# Optimal Scheduling of Urgent Preemptive Tasks

Ştefan Andrei
Department of Computer Science
Lamar University
Beaumont, TX, USA
E-mail: sandrei@cs.lamar.edu

Albert Cheng
Department of Computer Science
University of Houston
Houston, USA
E-mail: cheng@cs.uh.edu

Martin Rinard
Department of Electrical Engineering & Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
E-mail: rinard@lcs.mit.edu

Lawrence Osborne
Department of Computer Science
Lamar University
Beaumont, TX, USA
E-mail: ljosborne@my.lamar.edu

***Abstract*—Tasks' scheduling has always been a central problem in the embedded real-time systems community. As in general the scheduling problem is $\mathcal{NP}$-hard, researchers have been looking for efficient heuristics to solve the scheduling problem in polynomial time. One of the most important scheduling strategies is the Earliest Deadline First (EDF). It is known that EDF is optimal for uniprocessor platforms for many cases, such as: non-preemptive synchronous tasks (i.e., all tasks have the same starting time and cannot be interrupted), and preemptive asynchronous tasks (i.e., the tasks may be interrupted and may have arbitrary starting time). However, Mok showed that EDF is not optimal in multiprocessor platforms. In fact, for the multiprocessor platforms, the scheduling problem is $\mathcal{NP}$-complete in most of the cases where the corresponding scheduling problem can be solved by a polynomial-time algorithm for uniprocessor platforms. Coffman and Graham identified a class of tasks for which the scheduling problem can be solved by a polynomial-time algorithm, that is, two-processor platform, no resources, arbitrary partial order relations, and every task is non-preemptive and has a unit computation time.**

**Our paper introduces a new non-trivial and practical subclass of tasks, called urgent tasks. Briefly, a task is urgent if it is executed right after it is ready or it can only wait one unit time after it is ready. Practical examples of embedded real-time systems dealing with urgent tasks are all modern building alarm systems, as these include urgent tasks such as 'checking for intruders', 'sending a warning signal to the security office', 'informing the building's owner about a potential intrusion', and so on. By using propositional logic, we prove a new result in schedulability theory, namely that the scheduling problem for asynchronous and preemptive urgent tasks can be solved in polynomial time.**

***Keywords*-optimal scheduling; urgent task; polynomial-time algorithm**

Dr. Albert Cheng was supported in part by the National Science Foundation under Award No. 0720856.

## I. Introduction

Scheduling has a significant impact of our daily life, starting from logistics planning, workflow systems, space mission planning, entertainment, medical systems, and so on. Tasks' scheduling has always been a central problem in the embedded real-time systems community. As in general the scheduling problem is $\mathcal{NP}$-hard, researchers have been looking for efficient heuristics to solve the scheduling problem in polynomial time. There exist many models to define a task. In this paper, we consider that a task is characterized by three parameters: $s$ is called the *starting time* (also known as the *release time*), $c$ is called the *computation time* (also known as the *worst-case execution time*), and $d$ is called the *deadline*. For simplicity, we consider the tasks to be single-instance, hence there is no need to consider the tasks' period. Thus, the notions of task, task instance and job are equivalent and can be interchangeable used. In fact, the results and examples from this paper can be easily extended to periodic or sporadic tasks. Without loss of generality, we assume that $s$, $c$, and $d$, are non-negative integers, although a task may have rational values for some parameters when needed. Using these notations, a task $T$ is denoted as a triplet $(s, c, d)$, and it means that $T$ can be executed after time $s$ completing a total of $c$ time units by the deadline $d$. Given a task set $\mathcal{T} = \{T_1, ..., T_k\}$, then $\mathcal{T}$ is called *schedulable* by a scheduling algorithm SA if SA ensures that the timing constraints of all tasks in $\mathcal{T}$ are met. Algorithm SA is called *optimal* if whenever SA cannot find a schedule, then no other scheduling algorithm can [8].

Stankovic, Spuri, Di Natale, and Butazzo investigated the boundary between polynomial and NP-hard scheduling

problems [29]. There are only few subclasses of the general scheduling problem that have polynomial-time complexity optimal algorithms. Dertouzos showed that the Earliest Deadline First (EDF) algorithm has polynomial complexity and can solve the uniprocessor preemptive scheduling problem [10]. Mok discovered another optimal algorithm with polynomial complexity for the same subclass, that is, the Least Laxity First (LLF) algorithm [24]. Another polynomial algorithm was found by Lawler in 1983 for non-preemptive unit computation time tasks with arbitrary start time [18]. However, according to Graham, Lawler, Lenstra and Kan, when dealing with non-preemptive and non-unit computation time tasks, the scheduling problem becomes $\mathcal{NP}$-hard.

Despite the fact that EDF is an optimal method for uniprocessor platform, EDF is not optimal for multiprocessor platforms. Mok showed that for multiprocessor platforms, the scheduling problem is $\mathcal{NP}$-complete in most of the cases where the corresponding scheduling problem can be solved by a polynomial-time algorithm for the uniprocessor platforms [24]. Coffman and Graham identified a class of tasks for which the scheduling problem can be solved by a polynomial-time algorithm, that is, two-processor platform, no resources, arbitrary partial order relations, and every task is non-preemptive and has a unit computation time [9].

Anderson and Srinivasan discovered the Pfair scheduling technique where each task is broken into quantum-length subtasks, each of which must execute within a "window" of time slots [1]. ERfair is a variant of Pfair scheduling in which subtasks within the same job are allowed to execute before the Pfair window. The authors proved that the Pfair and ERfair are optimal scheduling techniques for intra-sporadic tasks on uniprocessor and two-processor platforms. More recently, Srinivasan and Anderson [28] showed that a simplified variant of the Pfair, called PD$^2$, is also optimal for scheduling "rate-based" tasks whose processing steps may be highly jittered. One of the differences between their techniques and ours is that their technique is a rate-based scheduling technique and our technique is based on a conversion to a special subset of propositional formulas. For simplicity in expressing the scheduling algorithm and corresponding proofs, our technique performs for each task an internal conversion to unit computation time sub-tasks. However, this conversion does not require any tight synchronization as in Anderson and Srinivasan's work [1] and is transparent to the method itself. In fact, we break tasks only in theory since we glue the corresponding sub-tasks back to contiguous entities whenever possible during the final execution assignment (details in Section II).

Our paper introduces a new non-trivial and practical subclass of asynchronous tasks, for which the scheduling problem can be solved in polynomial time. Briefly, given a task $T = (s, c, d)$, we say that $T$ is *urgent* if $s + c \leq d \leq s+c+1$. Practical examples of embedded real-time systems dealing with urgent tasks are all modern building alarm systems, as these include urgent tasks such as 'checking for intruders', 'sending a warning signal to the security office', 'informing the building's owner about a potential intrusion', and so on. Let us consider the following two task sets as running examples in our paper.

*Example 1.1:* By abstracting the previous alarm system, we can consider the preemptive task set $\mathcal{T}_1 = \{T_1, T_2, T_3\}$, where $T_1 = (0, 1, 1)$, $T_2 = (0, 1, 2)$, $T_3 = (0, 3, 3.5)$. Clearly, $\mathcal{T}_1$ contains only urgent tasks. In fact, the task set $\mathcal{T}_1$ is an adaptation of an example used by Mok to demonstrate the non-optimality of EDF scheduling for the multiprocessor platforms [24]. We consider a two-processor platform rather than a uniprocessor one as the above task set is not feasible if only one processor is used for scheduling. Obviously, $\mathcal{T}_1$ is not EDF-schedulable on a two-processor platform because $T_1$ will be assigned to the first processor, $T_2$ will be assigned to the second processor, hence $T_3$ will miss its deadline. However, we show in Section III that our method will find actually that these urgent tasks can be executed as follows: first $T_1$ and then $T_2$ on the first processor, and at the same time $T_3$ on the second processor. ∎

*Example 1.2:* Let us consider a second example taken from [6], that is, $\mathcal{T}_2 = \{T_1, T_2, T_3\}$ a preemptive task set given $T_1 = (0, 2, 3)$, $T_2 = (0, 2, 3)$, and $T_3 = (0, 2, 3)$. Carpenter et al. [6] showed that $\mathcal{T}_2$ is schedulable only using a fully dynamic and unrestricted migration scheduling algorithm. All the other combinations of priority and migration degrees fail to find a schedule for $\mathcal{T}_2$ on a two-processor platform [6]. Like $\mathcal{T}_1$ from Example 1.1, task set $\mathcal{T}_2$ is not EDF-schedulable [6]. On the contrary, our technique will identify $\mathcal{T}_2$ as an urgent task set and find the following schedule: $T_1$ for time interval $[0, 2)$ by the first processor, $T_2$ for time interval $[0, 1)$ by the second processor and $[2, 3)$ by the first processor, and $T_3$ for interval $[1, 3)$ by the second processor. ∎

To handle increasing task workload in embedded real-time systems ranging from automotive control to avionics, dual-core platforms are very popular, according to Stevenson and Hill [30] and Kim et al. [16]. This motivates the need for developing an efficient schedulability test and scheduling algorithm for a two-processor system. As in the alarm system, in a number of automotive applications described by Kopetz et al. [17], Leteinturier [20], Brodt [5] and avionics described by Ras and Cheng [25], Rice and Cheng [26], and Locke et al. [23], there are periodic task sets with long periods, short deadlines, and computation times close to the corresponding relative deadlines. These

are exactly the task characteristics we have captured in our model for which we provide a polynomial-time schedulability test and scheduler. There have been studies in the scheduling of periodic tasks with short deadlines and long periods for uniprocessor systems such as the one described by Audsley [3], but few on multiprocessors. This paper will tackle the tasks' scheduling on two-processor/dual-core systems.

**The structure of this paper:** Section II presents the definition and notations needed for the scheduling problem, and a conversion result of a urgent task set to an equivalent unit computation time urgent task set. Section III describes an efficient 2SAT encoding for urgent preemptive tasks using Algorithm **A** and its refined form, Algorithm **B**. Section IV presents a necessary condition for scheduling urgent tasks. The last two sections present related work and conclusions.

## II. The Scheduling Problem

There exists a few different, but similar, formulations for the scheduling problem. Although these formulations are in general equivalent, they might highlight some dimensions more than other dimensions. In this sense, our paper considers the two-processor platform, independent preemptive tasks, and no shared resources or overload.

For the sake of the presentation, we list some of the useful notations for the schedulability theory. A time interval is a set of time stamps with the property that any time stamp that lies between two time stamps in the set is also included in the set. For example, $[s, e)$ denotes a time interval that is left-closed and right-open. We say that task $T$ executes in the time interval $[s, e)_{(r)}$ if $T$ is ready to execute by processor $r$ at time $s$ and finishes its execution before time $e$, giving the possibility of next task to start its execution by processor $r$ at time $e$. The set with no elements is called the empty set and is denoted by $\emptyset$. We say that $[s, e)_{(r_1)} \cap [s', e')_{(r_2)} = \emptyset$ if and only if either $r_1 \neq r_2$ or $[s, e) \cap [s', e') = \emptyset$ in the mathematical sense (i.e., $[s, e) \cap [s', e') = \{x \mid x \in [s, e)$ and $x \in [s', e')\}$). From now on, $r_1$ and $r_2$ denote the two processors we are using. For a finite set $V$, we denote by $|V|$ the number of elements of $V$.

Here is a formal definition of the scheduling problem on a two-processor environment where each task has its own deadline. We consider in this paper a tast set denoted as $\mathcal{T}$ given by $\{T_1, ..., T_k\}$, where each task $T_i$ is given by $(s_i,\ c_i,\ d_i)$. According to [29], if each task has a deadline, the scheduling problem for the multiprocessor environment is exacerbated. This is actually one of the key points why the scheduling problem for multiprocessor is difficult. Definition 2.1 defines the execution assignment for the time interval $[0, D)$, where $D = \max\{d_i \mid T_i \in \mathcal{T}\}$. We refer to $D$ as the maximum deadline. The execution assignment is in fact similar to the notion of schedule defined by Srinivasan and Anderson [28]. The schedule there is represented as a predicate, whereas the execution assignment is expressed as a union of intervals.

*Definition 2.1:* Let us consider $\mathcal{T} = \{T_1, ..., T_k\}$ a task set, where each task $T_i$ is given by $(s_i,\ c_i,\ d_i)$. We say that the task set $\mathcal{T}$ is schedulable by processors $r_1$ and $r_2$ if and only if there exists an *execution assignment* (also known as *schedule*) denoted by $EA : \mathcal{T} \rightarrow [0, D)$, where in general $[s, e)_{(r)} \in EA(T)$ means the task $T$ executes by processor $r$ in time interval from time $s$ to time $e$, and satisfies the following two properties:

1) $\forall\ i \in \{1, ..., k\}$, we have $EA(T_i) = [s_i^{(1)}, e_i^{(1)})_{(r_{i,1})} \cup$ ... $\cup [s_i^{(n_i)}, e_i^{(n_i)})_{(r_{i,n_i})}$, where $r_{i,1}$, ..., $r_{i,n_i}$ are processors $r_1$ or $r_2$, $s_i^1 < e_i^1 \leq ... \leq s_i^{(n_i)} < e_i^{(n_i)}$, $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) = c_i$, $s_i \leq s_i^{(1)}$ and $e_i^{(n_i)} \leq d_i$;

2) $\forall\ i \in \{1, ..., k\}$, $\forall\ j \in \{1, ..., k\}$, $i \neq j$, we have $EA(T_i) \cap EA(T_j) = \emptyset$. ∎

Similar to the approach from [8], the scheduling problem presented in Definition 2.1 assumes that the tasks' constraints are known in advance, such as deadlines, computation times, and start times. This framework is called *static scheduling* [29]. Static scheduling is in contrast with dynamic scheduling, where the constraints may not be known in advance (e.g., start time). In fact, Mok showed that the scheduling problem for real-time systems with shared resources and no knowledge about the future start times of the tasks is undecidable [24]. It is important in practice to reduce task waiting time and context-switching time, especially when power dissipation is considered [19], [27]. However, for simiplicity Definition 2.1 assumes also that there is no context-switching time. Given $[s_i^{(u_i)}, e_i^{(u_i)})_{(r_i)} \in EA(T_i)$ and $[s_j^{(u_j)}, e_j^{(u_j)})_{(r_j)} \in EA(T_j)$, where $r_i$ and $r_j$ are processors, such that $e_i^{(u_i)} = s_j^{(u_j)}$, then we say that task $T_j$ is executed immediately after $T_i$. The superscript above means the block index, namely a task with an execution time of $c_i$ may be decomposed into different and distinct $n_i$ blocks executed in intervals $[s_i^{(1)}, e_i^{(1)})_{(r_{i,1})}$, ..., $[s_i^{(n_i)}, e_i^{(n_i)})_{(r_{i,n_i})}$.

Given two tasks $T_i$ and $T_j$, we recall that $T_i \rightarrow T_j$ is a precedence constraint between $T_i$ and $T_j$ if task $T_j$ must wait for task $T_i$ to finish its execution in order to get started.

Using the notation of Definition 2.1, we say that tasks $T_i$ are preemptive if $n_i \geq 1$ (the case $n_i = 1$ corresponds to non-preemptive tasks). Another special case is when the tasks have unit computation time. According to Definition 2.1, task $T_i$ has unit computation time if $c_i = 1$. A unit computation time is usually considered non-preemptive. Lawler proved that the scheduling problem for non-preemptive unit computation time tasks with

arbitrary start time can be solved by a polynomial time complexity algorithm [18]. However, according to Graham, Lawler, Lenstra and Kan, the scheduling problem with non-preemptive and non-unit computation time tasks becomes $\mathcal{NP}$-hard [14]. Next we recall the meaning of the urgent task set defined briefly in the Introduction.

*Definition 2.2:* Let us consider a task $T = (s, c, d)$. We say that task $T$ is *urgent* if and only if $s+c \leq d \leq s+c+1$. A task set containing only urgent tasks is called an *urgent task set*. ■

As shown in Examples 1.1 and 1.2 of the Introduction, the task sets $\mathcal{T}_1$ and $\mathcal{T}_2$ are urgent task sets. In addition, it is easy to check that $\mathcal{T}_1$ and $\mathcal{T}_2$ are not schedulable on a uniprocessor platform. Section III presents a scheduling algorithm able to generate a feasible schedule for $\mathcal{T}_1$ and $\mathcal{T}_2$ on a two-processor platform.

The next result shows the conversion of a preemptive urgent task set to a unit computation time urgent task set.

*Theorem 2.1:* Let $\mathcal{T} = \{T_1, ..., T_k\}$ be a preemptive urgent task set, where $k \geq 1$, and each $T_i$ is denoted as $(s_i, c_i, d_i)$. Let $\mathcal{T}' = \{T_1^{(1)}, ..., T_1^{(c_1)}, ..., T_k^{(1)}, ..., T_k^{(c_k)}\}$ be a task set such that $T_i^{(l)} = (s_i + l - 1, 1, d_i - c_i + l)$ and $T_i^{(1)} \rightarrow T_i^{(2)}$, ..., $T_i^{(c_i)-1} \rightarrow T_i^{(c_i)}$ are the precedence constraints, for all $i \in \{1, ..., k\}$, and $l \in \{1, ..., c_i\}$.

Then $\mathcal{T}$ is schedulable if and only if $\mathcal{T}'$ is schedulable.

**Proof** ($\Longrightarrow$) Let us suppose that $\mathcal{T}$ is schedulable. According to Definition 2.1, there exists $EA$, an execution assignment for $\mathcal{T}$, that satisfies conditions 1) and 2). We shall show that any arbitrary time interval that belongs to $EA$ leads to some unit time intervals that belong to the execution assignment for $\mathcal{T}'$. Let us consider an arbitrary time interval $[s_i^{(l_i)}, e_i^{(l_i)})_{(r_{i,l_i})} \in EA(T_i)$, where $l \in \{1, ..., n_i\}$. We show that there exist $e_i^{(l_i)} - s_i^{(l_i)}$ unit computation tasks in $\mathcal{T}'$ that get executed in the time intervals $[s_i^{(l_i)}, s_i^{(l_i)} + 1)_{(r_{i,l_i})}$, ..., $[e_i^{(l_i)} - 1, e_i^{(l_i)})_{(r_{i,l_i})}$. According to the unit computation tasks of $\mathcal{T}'$, this is equivalent with the following two conditions:

(a) $s_i + l_i - 1 \leq s_i^{(l_i)}$

(b) $s_i^{(l_i)} \leq d_i - c_i + l_i - 1$

If conditions (a) and (b) hold, the unit computation tasks $T_i^{(l_i)}, T_i^{(l_i)+1}, ..., T_i^{(l_i)+e_i^{(l_i)}-s_i^{(l_i)}-1}$, will execute the above unit intervals by processor $(r_{i,l_i})$. In other words:

$EA(T_i^{(l_i)}) = [s_i^{(l_i)}, s_i^{(l_i)} + 1)_{(r_{i,l_i})}$;

...

$EA(T_i^{(l_i)+e_i^{(l_i)}-s_i^{(l_i)}-1}) = [e_i^{(l_i)} - 1, e_i^{(l_i)})_{(r_{i,l_i})}$.

Hence, the precedence constraints $T_i^{(1)} \rightarrow T_i^{(2)}$, ..., $T_i^{(c_i)-1} \rightarrow T_i^{(c_i)}$ hold.

To prove (a), we consider condition 1) from Definition 2.1, that is, $s_i \leq s_i^1 < e_i^1 \leq ... \leq s_i^{(l_i)} < ... \leq s_i^{(n_i)} < e_i^{(n_i)}$.

Since $e_i^{(j)} \geq s_i^{(j)} + 1$, for any $j \in \{1, ..., l_i\}$, it follows that $s_i^{(l_i)} \geq s_i^{(l_i)-1} + 1 \geq ... \geq s_i^1 + l_i - 1 \geq s_i + l_i - 1$. Therefore $s_i + l_i - 1 \leq s_i^{(l_i)}$.

To prove (b), we use again condition 1). We get $s_i^{(l_i)} < e_i^{(l_i)} \leq e_i^{(l_i)+1} + 1 \leq ... \leq e_i^{(l_i)+(n_i)-(l_i)} + n_i - l_i \leq d_i + n_i - l_i$.

Condition 2) from Definition 2.1 holds for $\mathcal{T}'$ based on condition 2) for $\mathcal{T}$. Since $EA$ is an execution assignment schedulable for $\mathcal{T}'$ that satisfies conditions 1) and 2), it follows that $\mathcal{T}'$ is a schedulable task set.

($\Longleftarrow$) Let us suppose that $\mathcal{T}'$ is schedulable. That means there exists $EA$, an execution assignment for $\mathcal{T}'$ that satisfies conditions 1) and 2) from Definition 2.1:

1) $\forall\, i \in \{1, ..., k\}$, $\exists\, s_i^{(1)}, ..., s_i^{(c_1)}, ..., s_i^{(c_k)}$, such that $s_i \leq s_i^{(1)} < ... < s_i^{(c_1)} < ... < s_i^{(c_k)} < d_i$ and $EA(T_i^{(j)}) = [s_i^{(j)}, s_i^{(j)} + 1)_{(r_{i,j})}$, $\forall\, j \in \{1, ..., c_i\}$;

2) $\forall\, i \in \{1, ..., k\}, \forall\, i' \in \{1, ..., c_i\}, \forall\, j \in \{1, ..., k\}, \forall\, j' \in \{1, ..., c_j\}$, $i \neq j$, we have $EA(T_i^{(i')}) \cap EA(T_j^{(j')}) = \emptyset$.

Since any arbitrary task $T_i$, where $i \in \{1, ..., k\}$, of $\mathcal{T}$ is preemptive, the execution assignment $EA(T_i)$ can be easily defined using $EA(T_i^{(j)})$, where $j \in \{1, ..., c_i\}$. As such, $EA(T_i) = [s_i^{(1)}, s_i^{(1)} + 1)_{(r_{i,1})} \cup ... \cup [s_i^{(c_i)}, s_i^{(c_i)} + 1)_{(r_{i,c_i})}$, for all $i \in \{1, ..., c_i\}$. This is ensured by the precedence constraints: $T_i^{(1)} \rightarrow T_i^{(2)}$, ..., $T_i^{(c_i)-1} \rightarrow T_i^{(c_i)}$. Without loss of generality, we take $n_i = c_i$ in Definition 2.1 by identifying each execution interval as a unit computation time interval. Obviously, $EA(T_i)$ satisfies condition 1) from Definition 2.1 because $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) = \sum_{j=1}^{n_i} 1 = n_i = c_i$.

The second condition from Definition 2.1, $EA(T_i) \cap EA(T_j) = \emptyset$, for all $i \in \{1, ..., k\}$, $j \in \{1, ..., k\}$, $i \neq j$, holds due to the mutual exclusion of the unit computation tasks.

Therefore, it follows that $\mathcal{T}$ is a schedulable task set. ■

Considering the notations from Theorem 2.1, we can formally define a *conversion mapping* $\varphi : \mathcal{T} \rightarrow [\mathcal{T}']$ given by $\varphi(T_i) = [T_i^{(1)}, ..., T_i^{(c_i)}]$, for all $i \in \{1, ..., k\}$. Note that $[\mathcal{T}']$ means the set of all arrays with elements of $\mathcal{T}'$, The conversion mapping expresses also the precedence constraints between subtasks: $T_i^{(1)} \rightarrow T_i^{(2)}$, ..., $T_i^{(c_i)-1} \rightarrow T_i^{(c_i)}$.

The next two examples illustrate the conversion mapping of an urgent task set to a unit computation task set as described in Theorem 2.1.

*Example 2.1:* Let $\mathcal{T}_1 = \{T_1, T_2, T_3\}$ be the preemptive urgent task set defined in Example 1.1. By Theorem 2.1, $\mathcal{T}_1$ is converted to the unit computation task set $\mathcal{T}_1' = \{T_1', T_2', T_3', T_4', T_5'\}$, where $T_1' = (0, 1, 1)$, $T_2' = (0, 1, 2)$, $T_3' = (0, 1, 1.5)$, $T_4' = (1, 1, 2.5)$, and $T_5' = (2, 1, 3.5)$. The

conversion mapping is given by $\varphi(T_1) = [T_1']$, $\varphi(T_2) = [T_2']$, and $\varphi(T_3) = [T_3', T_4', T_5']$. According to Theorem 2.1, $\mathcal{T}_1$ is schedulable if and only if $\mathcal{T}_1'$ is schedulable. ■

*Example 2.2:* Let $\mathcal{T}_2 = \{T_1, T_2, T_3\}$ be the preemptive urgent task set defined in Example 1.2. By Theorem 2.1, $\mathcal{T}_2$ is converted to the unit computation task set $\mathcal{T}_2' = \{T_1', T_2', T_3', T_4', T_5', T_6'\}$, where $T_1' = (0, 1, 2)$, $T_2' = (1, 1, 3)$, $T_3' = (0, 1, 2)$, $T_4' = (1, 1, 3)$, $T_5' = (0, 1, 2)$, and $T_6' = (1, 1, 3)$. The conversion mapping is given by $\varphi(T_1) = [T_1', T_2']$, $\varphi(T_2) = [T_3', T_4']$, and $\varphi(T_3) = [T_5', T_6']$. According to Theorem 2.1, $\mathcal{T}_2$ is schedulable if and only if $\mathcal{T}_2'$ is schedulable. ■

We shall use in next section the conversion mapping and its inverse. The inverse of $\varphi$ is denoted as $\varphi^{(-1)} : [\mathcal{T}'] \rightarrow \mathcal{T}$ and is given by $\varphi^{(-1)} = T_i$. For instance, the inverse of conversion mapping for the task set from Example 2.2 is given by: $\varphi^{(-1)}(T_1') = T_1$, $\varphi^{(-1)}(T_2') = T_1$, $\varphi^{(-1)}(T_3') = T_2$, $\varphi^{(-1)}(T_4') = T_2$, $\varphi^{(-1)}(T_5') = T_3$, and $\varphi^{(-1)}(T_6') = T_3$.

## III. A 2SAT Encoding for Urgent Preemptive Tasks

The first part of this section defines the notion of over-schedulable schedulability and shows that an over-schedulable task set is schedulable, too. Then, this section describes Algorithm **A**, and its refined version, Algorithm **B**, that has as input a task set $\mathcal{T}$ of $k$ unit computation time urgent tasks and provides as output a 2SAT encoding (explained next paragraph) $F$ such that $\mathcal{T}$ is over-schedulable if and only if $F$ is satisfiable. Since the 2SAT satisfiability problem was proved by Aspvall, Plass, and Tarjan in 1979 to be solvable by a polynomial time complexity algorithm [2], it follows that the problem of scheduling unit computation time urgent tasks, and, by Theorem 2.1, the problem of scheduling urgent tasks can be solved by a polynomial time complexity algorithm.

The next part focuses on the SAT encoding associated with the task set. Let $\mathcal{LP}$ be the *propositional logic* over the finite set of *atomic formulæ* (variables) $V = \{A_1, ..., A_n\}$. A *literal* $L$ is an atomic formula $A$ (*positive* literal) or its negation $\neg A$ (*negative* literal).Any function $\mathcal{S} : V \rightarrow \{\texttt{false}, \texttt{true}\}$ is an *assignment* that can be uniquely extended in $\mathcal{LP}$ to a propositional formula $F$. The binary vector $(y_1, ..., y_n)$ is a truth assignment for $F$ over $V = \{A_1, ..., A_n\}$ if and only if $\mathcal{S}(F) = \texttt{true}$ such that $\mathcal{S}(A_i) = y_i$, $\forall\ i \in \{1, ..., n\}$. A formula $F$ is called *satisfiable* if and only if there exists an assignment $\mathcal{S}$ for which $\mathcal{S}(F) = \texttt{true}$; otherwise $F$ is called *unsatisfiable*. Any finite disjunction of literals is a *clause*. Any propositional formula $F \in \mathcal{LP}$ having $l$ clauses can be translated into the *conjunctive normal form* (CNF): $F = (L_{1,1} \vee ... \vee L_{1,n_1}) \wedge ... \wedge (L_{l,1} \vee ... \vee L_{l,n_l})$, where the $L_{i,j}$'s are literals. We can denote the above $F$ using the set representation $F = \{\{L_{1,1}, ..., L_{1,n_1}\}, ..., \{L_{l,1}, ..., L_{l,n_l}\}\}$, or simply $F = \{C_1, ..., C_l\}$, where $C_i = \{L_{i,1}, ..., L_{i,n_i}\}$. For instance, formula $F = (A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \vee \neg C)$ is represented in set notation as $F = \{\{A, \neg B, \neg C\}, \{\neg A, B, \neg C\}\}$. A clause $C$ with no literals is called the empty clause, and it is denoted as $\square$. If a propositional formula contains the empty clause, the entire formula is unsatisfiable (or contradictory). A clause $C$ with (at most) two literals is called a 2CNF clause. A formula containing only 2CNF clauses is called 2CNF formula. The SAT problem ('*Does a CNF propositional formula have a truth assignment?*') is called the 2SAT problem if the input is a 2CNF formula.

Aspvall, Plass and Tarjan proved in 1979 [2] that the 2SAT problem has a solution if and only if there is no strongly connected component of the implication graph that contains both some variable and its negation. Since strongly connected components may be found in linear time by an algorithm based on depth first search, the same linear time bound applies as well to the 2SAT problem.

We describe below Algorithm **A** that takes as input a task set and provides in the output a propositional formula that is satisfiable if and only if the task set is schedulable.

### Algorithm **A**

**The input:** $\mathcal{T}' = \{T_1', ..., T_k'\}$ an urgent task set, where each $T_i'$ is given by $(s_i, 1, d_i)$ and, $s_i$ and $d_i$ are non-negative integers;

**The output:** $F$ a 2SAT propositional formula such that $F$ is satisfiable if and only if $\mathcal{T}'$ is schedulable on a two-processor platform.

**The method:**

**1.** $F = \emptyset$;

**2.** add propositional clauses to $F$ specifying that each task cannot start earlier than the starting time for both processors;

**3.** add propositional clauses to $F$ specifying that each task cannot execute after its deadline;

**4.** add propositional clauses to $F$ specifying the mutual exclusion constraints, that is, a processor can execute at most one job at a time;

**5.** add propositional clauses to $F$ specifying that if job $T_i$ is executed by processor 1, then it cannot be executed at the same time by processor 2.

**6.** add propositional clauses to $F$ specifying that if two unit time jobs do not correspond to the same initial task, then they can be permuted (because they are not subject to precedence constraints);

**7.** add propositional clauses to $F$ specifying that if two time unit jobs correspond to the same initial task, then they have to follow the sequence/order on which they appear (e.g.,

the second unit time job cannot execute before the first unit time job).

We refine Algorithm **A** into Algorithm **B** by providing more implementation details about constructing a propositional formula able to provide a feasible schedule for the given urgent task set. Subsequently, we provide a correctness and time complexity result for Algorithm **B**.

Algorithm **B**

**The input:** $\mathcal{T}' = \{T'_1, ..., T'_k\}$ an urgent task set, where each $T'_i$ is given by $(s_i, 1, d_i)$ and, $s_i$ and $d_i$ are non-negative integers; $\varphi : \mathcal{T} \rightarrow \mathcal{T}'$ the conversion mapping;
**The output:** $F$ a 2SAT propositional formula such that $F$ is satisfiable if and only if $\mathcal{T}'$ is schedulable on a two-processor platform.
**The method:**
**1.** $C(F) = \emptyset$;
**2.** $V(F) = \{e^{(1)}_{i,j}, e^{(2)}_{i,j} \mid i \in \{1, ..., k\}, j \in \{0, ..., D-1\}\}$;
**3.** for ($i = 1$; $i <= k$; $i$++) {
**4.** for ($j = 0$; $j < s_i$; $j$++)
**5.** $C(F) = C(F) \cup \{\{\neg e^{(1)}_{i,j}\}\} \cup \{\{\neg e^{(2)}_{i,j}\}\}$;
**6.** for ($j = d_i + 1$; $j < D$; $j$++)
**7.** $C(F) = C(F) \cup \{\{\neg e^{(1)}_{i,j}\}\} \cup \{\{\neg e^{(2)}_{i,j}\}\}$;
**8.** for ($m = i + 1$; $m <= k$; $m$++) {
**9.** $maxT = max\{s_i, s_m\}$;
**10.** $minT = min\{d_i, d_m\}$;
**11.** for ($j = maxT$; $j < minT$; $j$++)
**12.** $C(F) = C(F) \cup \{\{\neg e^{(1)}_{i,j}, \neg e^{(1)}_{m,j}\}\} \cup \{\{\neg e^{(2)}_{i,j}, \neg e^{(2)}_{m,j}\}\}$;
}
}
**13.** for ($l = 0$; $l < D$; $l$++)
**14.** if (there exists $T'_i$ == $(l, 1, l+1)$ orelse $T'_i$ == $(l, 1, l+2)$) then {
**15.** if (there exists no $j \neq i$, such that $T'_j$ == $(l, 1, l+1)$ orelse $T'_j$ == $(l, 1, l+2)$) then
**16.** $C(F) = C(F) \cup \{\{e^{(1)}_{i,l}, e^{(2)}_{i,l}\}\} \cup \{\{\neg e^{(1)}_{i,l}, \neg e^{(2)}_{i,l}\}\}$ ;
**17.** else { // let $T'_j = (l, 1, l+1)$ orelse $T'_j = (l, 1, l+2)$, $j \neq i$
**18.** if ($\exists T'_j$ such that $\varphi^{-1}(T'_i) \neq \varphi^{-1}(T'_j)$) then
**19.** $C(F) = C(F) \cup \{\{e^{(1)}_{i,l}, e^{(1)}_{j,l}\}\} \cup \{\{e^{(1)}_{i,l}, e^{(2)}_{i,l}\}\} \cup \{\{e^{(1)}_{j,l}, e^{(2)}_{j,l}\}\} \cup \{\{e^{(2)}_{i,l}, e^{(2)}_{j,l}\}\}$;
**20.** else
**21.** if $T'_j$ == $(l, 1, l+2)$ then {
**22.** $C(F) = C(F) \cup \{\{e^{(1)}_{i,l}, e^{(2)}_{i,l}\}\} \cup \{\{\neg e^{(1)}_{i,l}, \neg e^{(2)}_{i,l}\}\}$ ;
**23.** replace $T'_j$ by $(l+1, 1, l+2)$;
}
**24.** else $C(F) = \square$; // $F$ is unsatisfiable as $T'_j = (l, 1, l+1)$)
**25.** if (there exist more tasks $(l, 1, l+1)$ other than $T'_i$ and $T'_j$) then
**26.** $C(F) = \square$; // $F$ is unsatisfiable
**27.** replace all tasks $(l, 1, l+2)$ other than $T'_i$ and $T'_j$ by $(l+1, 1, l+2)$;
}
}
**28.** return $F$ with $V(F)$ and $C(F)$ computed above;

The operator orelse from Algorithm **B** has the following meaning: $Cond_1$ orelse $Cond_2$ will evaluate first condition $Cond_1$. If this is true, then $Cond_2$ will not be evaluated. On the other hand, if $Cond_1$ is false, then $Cond_2$ will be evaluated and the value of $Cond_1$ orelse $Cond_2$ will be false if and only if both $Cond_1$ and $Cond_2$ are false.

The next result proves the correctness and complexity of Algorithm **B**. Note that functions $max()$ and $min()$ from Algorithm **B** have the traditional meaning: $min(a, b) = a$ if $a < b$, and $b$ otherwise; and $max(a, b) = a$ if $a > b$, and $b$ otherwise.

*Theorem 3.1:* Let us consider $\mathcal{T}'$ an urgent task set and $\varphi : \mathcal{T} \rightarrow \mathcal{T}'$ the conversion mapping as input for Algorithm **A**. Let $F$ be the output by Algorithm **B**. Then $\mathcal{T}'$ is schedulable on a two-processor platform if and only if $F$ is satisfiable. Moreover, $F$ has polynomial size of $\mathcal{T}'$ and Algorithm **B** has a polynomial-time complexity.

**Proof** We start with the complexity part, as it is easier to check. Obviously, $|V(F)| = 2 \cdot k \cdot D$. The number of clauses depends on each task's starting time and deadline. An upper bound for $|C(F)|$ added at statements from lines **3** to **12** is $2 \cdot k \cdot D + 2 \cdot k \cdot D + k \cdot (k-1)$. To find an upper bound for $|C(F)|$ added at statements from lines **13** to **27**, we suppose that at each iteration of the for statement from line **13**, we add 4 clauses as in the statement from line **19**. Therefore, an upper bound for $|C(F)|$ added from lines **13** to **27** is $4 \cdot D$. By summarizing these numbers, we get the total upper bound of $4 \cdot D \cdot (k+1) + k \cdot (k-1)$. Hence $F$ has a polynomial size of $\mathcal{T}'$.

In order to estimate the time complexity of Algorithm **B**, we recall that the Aspvall, Plass and Tarjan algorithm [2] for solving a 2CNF formula needs a time complexity of $n \cdot (n+m)$, where $n$ is the number of variables and $m$ the number of clauses of the propositional formula. Combining with the above results, it follows that Algorithm **B** has a time complexity of $2 \cdot k \cdot D \cdot [4 \cdot D \cdot (k+1) + k \cdot (k-1)]$.

For the correctness part, we shall prove that for any $i \in \{1, ..., k\}$, we have: $e^{(r)}_{i,j} =$ true if and only if task $T'_i$ is executed by processor $r$ at time interval $[j, j+1)$, where $j \in \{0, ..., D-1\}$ and $r \in \{1, 2\}$.

The statements from lines **4** and **5** consider each previous sub-interval by adding the unit clauses $\{\neg e^{(1)}_{i,j}\}$ and $\{\neg e^{(2)}_{i,j}\}$ for all time units before their start time. This is equivalent to: task $T'_i$ cannot execute in the sub-interval $[0, s_i)$. Similarly, the statements from lines **6** and **7** each and every subsequent sub-interval by adding the unit clauses $\{\neg e^{(1)}_{i,j}\}$ and $\{\neg e^{(2)}_{i,j}\}$ for all time units after their deadline time. This is equivalent to: task $T'_i$ cannot execute in the sub-interval $[d_i, D)$.

The statements from lines **8** to **12** correspond to mutual exclusion between tasks $T'_i$ and $T'_m$ executed by a processor, that is, one processor can execute either $T'_i$ or $T'_m$ at the same time. This is equivalent to adding to formula $F$

all the clauses $\{\neg e_{i,j}^{(1)}, \neg e_{m,j}^{(1)}\}$ and $\{\neg e_{i,j}^{(2)}, \neg e_{m,j}^{(2)}\}$ for all $j \in \{max\{s_i, s_m\},\ min\{d_i, d_m\}\}$.

The statements from lines **14** to **16** correspond to the case when there is only one task $T_i'$ asking for a processor available at time interval $[l, l+1)$. Since this can be done by either processor $r_1$ or $r_2$, the corresponding clauses added to $C(F)$ are $\{\{e_{i,l}^{(1)}, e_{i,l}^{(2)}\}\}$, and $\{\{\neg e_{i,l}^{(1)}, \neg e_{i,l}^{(2)}\}\}$. The last clause means that if $T_i'$ executes on processor $r_1$, then it cannot execute on processor $r_2$, and vice versa. The statements from lines **17** and **19** correspond to the case when there are two tasks $T_i'$ and $T_j'$ which do not belong to the same initial task (that is, $\varphi^{-1}(T_i') \neq \varphi^{-1}(T_j')$) asking for a processor for the time interval $[l, l+1)$. This correspond to $(e_{i,l}^{(1)} \wedge e_{j,l}^{(2)}) \vee (e_{j,l}^{(1)} \wedge e_{i,l}^{(2)})$, hence equivalent to the conjunctive normal form clauses $\{\{e_{i,l}^{(1)}, e_{j,l}^{(1)}\}\}$, $\{\{e_{i,l}^{(1)}, e_{i,l}^{(2)}\}\}$, $\{\{e_{j,l}^{(1)}, e_{j,l}^{(2)}\}\}$, $\{\{e_{i,l}^{(2)}, e_{j,l}^{(2)}\}\}$.

The statements from lines **20** to **24** deal with the case when $\varphi^{-1}(T_i') = \varphi^{-1}(T_j')$, namely $T_i'$ and $T_j'$ are subtasks of the same task. If $T_j'$ is specified as $(l, 1, l+1)$, then the schedule cannot be done so the formula $F$ is unsatisfiable. If $T_j'$ is specified as $(l, 1, l+2)$, then this is changed to $(l+1, 1, l+2)$. The task $T_i'$ is scheduled for either $[l.l+1)_{(r_1)}$ or $[l.l+1)_{(r_2)}$. Hence $C(F)$ will contain the corresponding clauses $\{\{e_{i,l}^{(1)}, e_{i,l}^{(2)}\}\}$, and $\{\{\neg e_{i,l}^{(1)}, \neg e_{i,l}^{(2)}\}\}$. The statement from line **22** corresponds to the case when condition $\varphi^{(-1)}(T_i') \neq \varphi^{(-1)}(T_j')$ does not hold. It means $T_i'$ and $T_j'$ are subtasks of the same original task, that is, $\varphi^{(-1)}(T_i')$. As such, it is also clear that since $T_j'$ was not scheduled for $[l, l+1)$, then the precedence constraints $T_i' \rightarrow T_j'$ was correctly implemented.

The statements from lines **25** and **26** cover the case when there is no processor available for the time interval $[l, l+1)$ since both processors $r_1$ and $r_2$ are taken. The statement from line **26** corresponds to the case when both processors are taken for the time interval $[l, l+1)$ hence its execution is shifted from $[l, l+1)$ to $[l+1, l+2)$ - this will be processed at the next iteration of the `for` statement from line **13**.

It is clear from lines **5** to **12** of Algorithm **B** that whenever $e_{i,j}^{(r)} = false$, then task $T_i'$ is not executed by processor $r$ at time interval $[j, j+1)$. Lines **16** and **22** ensure that if $e_{i,l}^{(r)} = true$ then task $T_i'$ executes at time interval $[l, l+1)_{(r)}$. Line **19** ensures that if $e_{i,l}^{(r)} =$`true` and $e_{j,l}^{(3-r)} =$`true`, then $T_i'$ executes in $[l, l+1)_{(r)}$ and $T_j'$ executes in $[l, l+1)_{(3-r)}$.

Line **28** returns the output of Algorithm **B**, hence the theorem is completely proved. ∎

Next, we show now the application of Algorithm **B** for our running task sets.

*Example 3.1:* We continue now the task set from Example 2.1. We recall that $\mathcal{T}_1' = \{T_1,\ T_2,\ T_3,\ T_4,\ T_5\}$. Without loss of generality, we truncate the deadlines to the integer value by considering the ceiling of the deadline. Therefore, $T_1 = (0, 1, 1)$, $T_2 = (0, 1, 2)$, $T_3 = (0, 1, 1)$, $T_4 = (1, 1, 2)$, and $T_5 = (2, 1, 3)$. By running Algorithm **B** on a two-processor platform, we get the formula $F$ given by the following clauses: $\{\neg e_{1,1}^{(1)}\}$, $\{\neg e_{1,2}^{(1)}\}$, $\{\neg e_{1,3}^{(1)}\}$, $\{\neg e_{1,1}^{(2)}\}$, $\{\neg e_{1,2}^{(2)}\}$, $\{\neg e_{1,3}^{(2)}\}$, $\{\neg e_{2,2}^{(1)}\}$, $\{\neg e_{2,3}^{(1)}\}$, $\{\neg e_{2,2}^{(2)}\}$, $\{\neg e_{2,3}^{(2)}\}$, $\{\neg e_{3,1}^{(1)}\}$, $\{\neg e_{3,2}^{(1)}\}$, $\{\neg e_{3,3}^{(1)}\}$, $\{\neg e_{3,1}^{(2)}\}$, $\{\neg e_{3,2}^{(2)}\}$, $\{\neg e_{3,3}^{(2)}\}$, $\{\neg e_{4,0}^{(1)}\}$, $\{\neg e_{4,2}^{(1)}\}$, $\{\neg e_{4,3}^{(1)}\}$, $\{\neg e_{4,0}^{(2)}\}$, $\{\neg e_{4,2}^{(2)}\}$, $\{\neg e_{4,3}^{(2)}\}$, $\{\neg e_{5,0}^{(1)}\}$, $\{\neg e_{5,1}^{(1)}\}$, $\{\neg e_{5,3}^{(1)}\}$, $\{\neg e_{5,0}^{(2)}\}$, $\{\neg e_{5,1}^{(2)}\}$, $\{\neg e_{5,3}^{(2)}\}$,

The mutual exclusion clauses are the following: $\{\neg e_{1,0}^{(1)}, \neg e_{2,0}^{(1)}\}$, $\{\neg e_{1,0}^{(2)}, \neg e_{2,0}^{(2)}\}$, $\{\neg e_{1,0}^{(1)}, \neg e_{3,0}^{(1)}\}$, $\{\neg e_{1,0}^{(2)}, \neg e_{3,0}^{(2)}\}$, $\{\neg e_{2,0}^{(1)}, \neg e_{3,0}^{(1)}\}$, $\{\neg e_{2,0}^{(2)}, \neg e_{3,0}^{(2)}\}$.

The clauses generated at steps **13** to **26** of Algorithm **B** are the following:

$\{e_{1,0}^{(1)}, e_{3,0}^{(1)}\}$, $\{e_{1,0}^{(1)}, e_{1,0}^{(2)}\}$, $\{e_{3,0}^{(1)}, e_{3,0}^{(2)}\}$, $\{e_{1,0}^{(2)}, e_{3,0}^{(2)}\}$, $\{e_{2,1}^{(1)}, e_{4,1}^{(1)}\}$, $\{e_{2,1}^{(1)}, e_{2,1}^{(2)}\}$, $\{e_{4,1}^{(1)}, e_{4,1}^{(2)}\}$, $\{e_{2,1}^{(2)}, e_{4,1}^{(2)}\}$, $\{e_{5,2}^{(1)}, e_{5,2}^{(2)}\}$,

A truth assignment for $F$ is: $\mathcal{S}(e_{1,0}^{(1)}) =$ `true`, $\mathcal{S}(e_{2,1}^{(1)}) =$ `true`, $\mathcal{S}(e_{3,0}^{(2)}) =$ `true`, $\mathcal{S}(e_{4,1}^{(2)}) =$ `true`, and $\mathcal{S}(e_{5,2}^{(1)}) =$ `true`. This truth assignment corresponds to the following schedulable schedule for $\mathcal{T}_1'$:

$EA(T_1') = \{[0, 1)_{(1)}\}$, $EA(T_2') = \{[1, 2)_{(1)}\}$, $EA(T_3') = \{[0, 1)_{(2)}\}$, $EA(T_4') = \{[1, 2)_{(2)}\}$, $EA(T_5') = \{[2, 3)_{(1)}\}$.

Coming back to the original task set $\mathcal{T}$, we get $EA(T_1) = \{[0, 1)_{(1)}\}$, $EA(T_2) = \{[1, 2)_{(1)}\}$, $EA(T_3) = \{[0, 2)_{(2)}$, $[2, 3)_{(1)}\}$. ∎

*Example 3.2:* We continue now the task set from Example 2.2. We recall that $\mathcal{T}_2' = \{T_1,\ T_2,\ T_3,\ T_4,\ T_5,\ T_6\}$. By running Algorithm **A** on a two-processor platform, we get the formula $F$ given by the following clauses: $\{\neg e_{1,2}^{(1)}\}$, $\{\neg e_{1,2}^{(2)}\}$, $\{\neg e_{2,0}^{(1)}\}$, $\{\neg e_{2,0}^{(2)}\}$, $\{\neg e_{3,2}^{(1)}\}$, $\{\neg e_{3,2}^{(2)}\}$, $\{\neg e_{4,0}^{(1)}\}$, $\{\neg e_{4,0}^{(2)}\}$, $\{\neg e_{5,2}^{(1)}\}$, $\{\neg e_{5,2}^{(2)}\}$, $\{\neg e_{6,0}^{(1)}\}$, $\{\neg e_{6,0}^{(2)}\}$.

Here there are some mutual exclusion clauses: $\{\neg e_{1,0}^{(1)}, \neg e_{3,0}^{(1)}\}$, $\{\neg e_{1,0}^{(2)}, \neg e_{3,0}^{(2)}\}$, $\{\neg e_{1,0}^{(1)}, \neg e_{5,0}^{(1)}\}$, $\{\neg e_{1,0}^{(2)}, \neg e_{5,0}^{(2)}\}$, $\{\neg e_{3,0}^{(1)}, \neg e_{5,0}^{(1)}\}$, $\{\neg e_{3,0}^{(2)}, \neg e_{5,0}^{(2)}\}$, $\{\neg e_{1,1}^{(1)}, \neg e_{2,1}^{(1)}\}$, $\{\neg e_{1,1}^{(2)}, \neg e_{2,1}^{(2)}\}$, and so on. The rest of the clauses are omitted because they are similar combinations with the above ones.

The clauses generated at steps **13** to **26** of Algorithm **B** are the following:

$\{e_{1,0}^{(1)}, e_{3,0}^{(1)}\}$, $\{e_{1,0}^{(1)}, e_{1,0}^{(2)}\}$, $\{e_{3,0}^{(1)}, e_{3,0}^{(2)}\}$, $\{e_{1,0}^{(2)}, e_{3,0}^{(2)}\}$, $\{e_{2,1}^{(1)}, e_{5,1}^{(1)}\}$, $\{e_{2,1}^{(1)}, e_{2,1}^{(2)}\}$, $\{e_{5,1}^{(1)}, e_{5,1}^{(2)}\}$, $\{e_{2,1}^{(2)}, e_{5,1}^{(2)}\}$, $\{e_{4,1}^{(1)}, e_{6,1}^{(1)}\}$, $\{e_{4,1}^{(1)}, e_{4,1}^{(2)}\}$, $\{e_{6,1}^{(1)}, e_{6,1}^{(2)}\}$, $\{e_{4,1}^{(2)}, e_{6,1}^{(2)}\}$.

A truth assignment for $F$ is: $\mathcal{S}(e_{1,0}^{(1)}) =$ `true`, $\mathcal{S}(e_{2,1}^{(1)}) =$ `true`, $\mathcal{S}(e_{3,0}^{(2)}) =$ `true`, $\mathcal{S}(e_{4,2}^{(1)}) =$ `true`, $\mathcal{S}(e_{5,1}^{(2)}) =$ `true`, and $\mathcal{S}(e_{6,2}^{(2)}) =$ `true`. This truth assignment corresponds to the following schedulable schedule for $\mathcal{T}_2'$:

$EA(T'_1) = \{[0,1)_{(1)}\}$, $EA(T'_2) = \{[1,2)_{(1)}\}$, $EA(T'_3) = \{[0,1)_{(2)}\}$, $EA(T'_4) = \{[2,3)_{(1)}\}$, $EA(T'_5) = \{[1,2)_{(2)}\}$, and $EA(T'_6) = \{[2,3)_{(2)}\}$.
Coming back to the original task set $\mathcal{T}_2$, we get $EA(T_1) = \{[0,2)_{(1)}\}$, $EA(T_2) = \{[0,1)_{(2)}, [2,3)_{(1)}\}$, $EA(T_3) = \{[1,3)_{(2)}\}$. ■

As an immediate implication of Theorem 3.1, Algorithm **B** is optimal. In other words, if $F$ provided as output by Algorithm **B** is unsatisfiable, then both task sets $\mathcal{T}$ and $\mathcal{T}'$ are not schedulable. In fact, Algorithm **B** schedules the task set $\mathcal{T}'$ in a similar way as Least Laxity First strategy. The resulted formula $F$ can actually provide more than just one schedule. In addition, Algorithm **B** has a polynomial-time complexity on a multiprocessor platform.

## IV. Necessary Conditions for Scheduling Urgent Tasks

This section is devoted to identifying large subclasses of urgent tasks that are not schedulable. The tasks that lead to non-schedulability are called *jammed* tasks. We prove that if a task set contains jammed tasks, then the task set cannot be schedulable. We describe some necessary conditions for tasks' scheduling on a two-processor platform. The next subsection presents some necessary conditions for tasks' scheduling on a uniprocessor platform. We saw in Section III that Algorithm **B** was able to check in a polynomial time complexity whether a task set is schedulable or not. However, the worst case has a time complexity of $2 \cdot k \cdot D \cdot [4 \cdot D \cdot (k+1) + k \cdot (k-1)]$, where $k$ is the number of tasks and $D$ is their maximum deadline. The conditions we present this section can be checked in linear time and space complexity.

*Definition 4.1:* Let us consider a two-processor platform. We say that a task set $\mathcal{T}$ is **jammed** if (at least) one of the following conditions hold:

a) there exist at least five urgent tasks in $\mathcal{T}$ with the same start time;

b) there exist at least four urgent tasks in $\mathcal{T}$ with start time $s$ and at least three other urgent tasks of $\mathcal{T}$ with start time $s+1$;

c) there exist at least three urgent tasks in $\mathcal{T}$ with start time $s$ and at least four other urgent tasks of $\mathcal{T}$ with start time $s+1$. ■

The following result represents a necessary condition for feasibility of urgent tasks. Theorem 4.1 is useful for schedulability analysis of urgent task sets.

*Theorem 4.1:* A jammed urgent task set is not schedulable on a two-processor platform.

**Proof** Let us consider $\mathcal{T}$ a jammed urgent task set. According to Definition 4.1, it means we have one of the following conditions fulfilled:

a) there exist at least $T_1$, $T_2$,, $T_3$, $T_4$, and $T_5 \in \mathcal{T}$ such that their start times equal $s$;

b) there exist $T_1$, $T_2$, $T_3$, $T_4 \in \mathcal{T}$ with start time $s$ and $T_5$, $T_6$, $T_7 \in \mathcal{T}$ with start time $s+1$;

c) there exist $T_1$, $T_2$, $T_3 \in \mathcal{T}$ with start time $s$ and $T_4$, $T_5$, $T_6$, $T_7 \in \mathcal{T}$ with start time $s+1$.

We need to prove that all the above conditions lead to unschedulable schedules. The computation time of each task is at least 1.

a) Without loss of generality, let us assume that $T_1$ executes in the time interval $[s, s+1)_{(r_1)}$, $T_2$ in $[s+1, s+2)_{(r_1)}$, $T_3$ in $[s, s+1)_{(r_2)}$, and $T_4$ in $[s+1, s+2)_{(r_2)}$. Task $T_5$ cannot be executed later than $s+2$ because it is an urgent task. At the same time, $T_3$ cannot be executed in either $[s, s+1)$ or $[s+1, s+2)$ as these two time intervals are taken by processors $r_1$ and $r_2$. Hence $T_5$ will miss its deadline.

b) Without loss of generality, let us assume that $T_1$ executes in $[s, s+1)_{(r_1)}$, $T_2$ in $[s+1, s+2)_{(r_1)}$, $T_3$ in $[s, s+1)_{(r_2)}$, $T_4$ in $[s+1, s+2)_{(r_2)}$, $T_5$ in $[s+2, s+3)_{(r_1)}$, and $T_6$ in $[s+2, s+3)_{(r_2)}$. Task $T_7$ cannot be executed later than $s+3$ because it is an urgent task with start time $s+1$. However, $T_7$ cannot be executed earlier than $s+3$. Hence $T_7$ will miss its deadline.

c) Case c) is similar to case b).

In conclusion, $\mathcal{T}$ is not schedulable as it contains jammed tasks. ■

In order to test whether a given urgent task set $\mathcal{T}$ is schedulable, we check first the applicability of Theorem 4.1 for jammed tasks. Obviously, this can be done in linear time and space complexity. In the affirmative case, we conclude that $\mathcal{T}$ is not schedulable. Otherwise, Algorithm **B** can be applied as an alternative to check whether the corresponding propositional formula is satisfiable.

### A. Scheduling conditions for the uniprocessor platform

This subsection presents some necessary conditions for tasks' scheduling on a uniprocessor platform. These conditions look much simpler than the corresponding conditions for tasks' scheduling on a two-processor platform. Likewise the scheduling conditions for the two-processor platform, and the corresponding scheduling conditions for the uniprocessor platform can be done in linear time and space complexity.

*Definition 4.2:* Let us consider a uniprocessor platform. We say that a task set $\mathcal{T}$ is **uni-jammed** if (at least) one of the following conditions hold:

a) there exist at least three urgent tasks in $\mathcal{T}$ with the same start time;

b) there exist two urgent tasks in $\mathcal{T}$ with start time $s$ and at least two other urgent tasks of $\mathcal{T}$ with start time $s+1$. ■

The following result represents a necessary condition for feasibility of urgent tasks. Theorem 4.2 is useful for schedulability analysis of urgent task sets.

*Theorem 4.2:* An uni-jammed urgent task set is not schedulable on a uniprocessor platform.

**Proof** Let us consider $\mathcal{T}$ a uni-jammed urgent task set. According to Definition 4.2, it means we have one of the following conditions fulfilled:

a) there exist at least $T_1$, $T_2$, and $T_3 \in \mathcal{T}$ such that their start times equal $s$;

b) there exist $T_1$, $T_2 \in \mathcal{T}$ with start time $s$ and $T_3$, $T_4 \in \mathcal{T}$ with start time $s+1$.

We need to prove that both conditions lead to unschedulable schedules. The computation time of each task is, of course, at least 1.

a) Without loss of generality, let us assume that $T_1$ executes in the time interval $[s, s+1)$ and $T_2$ in $[s+1, s+2)$. Task $T_3$ cannot be executed later than $s+2$ because it is an urgent task. At the same time, $T_3$ cannot be executed in either $[s, s+1)$ or $[s+1, s+2)$ as the processor executes $T_1$ and $T_2$, respectively. Hence $T_3$ will miss its deadline.

b) Without loss of generality, let us assume that $T_1$ executes in $[s, s+1)$, $T_2$ in $[s+1, s+2)$, and $T_3$ in $[s+2, s+3)$. Task $T_4$ cannot be executed later than $s+3$ because it is an urgent task with start time $s+1$. However, $T_4$ cannot be executed in any of the previous time intervals, namely $[s, s+1)$, $[s+1, s+2)$ or $[s+2, s+3)$. Hence $T_4$ will miss its deadline.

In conclusion, $\mathcal{T}$ is not schedulable as it contains uni-jammed tasks. ■

## V. Related and Future Work

Liu and Layland found a polynomial-time schedulability analysis test that ensures the Earliest Dealine First (EDF) optimality for synchronous tasks (i.e., all tasks have the same start time), and with relative deadlines equal to their respective periods [22]. However, Leung and Merrill proved that deciding if an asynchronous periodic task set, when deadlines are less or equal than the periods, is schedulable on one processor is $\mathcal{NP}$-hard [21].

There exist several polynomial-time algorithms for two-processor scheduling [12], [31], [32], but all these restrict tasks to have unit execution times (UET). Garey and Johnson [12] presented a test for determining whether there exists a schedule on two identical processors for this type of tasks with start times and deadlines, and provided an $\mathcal{O}(n^3)$ scheduling algorithm if such a schedule exists. The considered tasks are single-instance and hence not periodic. Vazirani [31] proposed a fast parallel (R-NC) algorithm for this problem. Wu and Jaffar [32] studied non-preemptive two-processor scheduling, again for UET tasks but with arbitrary precedence constraints, release times, and deadlines. They proposed an $\mathcal{O}(n^4)$ algorithm based on the key consistency notion known as successor-tree-consistency for solving the problem. Only single-instance tasks are considered. In contrast, our proposed polynomial-time schedulability test and algorithm works for urgent tasks with arbitrary execution times.

Moreover, Baruah, Rosier, and Howell proved in 1990 that the problem of deciding whether an asynchronous periodic task set, when deadlines are less than the periods, is schedulable on one processor is $\mathcal{NP}$-hard in the strong sense [4]. This even more negative result precludes the existence of pseudo-polynomial time algorithms for the solution of this feasibility decision problem, unless $\mathcal{P} = \mathcal{NP}$.

This result was extended in 1995 by Howell and Venkatrao who showed that the decision problem of determining whether a periodic task system is schedulable for all start times with respect to the class of algorithms using inserted idle times is NP-Hard in the strong sense, even when the deadlines are equal to the periods [15].

An interesting concept in scheduling theory motivated by parallel computing systems is to consider multiprocessor tasks which require more than one processor at the same time [11]. A generalization of the classsical uniprocessor and two-processor unit computation time tasks was addressed in [13]. Giaro and Kubale showed that, given a fixed set of either 1-element (it requires a single dedicated processor) or 2-element (it requires two dedicated procesors simultaneously), the scheduling problem of sparse instances of tasks with arbitrary start times and deadlines can be solved in polynomial time. We intend to consider this kind of scheduling framework and check whether the scheduling problem of urgent task sets can still be solved in polynomial time.

Chen and Hsueh [7] presented a model, called $T - L_{er}$ plane, to describe the behavior of tasks and processors. By allowing task migration, the authors described two optimal on-line algorithms based on $T - L_{er}$ plane to schedule real-time tasks with dynamic-priority assignment on uniform multiprocessors. Our work presented an optimal scheduling algorithm only for two-processor platforms, but we do not restrict the processors to be uniform.

Carpenter et al. presented in [6] nine combinations of priority and migration degrees taxonomy for scheduling algorithms. The task's priority can be (i) static, (ii) dynamic but fixed within a job, or (iii) fully dynamic. The task's degree migration can be (i) no migration (i.e., task partitioning), (ii) migration allowed, but only at the boundary (i.e., dynamic partitioning at the job level), and (iii) unrestricted migration (i.e., jobs are allowed to migrate). Example 1.2 describes the task set $\mathcal{T}_2$ taken from [6] to illustrate the power of a fully dynamic and unrestricted migration scheduling algorithm. On the other hand, all the

other eight combinations of priority and migration degree fail to find a schedule for $\mathcal{T}_2$ on a two-processor platform [6]. However, according to [6], this class of fully dynamic and unrestricted migration scheduling algorithm has a major drawback. The runtime overhead of the scheduling algorithms for this class may be unacceptably too high for some applications, in terms of runtime complexity, preemption frequency, and migration frequency. According to [6], migration is an important criteria in the design of multiprocessor real-time systems because it affects the true cost in terms of the final system produced. We plan as future work to investigate finding the best scheduling algorithm with minimum (or at least as minimum as possible) number of preemptions and migrations.

## VI. Conclusion

We identified and formally defined a non-trivial class of task sets, called *urgent tasks*, for which the scheduling problem can be solved in polynomial time. We presented an efficient algorithm for finding the schedule via an efficient 2SAT encoding. We identified a necessary efficient condition useful for schedulability analysis of urgent tasks.

## References


[1] James Anderson and Anand Srinivasan. Pfair scheduling: Beyond periodic task systems. In *in Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, 2000.

[2] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[4] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.

[5] J. Brodt. Revving up with automotive multicore. Technical report, EDN Magazine. NEC Electronics America, Inc., February 2008.

[6] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy 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] Shih-Ying Chen and Chih-Wen Hsueh. Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 147–156, Washington, DC, USA, 2008. IEEE Computer Society.

[8] A. M. K. Cheng. *Real-time systems. Scheduling, Analysis, and Verification*. Wiley-Interscience, U. S. A., 2002.

[9] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta lnformatica*, 1(3):200–213, 1972.

[10] M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74:807–813, 1974.

[11] Maciej Drozdowski. Scheduling multiprocessor tasks – an overview. *European Journal of Operational Research*, 94(2):215–230, October 1996.

[12] M. R. Garey and David S. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM J. Comput.*, 6(3):416–426, 1977.

[13] Krzysztof Giaro and Marek Kubale. Chromatic scheduling of 1- and 2-processor uet tasks on dedicated machines with availability constraints. In *Parallel Processing and Applied Mathematics*, pages 855–862, 2005.

[14] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[15] Rodney R. Howell and Muralidhar K. Venkatrao. On non-preemptive scheduling of recurring tasks using inserted idle times. *Information and Computation*, 117:117–50, 1995.

[16] Kwangsik Kim, Dohun Kim, and Chanik Park. Real-time scheduling in heterogeneous dual-core architectures. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 91–96, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Hermann Kopetz, Roman Obermaisser, Christian El Salloum, and Bernhard Huber. Automotive software development for a multi-core system-on-a-chip. In *SEAS '07: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[18] E. L. Lawler. Recent results in the theory of machine scheduling. *Mathematical Programming: The State of the Art. in M. Grtchel, A. Bachem, B. Korte (Eds.)*, pages 202–234, 1983.

[19] Tse Lee and Albert Mo Kim Cheng. Multiprocessor scheduling of hard-real-time periodic tasks with task migration constraints, 1994.

[20] P. Leteinturier. Multi-core processors: driving the evolution of automotive electronics architectures. *http://embedded.com/*, September 2007.

[21] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[22] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[23] C. Douglas Locke, David R. Vogel, and T. J. Mesler. Building a predictable avionics platform in ada: A case study. In *IEEE Real-Time Systems Symposium*, pages 181–189, 1991.

[24] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[25] Jim Ras and Albert M.K. Cheng. An evaluation of the dynamic and static multiprocessor priority ceiling protocol and the multiprocessor stack resource policy in an smp system. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, pages 13–22, 2009.

[26] L. E. P. Rice and A. M. K. Cheng. Timing analysis of the X-38 space station crew return vehicle avionics. In *Proceedings of the 5-th IEEE-CS Real-Time Technology and Applications Symposium*, pages 255–264, 1999.

[27] Zili Shao, Qingfeng Zhuge, Meilin Liu, Chun Xue, Edwin H. M. Sha, and Bin Xiao. Algorithms and analysis of scheduling for loops with minimum switching. *Int. J. Comput. Sci. Eng.*, 2(1/2):88–97, 2006.

[28] Anand Srinivasan and James H. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, 2006.

[29] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, 1995.

[30] Matt Stevenson and John Hill. Dual-core processing drives high-performance embedded systems. *RTC, The magazine of record for the embedded open systems industry*, January 2006.

[31] U Vazirani and V V Vazirani. The two-processor scheduling problem is in r-nc. In *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 11–21, New York, NY, USA, 1985. ACM.

[32] Hui Wu and Joxan Jaffar. Two processor scheduling with real release times and deadlines. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 127–132, New York, NY, USA, 2002. ACM.