An Efficient Scheduling Algorithm
for the Multiprocessor Platform

Stefan Andrei, Albert M.K. Cheng,
Gheorghe Grigoras, Vlad Radulescu

No. 1, January 2012
An Efficient Scheduling Algorithm for the Multiprocessor Platform

Ștefan Andrei
Lamar University
Department of Computer Science
Beaumont, TX, USA
Email: sandrei@cs.lamar.edu

Albert M.K. Cheng
University of Houston
Computer Science Department
Houston, TX, USA
Email: cheng@cs.uh.edu

Gheorghe Grigoraș
Cuza University of Iași
Department of Computer Science
Iași, Romania
Email: grigoras@infoiasi.ro

Vlad Rădulescu
Cuza University of Iași
Department of Computer Science
Iași, Romania
Email: rvlad@infoiasi.ro

Abstract—Given a task set \( T \), determining the number of processors leading to a feasible schedule for \( T \) is an important problem in the real-time embedded systems community. For periodic and independent task sets, the utilization rate represents a lower bound on the number of processors. A multiprocessor platform with fewer processors than the utilization rate of a given task set does not have a feasible schedule.

To the best of our knowledge, there is no estimation on the lower bound of the number of processors for a single-instance task set. The contribution of this paper is two-fold. Firstly, given a single-instance, non-preemptive and independent task set, we provide a lower bound on the number of processors such that there exists no feasible schedules on a multiprocessor platform with fewer processors than this lower bound. Secondly, we provide an efficient algorithm that finds a feasible schedule of a single-instance non-preemptive and independent task set on a multiprocessor platform having the number of processors equal to the lower bound.

I. INTRODUCTION

Finding optimal feasible schedules of task sets under various constraints has always been an important research area in the real-time embedded systems community. Stankovic, Spuri, Di Natale, and Butazzo investigated the boundary between polynomial and NP-hard scheduling problems [19]. 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 (i.e., a task is preemptive if its execution can be interrupted and resumed later) scheduling problem [8]. Mok discovered another optimal algorithm with polynomial complexity for the same subclass, that is, the Least Laxity First (LLF) algorithm [17]. Another polynomial algorithm was found by Lawler in 1983 for non-preemptive (i.e., a task is non-preemptive if its execution cannot be interrupted) unit computation time tasks with arbitrary start time [16]. However, according to Graham, Lawler, Lenstra and Kan [11], when dealing with non-preemptive and non-unit computation time tasks, the scheduling problem becomes NP-hard.

Scheduling non-preemptive tasks has received less attention in the real-time embedded systems community than preemptive scheduling. Despite this, non-preemptive scheduling is widely used in industry [14]. For example, non-preemptive scheduling algorithms have lower overhead than the corresponding preemptive scheduling algorithms because of the inter-task interference caused by caching and pipelining. The benefits of using non-preemptive tasks versus preemptive tasks are multiple on multiprocessor platforms, since the task migration overhead is higher and difficult to predict. This problem has less negative impact on non-preemptive scheduling because each task instance runs until its completion on the same processor and
task migration might occur at task instance boundaries. This paper first describes a criterion to determine the minimum number of processors for which there exists a feasible schedule. Given a task set $T$, determining the number of processors for finding a feasible schedule for $T$ is an important problem in the real-time embedded systems community. For periodic and independent (i.e., tasks are independent if there are no precedence constraints) task sets, the utilization rate represents a lower bound on the number of processors. In other words, a multiprocessor platform with fewer processors than this lower bound does not have a feasible schedule.

There exist many models to define a task. In this paper, we consider a task as 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’ periods. Thus, the notions of task, task instance and job are equivalent and can be interchangeably used. In fact, the results and examples from this paper can be 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 $(s, c, d)$, meaning that $T$ can be executed after time $s$ completing a total of $c$ time units by the deadline $d$. Given a task set $T = \{T_1, \ldots, T_n\}$, $T$ is called schedulable by a scheduling algorithm $Sch$ if $Sch$ ensures that the timing constraints of all tasks in $T$ are met. Algorithm $Sch$ is called optimal if whenever $Sch$ cannot find a schedule, then no other scheduling algorithm can [6]. These definitions will be detailed in Section III for multiprocessor platforms.

To the best of our knowledge, there is currently no available method for estimating the lower bound on the number of processors for meeting the deadlines of a single-instance task set. The contribution of this paper is two-fold. First, given a single-instance non-preemptive independent task set, we provide a lower bound on the number of processors such that there exists no feasible schedules on a multiprocessor platform with fewer processors than this lower bound. Second, we provide an efficient algorithm that finds a feasible schedule for any given single-instance, non-preemptive and independent task set on a multiprocessor platform having the number of processors equal to this lower bound. Our scheduling algorithm is able to provide feasible schedules for task sets for which the EDF and LLF methods cannot identify any schedule.

The remainder of this paper is organized as follows. Section II presents the related work with our paper. Section III describes the necessary definitions, notations and related results. Section IV defines our estimation of the number of processors for meeting the deadlines of a single-instance non-preemptive and independent task set. Section V describes an efficient algorithm to determine a schedule for a given task set, if one exists. Section VI concludes this paper.

II. RELATED WORK

Kubale proved that the scheduling problem for preemptive tasks on platforms with unbounded number of processors is strongly NP-hard even when each task requires the simultaneous use of at most 2 processors [15]. However, he showed that if the number of processors is fixed, the above mentioned problem can be solved using linear programming. Amoura et al. used a linear programming formulation to solve the above scheduling problem with a fixed number of processors in linear time [1].

Blakewicz et al. showed that the scheduling problem of non-preemptive tasks on a three-processor platform is strongly NP-hard [3]. An approximation algorithm with a performance ratio of $\frac{7}{6}$ for the above problem was proposed by Goemans [10]. The particular case where each task has a unit computation time leads to a polynomial time algorithm that solves the scheduling problem [13]. However, if the number of processors is an input variable, then the scheduling problem becomes NP-hard [13]. A polynomial time approximation scheme to solve the above scheduling problem for non-preemptive independent tasks was presented in [1].

Hence, it is obvious that the number of processors is indeed an important parameter for the scheduling problem. Our paper describes a method to calculate the lower bound of the number of processors for non-preemptive, independent, and single-instance tasks. We consider the number of processors to be fixed, however it is not required that the tasks have a unit computation time.

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 NP-
complete in most of the cases where the corresponding scheduling problem can be solved by a polynomial-time algorithm for the uniprocessor platforms [17].

Jeffay, Stanat, and Martel [14] proved that the scheduling problem for periodic non-preemptive task sets is NP-hard in the strong sense for uniprocessor platform. Cai and Kong [4] extended this complexity result for the task sets with arbitrary start times. As expected, the scheduling problem for multiprocessor platforms remains intractable.

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 [7].

Guan et al. studied the schedulability analysis problem for sporadic task sets on identical multiprocessors with non-preemptive scheduling [12]. A similar work was done by Baruah [2] for non-preemptive EDF scheduling on a multiprocessor platform. Carpenter et al. analyzed the trade-offs involved in scheduling independent, periodic real-time tasks on a multiprocessor platform [5]. A similar work was done by Dolev and Keizelman for non-preemptive tasks scheduling where there is no information about the tasks before they are released [9].

III. PRELIMINARIES

We present some concepts and notations to allow the remainder of the paper to be self-contained, by including some results and examples.

There exists a few different, but similar, formulations for the scheduling problem. Although these formulations are in general equivalent, they might highlight some aspects rather than other. Our paper considers the multiprocessor platform, independent preemptive tasks, and no shared resources or overhead.

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 lying 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 $[s, e)_p$ if $T$ is ready to execute on processor $p$ at time $s$ and finishes its execution before time $e$, allowing the next task to start its execution on processor $p$ at time $e$. The set with no elements is called the empty set and is denoted by $\emptyset$. We say that $[s, e)_p \cap [s', e')_p = \emptyset$ if and only if either $s_1 \neq s_2$ or $[s, e) \cap [s', e') = \emptyset$ in the mathematical sense (i.e., $[s, e) \cap [s', e') = \{x \mid x \in [s, e) \text{ and } x \in [s', e')\}$). We consider in our paper that the multiprocessor platform is simply denoted as $P = \{p_1, \ldots, p_m\}$, where $p_1, \ldots, p_n$ are processors. This notation may be omitted in the context of multiprocessor platform. For a finite set $V$, we denote by $|V|$ the number of elements of $V$.

Without loss of generality, 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 task set denoted as $T$ given by $\{T_1, \ldots, T_n\}$, where each task $T_i$ is represented by $(s_i, c_i, d_i)$. According to [19], if each task has a deadline, the scheduling problem for the multiprocessor environment becomes difficult. This is actually one of the key points that make the scheduling problem for multiprocessor platforms difficult. Definition 3.1 defines the execution assignment for the time interval $[0, D)$, where $D = \max\{d_i \mid T_i \in T\}$. 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 [18]. The schedule there is represented as a predicate, whereas the execution assignment is expressed as a union of intervals.

Definition 3.1: Let us consider a task set $T = \{T_1, \ldots, T_n\}$, where each task $T_i$ is given by $(s_i, c_i, d_i)$. We say that the task set $T$ is schedulable on a multiprocessor platform $P = \{p_1, \ldots, p_m\}$ if there exists an execution assignment (also known as schedule) denoted by $EA : T \rightarrow [0, D)$, where in general $(s, e)_p \in EA(T)$ means the task $T$ executes on processor $p$ in the time interval from time $s$ to time $e$, and satisfies the following two properties:

1) $\forall \ i \in \{1, \ldots, k\}$, we have $EA(T_i) = [s_i^{(1)}, e_i^{(1)})_{\{p_1, \ldots, p_{i-1}\}} \cup \ldots \cup [s_i^{(n_i)}, e_i^{(n_i)})_{\{p_{i+1}, \ldots, p_m\}}$, where $p_{i,1}, \ldots, p_{i,n_i}$ are processors from $P, s_i^{(1)} < e_i^{(1)} \leq \ldots \leq s_i^{(n_i)} < e_i^{(n_i)}, \sum_{j=1}^{n_i} (e_i^{(j)} - s_i^{(j)}) = c_i, s_i \leq s_i^{(1)}$ 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 $EA(T_i) \cap EA(T_j) = \emptyset$.

Similar to the approach from [6], the scheduling problem presented in Definition 3.1 assumes that the task constraints are known in advance, such as deadlines, computation times, and start times. This framework is called static scheduling [19]. Static scheduling is contrasting dynamic scheduling, where the con-
straints 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 [17]. Definition 3.1 also assumes that there is no context-switching time. Given \( [s_i^{(u_i)}, e_i^{(u_i)}] \) in \( EA(T_i) \) and \( [s_j^{(u_j)}, e_j^{(u_j)}] \) in \( EA(T_j) \), where \( p_i \) and \( p_j \) are processors, such that \( e_i^{(u_i)} = s_j^{(u_j)} \), we say that task \( T_j \) is executed immediately after \( T_i \).

Using the notation of Definition 3.1, we say that tasks \( T_i \) are preemptive if \( n_i \geq 1 \) (the case \( n_i = 1 \) corresponds to non-preemptive tasks). Note that the inequalities above are not contradictory, but instead the preemptive case includes the non-preemptive case. This is because a non-preemptive task \( T_i \) cannot have \( n_i > 1 \), but a preemptive task \( T_i \) may have \( n_i = 1 \). In other words, even if a task is preemptive, it might happen to run without being interrupted.

Another special case is when the tasks have unit computation time. According to Definition 3.1, 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 [16]. However, according to Graham, Lawler, Lenstra and Kan, the scheduling problem with non-preemptive and non-unit computation time tasks becomes NP-hard [11].

For simplicity, we shall consider that the tasks have their starting time equal to zero, namely all the tasks are ready to be executed from the beginning. From now on, a task will be denoted as \( (c, d) \), where \( c \) is its computation time and \( d \) is its deadline. Given a task set \( T = \{ T_1, \ldots, T_n \} \), where each task \( T_i \) is given by \( (c_i, d_i) \) for any \( i \in \{1, \ldots, n\} \), the EDF scheduling means the task with the earliest deadline, that is, \( d_i \), has the highest priority. The LLF scheduling means the task with the smallest laxity, that is, \( l_i = d_i - c_i \), has the highest priority.

IV. DETERMINING A LOWER BOUND ON THE NUMBER OF PROCESSORS FOR A MULTIPROCESSOR PLATFORM

The next result determines a lower bound of the number of processors on which a task set has the chance to be scheduled on that multiprocessor platform.

\[ \text{Theorem 4.1:} \quad \text{Let} \ \mathcal{T} = \{ T_1, \ldots, T_n \} \text{ be a single-instance non-preemptive and independent task set, where each task} \ T_i \text{ is given by} \ (c_i, d_i) \ \text{for} \ i \in \{1, \ldots, n\}. \ \text{The tasks are increasingly sorted according to their deadlines, that is,} \ d_1 \leq \ldots \leq d_n. \ \text{Let} \ \text{USI}_1 = \lceil \frac{c_1 + \ldots + c_n}{d_1} \rceil \ \text{be the partial Utilization of Single-Instance task set} \ \{ T_1, \ldots, T_n \}. \ \text{Let us denote by} \ \text{USI} \ \text{the maximum of} \ \text{USI}_1, \ldots, \text{USI}_n, \ \text{and call it the Utilization of Single-Instance task set. Then} \ \mathcal{T} \text{ is} \ \text{not schedulable on a multiprocessor platform with USI} - 1 \ \text{processors or less.} \]

\[ \text{Proof:} \quad \text{We assume that} \ \mathcal{T} \text{ is a schedulable task set on a} \ \text{USI} - 1 \ \text{multiprocessor platform. Let us consider} \ k \in \{1, \ldots, n\} \ \text{such that} \ \text{USI} = \text{USI}_k, \ \text{that is, an index that corresponds to the maximum of} \ \text{USI}_1, \ldots, \text{USI}_n. \ \text{We focus our attention on the subset of tasks} \ \{ T_1, \ldots, T_n \}. \ \text{Task} \ T_i \text{ needs to utilize} \ c_i \ \text{time units by the deadline} \ d_i. \ \text{Since the task set} \ \mathcal{T} \ \text{is schedulable, it means that the sum of the tasks utilizations should be less than or equal to the number of processors. It follows that} \ \frac{c_1}{d_1} + \ldots + \frac{c_k}{d_k} \leq \text{USI} - 1. \ \text{For any} \ i \in \{1, \ldots, k\}, \ \text{we have} \ d_i \leq d_k, \ \text{hence it results that} \ \frac{c_i}{d_i} \geq \frac{c_k}{d_k}. \ \text{Therefore,} \ \frac{c_1}{d_1} + \ldots + \frac{c_k}{d_k} \geq \frac{c_1 + \ldots + c_k}{d_k} \geq \text{USI}. \ \text{This contradicts the previous inequality. In conclusion, the task set} \ \mathcal{T} \ \text{is not schedulable on a} \ \text{USI} - 1 \ \text{multiprocessor platform.} \]

\[ \text{Theorem 4.1 can also be proved directly, avoiding the ‘reduction ad absurdum’ method. Since the tasks are increasingly sorted after their deadlines, it means that} \ c_1 + \ldots + c_i \ \text{is the total unit time needed for one processor to execute the tasks whose deadlines are less than or equal to} \ d_i. \ \text{This implies that for} \ N \ \text{processors, the total time available by} \ d_i \ \text{is} \ N \cdot d_i. \ \text{If} \ N < (c_1 + \ldots + c_i)/d_i, \ \text{then} \ N \cdot d_i < c_1 + \ldots + c_i, \ \text{hence it is impossible for all tasks} \ T_1, \ldots, T_n \ \text{to meet their deadlines by} \ d_i. \ \text{As a consequence, it follows that} \ N \geq (c_1 + \ldots + c_i)/d_i, \ \text{that is,} \ N \geq [(c_1 + \ldots + c_i)/d_i] = \text{USI}_i. \ \text{Since this inequality holds for all} \ i \in \{1, \ldots, n\}, \ \text{it follows that we cannot have fewer than} \ \text{USI} \ \text{processors to schedule the task set} \ \{ T_1, \ldots, T_n \}. \]

The following example motivates why we need to consider the maximum of all partial \( \text{USI}_i \), where \( i \in \{1, \ldots, n\} \).

\[ \text{Example 4.1:} \quad \text{Let} \ \mathcal{T} = \{ T_1, T_2, T_3, T_4 \} \ \text{be a single-instance and non-preemptive task set given by:} \ T_1 = (1, 1), \ T_2 = (2, 2), \ T_3 = (2, 2), \ \text{and} \ T_4 = (3, 5). \ \text{It is easy to calculate} \ \text{USI}_1 = 1, \ \text{USI}_2 = 2, \ \text{USI}_3 = 3, \ \text{and} \ \text{USI}_4 = 2. \ \text{According to Theorem 4.1, a multiprocessor platform with 2 processors or} \]

less cannot schedule \( \mathcal{T} \). In addition, this example demonstrates that \([USI_1, ..., USI_n]\) is not necessarily an increasing list of values.

The following example shows a task set for which the EDF method fails to provide a schedule on a four-processor platform.

**Example 4.2:** Let \( \mathcal{T} = \{T_1, ..., T_{12}\} \) be a single-instance and non-preemptive task set given by: \( T_1 = (1, 1), T_2 = (1, 2), T_3 = (2, 3), T_4 = (2, 3), T_5 = (2, 4), T_6 = (2, 4), T_7 = (4, 5), T_8 = (1, 5), T_9 = (1, 5), T_{10} = (1, 5), T_{11} = (2, 5), \) and \( T_{12} = (1, 5) \). The deadlines are sorted increasingly, hence we get \( USI_1 = 1, USI_2 = 1, USI_3 = 2, USI_4 = 2, USI_5 = 2, USI_6 = 3, USI_7 = 3, USI_8 = 3, USI_9 = 4, USI_{10} = 4, USI_{11} = 4, \) and \( USI_{12} = 4 \). Hence \( USI = \max\{USI_1, ..., USI_{12}\} = 4 \). The EDF method fails to provide a feasible schedule for \( \mathcal{T} \) on a four-processor platform. This is because tasks \( T_1, T_2, T_3, \) and \( T_4 \) will be chosen to be first executed on processors 1, 2, 3, and 4, respectively. Then, \( T_5 \) and \( T_6 \) will be scheduled for processors 1 and 2. Therefore, task \( T_7 \) will miss its deadline because all the processors have left less than 3 available time units to execute.

The following example shows a task set for which the LLF method fails to provide a schedule on a two-processor platform.

**Example 4.3:** Let \( \mathcal{T} = \{T_1, ..., T_6\} \) be a single-instance and non-preemptive task set given by: \( T_1 = (2, 2), T_2 = (2, 2), T_3 = (3, 6), T_4 = (3, 6), T_5 = (1, 5), \) and \( T_6 = (1, 5) \). The laxities are sorted increasingly, namely \( l_1 = 0, l_2 = 0, l_3 = 3, l_4 = 3, l_5 = 4, \) and \( l_6 = 4 \). It is easy to see that \( USI = 2 \). The LLF method fails to provide a feasible schedule for \( \mathcal{T} \) on a two-processor platform. According to the LLF scheduling strategy, tasks \( T_1 \) and \( T_2 \) will be chosen to be first executed on processors 1 and 2, respectively. Then, \( T_3 \) and \( T_4 \) will be scheduled on processors 1 and 2, respectively. As a result of the LLF strategy, both tasks \( T_5 \) and \( T_6 \) cannot be scheduled because they will miss their deadline of 5. Hence, the LLF scheduling method fails to provide a feasible schedule for the above task set.

The task set of Example 4.2 does not have an EDF schedule, but it has a LLF schedule. On the other hand, the task set of Example 4.3 does not have a LLF schedule, but it has an EDF-schedule. The next example represents a task set that has neither an EDF schedule nor a LLF schedule.

**Example 4.4:** Let \( \mathcal{T} = \{T_1, ..., T_7\} \) be a single-instance and non-preemptive task set given by: \( T_1 = (2, 2), T_2 = (7, 7), T_3 = (8, 9), T_4 = (3, 6), T_5 = (1, 5), T_6 = (5, 12), \) and \( T_7 = (3, 11) \). It is easy to see that \( USI = 3 \). The laxities are sorted increasingly, hence the LLF scheduler will first assign tasks \( T_1, T_2, \) and \( T_3 \) to be executed on processors 1, 2, and 3, respectively. Then, task \( T_4 \) will be scheduled for processor 1. Hence, task \( T_5 \) will miss its deadline.

By following the EDF scheduling method, \( T_1, T_5, \) and \( T_4 \) will be assigned to processors 1, 2, and 3, respectively. Hence, task \( T_2 \) will miss its deadline. In conclusion, the task set \( \mathcal{T} \) is neither LLF-schedulable nor EDF-schedulable.

## V. A Scheduling Algorithm

We describe in this section our scheduling algorithm, called Algorithm A, that takes a single-instance non-preemptive and independent task set and returns a feasible schedule, if one exists.

We define first the ordering relation for the task sets. This ordering relation is actually based on task laxities, i.e., \( l = d - c \), where \( T = (c, d) \) is a given task.

**Definition 5.1:** Given two tasks \( T_1 = (c_1, d_1) \) and \( T_2 = (c_2, d_2) \), we say that \( T_1 < T_2 \) if \( d_1 - c_1 < d_2 - c_2 \) or \( (d_1 - c_1 = d_2 - c_2 \) and \( d_1 < d_2) \). We say that \( T_1 = T_2 \) if \( c_1 = c_2 \) and \( d_1 = d_2 \). We say that \( T_1 \leq T_2 \) if \( T_1 < T_2 \) or \( T_1 = T_2 \).

For example, given \( T_1 = (1, 3), T_2 = (2, 5), \) and \( T_3 = (2, 4) \), we have \( T_1 < T_2, T_1 \leq T_3, \) and \( T_3 \leq T_2 \).

Next, we define a task order restriction that might influence the order of two tasks \( T_1 \) and \( T_2 \) such that \( T_1 \leq T_2 \) as specified in Definition 5.1.

**Definition 5.2:** Given two tasks \( T_1 = (c_1, d_1) \) and \( T_2 = (c_2, d_2) \) such that \( T_1 \leq T_2 \), we say that \( T_1 \not< x T_2 \) if \( d_2 < x + c_1 + c_2 \leq d_1 \). Given a task set \( \mathcal{T} = \{T_1, ..., T_n\} \), we denote by \( TOR(\mathcal{T}) = \{T_i \not< x T_j \mid 1 \leq i < j \leq n, x \geq 0\} \).

Definition 5.2 says that task \( T_2 \) cannot be executed after task \( T_1 \) on the same processor as \( T_1 \) if the computation time of all tasks prior to \( T_1 \) is greater or equal than \( x \). In other words, the relation \( x \not< T \) holds if \( T_2 \) cannot be executed after \( T_1 \) in time \( x \). In fact, task \( T_1 \) may be executed after task \( T_2 \) under the above conditions. The ordering relation \( \not< x \) is useful for the cases when the LLF method cannot provide a schedule and the EDF method can be applied instead.

**Example 5.1:** Let us consider the task set from Example 4.4. The laxities \( l_i = d_i - c_i \) for all \( i \in \{1, ..., 10\} \).
are, in order, \( l_1 = 0 \), \( l_2 = 0 \), \( l_3 = 1 \), \( l_4 = 3 \), \( l_5 = 4 \), \( l_6 = 7 \), and \( l_7 = 8 \). Hence, the task set is already increasingly sorted after their laxities. According to Definition 5.2, \( \text{TOR}(T) = \{ T_3 \not\in 0 \ T_3 \not\in 2 \ T_5 \not\in 4 \ T_6 \not\in 4 \ T_7 \}. \)

As a remark for Example 5.1, the value of \( x \) for each of the three task order restrictions was unique. However, it may be possible to have more than just one value. For example, given \( T_1 = (6, 13) \) and \( T_2 = (3, 11) \), it is clear that \( T_1 \leq T_2 \). In order to investigate what values \( x \) can get, we consider the inequality from Definition 5.2, that is, \( 11 < x + 6 + 3 \leq 13 \). Hence, \( x \) may take two possible values, namely \( x = 3 \) and \( x = 4 \).

The ordering relations “\( \leq \)” and “\( \not\in \)” between tasks will be used in Algorithm A. In addition, we consider a chain of tasks \( C \) as \( [T_1, \ldots, T_k] \), a list of tasks from the given task set \( T \). We denote the computation of the chain \( c(C) \) as \( c(T_1) + \ldots + c(T_k) \) and \( last(C) \) as \( T_k \). We denote by \( C - last(C) \) the chain \( C \) obtained after removing its last task. We denote by \( C + T \) the chain obtained by catenating \( C \) and task \( T \). Given a task set, the algorithm below tries to generate a schedule on a multiprocessor platform. It can be viewed as a beneficial combination of two “complementary” scheduling techniques, LLF and EDF. For example, the algorithm below will be able to generate schedules for all task sets from Examples 4.2, 4.3, and 4.4, for which EDF or LLF or both were not able to provide schedules.

**Algorithm A**

**The input:** A set of single-instance non-preemptive and independent tasks \( T = \{ T_1, \ldots, T_n \} \), where each task \( T_i = (c_i, d_i) \), for all \( i \in \{ 1, \ldots, n \} \), such that \( d_1 \leq \ldots \leq d_n \).

**The output:** A schedule for the task set \( T \) on a platform having \( USI \) processors, if \( \text{T} \) is feasible. Otherwise, display that \( \text{T} \) is infeasible.

**The method:**

1. \( USI = [c_i / d_i] \);
2. for \( (i=2; i \leq n; i++) \) {  
   3. \( USI_i = [(c_i + \ldots + c_i) / d_i] \);  
   4. if \( (USI < USI_i) \) \( USI = USI_i \);  
5. Sort lexicographically tasks \( T_1, \ldots, T_n \) under \( d_1 - c_i \) as a primary key and \( d_i \) as a second key. The obtained list is \( T' = \{ T_{\pi(1)}, \ldots, T_{\pi(n)} \} \), where \( \pi \) is the corresponding permutation such that \( T_{\pi(i)} \leq T_{\pi(i+1)} \), for all \( i \in \{ 1, \ldots, n - 1 \} \).
6. \( \text{TOR}(T') = \emptyset \);
7. for \( (i=1; i < n; i++) \)  
8. if \( (d(T_{\pi(i)}) - c(T_{\pi(i)}) > 0) \)  
9. for \( (j=i+1; j \leq n; j++) \)  
10. if \( (\exists x \geq 0 \text{ such that } d(T_{\pi(j)}) < x + c(T_{\pi(i)}) + c(T_{\pi(j)}) \leq d(T_{\pi(i)}) \)  
11. \( \text{TOR}(T') = \text{TOR}(T') \cup \{ T_{\pi(i)} \not\in x T_{\pi(j)} \} \);
12. Choose \( T_{\pi(1)}, \ldots, T_{\pi(USI)} \) from the list \( T' \);
13. Remove \( T_{\pi(1)}, \ldots, T_{\pi(USI)} \) from the list \( T' \);
14. \( \text{feasible} = \text{true} \);
15. while \( (T' \) is a non-empty set \&\& feasible) \{  
16. Let \( T \) be the first task from \( T' \);
17. Choose a chain \( C \) such that \( c(T) + c(C) \leq d(T) \);
18. if \( (\text{such a chain exists}) \) \{  
19. Remove \( T \) from \( T' \);
20. Add \( T \) to chain \( C \);
21. \} else \{  
22. Choose a chain \( C \) such that \( \text{last}(C) \not\in T \) is in \( \text{TOR}(T') \), where \( x = c(C - \text{last}(C)) \);
23. if \( (\text{such a chain exists}) \) \{  
24. Add \( T \) to chain \( C \), but switch \( T \) and \( \text{last}(C) \);
25. Remove \( T \) from \( T' \);  
26. \} else \{  
27. Print \( T' \) is infeasible on a USI-processor platform';
28. \( \text{feasible} = \text{false} \);
29. \} \}
30. Print \( T' \) is feasible on a USI-processor platform';
31. Print all chains \( C \) representing the schedule.
}

The following example considers the task set from Example 4.2 by showing how Algorithm A can be applied to generate a feasible schedule.

**Example 5.2:** Let us consider the task set from Example 4.2. The first four steps of Algorithm A will calculate \( USI \) as 4. Then step 5 will consider the laxities \( l_i = d_i - c_i \) for all \( i \in \{ 1, \ldots, 12 \} \), the primary key for sorting the list of tasks. These laxities are, in order, \( l_1 = 0 \), \( l_2 = 1 \), \( l_3 = 1 \), \( l_4 = 1 \), \( l_5 = 2 \), \( l_6 = 2 \), \( l_7 = 1 \), \( l_8 = 4 \), \( l_9 = 4 \), \( l_{10} = 4 \), \( l_{11} = 3 \), and \( l_{12} = 4 \). Therefore, the task set is rewritten as a permutation \( T' = \{ T_1, T_2, T_3, T_4, T_7, T_5, T_6, T_{11}, T_8, T_9, T_{10}, T_{12} \} \). The task order restriction set is empty, that is, \( \text{TOR}(T) = \emptyset \). The first four chosen tasks are, in order: \( C_1 = [T_1], C_2 = [T_2], C_3 = [T_3], \) and \( C_4 = [T_4] \). They represent the roots of the chains of computation times \( c[C_1] = 1, c[C_2] = 1, c[C_3] = 2, \) and \( c[C_4] = 2 \), respectively. Each chain will be executed on a distinct processor (for simplicity, we assume chain \( C_1 \) is executed on processor 1, and so on).
According to Algorithm A, the chains subject to be extended are $C_1$ and $C_2$, as they have the minimum computation cost. The candidates are clearly $T_7$ and $T_9$. We get the following chains: $C_1 = [T_1, T_2]$ of cost 5, $C_2 = [T_3, T_5, T_9]$ of cost 3, $C_3 = [T_3, T_5]$ of cost 2, and $C_4 = [T_1]$ of cost 2. Note that chain $C_1$ is now final (that is, it cannot be further extended) as it has a computation time of 5. We now extend chain $C_3$ with $T_6$ and chain $C_4$ with $T_{11}$. Then chain $C_2$ is extended with $T_8$. Lastly, chain $C_2$ can be extended with $T_9$, chain $C_3$ with $T_{10}$, and chain $C_4$ with $T_{12}$. Hence, the task set $T'$ is schedulable and has the following schedule given by the below four chains, all having a computation time of 5: $C_1 = [T_1, T_7]$, $C_2 = [T_2, T_5, T_8, T_9]$, $C_3 = [T_3, T_5, T_{10}]$, and $C_4 = [T_4, T_{11}, T_{12}]$. The execution assignment of the task set $T'$, as well as $T$, is given by: $EA(T_1) = \{ [0, 1] \}$, $EA(T_2) = \{ [1, 5] \}$, $EA(T_3) = \{ [1, 3] \}$, $EA(T_5) = \{ [4, 5] \}$, $EA(T_6) = \{ [2, 4] \}$, $EA(T_7) = \{ [0, 1] \}$, $EA(T_8) = \{ [3, 4] \}$, $EA(T_9) = \{ [0, 2] \}$, $EA(T_{10}) = \{ [4, 5] \}$, $EA(T_{11}) = \{ [2, 4] \}$, and $EA(T_{12}) = \{ [4, 5] \}$.

Even if our strategy and the Least Laxity First (LLF) scheduling technique share many similarities, they are however different. The main differences between them are:

1. Algorithm A above calculates laxities only once whereas the LLF technique needs to update their laxities at every step of generating the schedule.
2. The LLF strategy solves the ties randomly, whereas Algorithm A has a more refined and precise way to solve the non-deterministic schedule, namely the task having the earliest deadline is chosen.
3. Even if the task priorities are initially decided by the task laxities, our technique consider changing a task priority based on the ordering relation $
eq$ defined over the task set. Example 4.4 represents a task set that is neither EDF nor LLF schedulable, but it can be scheduled by our algorithm (the schedule is given in Example 5.3).

Example 4.2 describes a task set for which the EDF technique does not provide a feasible schedule, but Algorithm A does. Hence Algorithm A is able to provide feasible schedules for a larger class of task sets. In fact the next result proves that Algorithm A is correct and efficient. Algorithm A takes the advantages of both LLF and EDF techniques by taking into account first the laxities and solving the tie cases using their earlier deadline. The following three examples represent instances of task sets supplied as input to Algorithm A and provide various answers.

**Example 5.3:** Let $T = \{ T_1, ..., T_7 \}$ be the task set from Example 4.4. Example 4.4 indicated that this task set $T$ is neither LLF-schedulable nor EDF-schedulable. We determined in Example 5.1 that $TOR(T) = \{ T_3 \not\rightarrow 0, T_3 \not\rightarrow 2, T_5 \not\rightarrow 4 \}$. Algorithm A will only use the task order restriction $T_4 \not\rightarrow 2$ $T_5$, hence $T_4$ and $T_5$ are switched when building the schedule. The reason for which Algorithm A does not need the other two task order restrictions is because the tasks $T_3$ and $T_5$, as well as the tasks $T_5$ and $T_7$, are executed on different processors. The schedule provided by Algorithm A is given by the following chains: $C_1 = [T_1, T_5, T_4, T_6]$, $C_2 = [T_2, T_7]$, and $C_3 = [T_3]$.

The next example is a variation of Example 5.2 for which tasks $T_8$ and $T_9$ are merged.

**Example 5.4:** Let $T = \{ T_1, ..., T_{13} \}$ be a single-instance and non-preemptive task set given by: Let $T = \{ T_1, ..., T_{12} \}$ be a single-instance and non-preemptive task set given by: $T_1 = (1, 1), T_2 = (1, 2), T_3 = (2, 3), T_4 = (2, 3), T_5 = (2, 3), T_6 = (2, 4), T_7 = (4, 5), T_8 = (2, 4), T_9 = (1, 5), T_{10} = (2, 5), T_{11} = (1, 5)$. Obviously, $USI = 4$. The EDF method fails again to provide a feasible schedule for $T$ on a four-processor platform because of the same reason explained in Example 4.2. By applying Algorithm A, the task set $T$ is schedulable and has the schedule given by the four chains below, all having a computation time of 5: $C_1 = [T_1, T_7]$, $C_2 = [T_2, T_5, T_{10}]$, $C_3 = [T_3, T_6, T_9]$, and $C_4 = [T_4, T_{11}, T_{12}]$. The execution assignment of the task set $T$ is given by: $EA(T_1) = \{ [0, 1] \}$, $EA(T_2) = \{ [1, 5] \}$, $EA(T_3) = \{ [1, 3] \}$, $EA(T_4) = \{ [3, 5] \}$, $EA(T_5) = \{ [0, 2] \}$, $EA(T_6) = \{ [2, 4] \}$, $EA(T_7) = \{ [4, 5] \}$, $EA(T_8) = \{ [0, 2] \}$, $EA(T_9) = \{ [2, 4] \}$, and $EA(T_{11}) = \{ [4, 5] \}$. The LLF method will also be able to provide the same feasible schedule as Algorithm A. This is because Algorithm A did not use any tasks inversion of type $\not\rightarrow$. 

The next example represents an infeasible task set.

**Example 5.5:** Let $T = \{ T_1, ..., T_5 \}$ be a single-instance and non-preemptive task set given by: $T_1 = (1, 1), T_2 = (1, 2), T_3 = (2, 3), T_4 = (2, 3), T_5 = (2, 4), T_6 = (2, 4), T_7 = (4, 5), T_8 = (3, 5)$, and $T_9 = (3, 5)$. Obviously, $USI = 4$. The EDF method
fails again to provide a feasible schedule for $\mathcal{T}$ on a four-processor platform because of the same reason explained in Example 4.2.

Similarly, LLF fails to provide a feasible schedule for a four-processor platform. According to their laxities, tasks $T_1$, $T_2$, $T_3$, and $T_4$ will be first assigned to be executed by processors 1, 2, 3, and 4, respectively. Then $T_5$, $T_6$, and $T_8$ will be assigned, in this order, to processors 1, 2, 3, and 4. The chains of tasks have now the following execution times: 5, 3, 3, and 5, respectively. Hence none of them can execute task $T_9$ for 3 time units without missing its deadline of 5.

By applying Algorithm A, the obtained answer is $\mathcal{T}$ is infeasible on a 4-processor platform’. This is because there is no schedule for the above task set. Note that USI was replaced in the above string by 4.

Provided that we do not change the execution times and deadlines of the task set from Example 5.5, one way to have the task set feasible is to consider a five-processor platform instead of a four-processor platform. Obviously, it can be easily checked that all three scheduling algorithms considered for comparison in this paper, i.e., EDF, LLF, and Algorithm A, provide a feasible schedule on a five-processor platform.

VI. Conclusions

Given a single-instance non-preemptive and independent task set, we determined a lower bound of the number of processors for which there exists no feasible schedules on a multiprocessor platform. In addition, we provided an efficient algorithm that finds a feasible schedule for the single-instance non-preemptive and independent task set on a multiprocessor platform having the number of processors equal to the lower bound, if one exists. Our algorithm represents a better alternative than EDF and LLF scheduling strategies for the single-instance, non-preemptive and independent task sets on a multiprocessor platform.

References


