Estimating the number of processors towards an efficient non-preemptive scheduling algorithm

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

No. 2, April 2012
Abstract—Given a task set $\mathcal{T}$, finding a feasible schedule for $\mathcal{T}$ is one of the most significant problems of real-time embedded systems. The research community has found a lot of important results for the scheduling problem on both uniprocessor and multiprocessor platforms. To the best of our knowledge, there is still room for research in determining the number of processors for a given task set, where each task is specified by its starting time, computation time, and deadline.

The present paper describes an improved lower bound on the number of processors that allows a feasible schedule for a single-instance, non-preemptive, and independent task set $\mathcal{T}$. In addition, an efficient algorithm is described, in order to achieve a feasible schedule for $\mathcal{T}$ for the real-time system with the number of processors equal to the lower bound previously determined.

Both the above facts are extensions of the results presented in [3]. Among many improvements, one of the main differences consists in considering the start times of the tasks when defining the task set, hence by allowing the considered model to cover more practical situations.

Keywords: scheduling problem; multiprocessor platform; lower bound of the number of processors; scheduling algorithm.

I. INTRODUCTION

The scheduling problem represents one of the most studied problems in the real-time embedded systems community. The main interest related to the scheduling problem is finding optimal feasible schedules of task sets under various constraints. Despite that, another important problem is estimating the number of processors for a platform for which the scheduling problem may be solvable. Given the scheduling problem, Dhall and Liu mentioned that the problem of determining the minimum number of processors is difficult [11]. They presented two heuristic algorithms that yield a number of processors that is reasonably close to the minimum number of processors.

Most of the theoretical results and efficient scheduling algorithms deal with the preemptive tasks, although in practice it is known that the non-preemptive tasks are used more than their preemptive counterparts. Scheduling non-preemptive tasks has received less attention in the real-time embedded systems community than preemptive scheduling. However, non-preemptive scheduling is widely used in industry [21]. 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. The scheduling problem for non-preemptive tasks is \(\mathcal{NP}\)-hard [21; 6; 12; 7; 4; 16; 27; 31; 19] and many researchers investigated whether various constraints or particular cases make it tractable [32; 26; 22; 15; 23; 17; 9; 18; 2].

Stankovic, Spuri, Di Natale, and Butazzo investigated the boundary between polynomial and \(\mathcal{NP}\)-hard scheduling problems [30]. There are only few subclasses of the general scheduling problem that have polynomial-time complexity optimal algorithms. Diotouzos 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 [10]. For the EDF strategy, the task with earliest deadline will be the next scheduled task. Mok discovered another optimal algorithm with polynomial complexity for the same subclass, that is, the Least Laxity First (LLF) algorithm [28]. For the LLF strategy, the task with lowest laxity (that is, the difference between deadline and computation time) will be the next scheduled task. 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 [25]. However, according to Graham, Lawler, Lenstra and Kan [14], 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 [28].

Kubale proved that the scheduling problem for preemptive tasks on platforms with unbounded number of processors is strongly \(\mathcal{NP}\)-hard even when each task requires the simultaneous use of at most 2 processors [24]. 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 \(\mathcal{NP}\)-hard [5]. An approximation algorithm with a performance ratio of \(\frac{7}{6}\) for the above problem was proposed by Goemans [13]. The particular case where each task has a unit computation time leads to a polynomial time algorithm that solves the scheduling problem [20]. However, if the number of processors is an input variable, then the scheduling problem becomes \(\mathcal{NP}\)-hard [20]. 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. Considering the determined lower bound to be the number of processors, an efficient algorithm for computing a feasible schedule is described.

Both of the above facts are extensions of the results presented in [3]. Among many improvements, one of the main differences consists in considering the start times of the tasks when defining the task set, hence by allowing the considered model to cover more practical situations. The next section describes a new lower bound on the number of processors for a multiprocessor platform and a comparison with the previous estimation from [3].

II. Determining a Lower Bound on the Number of Processors for a Multiprocessor Platform

Given a task set with arbitrary starting times, computation time, and deadlines, estimating the number of processors necessary for allowing a feasible schedule for a task set is still an open question. In this paper, we focus on the case of single-instance, non-preemptive, and independent task sets. Obviously, an upper bound is given by the number of tasks. Of a bigger interest, however, is to achieve a lower bound of the number of processors, which would lead to a more efficient utilization of resources.

An estimation of the lower bound has been proposed in [3] for the particular case where the start times of all tasks are 0 (i.e., start times can be ignored). We
briefly present the result from [3], which will be further extended for a more general case.

Let \( T = \{ T_1, ..., T_n \} \) be a single-instance, non-preemptive, and independent task set, where each task \( T_i \) is given by \((c_i, d_i)\) for \( i \in \{1, ..., n\} \), \( c_i \) being the computation time of task \( T_i \) and \( d_i \) being the deadline of task \( T_i \). The tasks are increasingly sorted according to their deadlines, that is, \( d_1 \leq ... \leq d_n \). For each \( i \in \{1, ..., n\} \), let \( USI_i = [(c_1 + ... + c_i)/d_i] \) be the partial Utilization of Single-Instance task set \( \{ T_1, ..., T_n \} \). Then \( T \) is not schedulable on a multiprocessor platform with less than \( USI = \max(USI_1, ..., USI_n) \) processors. \( USI \) is called the Utilization of Single-Instance task set \( T \).

A. Calculating a closer estimation than \( USI \)

When start times are considered, determining a lower bound of the number of processors becomes more restrictive. The formula described in [3] is based on the underlying assumption that the scheduling algorithm is capable of achieving a CPU utilization of 100\%. However, when start times are different from 0, there may be periods of time when the processors cannot be used at their full capacity, simply because there are fewer tasks to be executed. Thus, it is necessary to determine the amount of processor time required for executing the tasks, for each time interval determined by start times and deadlines. We first introduce two definitions necessary for expressing the new lower bound.

**Definition 2.1:** For each deadline \( d_i \), we define the requested execution time, denoted by \( R_i \), as the amount of processor time required for the execution of tasks until \( d_i \), in order to allow all tasks to meet their deadlines.

In order to compute \( R_i \), the following values must be considered:
- For each task \( j \) with \( d_j \leq d_i \), the whole computational cost \( c_j \) is considered.
- For each task \( j \) with \( d_j < d_i < d_i + c_j \), an execution time of \( c_j - (d_j - d_i) \) is required before \( d_i \); otherwise, the task will not have the chance to meet its deadline.

We assume the tasks are sorted by their deadlines. we then consider \( R_i = \sum_{j=1}^{i} c_j + \sum_{j=i+1, d_j < d_i + c_j} c_j - d_j \).

As a remark, the tasks for which \( d_j = d_i + c_j \) are not taken into consideration because \( c_j + d_i - d_j \) is a null quantity, hence it does not change the value of \( R_i \).

**Definition 2.2:** For each start time \( s_i \), we define the available execution time, denoted by \( A_i \), as the amount of time available for execution before \( s_i \); that is, the total time that the tasks with start times before \( s_i \) could be executed until \( s_i \) (provided there are enough processors in the real-time system).

The following values must be considered in order to compute \( A_i \):
- For each task \( j \) with \( d_j \leq s_i \), the whole computational cost \( c_j \) is considered.
- For each task \( j \) with \( d_j > s_i \) and \( s_j < s_i \), an execution time of \( \min(c_j, s_i - s_j) \) is available before \( s_i \).

In conclusion, we get:

\[
A_i = \sum_{j=1, d_j \leq s_i} c_j + \sum_{j=1, d_j > s_i, s_j < s_i} c_j - \min(c_j, s_i - s_j).
\]

**Theorem 2.1:** Let \( T = \{ T_1, ..., T_n \} \) be a single-instance, non-preemptive, and independent task set, where each task is given by \((s_i, c_i, d_i)\) for any \( i \in \{1, ..., n\} \). The tasks are increasingly sorted by their deadlines, that is, \( d_1 \leq ... \leq d_n \). For any \( i \in \{1, ..., n\} \), let \( EUSI_i = \max_{d_j > s_i, j \in \{1, ..., n\}} [(R_i - A_j)/(d_i - s_j)] \) be the partial Extended Utilization of Single-Instance task set \( T \) until deadline \( d_i \), where \( \max(A) \) is the maximum number from set \( A \). Then there is no feasible schedule for the task set \( T \) on a real-time system with less than \( EUSI = \max_i(EUSI_i) \) processors.

**Proof** Let us consider deadline \( d_i \), where \( i \in \{1, ..., n\} \). For any \( j \in \{1, ..., n\} \) such that \( s_j < d_i \), the minimum amount of time required for execution between moments \( s_j \) and \( d_i \) is \( R_i - A_j \). This is because \( R_i \) represents the minimum amount of time required for execution between moments 0 and \( d_i \), while \( A_j \) is the maximum amount of time that can be executed between moments 0 and \( s_j \). So, if we have less than \([R_i - A_j]/(d_i - s_j)] \) processors, it is not possible for all tasks to execute until \( d_i \), in order to be able to meet their deadlines:

- Either at least one task \( j \) with \( d_j < d_i \) will not be able to complete before \( d_j \).
- Or at least one task \( j \) with \( d_j > d_i \) will execute a time too short before \( d_i \), so that it will not have the chance to complete before \( d_j \).
This restriction must be verified for all start times \( s_j \) that occur before deadline \( d_i \), so we consider the worst case for \( USI_i \).

At the same time, the number of available processors must not be exceeded by any \( EUSI_i \) value, which means we need at least \( EUSI_i \) processors to get a feasible schedule.

**Remark 2.1:** Given a task deadline \( d_i \) and a start time \( s_j < d_i \), it is sometimes possible that \( R_i < A_j \). In this case, \( \lceil (R_i - A_j)/(d_i - s_j) \rceil < 0 \). Such negative values must not be considered when calculating \( EUSI_i \).

We consider two important questions about Theorem 2.1 and Remark 2.1. The first one is about the relationship between \( EUSI \) and \( USI \), namely whether the newly defined estimation of the number of processors is better than the previous estimation (that is, the one from [3]). The second question is whether \( EUSI \) is well-defined for any task set, that is, whether the situation described in Remark 2.1 could occur simultaneously for all start times and deadlines. Both questions are answered in Lemma 2.1 below.

**Lemma 2.1:** For any task set \( T = \{T_1, ..., T_n\} \), the value of \( EUSI \) is well-defined (that is, it can be calculated). Moreover, \( EUSI \geq USI \).

**Proof** Without loss of generality, we assume that in any task set, there is always at least one task whose start time is 0. This assumption does not reduce from its generality as the tasks can be shifted to the left with the minimum amount of start time. More precisely, if the initial description of the task set contains no such tasks, then we simply shift the time scale so that the lowest start time becomes 0; that is, we subtract the value of the lowest start time from the start times and deadlines of all tasks. This does not change the scheduling problem, as no task can execute before the lowest start time.

Let \( m \) be a task with \( s_m = 0 \). Hence, obviously, \( A_m = 0 \), as no processor time is available for execution before time 0. Considering an arbitrary task \( i \), we attempt to calculate \( EUSI_i \). There is at least one term that we can consider, namely \( \lceil (R_i - A_m)/(d_i - s_m) \rceil = \lceil R_i/d_i \rceil > 0 \). In conclusion, it is always possible to calculate \( EUSI_i, \forall i \), and thus \( EUSI \).

We make the observation that \( R_i \geq c_1 + \ldots + c_i \). Indeed, by definition, \( R_i \) includes the computational costs of all tasks \( j \) such that \( d_j \leq d_i \), that is, tasks 1, 2, \ldots, \( i \) (we remind that tasks are sorted by their deadlines). Therefore, we get \( \lceil R_i/d_i \rceil \geq \lceil (c_1 + \ldots + c_i)/d_i \rceil = USI_i \). In conclusion, \( EUSI_i \geq USI_i, \forall i \in \{1, ..., n\} \). Hence, \( EUSI \geq USI \).

The next example illustrates the computation of \( USI \) and \( EUSI \) for a task set as well as a comparison between them.

**Example 2.1:** Let us consider a single-instance, non-preemptive, and independent task set \( T = \{T_1, T_2, T_3, T_4\} \), where the tasks are given by: \( T_1 = (0, 3, 3), T_2 = (2, 3, 5), T_3 = (3, 4, 7), T_4 = (2, 6, 10) \). If we ignore the start times and compute the number of processors with the formula presented in [3], we get:

\[
USI_1 = [3/3] = 1,
USI_2 = [(3 + 3)/5] = 2,
USI_3 = [(3 + 3 + 4)/7] = 2,
USI_4 = [(3 + 3 + 4 + 6)/10] = 2
\]

\[USI = \max(USI_1, USI_2, USI_3, USI_4) = 2 \]

Let us now consider the start times:

\[
s_1 = 0 \rightarrow A_1 = 0
s_2 = s_4 = 2 \rightarrow A_2 = A_4 = 2
s_3 = 3 \rightarrow A_3 = 3 + 1 + 1 = 5
\]

\[
d_1 = 3 \rightarrow R_1 = 3 + 1 = 4
\]

\[
d_2 = 5 \rightarrow R_2 = 3 + 3 + 2 + 1 = 9
\]

\[
d_3 = 7 \rightarrow R_3 = 3 + 3 + 4 + 3 = 13
\]

\[
d_4 = 10 \rightarrow R_4 = 3 + 3 + 4 + 6 = 16
\]

\[
EUSI_1 = \max\{(R_1 - A_1)/(d_1 - s_1)\}, \quad \{(R_1 - A_4)/(d_1 - s_4)\} = \max(2, 2, 2) = 2
\]

\[
EUSI_2 = \max\{(R_2 - A_1)/(d_2 - s_1)\}, \quad \{(R_2 - A_3)/(d_2 - s_3)\}, \quad \{(R_2 - A_4)/(d_2 - s_4)\} = \max(2, 2, 2, 2, 3) = 3
\]

\[
EUSI_3 = \max\{(R_3 - A_1)/(d_3 - s_1)\}, \quad \{(R_3 - A_2)/(d_3 - s_2)\}, \quad \{(R_3 - A_4)/(d_3 - s_4)\} = \max(2, 2, 2, 2, 2, 3) = 3
\]

\[
EUSI_4 = \max\{(R_4 - A_1)/(d_4 - s_1)\}, \quad \{(R_4 - A_2)/(d_4 - s_2)\}, \quad \{(R_4 - A_3)/(d_4 - s_3)\}, \quad \{(R_4 - A_4)/(d_4 - s_4)\} = \max(2, 2, 2, 2) = 2
\]

This time we get \( EUSI = 3 \), which is different from the value determined by the old formula.

One can see that the difference was made during both the time interval \([2, 5]\) (that is, between start times \( s_2 = s_4 \) and deadline \( d_2 \)) and the time interval \([2, 7]\) (between start times \( s_2 = s_4 \) and deadline \( d_3 \)). Within the interval \([2, 5]\), for example, the necessary execution times for the tasks are as below:

- \( T_1 \): 1 time unit (it could have executed at most 2 time units before time 2)
- $T_2$: 3 time units (it could not have executed at all before time 2 and it must complete by time 5)
- $T_3$: 2 time units (it could not have executed at all before time 3 and it has to execute at least 2 time units by time 5)
- $T_4$: 1 time unit (it could not have executed at all before time 2 and it has to execute at least 1 time unit by time 5)

We then need an overall 7 time units to be executed, during a period of 3 time units. In conclusion, the number of processors may not be less than 3.

III. THE SCHEDULING ALGORITHM

We have presented in [3] an efficient algorithm, called Algorithm A, that takes a single-instance, non-preemptive, and independent task set and returns a feasible schedule, if one exists. This section presents an improved extension of Algorithm A, called Algorithm C. Unlike its predecessor, Algorithm C uses $EUSI$ instead of $USI$ as the estimation of the numbers of processors, and it also considers the effect of the different start times of the tasks on the scheduling process.

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

**Definition 3.1:** Given two tasks $T_1 = (s_1, c_1, d_1)$ and $T_2 = (s_2, c_2, d_2)$, 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 $T_1 \leq T_2$, $T_1 \leq T_3$, and $T_3 \leq T_2$.

Up to this point, the ordering relation follows the principles of the LLF technique. We now introduce a task order restriction that may alter the order of two tasks, as established by the ordering relation.

**Definition 3.2:** Given two tasks $T_1 = (s_1, c_1, d_1)$ and $T_2 = (s_2, c_2, d_2)$ such that $T_1 \leq T_2$, we say that $T_1 \not\rightarrow x T_2$ if $d_2 < x + c_1 + c_2 \leq d_1$, $s_1 \leq x$, $s_2 \leq x$, and $x$ is a positive integer.

Definition 3.2 is useful for identifying the situations where the ordering relation is not capable to provide a feasible schedule, but an improvement can still be made. If we have $T_1 \leq T_2$, we would normally expect task $T_1$ to be scheduled before task $T_2$. However, if $x$ (or more) time units have already been executed on processor $P$ and $T_1 \not\rightarrow x T_2$, it is no longer possible to schedule task $T_1$ and then task $T_2$ on processor $P$, simply because the latter will miss its deadline. On the other hand, the task order restriction does not forbid other possible scheduling decisions:

- It may be possible to schedule task $T_1$ and then task $T_2$ on the same processor $P$, if the amount of time previously executed on $P$ is lower than $x$.
- It may also be possible to schedule task $T_1$ and then task $T_2$, even after moment $x$, on different processors.
- If $x$ (or less) time units have already been executed on processor $P$, it is possible to schedule task $T_2$ and then task $T_1$ on processor $P$ and both tasks $T_1$ and $T_2$ will not miss their deadlines.

In fact, the task order restriction was specifically introduced to handle a special case: a task $T_1$ has already been scheduled on a processor, another task $T_2$ cannot be scheduled right after $T_1$ on the same processor, but switching the order of the two tasks allows both of them to meet their deadlines.

It is easy to see that, given two tasks $T_1$ and $T_2$ such that $T_1 \leq T_2$, if $T_1 \not\rightarrow x T_2$, then the values of $x$ for which the restriction holds belong to the interval $(d_2 - c_1 - c_2, d_1 - c_1 - c_2)$.

The ordering relations “$\leq$” and “$\not\rightarrow$” between tasks will be used in Algorithm C. In addition, we consider a chain of tasks $C = [T_1, \ldots, T_k]$ is 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 also denote by $C - last(C)$ the chain $C$ obtained after removing its last task. We denote by $C + T$ the chain obtained by catenating task $T$ to the end of chain $C$. 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.

In the process of executing the tasks, a particular situation may occur. At some point, it may not be possible to schedule new tasks immediately on some processors, simply because the computation times of the corresponding chains are lower than the start times of all the new tasks. In such a case, those chains must wait in an idle state until new tasks may be launched, as they reach their start times. Obviously, it is not convenient to have gaps between the tasks in a chain. So, in order to allow the algorithm to handle the wait intervals, we model them by introducing generic idle tasks, whose execution times can be chosen to be as long as necessary - that is, until new tasks reach their
start moments.

One can see that, besides the ordering provided by task laxities and task order restrictions, start times also play a role in deciding which tasks are scheduled at a certain moment. Indeed, let us consider two tasks \( T_1 \) and \( T_2 \), such that \( T_1 \leq T_2 \) and there is no task order restriction \( T_1 \not\rightarrow_x T_2 \), but \( s_2 < s_1 \). If at least one processor is idle at time \( s_2 \), then it is possible for task \( T_2 \) to be scheduled before task \( T_1 \), namely, at time \( s_2 \). The algorithm works this way because the underlying assumption in computing the number of processors (EUSI) is that all the processors are always used when there are available tasks. Thus, if we wait until time \( s_1 \), in order to schedule task \( T_1 \) before task \( T_2 \), the processor will be idle for more time than strictly necessary, which increases the risk that the schedule will not be feasible.

Algorithm A from [3] uses USI processors for the considered platform. We present below Algorithm C, an improved version of Algorithm A based on EUSI as the number of processors, instead of USI. Algorithm C shares the general features with Algorithm A and has the same worst-case polynomial-time complexity as Algorithm A, that is, \( O(n^2) \), where \( n \) is the number of tasks.

Algorithm C

The input: A set of single-instance non-preemptive and independent tasks \( \mathcal{T} = \{T_1, \ldots, T_n\} \), 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 \( \mathcal{T} \) on a platform having EUSI processors, if \( \mathcal{T} \) is feasible. Otherwise, display that \( \mathcal{T} \) is infeasible.

The method:
1. \( \text{EUSI}_1 = 0; \)
2. for \( (i = 1; i \leq n; i = i + 1) \) \{ 
   3. \( R_i = 0; \)
   4. \( A_i = 0; \)
   5. for \( (j = 1; j \leq n; j = j + 1) \) \{ 
      6. if \( (j \leq i) \) \{ 
        7. \( R_i = R_i + c_j; \)
      8. else if \( (c_j + d_j > d_i) \) \{ 
        9. \( R_i = R_i + (c_j - (d_j - d_i)); \)
      10. if \( (s_j \leq s_i) \) \{ 
        11. \( A_i = A_i + c_j; \)
      12. else if \( (s_1 < s_i) \) \{ 
        13. \( A_i = A_i + \min(c_j, s_i - s_j)); \)
      \}
   14. \( \text{EUSI}_i = 0; \)
   15. for \( (i = 1; i \leq n; i = i + 1) \) \{ 
      16. \( \text{EUSI}_i = 0; \)
      17. for \( (j = 1; j \leq n; j = j + 1) \) \{ 
         18. if \( (d_i > s_j) \) \{ 
            19. if \( (\text{EUSI}_i < [(R_i - A_j)/(d_i - s_j)]) \) \{ 
               20. \( \text{EUSI}_i = [(R_i - A_j)/(d_i - s_j)]; \)
            21. if \( (\text{EUSI}_i < \text{EUSI}_j) \) \{ 
               22. \( \text{EUSI}_i = \text{EUSI}_j; \)
            \}
         \}
      \}
   \}
23. Sort lexicographically tasks \( T_1, \ldots, T_n \) under \( d_i - c_i \) as a primary key and \( d_i \) as a second key. The obtained list is \( \mathcal{T}' = \{T_{\pi(1)}, \ldots, T_{\pi(n)}\} \), where \( \pi \) is the corresponding permutation such that \( T_{\pi(i)} \leq T_{\pi(i+1)} \), for all \( i \in \{1, \ldots, n-1\} \);
24. Initialize all chains \( (1, \ldots, \text{EUSI}) \) as void;
25. \( \text{feasible} = \text{true}; \)
26. while \((\mathcal{T}' \) is a non-empty set & feasible) \{ 
   27. Let \( S \) be the lowest start time of a task from \( \mathcal{T}' \);
   28. for each \( (\text{chain } C) \) \{ 
      29. if \( (S > c(C)) \) \{ 
         30. Add idle tasks to chain \( C \) until time \( S \);
      31. if \( (\text{the last two tasks in chain } C \text{ are idle}) \) 
         32. Merge the two tasks;
      \}
   33. Let \( T \) be the first task from \( \mathcal{T}' \) such that there is at least one chain \( C \) with \( s(T) \leq c(C) \);
   34. Choose a chain \( C \) such that \( c(T) + c(C) \leq d(T) \);
   35. if \( (\text{such a chain exists} \) \{ 
      36. Remove \( T \) from \( \mathcal{T}' \);
      37. Add \( T \) to chain \( C \);
   \}
   38. else \{ 
      39. Choose a chain \( C \) such that \( \text{last}(C) \not\rightarrow_x T \), where \( x = c(C) - \text{last}(C) \);
      40. if \( (\text{such a chain } C \text{ exists}) \) \{ 
         41. Add \( T \) to chain \( C \) and switch between \( T \) and \( \text{last}(C) \);
         42. Remove \( T \) from \( \mathcal{T}' \);
      \}
      43. else \{ 
         44. Print ‘\( \mathcal{T}' \) is infeasible on an EUSI-processor platform’;
         45. \( \text{feasible} = \text{false}; \)
      \}
   \}
   46. if \( (\text{feasible}) \) \{ 
      47. Print ‘\( \mathcal{T}' \) is feasible on an EUSI-processor platform’;
      48. Print all chains \( C \) representing the schedule.
   \}
\}

Let us now consider an example where the LLF technique fails, while EDF and Algorithm C are capable of providing a feasible schedule.

Example 3.1: Let \( \mathcal{T} = \{T_1, T_2, T_3, T_4, T_5, T_6\} \) be a single-instance and non-preemptive task set given by: \( T_1 = (0, 3, 3), T_2 = (0, 2, 3), T_3 = (1, 1, 6), T_4 = (1, 1, 6), T_5 = (2, 3, 7), T_6 = (2, 3, 7) \). Just
as in the previous example, we get $EUSI = 2$. The LLF method fails to provide a feasible schedule for $T$ on a two-processor platform. Obviously, tasks $T_1$ and $T_2$ are first scheduled on processors 1 and 2, respectively. In the sequel, tasks $T_5$ and $T_6$ are scheduled on processors 1 and 2, respectively, as they have lower laxities than tasks $T_3$ and $T_4$. Next, task $T_3$ is scheduled on processor 2. However, task $T_4$ cannot be scheduled on any processor in such a way that it meets its deadline.

The EDF technique, however, is successful. It also starts by scheduling tasks $T_1$ and $T_2$ on processors 1 and 2, respectively. Then, tasks $T_3$ and $T_4$ are scheduled on processors 1 and 2, respectively. In the end, tasks $T_5$ and $T_6$ have enough time to execute on processors 1 and 2, respectively.

Algorithm C behaves much like LLF. Nevertheless, when task $T_4$ becomes impossible to schedule, a switch between the tasks $T_4$ and $T_5$ is made on processor 1. In this way, all tasks can complete by their deadlines.

In the next example, a task set is not schedulable by either EDF or LLF, but is schedulable by Algorithm C.

**Example 3.2:** Let $T = \{T_1, T_2, T_3, T_4, T_5\}$ be a single-instance and non-preemptive task set given by: $T_1 = (0, 3, 3)$, $T_2 = (1, 1, 6)$, $T_3 = (2, 3, 7)$, $T_4 = (1, 7, 8)$, $T_5 = (1, 8, 9)$. This time $EUSI = 3$. If we use the EDF method, tasks $T_1$, $T_2$, and $T_3$ are scheduled on processors 1, 2 and 3, respectively. At this point, neither task $T_4$, nor task $T_5$ can be scheduled on any processor.

The LLF technique is also unsuccessful. In this case, tasks $T_1$, $T_4$, and $T_5$ are scheduled on processors 1, 2, and 3, respectively. Then task $T_3$ is scheduled on processor 1, which further makes it impossible for task $T_2$ to be scheduled on any processor.

If we use Algorithm C, again, the behavior resembles that of LLF. The difference is that, when task $T_2$ must be scheduled, it is switched with task $T_3$. Thus, all task will meet their deadlines.

**IV. Conclusions**

In this paper, we propose an estimation on the lower bound of the number of processors for a single-instance task set, derived from the formula presented in [3]. The improvement consists in taking into consideration the start times of the tasks, which are no longer considered to be 0.

In addition, an efficient algorithm is provided for finding a feasible schedule for a single-instance, non-preemptive, and independent task set on a multiprocessor platform, having a number of processors equal to the lower bound. It extends Algorithm C from [3], also by considering the effect of the different start times of the tasks on the scheduling process. As a result, Algorithm C exhibits better behavior than the classic EDF and LLF algorithms in many instances of the problem.

The future work will focus on investigating the application of the principles behind the algorithm on more general models (e.g., periodic and sporadic tasks).

**References**


