time for current instance of τ_i.

Example 1: For the following task set and a fixed priority policy on two processors (P_1, P_2), the real execution is often represented as the Gantt diagram below:

Task set

	T	C	O	D	prio
τ_0	5	5	0	5	1
τ_1	5	2	0	5	2
τ_2	5	2	0	5	3



This is equivalent to describe the sequence of configurations:

time	0	1	2	...
τ_0	$(5, 5, 0, 5)$	$(4, 4, 0, 4)$	$(3, 3, 0, 3)$	
τ_1	$(5, 2, 0, 5)$	$(4, 1, 0, 4)$	$(3, 0, 0, 3)$	
τ_2	$(5, 2, 0, 5)$	$(4, 2, 0, 4)$	$(3, 2, 0, 3)$	

At time t, when some task instance ends and a new job starts, therefore we implicitely have two configurations $(0, 0, 0, 0) \rightarrow (T_i, C_i, D_i, 0)$.

We only consider integer times since all the decisions are taken at integer dates. This is the same assumption as Guan and al. [23] who have shown that a discrete model checking is sufficient.

B. Window of study

The first lemma shows that the behaviour is repetitive once all the tasks are awake.

Lemma 1: $\forall i \in [1, n]$ and $\forall t \geq max_{i \leq n}(O_i)$, we have: $T_i^c(t + H) = T_i^c(t)$ and $D_i^c(t + H) = D_i^c(t)$ where $H = lcm_{i \leq n}(T_i)$ is the hyper-period.

Proof: This is straightforward from the definition of T_i^c. Let $t \geq max_{i \leq n}(O_i)$ and $i \in [1, n]$, we have $T_i^c(t) = T_i + ((t - O_i) \ mod \ T_i)$. Thus, $T_i^c(t + H) = T_i + ((t + H - O_i) \ mod \ T_i) = T_i^c(t)$ since $H \ mod \ T_i = 0$. Moreover $D_i^c(t) = max(0, T_i^c(t) - (T_i - D_i))$. ■

Property 1: There is a finite number of states of an executing system. It is sufficient to study at most the window $[0, max_{i \leq n}(O_i) + p \times H]$ where $p = \prod_i (C_i + 1)$ to analyse the task set feasibility.

Proof: It is simply an enumeration question. We count the number of possible states at time $t - max_{i \leq n}(O_i) \equiv 0 \ mod \ (H)$ since we know that $O_i^c(t) = 0$, T_i^c and D_i^c are constant at these times. The only parameters that can change are $C_i^c \in [0, C_i]$. Thus there are $C_i + 1$ possible states for each task τ_i and $p = \prod_i (C_i + 1)$ states for the executing system.

Therefore, if we make a schedulability analysis or an off-line computation, the algorithm must compute all the reachable states at $max_{i \leq n}(O_i) + H$ and at $max_{i \leq n}(O_i) + 2H$. If they are the same, the exploration is sufficient, otherwise, we should go till $max_{i \leq n}(O_i) + 3H$ and so on until a fixed point. Since we have counted the number of states, we know that at worst, we find a new state at each step and it would require $\prod_i (C_i + 1)$ steps to a complete exploration. ■

This ensures that the exploration with UPPAAL will always conclude. Practically, it may fail because of state space size and the fact that memory consumption is bounded.

III. OFF-LINE OPTIMAL POLICY CONSTRUCTION

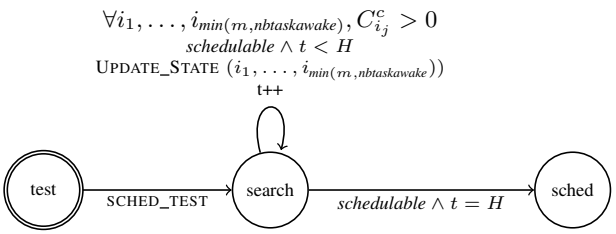
In this section we describe the exhaustive search among the behaviours for finding a valid schedule with preemption and

migration. The method is optimal since it will always construct a valid schedule if any exists. The idea is to a use a model checker that explores all the schedules which respect the real-time constraints and the precedences. Thanks to the exhaustive nature of the search, we are ensured to cross a solution if any exists. We have encoded the problem so that a schedule can be found if an automaton can reach a particular state. Thus, finding a solution, consists in providing a counter example to a reachability property.

A. Off-line optimal scheduling of independent task set

The temporal exploration is encoded in an automaton composed of three states (*test* which is initial, *search* and *sched*), four transitions and several C functions. There are two possibilities: either the task set is synchronous or it is asynchronous.

1) Synchronous task sets: This case has been treated in [16], [17]. A branch and bound method can be applied to find an off-line schedule on the hyper-period. The found schedule on the hyper-period can be used repeatedly. It is equivalent to find an order on a finite set of jobs. This can also be encoded in UPPAAL as follows:



Initially the automaton is in the *test* state. It can reach the *search* state if some necessary scheduling tests are satisfied. SCHED_TEST verifies that the load is less than the number of processors. Since UPPAAL does not manipulate float, we use the formula:

$$\Sigma_{i \leq n} C_i \times H/T_i \leq m \times H$$

We also verify the load on each deadline by the formula:

$$\forall i, \Sigma_{j \leq n, D_j \leq D_i} C_j \leq m \times D_i$$

At that point, the user can add as many necessary conditions as wanted. It will help rejecting non feasible task sets.

The *search* state describes the temporal executions. The variable t counts the time since the beginning till the hyper-period. The loop transition on *search* manages the temporal progression. It is firable only if the task set is still schedulable, i.e. if the Boolean variable *schedulable=true*. In that case, the function UPDATE_STATE updates the value of the task parameters as follows:

- $T^c_i(t+1) := T^c_i(t) - 1 + T_i \times \delta_{T^c_i(t)==1}$ where $\delta_c = 1$ iff the condition c is true and 0 otherwise,
- $D^c_i(t+1) := max(0, T^c_i(t) - (T_i - D_i))$

The loop is a non deterministic transition. We consider any combination of k active tasks where k is the minimum between the number of processors m and the number of active tasks denoted by *nbtaskawake*. These k tasks are assumed to access the processors at time t. Thus,

- for all $i \in \{i_1, \ldots, i_k\}$, $C^c_i(t+1) := C^c_i(t) - 1$
- otherwise $C^c_i(t+1) := C^c_i(t)$
- we must have $C^c_i(t+1) \leq D^c_i(t+1)$, otherwise the task will not be correctly scheduled. If this condition does not hold, *schedulable:=false*,
- if $T^c_i(t) = 1$, a new job starts. If $C^c_i(t+1) > 0$, *schedulable:=false* otherwise $C^c_i(t+1) := C_i$.

At each step, the time is increased by one $t := t + 1$ and the new scheduling states are developed. The symmetric solutions are automatically detected and reduced by UPPAAL.

The question asked to the model checker is whether there is a path reaching the state *sched*:

$$E <> sched$$

Not that if the task set has implicit deadline and is feasible, it is sufficient to simulate a PFair or an LLREF simulation. This solution has a linear complexity.

2) Asynchronous task sets: The asynchronous case must deal with the cyclicity. We use an additional variable which stores the configurations at precise times: at $t = max_{i \leq n}(O_i)\ mod\ (H)$. Thanks to lemma 1, we know that $O_i = 0$, T^c_i and D^c_i are constants at that time. So it is sufficient to save the values of the remaining execution times $\langle C^c_0(t), \ldots, C^c_n(t) \rangle$ The automaton is represented in the figure 2.

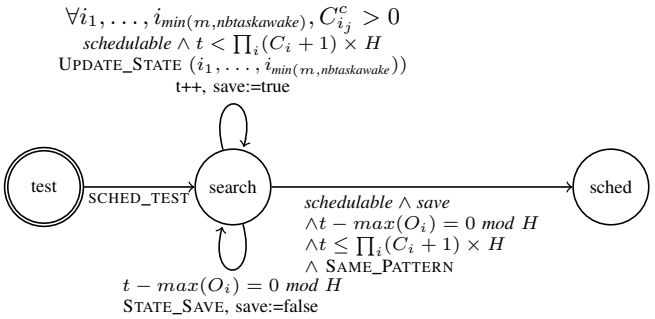


Fig. 2. Exploration in UPPAAL

It is almost the same as the one of the synchronous case except:

- the time t is bounded by the upper approximation $\prod_i (C_i + 1)$,
- the loop transition STATE_SAVE may or not be fired at time $t = max_{i \leq n}(O_i)\ mod\ (H)$. If it is fired, the current configuration is saved in the special variable and the transition SAME_PATTERN cannot be fired till the variable *save* has been set to true again, that is after an update,
- the condition on the transition SAME_PATTERN states that the task set must be schedulable, the backup has not been modified in the same instant and the previous

backup is exactly equal to the current configuration, meaning that we have found a repetitive schedule,

- the transition UPDATE_STATE manages the release times
  - $\forall i \leq n$, such that $O_i^c(t) > 0$, $O_i^c(t+1) := O_i^c(t) - 1$
  - $\forall i \leq n$, such that $O_i^c(t) = 0$,
    - $T_i^c(t+1) := T_i^c(t) - 1 + T_i \times \delta_{T_i^c(t)==1}$
    - $D_i^c(t+1) := max(0, T_i^c(t) - (T_i - D_i))$

The question asked to the model checker is whether there is a path reaching the state *sched* in less than $max_i(O_i) + \prod_i (C_i + 1) \times H$ steps. The logical formula is the same as in the synchronous case.

Example 2: Let us consider the following task set.

τ_i	T_i	C_i	O_i	D_i
τ_0	5	5	1	5
τ_1	5	2	0	5
τ_2	5	2	0	5

It could be scheduled by a fixed priority, it is a variation of example 1 page 3 with a release time for task τ_0. A path in the automaton is the following:

time	0	1	2	3	4	5	6
τ_0	$(5,5,1,5)$	$(5,5,0,5)$	$(4,4,0,4)$	$(3,3,0,3)$	$(2,2,0,2)$	$(1,1,0,1)$	$(0,0,0,0)$ $(5,5,0,5)$
τ_1	$(5,2,0,5)$	$(4,1,0,4)$	$(3,0,0,3)$	$(2,0,0,2)$	$(1,0,0,1)$	$(0,0,0,0)$ $(5,2,0,5)$	$(4,1,0,4)$
τ_2	$(5,2,0,5)$	$(4,2,0,4)$	$(3,2,0,3)$	$(2,1,0,2)$	$(1,0,0,1)$	$(0,0,0,0)$ $(5,5,0,5)$	$(4,2,0,4)$
saved conf		$(5,1,2)$					

We note that at time $t = 6$, the state *sched* is reachable. We thus obtain an off-line sequencing of length $1 + H$ which repetitive pattern if of length H.

B. Off-line optimal fixed priority scheduling

Cucu and Goossens [14] have proposed the exploration of the fixed priority assignments to define an optimal fixed priority schedule. This can be implemented in UPPAAL with our configuration encoding. We simply need to do some modifications:

- we use a variable *list_prio* which imposes the order of the tasks,
- we encode a schedulability analysis.

Again, we distinguish the synchronous and the asynchronous cases. In the synchronous case, as shown in [14], the feasibility is $[0, H]$. Thus, the automaton is shown below.

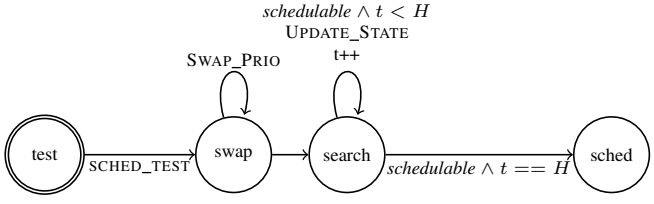


Fig. 3. Valid fixed priority assignment for synchronous task set

Initially, the automaton is in the *test* state. The variable *list_prio* is initialised to $[1, \ldots, n]$. The necessary scheduling test are applied and may modify the Boolean variable *schedulable* if not satisfied. The loop on the *swap* states allows to permute the values of *list_prio*[i] and *list_prio*[j] to explore every scheduling possibilities. Once the *list_prio* is fixed, the deterministic execution of the tasks is made until the time H. If the Boolean variable *schedulable* is still true, then the task set is schedulable for the priority assignment given by *list_prio*.

Thus, for the model checker it is again a reachability problem.

$$E <> sched$$

In that case, what is important in the counter example is the value of *list_prio*.

In the asynchronous case, either we can apply the saving or make the simulation till the feasibility window proposed in the paper. The figure 4 shows the schedulability test using the saving. The automaton is almost the same as in case synchronous and the reachability problem is again the same.

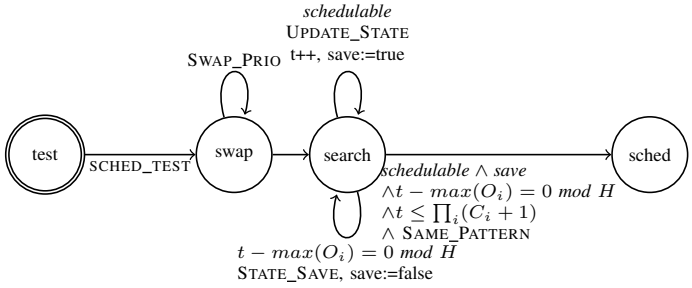


Fig. 4. Execution in UPPAAL

Example 3: For the task set of example 2, the model checker quickly finds the solution *list_prio*$= [1, 2, 3]$.

C. Off-line optimal scheduling of dependent task set

The previous sections propose automatic solutions for finding a sequencing of an independent task set or a fixed priority assignment. We extend the approaches for dependent task sets. In the following, we consider extended precedences.

1) Precedences definition:

Definition 1 (Extended precedence): An extended precedence [8] between τ_i and τ_j is defined by a finite set of pairs of integers $M_{i,j} \subseteq [0, p_{i,j}/T_i] \times [0, p_{i,j}/T_j]\mathbb{N}^2$ with $p_{i,j} = lcm(T_i, T_j)$ is the least common multiple of T_i and T_j. The set $M_{i,j}$ imposes all the job precedences $\tau_i[n] \rightarrow \tau_j[n']$ where:

$$\exists k \in \mathbb{N}, \exists (m, m') \in M_{i,j}(n, n') = (m, m') + \left(k\frac{p_{i,j}}{T_i}, k\frac{p_{i,j}}{T_j}\right)$$

An extended precedence is denoted by $\tau_i \xrightarrow{M_{i,j}} \tau_j$.
A simple precedence $\tau_i \rightarrow \tau_j$ is a particular case of extended precedences $\tau_i \xrightarrow{M_{i,j}} \tau_j$ with $M_{i,j} = \{(0, 0)\}$.

Example 4: We describe several patterns of extended precedences $\tau_i \xrightarrow{M} \tau_j$ hereafter in figure 5.

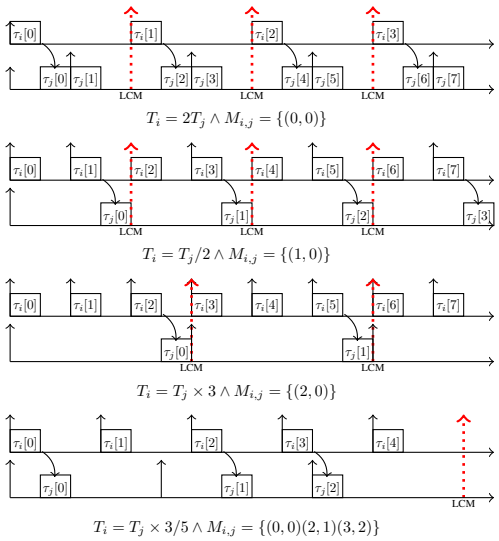


Fig. 5. Example of extended precedences

A task instance can be constrained by several precedences but the complete graph of precedences must not contain any cycle, i.e. we cannot have $\tau_i[k_i] \rightarrow \tau_j[k_j] \rightarrow \cdots \rightarrow \tau_i[k_i]$.

Let us introduce some notations. *preds(τ_i)* is composed of all the predecessors of τ_i: $preds(\tau_j) = \{\tau_i | (\tau_i, \tau_j) \in \rightarrow\}$. In the same way, *succs(τ_j)* is composed of all the successors of τ_j: $succs(\tau_i) = \{\tau_j | (\tau_i, \tau_j) \in \rightarrow\}$.

2) Precedences encoding: A precedence constrain $\tau_i \xrightarrow{M_{i,j}} \tau_j$ is formally represented by $\langle \tau_i, \tau_j, M_{i,j} \rangle$. To store these data we use a two-dimensional table. A row contains all necessary data for describing all the precedences of a task. That is, the row i is composed by $\langle \tau_j, M_{i,j} \rangle \forall j \in preds(\tau_i)$. This table represents at any the time the current state of the precedences. The precedence constraints evolution is performed as follow:

- if the instance $\tau_i[a]$ finishes its execution at time t, for all the other jobs awaiting for it $\forall j, \tau_i[a] \in preds[\tau_j](t-1)$, they are free of this precedence $\tau_i[a] \notin preds[\tau_j](t)$;
- the current instance of τ_i cannot execute at time t if $preds[\tau_i](t) \neq \emptyset$;
- when a task wakes up, the precedence constraints must be refreshed.

We can add necessary scheduling test for the precedences. First, if a predecessor starts after the deadline of a successor then the task set is not schedulable

$$\forall \tau_i \xrightarrow{M_{i,j}} \tau_j, \forall (n, n') \in M_{i,j},$$
$$O_i + (n-1)T_i + C_i + C_j \leq O_j + (n'-1)T_j + D_j$$

Second, it may that a precedence is not constraining. We remove these constraints. For a precedence $\forall \tau_i \xrightarrow{M_{i,j}} \tau_j$, we encode $M'_{i,j}$ where initially, $M'_{i,j} = M_{i,j}$. Then, we simplify the set as follows $\forall (n, n') \in M_{i,j}$:

$$O_i + (n-1)T_i + D_i \leq O_j + (n'-1)T_j$$
$$\implies M'_{i,j} = M'_{i,j} \setminus \{(n, n')\}$$

Example 5: Let us consider the following task set with simple precedences.

	T	C	O	D
τ_0	6	1	0	6
τ_1	6	2	0	6
τ_2	6	2	1	6
τ_3	6	1	1	6

$\tau_0 \rightarrow \tau_1$
$\tau_0 \rightarrow \tau_2$
$\tau_0 \rightarrow \tau_3$
$\tau_1 \rightarrow \tau_3$
$\tau_2 \rightarrow \tau_3$

The model checker provides the fixed priority assignment *list_prio*= $[0, 1, 2, 3, 4]$ and the counter example described in the figure 6.

IV. Performance Evaluation

In order to make the approach user friendly, we have developed a converter that transforms a simple task set description into an equivalent UPPAAL model. It has been incorporated in the tool box developed at ONERA for analysing task sets and generating code.

A. Automatic conversion

The prototype `converter` which has been described in [22] has been extended for taking into account the precedences. Example 5 can be described by:

```
# Example1 task set
# Task "Name" T C D O
Task "Tau0" 6 1 6 0
Task "Tau1" 6 2 6 0
Task "Tau2" 6 2 6 1
Task "Tau3" 6 1 6 1
# Precedence
# Dependency "SuccessorTask" "PredecessorTask"
Dependency "Tau1" "Tau0"
Dependency "Tau2" "Tau0"
Dependency "Tau3" "Tau0"
Dependency "Tau3" "Tau1"
Dependency "Tau3" "Tau2"
```

The new task file format can be used to specify extended dependencies too.

Example 6: Let us consider the following task set with simple extended precedences.

	T	C	O	D
τ_0	5	2	0	5
τ_1	10	3	0	10
τ_2	20	7	0	20

$\tau_0 \xrightarrow{(0,0)(3,1)} \tau_1$
$\tau_0 \xrightarrow{(2,0)} \tau_2$

The task description file used for the example 6 would be:

```
# Example2 task set
# Task "Name" T C D O
Task "Tau0"  5 2  5 0
Task "Tau1" 10 3 10 0
Task "Tau2" 20 7 20 0
# Precedence
# Dependency "SuccessorTask" "PredecessorTask"
Dependency "Tau1" "Tau0" 0 0 3 1
Dependency "Tau2" "Tau0" 2 0
```

B. Task set generation

We have also included an automatic task set generator with varying parameters such as the number of tasks, the geometric progression of period (r), the period limit (l) and the WCET ratio (w). The real-time attributes are generated by the rules:

time	0	1	2	3	4	5	6	7
τ_0	(6,1,0,6)	(5,0,0,5)	(4,0,0,4)	(3,0,0,3)	(2,0,0,2)	(1,0,0,1)	(0,0,0,0) (6,1,0,6)	(5,0,0,5)
prec(τ_0)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
τ_1	(6,2,0,6)	(5,2,0,5)	(4,1,0,4)	(3,0,0,3)	(2,0,0,2)	(1,0,0,1)	(0,0,0,0) (6,2,0,6)	(5,2,0,5)
prec(τ_1)	$\{0\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$
τ_2	(6,2,1,6)	(6,2,0,6)	(5,1,0,5)	(4,0,0,4)	(3,0,0,3)	(2,0,0,2)	(1,0,0,1)	(0,0,0,0) (6,2,0,6)
prec(τ_2)	$\{0\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
τ_3	(6,1,1,6)	(6,1,0,6)	(5,1,0,5)	(4,1,0,4)	(3,0,0,3)	(2,0,0,2)	(1,0,0,1)	(0,0,0,0) (6,1,0,6)
prec(τ_3)	$\{0,1,2\}$	$\{1,2\}$	$\{1,2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{1,2\}$

Fig. 6. Counter example of a fixed priority assignment for dependent task set

$$O = rand() \mod D \quad (1)$$

$$T = \begin{cases} l * (1 + (rand() \mod r)), \text{ if } r > 0 \\ 1 + (rand() \mod l), \text{ otherwise} \end{cases} \quad (2)$$

$$D = T \quad (3)$$

$$C = 1 + ((rand() \mod (D - r))/w) \quad (4)$$

At the case of simple precedences the generator produces task sets such that:

$$Predecessor = rand() \mod nbTask \quad (5)$$

$$Successor = rand() \mod nbTask \quad (6)$$

However, for extended precedences the task set can be multi-periodic, and the generator produces tasks sets such that:

$$Predecessor = rand() \mod nbTask \quad (7)$$

$$Successor = rand() \mod nbTask \quad (8)$$

$$words = word_1 \cdots word_k, k \leq \min(\frac{H}{T_{Succ}}, \frac{H}{T_{Pred}}) \quad (9)$$

$$word_i = (0 \leq \frac{H}{T_{Pred}}, 0 \leq \frac{H}{T_{Succ}}) \quad (10)$$

Where $rand()$ gives 32 bit integer random numbers (our current Linux implementation uses `/dev/urandom`).

C. Experiments

The experiments were run on a 64 bits Debian Linux host (Intel Xeon X5472 @3.00GHz) with 16GB of DDR2 memory and UPPAAL v4.0.11 binary distribution. We used the task set generator in order to evaluate our off-line scheduler producer with more than 10000 task sets For each task set we used our UPPAAL model generator tool in order to generates the corresponding UPPAAL model and then launch the model checker. Theoretically, the model checker can either:

(i) terminate with a counter example, which means we have found a valid schedule,
(ii) terminate after exhausting the whole space state, which means that the task set is not schedulable,
(iii) aborts without conclusion, which means the space state dimension to be kept in memory is too large and we run out-of-memory.

Practically, the current version of our generator do generate almost only schedulable task set, we will work on this issue in the future in order to generate more various task sets. As already noticed in [22] the two determinant factors in terms of complexity are the number of tasks and H. This can be observed in figures 7 and 8. Furthermore we can see that the asynchronous case costs more. This is a clear confirmation of the theoretical result of the repetition window size for asynchronous case given in section II.

The results are promising because despite of the expected exponential growth in time we can find solution for task set consisting of up to 50 tasks in the asynchronous case (see Fig. 8) and more than 90 tasks in the synchronous case (see Fig. 8).

Figure 9 shows the proportion of conclusive search with respect to the CPU load. Abort cases mean that the model checker did run out ot memory before the end of the search. This figure shows that we can find solutions up to 70% of global CPU load. This is an interesting result for industrial users who may want to include margin in their future multiprocessor architecture design. In the monoprocessor case industrial do include margin, but they do have optimal scheduling test, on the contrary in the multiprocessor they do not but we can say that as of today it is difficult to find a valid schedule with more than 70% global load.

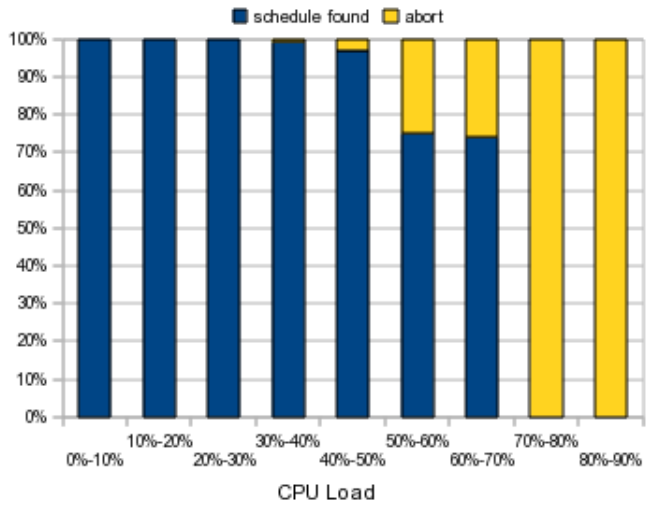


Fig. 9. Rate of schedule found vs CPUs load

V. CONCLUSION

We have studied an automatic manner for providing an off-line schedule for asynchronous dependent periodic task

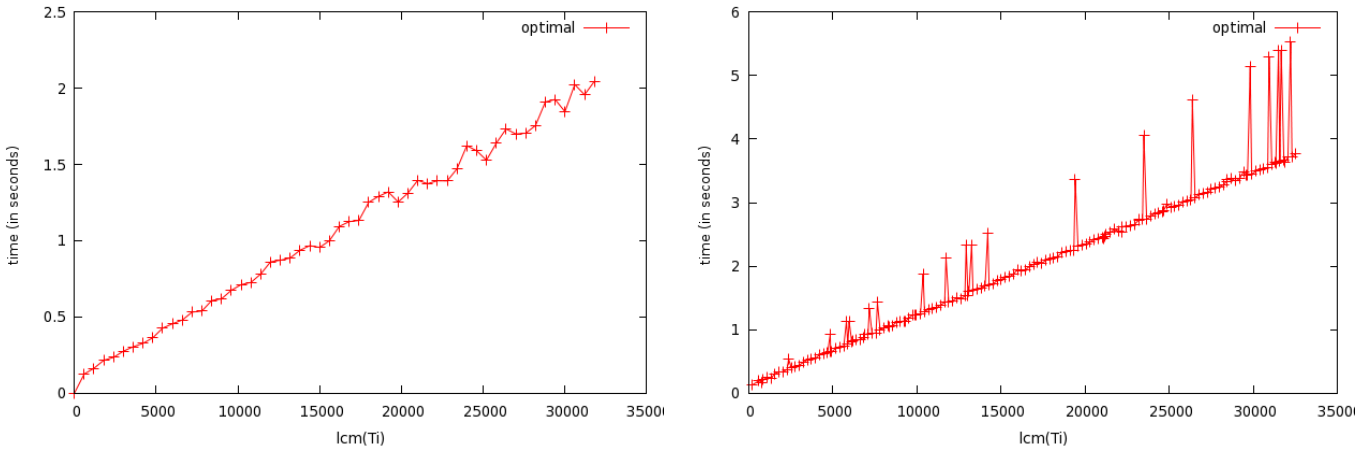


Fig. 7. Time vs H (left = synchronous / right = asynchronous)

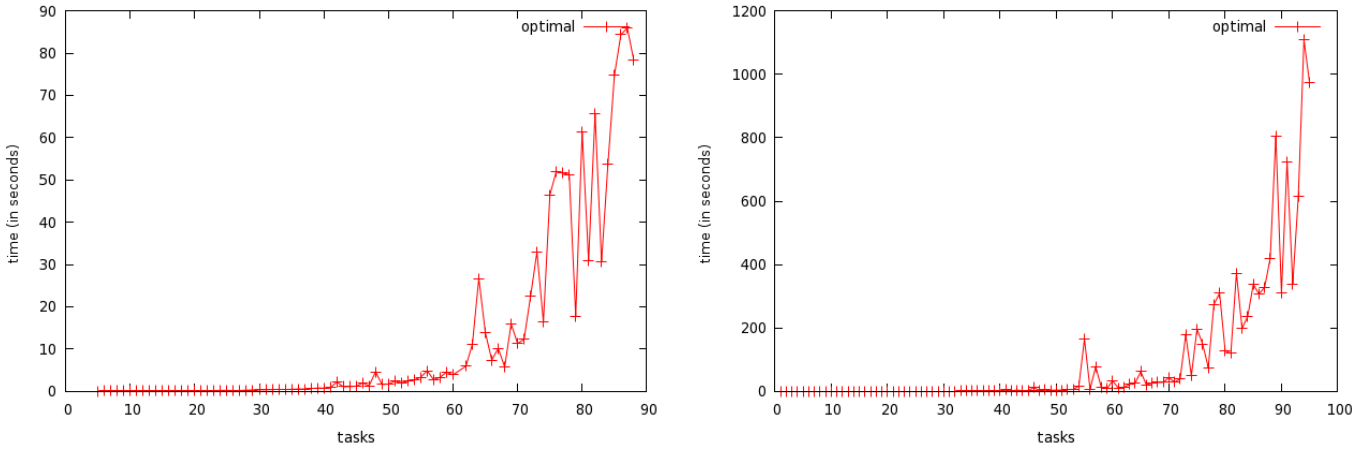


Fig. 8. Time vs nb Tasks (left = synchronous / right = asynchronous)

sets using a model checker. The schedule is the counter example computed by the tool. The next step will be to use a constraint solver for improving the generated sequence. The model checker gives a tractable bound for the solver compared to our theoritical bound. The solver will balance the sequence to minimize criteria such as the number of preemption and migration, the length of the schedule.

REFERENCES

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] K. S. Hong and J. Y. T. Leung, "On-line scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 41, pp. 1326–1331, 1988.

[3] S. K. Baruah, "Fairness in periodic real-time scheduling," *Real-Time Systems Symposium, IEEE International*, vol. 0, p. 200, 1995.

[4] J. H. Anderson and A. Srinivasan, "Mixed pfair/erfair scheduling of asynchronous periodic tasks," *J. Comput. Syst. Sci.*, vol. 68, no. 1, pp. 157–204, 2000.

[5] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," *Real-Time Systems Symposium, IEEE International*, vol. 0, pp. 101–110, 2006.

[6] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[7] E. Grolleau, "Modélisation précise des applications temps réel en vue de leur validation temporelle," Ph.D. dissertation, Habilitation à Diriger les Recherches, ENSMA, 2009.

[8] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," *In 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, April 2010.

[9] M. T. Jones, "Inside the linux 2.6 completely fair scheduler," December 2009, http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/.

[10] B. Brandenburg and J. Anderson, "On the implementation of global real-time schedulers," in *Proceedings of the 30th IEEE Real-Time Systems Symposium*, December 2009, pp. 214–224.

[11] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, pp. 416–429, 1969.

[12] Y. Cho, S. Yoo, K. Choi, N.-E. Zergainoh, and A. A. Jerraya, "Scheduler implementation in mp soc design," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '05. New York, NY, USA: ACM, 2005, pp. 151–156. [Online]. Available: http://doi.acm.org/10.1145/1120725.1120793

[13] M. Vetromille, L. Ost, C. Marcon, C. Reif, and F. Hessel, "Rtos scheduler implementation in hardware and software for real time applications," in *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on*, 2006, pp. 163 –168.

[14] L. Cucu and J. Goossens, "Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems," in *DATE'07:*

Proceedings of the conference on Design, automation and test in Europe. San Jose, CA, USA: EDA Consortium, 2007, pp. 1635–1640.
[15] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, *Scheduling in real-time systems*. John Wiley & Sons, October 2002.
[16] J. Xu and D. Parnas, "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 360–369, March 1990. [Online]. Available: http://portal.acm.org/citation.cfm?id=78266.78285
[17] J. Xu and D. L. Parnas, "On satisfying timing constraints in hard-real-time systems," *IEEE Transaction Software Engineering*, vol. 19, pp. 70–84, January 1993.
[18] E. Grolleau and A. Choquet-Geniet, "Off-line computation of real-time schedules by means of petri nets," in *Workshop On Discrete Event Systems, WODES2000*, ser. Discrete Event Systems: Analysis and Control. Ghent, Belgium: Kluwer Academic Publishers, 2000, pp. 309–316. [Online]. Available: http://www.lisi.ensma.fr/ftp/pub/documents/papers/2000/copyrighted/2000-WODES2000-Grolleau.pdf
[19] T. Shepard and J. A. M. Gagné, "A pre-run-time scheduling algorithm for hard real-time systems," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 669–677, July 1991. [Online]. Available: http://dx.doi.org/10.1109/32.83903
[20] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Optimal scheduling using priced timed automata," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, pp. 34–40, March 2005. [Online]. Available: http://doi.acm.org/10.1145/1059816.1059823
[21] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, ser. LNCS, M. Bernardo and F. Corradini, Eds., no. 3185. Springer–Verlag, September 2004, pp. 200–236.
[22] M. Cordovilla, F. Boniol, E. Noulard, and C. Pagetti, "Multiprocessor schedulability analyser," in *SAC*, S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, Eds. ACM, 2011.
[23] N. Guan, Z. Gu, M. Lv, Q. Deng, and G. Yu, "Schedulability analysis of global fixed-priority or edf multiprocessor scheduling with symbolic model-checking," in *ISORC*. IEEE Computer Society, 2008, pp. 556–560.

APPENDIX

In the case of monoprocessor, off-line solutions are not better than on-line policies in term of optimality since EDF and LLF are optimal. In the case of multiprocessor, off-line solutions are known to be better [2] since an optimal strategy must be clairvoyant. Even in the case of synchronous task set, we do not currently know an optimal on-line policy.

To illustrate this point, consider PFair [3] and LLREF [5]. These two scheduling policies are the two currently known on-line policies for multiprocessor platforms. The aim of this appendix is to exhibit two counter examples that prove they are not optimal for synchronous deadline constrained task sets ($O_i = 0$).

A. PFair execution

Let us consider the task set composed of a unique task $\tau = (C = 2, D = 2, T = 20)$ which is schedulable.

Reminder. An execution is fair if $\forall t, \forall \tau, lag(\tau, t) \in [-1, 1]$ where

$$\begin{cases} lag(\tau,t) = WT(\tau,t) - sched(\tau,t) \\ with\ WT(\tau,t) = t \times C/T \end{cases}$$

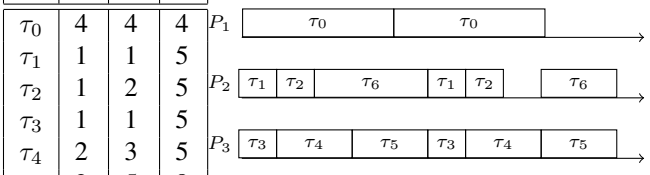
It means that the current execution time, $sched(\tau, t)$, is close to the ideal scheduling represented by *WT*.

Application on the case study. A feasible schedule will terminate the task at $t = 2$. Thus, $lag(\tau, 2) = WT(\tau, 2) - sched(\tau, 2) = 4/20 - 2 = -9/5 < -1$. This entails that any feasible scheduling will be unfair.

B. LLREF execution

LLREF (Largest Local Remaining Execution First) has been introduced by Cho and al. [5]. Let us consider the task set $\mathcal{S}$ which is schedulable using a fixed priority policy. The simulation shows the beginning of the execution. We use our schedulability analyser [22] to verify the schedulability.

$\mathcal{S}$	C	D	T
τ_0	4	4	4
τ_1	1	1	5
τ_2	1	2	5
τ_3	1	1	5
τ_4	2	3	5
τ_5	2	5	8
τ_6	3	7	8



Reminder. In LLREF, the execution is divided in time slots whose length depends on the closest next awakening or next deadline of a job. At instant t, let us denote the next time $ND(t)$, which is equal to the closest next awakening or next deadline. In order to assign the priority, two parameters are computed:

1) l_τ represents the *local remaining execution time* of τ in the interval $[t, ND(t)]$. It is a fair execution to be done:

$$l_\tau(t) = C_\tau(t) \times \frac{ND(t) - t}{D_\tau - t} \tag{11}$$

2) L_τ is the *local laxity* of τ in the interval $[t, ND(t)]$:

$$L_\tau(t) = ND(t) - t - l_\tau(t) \tag{12}$$

The policy works as follows:

1) if the laxity is zero, that is $L_\tau(t) = 0$, the remaining time till $ND(t)$ is necessary to complete the execution of τ. Thus, the execution is urgent and the task is assigned the highest priority;
2) for the set of tasks which laxity is not null, the highest priority is given to those with the highest fair execution l_τ.

Extension to constrained deadline task set. Since LLREF has been defined for implicit deadline tasks, it is not clearly defined in the original paper for constrained deadline tasks if we have to consider D or T in the formulas. Let us imagine that we keep T:

$$l_\tau(t) = C_\tau(t) \times \frac{ND(t) - t}{T_\tau - t} \tag{13}$$

Let us apply LLREF to the task set $\mathcal{S}$. The next awakening of a task is at $t = 4$. Thus, $ND(0) = 4$. Let us compute the highest priority tasks with equation 13.

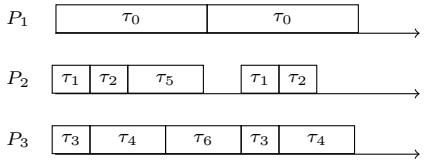
ND(0)	$l_{\tau_0}(0)$	$l_{\tau_i}(0)$ i = 1..3	$l_{\tau_4}(0)$	$l_{\tau_5}(0)$	$l_{\tau_6}(0)$
4	4 $L_{\tau_0}(0) = 0$	4/5	8/5	1	3/2

According to the LLREF policy, τ_0, τ_4 and τ_6 are elected. Thus both tasks τ_1 and τ_3 will miss their deadline. This is the reason why we consider the equation (11).

Application on the case study. Let us describe the behaviour of the task set $\mathcal{S}$ with LLREF. The next awakening of a task is at $t = 4$. Thus, $ND(0) = 4$. We compute the values of the parameters in Fig.10 and draw the execution below.

t	0	1	2	3	4	5	6	7
ND(t)	4	4	4	4	5	8	8	8
$l_{\tau_0}(t)$	4	3	2	1	1	3	2	1
$L_{\tau_0}(t)$	0	0	0	0	0	0	0	0
$l_{\tau_1}(t)$	4	-	-	-	-	3	-	-
$L_{\tau_1}(t)$	0	-	-	-	-	0	-	-
$l_{\tau_2}(t)$	2	3	-	-	-	3/2	2	-
$L_{\tau_2}(t)$	2	0	-	-	-	3/2	0	-
$l_{\tau_3}(t)$	4	-	-	-	-	3	-	-
$L_{\tau_3}(t)$	0	-	-	-	-	0	-	-
$l_{\tau_4}(t)$	8/3	3	2	-	-	2	2	1
$L_{\tau_4}(t)$	4/3	0	0	-	-	1	0	0
$l_{\tau_5}(t)$	8/5	3/2	4/3	1/2	-	-	-	-
$L_{\tau_5}(t)$	12/5	3/2	2/3	1/2	-	-	-	-
$l_{\tau_6}(t)$	12/7	3/2	6/5	3/4	2/3	3/2	2	pb
$L_{\tau_6}(t)$	16/7	3/2	4/5	1/4	1/3	3/2	0	

Fig. 10. LLREF parameters



At time $t = 6$, 4 tasks are urgent for 3 processors. Necessarily, one of them (in the execution it is τ_6) will miss its deadline. The problem comes from time $t = 2$, where τ_5 gets a higher priority than τ_6.