Optimal Scheduling of Urgent Preemptable Tasks

Stefan Andrei

No. 1, January 2009
Optimal Scheduling of Urgent Preemptable Tasks

Ștefan Andrei
Department of Computer Science
Lamar University, TX 77710, U.S.A.
sandrei@cs.lamar.edu
January 7, 2009

Abstract

Tasks’ scheduling has always been a central problem in embedded real-time systems community. As in general the scheduling problem is \( \mathcal{NP} \)-hard, researchers have been looking for efficient heuristics for solving the scheduling problem in polynomial time. Liu and Layland found in 1973 a polynomial-time schedulability analysis test that ensures the Earliest Deadline First (EDF) optimality for synchronous tasks (i.e., all tasks have the same start time), and with relative deadlines equal to their respective periods. However, Leung and Merrill proved in 1980 that deciding if an asynchronous periodic task set, when deadlines are less or equal than the periods, is feasible on one processor is \( \mathcal{NP} \)-hard. 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 feasible on one processor is \( \mathcal{NP} \)-hard in the strong sense. 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.
Our paper deals with this fundamental issue in scheduling theory. We have identified a new non-trivial and practical subclass of asynchronous periodic tasks, when deadlines are less or equal than the periods, for which the scheduling problem can be solved in polynomial time. Briefly, denoting the deadline of task $T$ by $d$, its start time by $s$ and its computation time by $c$, 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.

1 Introduction

Tasks’ scheduling has always been a central problem in embedded real-time systems community. As in general the scheduling problem is $\mathcal{NP}$-hard, researchers have been looking for efficient heuristics for solving the scheduling problem in polynomial time. Liu and Layland found in 1973 a polynomial-time schedulability analysis test that ensures the Earliest Deadline First (EDF) optimality for synchronous tasks (i.e., all tasks have the same start time), and with relative deadlines equal to their respective periods [10]. However, Leung and Merrill proved in 1980 that deciding if an asynchronous periodic task set, when deadlines are less or equal than the periods, is feasible on one processor is $\mathcal{NP}$-hard [9].

$\mathcal{NP}$ is the class of all decision problems that can be solved by a non-deterministic Turing machine in polynomial time. A decision problem $P$ is $\mathcal{NP}$-complete if $R \in \mathcal{NP}$ and all other problems in $\mathcal{NP}$ are polynomial reducible to $P$. A decision problem $P$ is $\mathcal{NP}$-hard if all problems in $\mathcal{NP}$ are polynomial reducible to $P$, but it is uncertain that $P \in \mathcal{NP}$. The class of decision problems solvable by a deterministic Turing machine in polynomial-time complexity is denoted as $\mathcal{P}$. In fact, the most important open question of complexity theory is whether the complexity class $\mathcal{P}$ is the same as the complexity class $\mathcal{NP}$, or is just a strict subset as is generally assumed.

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 feasible on one processor is $\mathcal{NP}$-hard in the strong sense [2]. This even more negative result precludes the existence of pseudo-polynomial time algorithms for the solution of this feasibility decision.
problem, unless $P = NP$.

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 [7].

Our paper deals with this fundamental issue in scheduling theory. We have identified a new non-trivial and practical subclass of asynchronous periodic tasks, when deadlines are less or equal than the periods, for which the scheduling problem can be solved in polynomial time. Briefly, denoting the deadline of task $T$ by $d$, its start time by $s$ and its computation time by $c$, 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.

**The motivation of this paper:** We identify and formally define a non-trivial class of task sets, called *urgent tasks*, for which the scheduling problem can be solved in polynomial time. We present an efficient algorithm for finding the schedule via an efficient 2SAT encoding (Algorithm A) and a correctness and completeness result for it (Theorem 3.1). We identify a necessary efficient condition useful for schedulability analysis of urgent tasks.

**The structure of this paper:** Section 2 presents the definition and notations needed for the scheduling problem, feasibility, and over-feasibility. Section 3 describes an efficient 2SAT encoding for urgent preemptable tasks using Algorithm A. Section 4 presents a necessary condition for scheduling urgent tasks. The last two sections present related work and conclusions.

## 2 The Scheduling Problem

There exist 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 uni-processor platform instead of multi-processor platform, independent preemptable tasks rather than precedence constraints, shared resources or overload, and so on. We consider also the context of static scheduling instead of dynamic scheduling. That means the scheduling algorithm has prior
knowledge about task set constraints such as deadlines, start times, computation times, and periods. In fact, Mok showed in 1983 that the scheduling problem for real-time systems with shared resources and no knowledge about the future start times of the tasks is undecidable [11].

For the sake of the presentation, we list some of the useful notations in schedulability theory. We denote by $L = LCM(p_1, \ldots, p_k)$ the Lowest Common Multiple of the integers $p_1, \ldots, p_k$. The smallest common multiple of two or more integers is called the lowest common multiple. 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 $[s, e)$ if $T$ is ready to execute at time $s$ and finishes its execution before time $e$, giving the possibility of next task to start its execution at time $e$. The set with no elements is called the empty set and is denoted by $\emptyset$. 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 uni-processor environment where each task has its own deadline. Note that in many scheduling definitions, all tasks have the same deadline as an assumption to restrict the scheduling problem. According to [12], if each task has a deadline, the scheduling problem for the multi-processor environment is exacerbated.

**Definition 2.1** Let us consider $\mathcal{T} = \{T_1, \ldots, T_k\}$ a task set, where each task $T_i$ is given by $(s_i, c_i, d_i, p_i)$ with the following meaning: $s_i$ is called $T_i$’s starting time (also known as release time), $c_i$ is called $T_i$’s computation time (also known as worst-case execution time), $d_i$ is called $T_i$’s deadline, and $p_i$ is called $T_i$’s period. We say that the task set $\mathcal{T}$ is feasible if and only if there exists an execution assignment (also known as schedule) denoted by $EA : \mathcal{T} \rightarrow [0, L)$, where in general $[s, e) \in EA(T)$ means the task $T$ executes in time interval from time $s$ to time $e$, and satisfies the following two properties:

1) $\forall i \in \{1, \ldots, k\}$, we have $EA(T_i) = [s_i^{(1)}, e_i^{(1)}) \cup \ldots \cup [s_i^{(n_i)}, e_i^{(n_i)})$, where $s_1^1 < e_1^1 \leq \ldots \leq s_i^{(n_i)} < e_i^{(n_i)}$, $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) = c_i$, $s_i \leq s_i^{(1)}$ and $e_i^{(n_i)} \leq d_i$;

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

4
Similar to the approach from [3], the scheduling problem presented in Definition 2.1 assumes that the tasks’ constraints are known in advance, such as deadlines, computation times, start times, and periods. This framework is called static scheduling [12]. Definition 2.1 assumes also that there is no context-switching time. Given \([s_i^{(ui)}], e_i^{(ui)}\) \(\in EA(T_i)\) and \([s_j^{(ui)}, e_j^{(ui)}] \in EA(T_j)\) such that \(e_i^{(ui)} = s_j^{(ui)}\), then we say that task \(T_j\) is executed immediately after \(T_i\).

Using the notations of Definition 2.1, we say that tasks \(T_i\) are preemptable if \(n_i \geq 1\) (the case \(n_i = 1\) corresponds to non-preemptable 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-preemptable. The scheduling problem for non-preemptable unit computation time tasks with arbitrary start time was proved to be in \(P\) by Lawler in 1983 [8]. However, according to Graham, Lawler, Lenstra and Kan (1979), when dealing with non-preemptable and non-unit computation time tasks, the scheduling problem becomes \(NP\)-hard [6].

**Theorem 2.1** Let \(T = \{T_1, ..., T_k\}\) be a preemptable urgent task set, where \(k \geq 1\), and each \(T_i\) is denoted as \((s_i, c_i, d_i, p_i)\). Let \(T' = \{T_1^{(1)}, ..., T_1^{(c_i)}, ..., T_k^{(1)}, ..., T_k^{(c_k)}\}\) be a task set such that \(T_i^{(l)} = (s_i + l - 1, 1, d_i - c_i + l, p_i)\) for all \(i \in \{1, ..., k\}\), and \(l \in \{1, ..., c_i\}\).

Then \(T\) is feasible if and only if \(T'\) is feasible.

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

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

(b) \(s_i^{(l)} \leq d_i - c_i + l_i - 1\)
If conditions (a) and (b) hold, the unit computation tasks \( T_i^{(l_i)}, T_i^{(l_i)+1}, \ldots, T_i^{(l_i)+e_i^{(l_i)}-s_i^{(l_i)}-1} \), will execute the above unit intervals. In other words:

\[
EA(T_i^{(l_i)}) = [s_i^{(l_i)}, s_i^{(l_i)} + 1);
\]

\[
\ldots
EA(T_i^{(l_i)+e_i^{(l_i)}-s_i^{(l_i)}-1}) = [e_i^{(l_i)} - 1, e_i^{(l_i)}).
\]

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

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

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

Condition 2) from Definition 2.1 holds for \( T' \) based on condition 2) for \( T \).

Since \( EA \) is an execution assignment feasible for \( T' \) that satisfies conditions 1) and 2), it follows that \( T' \) is a feasible task set.

(\( \iff \)) Let us suppose that \( T' \) is feasible. That means there exists \( EA \), an execution assignment for \( T' \) that satisfies conditions 1) and 2) from Definition 2.1:

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

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

Since any arbitrary task \( T_i \), where \( i \in \{1, \ldots, k\} \), of \( T \) is preemptable, the execution assignment \( EA(T_i) \) can be easily defined using \( EA(T_i^{(j)}) \), where \( j \in \{1, \ldots, c_i\}. \) As such, \( EA(T_i) = [s_i^{(1)}, s_i^{(1)} + 1) \cup \ldots \cup [s_i^{(c_i)}, s_i^{(c_i)} + 1), \) for all \( i \in \{1, \ldots, c_i\}. \) 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 \( i \in \{1, \ldots, k\}, j \in \{1, \ldots, k\}, i \neq j, \) holds due to the mutual exclusion of the unit computation tasks.

Therefore, it follows that \( T \) is a feasible task set.
The next example illustrates the conversion of an urgent task set to a unit computation task set as described in Theorem 2.1.

Example 2.1 Let $T = \{T_1, T_2, T_3\}$ be a preemptable urgent task set given by $T_1 = (0, 3, 4, 6), T_2 = (3, 2, 6, 6), \text{ and } T_3 = (4, 1, 6, 6)$. It can be easily checked that all three tasks are urgent. By Theorem 2.1, $T$ can be converted to the unit computation task set $T' = \{T_1^{(1)}, T_1^{(2)}, T_1^{(3)}, T_2^{(1)}, T_2^{(2)}, T_3\}$, where $T_1^{(1)} = (0, 1, 2, 6), T_1^{(2)} = (1, 1, 3, 6), T_1^{(3)} = (2, 1, 4, 6), T_2^{(1)} = (3, 1, 5, 6), \text{ and } T_2^{(2)} = (4, 1, 6, 6)$. In order to anticipate the corresponding SAT encoding, we can now rename the tasks $T_1^{(1)}$ by $T_1, T_1^{(2)}$ by $T_2, T_1^{(3)}$ by $T_3, T_2^{(1)}$ by $T_4, T_2^{(2)}$ by $T_5, \text{ and } T_3$ by $T_6$. Hence $T' = \{T_1, \ldots, T_6\}$, where $T_1 = (0, 1, 2, 6), T_2 = (1, 1, 3, 6), T_3 = (2, 1, 4, 6), T_4 = (3, 1, 5, 6), T_5 = (4, 1, 6, 6), \text{ and } T_6 = (4, 1, 6, 6).

According to Theorem 2.1, $T$ is feasible if and only if $T'$ is feasible.

3 A 2SAT Encoding for Urgent Preemptable Tasks

The first part of this section defines the notion of over-feasible schedulability and shows that an over-feasible task set is feasible, too. Then, this section describes Algorithm A that has as input a task set $T$ of $k$ unit computation time urgent tasks and provides as output a 2SAT encoding $F$ such that $T$ is over-feasible if and only if $F$ is satisfiable. Since the 2SAT satisfiability problem was proved by Aspvall, Plass, and Tarjan in 1979 to be polynomial [1], 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 in polynomial time.

We say that a task set $T$ is over-feasible if the processor may execute some tasks more or equal than their computation times. More formally, a task set is over-feasible if the equality $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) = c_i$ of Definition 2.1 is replaced by the inequality $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) \geq c_i$. The next result proves that an over-feasible task set is feasible, too.

Lemma 3.1 An over-feasible task set is feasible.

Proof Let $T = \{T_1, \ldots, T_k\}$ be an over-feasible task set, where each task $T_i$ is given by $(s_i, c_i, d_i, p_i), \forall i \in \{1, \ldots, k\}$. That means there exists an
execution assignment $EA : \mathcal{T} \rightarrow [0, L)$, where $L = LCM(p_1, \ldots, p_k)$, and satisfies the following two properties:

1) $\forall i \in \{1, \ldots, k\}$, we have $EA(T_i) = [s_i^{(1)}, e_i^{(1)}] \cup \ldots \cup [s_i^{(n_i)}, e_i^{(n_i)})$, where $s_i^1 < e_i^1 \leq \ldots \leq s_i^{(m_i)} < e_i^{(m_i)}$, $\sum_{j=1}^{m_i} (e_i^{(j)} - s_i^{(j)}) = c_i, s_i \leq s_i^{(1)}$ and $e_i^{(n_i)} \leq d_i$.

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

Given a task $T_i$, we distinguish two cases. We define $EA' : \mathcal{T} \rightarrow [0, L)$ as follows.

a) If $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) = c_i$, then $EA'(T_i) = EA(T_i)$;

b) If $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) > c_i$, then $EA'(T_i)$ is obtained by the following pseudocode:

\[
m = n_i;
stop = \text{false};
while (m >= 1) and (stop == \text{false}) \{
    if $\sum_{j=1}^{m} (e_i^{(j)} - s_i^{(j)}) - c_i \geq e_i^{(m)} - s_i^{(m)}$ then $m--;
    \text{else} \{
        e_i^{(m)} = e_i^{(m)} + c_i - \sum_{j=1}^{m} (e_i^{(j)} - s_i^{(j)});
        stop = \text{true};
    \}
\}
\]

$EA'(T_i) = [s_i^{(1)}, e_i^{(1)}] \cup \ldots \cup [s_i^{(n_i)}, e_i^{(n_i)});$

$n_i = m$.

To show that $EA'$ is a proper execution assignment for $\mathcal{T}$, we have only to show that given $EA'(T_i) = [s_i^{(1)}, e_i^{(1)}] \cup \ldots \cup [s_i^{(n_i)}, e_i^{(n_i)})$, then $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) = c_i$, all the other conditions being obviously fulfilled. The proof for case a) is straightforward.

We show now that proof for case b). Let $m$ be the integer from the else branch of the if statement from the above algorithm. The new value of $e_i^{(m)}$ is obtained from the previous value of $e_i^{(m)}$ by adding $c_i - \sum_{j=1}^{m} (e_i^{(j)} - s_i^{(j)})$, that is, a negative integer. Since $n_i$ equals to $m$ now, then $\sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) =
\[ \sum_{j=1}^{m} (e_i^{(j)} - s_i^{(j)}) + c_i - \sum_{j=1}^{m} (e_i^{(j)} - s_i^{(j)}) = c_i. \]

In conclusion, we showed that \( EA' \) is an execution assignment indicating that \( T \) is a feasible task set. ■

The next part focuses on the SAT encoding associated to 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), that is, \( \mathcal{V}(L) = \mathcal{V}(\neg L) = A \). Any function \( S : V \rightarrow \{0, 1\} \) 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 \( S(F) = 1 \) such that \( S(A_i) = y_i, \forall i \in \{1, ..., n\} \). A formula \( F \) is called satisfiable if and only if there exists an assignment \( S \) for which \( S(F) = 1 \); otherwise \( F \) is called unsatisfiable. Any finite disjunction of literals is a clause. The set of atomic formulæ whose literals belong to clause \( C \) and formula \( F \) are denoted by \( \mathcal{V}(C) \) and \( \mathcal{V}(F) \), respectively. Any propositional formula \( F \in \mathcal{LP} \) having \( l \) clauses can be translated into the conjunctive normal form (CNF): \( F = (L_{1,1} \lor ... \lor L_{1,n_1}) \land ... \land (L_{l,1} \lor ... \lor L_{l,n_l}) \), 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 \( C_i = \{L_{i,1}, ..., L_{i,n_i}\} \). A clause \( C \) with 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.

As Aspvall, Plass and Tarjan proved in 1979 [1], 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.

Algorithm A

**The input:** \( T = \{T_1, ..., T_k\} \) an urgent task set, where each \( T_i \) is given by \((s_i, 1, d_i, p_i)\);

**The output:** \( F \) a 2SAT propositional formula such that \( F \) is satisfiable if and only if \( T \) is over-feasible.

**The method:**

1. compute \( L = LCM(p_1, ..., p_k); C(F) = \emptyset \);
2. \( V(F) = \{ e_{i,j} \mid i \in \{1, \ldots, k\}, j \in \{0, \ldots, L - 1\}\} \)
3. \( \text{for (i = 1; i <= k; i++)} \)
4. \( \text{for (l = 0; l < L/p_i; l++)} \) 
5. \( \text{for (j = p_i * l; j < s_i + p_i * l; j++)} \)
6. \( C(F) = C(F) \cup \{ \neg e_{i,j}\} \)
7. \( \text{for (j = d_i + p_i * l; j < p_i * (l + 1); j++)} \)
8. \( C(F) = C(F) \cup \{ \neg e_{i,j}\} \)
9. \( \text{if (d_i > s_i + c_i) then} \)
10. \( C(F) = C(F) \cup \{ e_{i,s_i+p_i*l} \lor e_{i,s_i+p_i*l+1}\} \)
11. \( \text{else C(F) = C(F) \cup \{ e_{i,s_i+p_i*l}\}} \)
12. \( \text{for (m = i + 1; m <= k; m++)} \)
13. \( \text{maxT} = \max\{s_i + p_i * l, s_m + p_m * l\} \)
14. \( \text{minT} = \min\{d_i + p_i * l, d_m + p_m * l\} \)
15. \( \text{for (j = maxT; j < minT; j++)} \)
16. \( C(F) = C(F) \cup \{ \neg e_{i,j} \lor \neg e_{m,j}\} \)

17. \( \text{return } F \text{ with } V(F) \text{ and } C(F) \text{ computed above} \)

The next result proves the correctness and complexity of Algorithm A. Note that functions max() and min() from Algorithm A 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 \( T \) an urgent task set as input for Algorithm A and \( F \) provided as output by Algorithm A. Then \( T \) is feasible if and only if \( F \) is satisfiable. Moreover, \( F \) has polynomial size of \( T \) and Algorithm A has a polynomial-time complexity.

**Proof** We start with the complexity part, as it is easier to check. Obviously, \( |V(F)| = k \cdot L \), where \( L = LCM(p_1, \ldots, p_k) \). The number of clauses depends on each task’s starting time and deadline. An upper bound for \( |C(F)| \) can be obtained by taking \( s_i = 0 \) and \( p_i = d_i \), that is, \( k \cdot (k \cdot L + 3 \cdot L + 2) \). Thus \( |C(F)| \leq k \cdot (k \cdot L + 3 \cdot L + 2) \). Since both \( V(F) \) and \( C(F) \) have polynomial size in terms of \( T \), then \( F \) has polynomial size of \( T \). For estimating the time complexity of Algorithm A, we observe that the upper bound of the number of nested loops can be obtained by the nested for statements from lines 3, 4, 12, and 15. It can be easily seen that this number is less than \( k^2 \cdot L \cdot \max\{ c_i \mid i \in \{1, \ldots, k\}\} \).
For the correctness part, we shall prove that for any \( i \in \{1, ..., k\} \), we have: \( e_{i,j} = \text{true} \) if and only if task \( T_i \) is executed at time interval \([j, j+1)\), where \( j \in \{0, ..., L-1\} \).

The for statement from line 4 considers splitting the interval \([0, L]\) in \(L/p_i\) sub-intervals: \([0, p_i), [p_i, 2 \cdot p_i), ..., [p_i \cdot ((L/p_i) - 1), L)\). The statements from lines 5 and 6 consider each and every previous sub-interval by adding the unit clause \( \{\neg e_{i,j}\} \) for all time units before their start time. This is equivalent to: task \( T_i \) cannot execute in the sub-interval \([p_i \cdot l, s_i + p_i \cdot l]\), for any \( l \in \{0, ..., L/p_i\} \). Similarly, the statements from lines 7 and 8 each and every subsequent sub-interval by adding the unit clause \( \{\neg e_{i,j}\} \) for all time units after their deadline time. This is equivalent to: task \( T_i \) cannot execute in the sub-interval \([d_i + p_i \cdot l, p_i \cdot (l+1)\)\), for any \( l \in \{0, ..., L/p_i\} \).

The statement from line 10 states that the disjunctive clause \( \{e_{i,s_l+p_i,s_l+1} \lor e_{i,s_l+p_i,s_l+1}\} \) in case \( d_i > s_i + c_i \) (the test condition from line 9), otherwise the statement from line 11 adds the unit clause \( \{e_{i,s_l+p_i,s_l}\} \). This is equivalent to the fact that task \( T_i \) executes a unit time in one of these two time intervals \([s_i + p_i \cdot l, s_i + p_i \cdot l + 1)\) or \([s_i + p_i \cdot l + 2, s_i + p_i \cdot l + 2)\) in case \( d_i > s_i + c_i \) and only the time interval \([s_i + p_i \cdot l, s_i + p_i \cdot l + 1)\) in case \( d_i = s_i + c_i \).

The statements from lines 12 to 16 correspond to mutual exclusion between tasks \( T_i \) and \( T_m \), that is, the processor can execute either \( T_i \) or \( T_m \). This is equivalent to adding to formula \( F \) all the clauses \( \{\neg e_{i,j} \lor \neg e_{m,j}\} \) for any \( j \in \{\max\{s_i + p_i \cdot l, s_m + p_m \cdot l\}, \min\{d_i + p_i \cdot l, d_m + p_m \cdot l\}\} \).

Line 17 returns the output of Algorithm A, so the theorem is therefore proved.

We continue Example 2.1 with three initial non-unit urgent tasks reduced to six unit computation time urgent tasks. By applying Algorithm A, we get the following clauses:
\[
\{\neg e_{1,2}\}, \{\neg e_{1,3}\}, \{\neg e_{1,4}\}, \{\neg e_{1,5}\}, \{e_{1,0} \lor e_{1,1}\}, \{\neg e_{2,0}\}, \{\neg e_{2,3}\}, \{\neg e_{2,4}\}, \\
\{\neg e_{2,5}\}, \{e_{2,1} \lor e_{2,2}\}, \{\neg e_{3,0}\}, \{\neg e_{3,1}\}, \{\neg e_{3,4}\}, \{\neg e_{3,5}\}, \{e_{3,2} \lor e_{3,3}\}, \{\neg e_{4,0}\}, \\
\{\neg e_{4,1}\}, \{\neg e_{4,2}\}, \{\neg e_{4,3}\}, \{e_{4,3} \lor e_{4,4}\}, \{\neg e_{5,0}\}, \{\neg e_{5,1}\}, \{\neg e_{5,2}\}, \{\neg e_{5,3}\}, \\
\{e_{5,4} \lor e_{5,5}\}, \{\neg e_{6,0}\}, \{\neg e_{6,1}\}, \{\neg e_{6,2}\}, \{\neg e_{6,3}\}, \{e_{6,4} \lor e_{6,5}\}, \{\neg e_{1,1} \lor \neg e_{2,1}\}, \\
\{\neg e_{2,2} \lor \neg e_{3,2}\}, \{\neg e_{3,3} \lor \neg e_{4,3}\}, \{\neg e_{4,4} \lor \neg e_{5,4}\}, \{\neg e_{5,4} \lor \neg e_{6,4}\}, \{\neg e_{5,5} \lor \neg e_{6,5}\}.
\]

An assignment for \( F \) is: \( S(e_{1,0}) = \text{true}, S(e_{2,1}) = \text{true}, S(e_{3,2}) = \text{true}, S(e_{4,3}) = \text{true}, S(e_{5,4}) = \text{true}, \) and \( S(e_{6,5}) = \text{true} \) (the rest of the variables can be \text{false}). This assignment corresponds to the following feasible schedule for \( T' \):
\[
EA(T_1) = [0, 1), EA(T_2) = [1, 2), EA(T_3) = [2, 3), EA(T_4) = [3, 4),
\]
\( EA(T_5) = [4, 5), EA(T_6) = [5, 6). \)

Coming back to the original task set \( T \), we get \( EA(T_1) = \{[0, 3]\}, EA(T_2) = \{[3, 5]\}, EA(T_3) = \{[5, 6]\}. \)

4 A Necessary Condition for Scheduling Urgent Tasks

This section is dedicated to identifying large subclasses of urgent tasks that are not feasible. We called the tasks that lead to un-feasibility, as jammed tasks. We shall prove that if a task set contains jammed tasks, then the task set cannot be feasible.

Definition 4.1 We say that a task set \( T \) is jammed if (at least) one of the following conditions hold:

a) there exist at least three urgent tasks in \( T \) with the same start time;

b) there exist two urgent tasks in \( T \) with start time \( s \) and at least two other urgent tasks of \( T \) 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 feasible.

Proof Let us consider \( T \) a jammed urgent task set. According to Definition 4.1, it means we have one of the following condition fulfilled:

a) there exist at least \( T_1, T_2, \) and \( T_3 \in T \) such that their start times equal with \( s \)

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

We need to prove that both conditions lead to unfeasible 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 feasible as it contains jammed tasks.

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

5 Related and Future Work

The urgent tasks have been formally formulated in this paper. But they have actually appeared in various contexts, including theoretical and practical frameworks. For example, according to [12], Mok used the following task set as a counter-example for the EDF optimality in the multiprocessor case [11]: $\mathcal{T} = \{T_1, T_2, T_3\}$, where $c_1 = 1, d_1 = 1; c_2 = 1, d_2 = 2; c_3 = 3, d_3 = 3.5$; (note that the start times and periods are omitted from this example). Clearly, $\mathcal{T}$ is not EDF-feasible 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). In fact, $\mathcal{T}$ is actually an urgent task set (of course, taking $s_1 = s_2 = s_3 = 0$) on a two-processor platform. For example, these urgent tasks can be executed as follows: first $T_1$ and then $T_2$ on processor $P_1$, and at the same time $T_3$ on processor $P_2$. We plan to extend the urgent task set class to the multi-processor platform to look for a more precise boundary where the scheduling problem transits from $\mathcal{P}$ to $\mathcal{NP}$-hard. This kind of investigations have always received interest from the real-time systems community [12].

A relatively new concept in scheduling theory motivated by parallel computing systems is to consider multiprocessor tasks which require more than one processor at the same time [4]. A generalization of the classsical uni-processor and two-processor unit computation time tasks was addressed in [5]. 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 processors 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.

6 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


