A new scheduling algorithm for non-preemptive independent tasks on a multi-processor platform

Stefan Andrei
Lamar University
Department of Computer Science
Beaumont, TX, USA
stefan.andrei@lamar.edu

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

Vlad Radulescu
Cuza University of Iasi
Department of Computer Science
Iasi, Romania
rvlad@infoiasi.ro

Sharfuddin Alam
Lamar University
Department of Computer Science
Beaumont, TX, USA
salam3@lamar.edu

Suresh Vadlakonda
Lamar University
Department of Computer Science
Beaumont, TX, USA
svadlakonda@lamar.edu

***Abstract*— The abort-and-restart scheme from the Priority-based Functional Reactive Programming (PFRP) paradigm eliminates the priority inversion problem. This paper is similar by solving the priority inversion problem using the task order restrictions sets of relations. One of the core problems in real-time systems is finding a feasible schedule for a task set on a multiprocessor platform. While preemptive scheduling has benefited from a large number of significant results, the non-preemptive case has still room for improvement.**

Despite the fact that the well-known EDF and LLF scheduling techniques are optimal for preemptive uniprocessor platform there is no known optimal scheduling algorithm for non-preemptive task sets on a multi-processor platform. Our paper continues our previous works [1] and [2] by describing the experimental results together with their findings about our alternate scheduling method to the EDF and LLF techniques. Our algorithm, called 𝔸, is able to schedule all considered non-preemptive independent tasks on a multi-processor platform, while EDF and LLF fail to sometimes provide a feasible schedule. The experiments indicate no execution time overhead for the cases where all the scheduling algorithms were able to provide the schedule. However, when EDF and LLF fail to provide a schedule, 𝔸 gives the schedule with a 60% execution time overhead.

Keywords—scheduling algorithm, multiprocessor platform, non-preemptive task

I. INTRODUCTION

The Priority-based Functional Reactive Programming (PFRP) paradigm [14,15,16] is a new declarative approach to modeling and building reactive systems. The PFRP paradigm intends to improve the programming of real-time embedded controllers, typically programmed using C programming language or assembly language. PFRP programs are written as a collection of functions (hence, stateless).

The PFRP compiler generates resource-bounded C code that comprises in a group of event handlers, where each event handler is responsible for a single interrupt or event source. The semantics of PFRP requires that each event handler executes atomically. This requirement facilitates reasoning about PFRP programs and thus is a desirable feature of the language. However, this means that a lower-priority task is aborted when a higher-priority task arrives. The task is restarted once the higher priority task completes. By its nature, PFRP, with its abort-and-restart scheme, eliminates the priority inversion problem. This paper is similar by solving the priority inversion problem using the task order restrictions sets of relations (Section 3).

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 [3]. 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 [4]. Mok discovered another optimal algorithm with polynomial complexity for the same subclass, that is, the Least Laxity First (LLF) algorithm [5]. 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 [6]. However, according to Graham, Lawler, Lenstra and Kan [7], 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 [8]. 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.

There exist many models to define a task and a job. For simplicity, we consider only single-instance tasks, that is, without considering their periods. That is why, we use throughout this paper the terms *task* and *job* interchangeably. In this paper, we consider that a task T is denoted by (s, c, d) and characterized by three parameters: s is called the *start 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*. The meaning is that task T can be executed after s time units, completing a total of c time units by the deadline d. For simplicity, we consider the tasks to be single instance. Hence there is no need to consider the tasks' periods. 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. Given a task set $\mathcal{T} = \{T_1, ..., T_n\}$, $\mathcal{T}$ is called *schedulable* by a scheduling algorithm if all the timing constraints of all tasks in $\mathcal{T}$ are met. The scheduling algorithm is called *optimal* if whenever it cannot find a schedule, then no other scheduling algorithms can [9].

Our scheduling algorithm, called $\mathbb{A}$, is able to provide feasible schedules for all task sets for which the EDF and LLF methods identified a feasible schedule. In fact, for many task sets for which the EDF and LLF methods could not identify a feasible schedule, Algorithm $\mathbb{A}$, was able to provide feasible schedule. This is supported by Lemma 3.1 and the experimental results section, where we provide a more accurate comparison between $\mathbb{A}$ with EDF or LLF.

The remainder of this paper is organized as follows. The next section describes the notations, definitions, and some examples needed for the scheduling problem, in particular EDF, LLF, and Algorithm $\mathbb{A}$. Section III presents $\mathbb{A}$, followed by Section IV, called Experimental Results. Conclusion and References end the paper.

II. PRELIMINARIES

Our paper considers the multiprocessor platform, independent non-preemptive tasks, and no shared resources, overhead, or context-switching time. Similar to the approach from [9], we assume that the task constraints are known in advance, such as deadlines, computation times, and start times.

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 just 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 $[s, e)_{(p1)} \cap [s', e')_{(p2)} = \emptyset$ if and only if either $p_1 \neq p_2$ or $[s, e) \cap [s', e') = \emptyset$ in the mathematical sense (symbol $\cap$ means the interval intersection). We consider in our paper that the multiprocessor platform is simply denoted as $\mathcal{P} = \{p_1, ..., p_m\}$, where p_1, ..., p_m are processors. In this paper, we consider $m < n$, otherwise the scheduling problem would be trivial. For a finite set V, we denote by $|V|$ the number of elements of V. We consider in this paper a task set denoted as $\mathcal{T}$ given by $\{T_1, ..., T_n\}$, where each task T_i is represented by (s_i, c_i, d_i). Since each task has a deadline, the scheduling problem for the multiprocessor environment becomes difficult [10]. We say that the task set $\mathcal{T}$ is schedulable on the multiprocessor platform $\mathcal{P}$ if there exists a *schedule* for each task T_i such that T_i executes in interval $[s, e)_{(p)}$, that is, task T_i executes on processor p in the time interval from time s to time e, and satisfies the following two properties:

1) $s_i \leq s < e \leq d_i$ and $e - s = c_i$;
2) there is no other task T' executed by processor p within interval $[s, e)$.

Given a task set $\mathcal{T} = \{T_1, ..., T_n\}$, where each task T_i is given by (s_i, c_i, d_i) for any $i \in \{1, ..., n\}$, 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, $l_i = d_i - c_i - s_i$ has the highest priority. The definition of laxity for preemptive tasks includes the time instance because a task's priority may change during its execution. However, due to the fact that the tasks are non-preemptive, the tasks' priorities do not change during their execution.

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

Example 2.1. Let $\mathcal{T}_1 = \{T_1, ..., T_{12}\}$ be a single instance and non-preemptive task set given by: $T_1 = (0, 1, 1)$, $T_2 = (0, 1, 2)$, $T_3 = (0, 2, 3)$, $T_4 = (0, 2, 3)$, $T_5 = (0, 2, 4)$, $T_6 = (0, 2, 4)$, $T_7 = (0, 4, 5)$, $T_8 = (0, 1, 5)$, $T_9 = (0, 1, 5)$, $T_{10} = (0, 1, 5)$, $T_{11} = (0, 2, 5)$ and $T_{12} = (0, 1, 5)$. The deadlines are sorted increasingly. The EDF method fails to provide a feasible schedule for $\mathcal{T}_1$ on a four-processor platform, $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$. This is because tasks T_1, T_2, T_3, and T_4 will be chosen to be first executed on processors p_1,

p_2, p_3, and p_4, respectively. Then, T_5 and T_6 will be scheduled for processors p_1 and p_2. Therefore, task T_7 will miss its deadline because all the processors have left less than 3 available time units to execute.

However, $\mathcal{T}_1$ is LLF-schedulable because T_1 executes in $[0, 1)_{(p1)}$, T_2 executes in $[0, 1)_{(p2)}$, T_3 executes in $[0, 2)_{(p3)}$, T_4 executes in $[0, 2)_{(p4)}$, T_7 executes in $[1, 5)_{(p1)}$, T_5 executes in $[1, 3)_{(p2)}$, T_6 executes in $[2, 4)_{(p3)}$, T_{11} executes in $[2, 4)_{(p4)}$, T_8 executes in $[3, 4)_{(p2)}$, T_9 executes in $[4, 5)_{(p2)}$, T_{10} executes in $[4, 5)_{(p3)}$, and T_{12} executes in $[4, 5)_{(p4)}$. ■

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

Example 2.2. Let $\mathcal{T}_2 = \{T_1, ..., T_6\}$ be a single instance and non-preemptive task set on a two-processor platform $\mathcal{P} = \{p_1, p_2\}$ given by: $T_1 = (0, 2, 2)$, $T_2 = (0, 2, 2)$, $T_3 = (0, 3, 6)$, $T_4 = (0, 3, 6)$, $T_5 = (0, 1, 5)$, and $T_6 = (0, 1, 5)$. The laxities are sorted increasingly, namely $l_1 = 0$, $l_2 = 0$, $l_3 = 3$, $l_4 = 3$, $l_5 = 4$, and $l_6 = 4$. The LLF method fails to provide a feasible schedule for $\mathcal{T}_2$ 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 p_1 and p_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.

However, $\mathcal{T}_2$ is EDF schedulable because T_1 executes in $[0, 2)_{(p1)}$, T_2 executes in $[0, 2)_{(p2)}$, T_5 executes in $[2, 3)_{(p1)}$, T_6 executes in $[2, 3)_{(p2)}$, T_3 executes in $[3, 6)_{(p1)}$, and T_4 executes in $[3, 6)_{(p2)}$. ■

The next example shows a task set that is neither LLF schedulable nor EDF schedulable on a two processor platform.

Example 2.3. Let $\mathcal{T}_3 = \{T_1, ..., T_7\}$ be a single instance and non-preemptive task set on a three-processor platform, $\mathcal{P} = \{p_1, p_2, p_3\}$ given by: $T_1 = (0, 2, 2)$, $T_2 = (0, 7, 7)$, $T_3 = (0, 8, 9)$, $T_4 = (0, 3, 6)$, $T_5 = (0, 1, 5)$, $T_6 = (0, 5, 12)$, and $T_7 = (0, 3, 11)$. The laxities are sorted in an increasing order. 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 $\mathcal{T}_3$ is neither LLF-schedulable nor EDF-schedulable. ■

III. OUR SCHEDULING ALGORITHM $\mathbb{A}$

We describe Algorithm $\mathbb{A}$ that takes a single-instance non-preemptive and independent task set and returns a feasible schedule. It may happen however that the task set to be feasible, but Algorithm $\mathbb{A}$ would not be able to provide a feasible schedule. We define first the ordering relation for the task sets. This ordering relation is actually based on task laxities.

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 $d_1 - c_1 - s_1 < d_2 - c_2 - s_1$ or $(d_1 - c_1 - s_1 = d_2 - c_2 - s_1$ and $d_1 < d_2)$. We say that $T_1 \leq T_2$ if $T_1 < T_2$ or $T_1 = T_2$. ■

Example 3.1. Given a task set $\mathcal{T} = \{T_1, T_2, T_3\}$ where $T_1 = (0, 1, 3)$, $T_2 = (0, 2, 5)$ and $T_3 = (0, 2, 4)$, we have $T_1 \leq T_2$, $T_1 \leq T_3$ and $T_3 \leq T_2$. ■

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 $T_1 < T_2$, we say that $T_1 \lessdot_x T_2$ if $d_2 < x + c_1 + c_2 \leq d_1$, $s_1 \leq x$ and $s_2 \leq x$. For a task set $\mathcal{T} = \{T_1, ..., T_n\}$, we define the set of *task order restrictions* relation $TOR(\mathcal{T}) = \{T_i \lessdot_x T_j \mid$ where $i, j \in \{1, 2, \ldots, n\}\}$. ■

Definition 3.2 provides a way of handling the situations where the main ordering relation cannot find a feasible schedule. For two tasks T_1 and T_2, with $T_1 < T_2$, the first attempt is always to schedule T_1 and then T_2.

However, if x (or more) time units have already been executed on processor p and $T_1 \lessdot_x T_2$, 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.

Nevertheless, if $d_2 < d_1$, it is sometimes possible that T_2 cannot meet its deadline with this scheduling order, but T_1 and T_2 can both meet their deadlines if T_2 is scheduled first.

On the other hand, the task order restriction does not forbid other possible scheduling decisions:

1. 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;
2. It may also be possible to schedule task T_1 and then task T_2, even after moment x, on different processors;
3. More important for our goal, at moment x, it is definitely possible to schedule task T_2 and then task T_1 on the same processor, such that none of them misses its deadline.

In fact, the task order restriction relation 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 $T_1 \lessdot_x T_2$, then the values of x for which the restriction holds belong to the interval $(d_2 - c_1 - c_2, d_1 - c_1 - c_2]$.

We note that both ordering relations defined above, $<$ and $\prec_x$, are concerned with comparing and switching tasks on the same processor. Example 3.2 describe how to construct and use the ordering relation $\prec_x$ for a task set.

Example 3.2. Let us consider the task set from Example 2.3. The laxities $l_i = d_i - c_i - s_i$ for all $i \in \{1, \ldots, 7\}$ 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 sorted in an increasing order upon their laxities. According to Definition 3.2, $TOR(\mathbb{T}_3) = \{T_3 \prec_0 T_5; T_4 \prec_2 T_5; T_6 \prec_4 T_7\}$. However, it may be possible to have more than just one value. For example, given $T_1 = (0, 6, 13)$ and $T_2 = (0, 3, 11)$, it is clear that $T_1 \leq T_2$. In order to investigate what values x can get, we consider the inequality from Definition 3.2, 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 "$\prec_x$" between tasks will be used in Algorithm $\mathbb{A}$. In addition, we consider a chain of tasks C as $[T_1, \ldots, T_k]$, a list of tasks from the given task set $\mathbb{T}$. 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 concatenating C and task T. Algorithm $\mathbb{A}$ tries to generate a schedule for $\mathbb{T}$ on a multiprocessor platform. It can be viewed as a beneficial combination of two "complementary" scheduling techniques, LLF and EDF. Algorithm $\mathbb{A}$ below (extended from [1] and [2]) will be able to generate schedules for all task sets from Examples 2.1, 2.2, and 2.3.

Algorithm $\mathbb{A}$

Input: A set of single-instance non-preemptive and independent tasks $\mathbb{T} = \{T_1, \ldots, T_n\}$ on a $m = |\mathbb{P}|$ processor platform, where each task $T_i = (s_i, c_i, d_i)$, for all $i \in \{1, \ldots, n\}$, such that $d_1 \leq \ldots \leq d_n$.

Output: A schedule for the task set $\mathbb{T}$ on a m-processor platform. Otherwise, display that $\mathbb{T}$ is infeasible using $\mathbb{A}$-algorithm.

1. Sort lexicographically tasks $T_1, .., T_n$ under $d_i - c_i - s_i$ as a primary key and d_i as a second key. The obtained list is $T' = [T_{\pi(1)}, \ldots, T_{\pi(n)}]$, where π is the corresponding permutation such that $T_{\pi(i)} \leq T_{\pi(i+1)}$ for all $i \in \{1, .., n-1\}$;
2. $TOR(T') = \emptyset$;
3. `for` $(i = 1; i < n; i{+}{+})$
4. `if` $(d(T_{\pi(i)}) - c(T_{\pi(i)}) - s(T_{\pi(i)}) \geq 0)$
5. `for` $(j = i + 1; j \leq n; j{+}{+})$
6. `if` $(\exists x \geq 0, d(T_{\pi(j)}) < x + c(T_{\pi(i)}) + c(T_{\pi(j)}) \leq d(T_{\pi(i)}))$
7. $TOR(T') = TOR(T') \cup \{T_{\pi(i)} \prec_x T_{\pi(j)}\}$;
8. Choose $T_{\pi(1)}, \ldots, T_{\pi(m)}$ as the set of initial tasks for the first p chains;
9. Remove $T_{\pi(1)}, \ldots, T_{\pi(m)}$ from the list T';
10. *feasible* = `true;`
11. `while` (T' is a non-empty set && *feasible*) {
12. Let T be the first task from T';
13. Choose a chain C such that $c(T) + c(C) \leq d(T)$;
14. `if` (such a chain exists) {
15. Remove T from T';
16. Add T to chain C;
}
else {
17. Choose a chain C such that $last(C) \prec_x T$ is in $TOR(T')$, where $x \geq c(C - last(C))$;
18. `if` (such a chain C exists) {
19. Add T to chain C, but switch T and $last(C)$;
20. Remove T from T';
}
else {
21. `print` '$\mathbb{A}$ was unable to find a schedule';
22. feasible = `false;`
}
}
23. `if` (feasible) {
24. print '$\mathbb{T}$ is feasible and its schedule';
} } }

Lemma 3.1. If a task set $\mathbb{T}$ is LLF-schedulable, then Algorithm $\mathbb{A}$ will produce the same schedule. ■

Proof. If the input task set $\mathbb{T}$ is LLF-schedulable, then the statements from lines 17 to 22 will never be executed because it will always be a chain such that $c(T) + c(C) \leq d(T)$. Hence, the schedule provided by Algorithm $\mathbb{A}$ will be exactly the same as the LLF schedule. ■

Theorem 3.1. (**correctness of Algorithm $\mathbb{A}$**) Let us consider a set of single-instance non-preemptive and independent tasks $\mathbb{T} = \{T_1, \ldots, T_n\}$ on a $m = |\mathbb{P}|$ processor platform, where each task $T_i = (s_i, c_i, d_i)$, for all $i \in \{1, \ldots, n\}$, such that $d_1 \leq \ldots \leq d_n$.

Upon its execution, Algorithm $\mathbb{A}$ will produce a correct schedule for the task set $\mathbb{T}$ on a p-processor platform. Otherwise, display that $\mathbb{A}$ was unable to find a feasible schedule. ■

Proof. Line 1 sorts the tasks according to their laxities, as stated in the LLF scheduling technique. Obviously, the tasks may be reordered from the lowest laxity (hence the highest priority) to the highest laxity (so the lowest priority) as described by the permutation π. Hence, the obtained list is $T' = [T_{\pi(1)}, \ldots, T_{\pi(n)}]$, where π is the corresponding permutation such that $T_{\pi(i)} \leq T_{\pi(i+1)}$ for all $i \in \{1, \ldots, n-1\}$.
The statements from lines 2 to 7 construct the task order restriction relation as described in Definition 3.2. Line 8 assigns to each processor a task that is ready to execute. So, task $T_{\pi(1)}$ will be assigned to processor 1, task $T_{\pi(2)}$ will be assigned to processor 2, and so on. These allocated tasks will be removed from the local variable T' (line 9). Testing whether variable T' is empty is part

of the termination condition of the `while` loop starting at line 11. We assume initially that the task set $\mathbb{T}$ is feasible (line 10). Lines 12 to 16 will choose a chain of tasks (hence a processor) for task T. Since the tasks are ordered in an ascending order of their laxities, it means that the statements from lines 12 to 16 follow the LLF scheduling technique (Lemma 3.1).

Lines 17 to 20 handle the case when there is no chain C such that $c(T) + c(C) \leq d(T)$, then it will be subject to switching the task order in one of the chains which satisfy the task order restriction. If there is a chain C such that $last(C) \not\prec_x T$ is in $TOR(T')$, where $x \geq c(C - last(C))$, then it means that we can switch T and $last(C)$. If $T_k = last(C)$, it means that $d_T < x + c_{Tk} + c_T \leq d_{Tk}$ (Definition 3.2). The interval $[0, x)$ is reserved to previous executed tasks, including T_k. Because task T will miss its deadline if it executed after T_k, then line 19 of Algorithm $\mathbb{A}$ switches tasks T_k and T. Then task T_k will be executed after T without missing its deadline because $d_T < x + c_{Tk} + c_T \leq d_{Tk}$.

Lines 21 and 22 handle the case when there is no chain C such that $last(C) \not\prec_x T$ is in $TOR(T')$. In this case, Algorithm $\mathbb{A}$ will display '$\mathbb{A}$ was unable to find a schedule' and stop the `while` loop starting at line 11. ■

Theorem 3.2. (**complexity of Algorithm $\mathbb{A}$**) Let us consider a set of single-instance non-preemptive and independent tasks $\mathbb{T} = \{T_1, ..., T_n\}$ on a $m = |\mathbb{P}|$ processor platform, where each task $T_i = (s_i, c_i, d_i)$, for all $i \in \{1; ..., n\}$, such that $d_1 \leq ... \leq d_n$. Algorithm $\mathbb{A}$ has both time and space complexities of $O(n^2)$. ■

Proof. The statement from line 1 can be implemented in $O(n \log n)$ as the optimal time complexity of sorting algorithms. The statements from lines 2-10 can be done in $O(n^2)$ time complexity. The statements from lines 11 to 24 can be implemented in $O(n \times m)$ time complexity. Since $m \leq n$, we conclude that the time complexity of Algorithm $\mathbb{A}$ is $O(n^2)$. Due to a similar justification, the space complexity of Algorithm $\mathbb{A}$ is $O(n^2)$. ■

The exact comparison between $\mathbb{A}$-schedule and the EDF-schedule is subject to future research.

Example 3.3. Let $\mathbb{T}_3 = \{T_1, ..., T_7\}$ be a single instance and non-preemptive task set on a three-processor platform, $\mathbb{P} = \{p_1, p_2, p_3\}$ given by: $T_1 = (0, 2, 2)$, $T_2 = (0, 7, 7)$, $T_3 = (0, 8, 9)$, $T_4 = (0, 3, 6)$, $T_5 = (0, 1, 5)$, $T_6 = (0, 5, 12)$, and $T_7 = (0, 3, 11)$. Example 2.3 indicates that the task set $\mathbb{T}_3$ is neither EDF-schedulable, not LLF-schedulable. $\mathbb{T}_3$ is $\mathbb{A}$-schedulable because T_1 executes in $[0, 2)_{(p1)}$, T_2 executes in $[0, 7)_{(p2)}$, T_3 executes in $[0, 8)_{(p3)}$, T_5 executes in $[2, 3)_{(p1)}$, T_4 executes in $[3, 6)_{(p1)}$, T_6 executes in $[6, 11)_{(p1)}$, and T_7 executes in $[7, 10)_{(p2)}$. The priority inversion between T_4 and T_5 was possible because $TOR(\mathbb{T}_3) = \{T_3 \not\prec_0 T_5; T_4 \not\prec_2 T_5; T_6 \not\prec_4 T_7\}$ (Example 2.3). Processor p_1 reaches time unit of 2 before task T_4 was scheduled to be executed. Since $T_4 \not\prec_2 T_5$ belongs to $TOR(\mathbb{T}_3)$ and the current time was 2, according to line 19 of Algorithm $\mathbb{A}$, we can switch T_4 and T_5 and get the previous $\mathbb{A}$-schedule. ■

Paper [14] describes a similar scheduling algorithm that solves the problem of finding a feasible non-preemptive schedule whenever one exists on M identical processors for a given set of processes such that each process starts executing after its release time and completes its computation before its deadline. However, the scheduling algorithm of [14] has the exclusion relations defined on ordered pairs of process segments given before the algorithm starts. In addition, these exclusions only apply for segments of an interval and their scheduling algorithm is exponential in the worst case as it is based on branch-and-bound technique. In contrast, Algorithm $\mathbb{A}$ compute similar task order restrictions during the scheduling algorithm and has a polynomial time complexity in the worst case.

IV. Experimental results

We have implemented all three scheduling algorithms (EDF, LLF, $\mathbb{A}$) in Java programming language on a Windows 64Mb system, having a processor of 3.3MHz and a memory 6GB of RAM. Table 1 shows the average execution time of EDF, LLF, and $\mathbb{A}$ in milliseconds together with the answer to the feasibility question. To maintain its accuracy, the experiments are executed 1000 times to take the average time. The task sets from Table 1 are uniformly selected from real-time Benchmarks of UH, SNU, and LU [10].

No	$\#\mathbb{T}$	$\#\mathbb{P}$	EDF		LLF		$\mathbb{A}$	
			Feas?	Time	Feas?	Time	Feas?	Time
1	4	3	Yes	0.015	Yes	0.015	Yes	0.015
2	12	4	Yes	0.040	Yes	0.041	Yes	0.042
3	6	2	Yes	0.016	Yes	0.018	Yes	0.018
4	4	4	Yes	0.015	Yes	0.013	Yes	0.013
5	8	6	Yes	0.021	Yes	0.026	Yes	0.025
6	24	8	Yes	0.085	Yes	0.074	Yes	0.077
7	12	4	Yes	0.036	Yes	0.041	Yes	0.043
8	8	8	Yes	0.022	Yes	0.032	Yes	0.035
9	12	9	Yes	0.037	Yes	0.050	Yes	0.047
10	36	12	Yes	0.123	Yes	0.097	Yes	0.096
11	12	4	No	0.030	Yes	0.043	Yes	0.048
12	6	4	No	0.017	Yes	0.016	Yes	0.018
13	12	8	No	0.045	Yes	0.048	Yes	0.046
14	24	8	No	0.094	Yes	0.080	Yes	0.080
15	6	2	Yes	0.018	No	0.018	Yes	0.032
16	12	4	Yes	0.043	No	0.042	Yes	0.083
17	18	6	Yes	0.066	No	0.056	Yes	0.124

18	7	3	No	0.022	No	0.019	Yes	0.037
19	14	6	No	0.053	No	0.054	Yes	0.112
20	21	9	No	0.052	No	0.060	Yes	0.139

Table 1. Task set scheduling results

We concluded the following five findings from Table 1:

1. When all tasks are LLF and EDF schedulable, 𝔸 takes about the same amount of time as LLF and EDF. The average is as follows:
executionTime_𝔸 0.015621216 ms
executionTime_EDF 0.01813721 ms
executionTime_LLF 0.015517794 ms
2. When LLF succeeds but EDF fails, 𝔸 succeeds:
executionTime_𝔸 0.016765054 ms
executionTime_EDF 0.015995821 ms
executionTime_LLF 0.01609016 ms
3. When LLF fails and EDF succeed, 𝔸 succeeds:
executionTime_𝔸 0.051071003 ms
executionTime_EDF 0.031431895 ms
executionTime_LLF 0.013447051 ms
4. When both LLF and EDF fail, 𝔸 succeeds:
executionTime_𝔸 0.07502317 ms
executionTime_EDF 0.057403896 ms
executionTime_LLF 0.018161282 ms
5. The overall execution time average for both schedulable and non-schedulable task sets is:
executionTime_ 𝔸 0.024385475 ms
executionTime_EDF 0.015448866 ms
executionTime_LLF 0.012151535 ms

V. Conclusion

When EDF and LLF cannot provide a schedule, the overall execution time of Algorithm 𝔸 is increased by about 60% compared to EDF and LLF. However, if a task set is LLF-schedulable (regardless whether it is EDF-schedulable) algorithm 𝔸 will take about the same time as LLF. Moreover, if a task set is EDF-schedulable but not LLF-schedulable, then Algorithm 𝔸 takes 60% more execution time than EDF on average.

References

[1] Ş. Andrei, A. Cheng, G. Grigoras, and V. Radulescu. An Efficient Scheduling Algorithm for the Multiprocessor Platform. In *Proceedings of 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (*SYNASC'10*), pages 245–252, IEEE Computer Society, Timisoara, Romania, 2010.

[2] Ş. Andrei, A. Cheng, and V. Radulescu. An Improved Upper-bound Algorithm for Non-preemptive Task Scheduling. In *Proceedings of 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (*SYNASC'15*), pages 245–252, IEEE Computer Society, Timisoara, Romania, 2015.

[3] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, 1995.

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

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

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

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

[8] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 129–139. IEEE Computer Society, 1991.

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

[10] ***: Benchmark of real-time embedded systems tasks, available online at galaxy.lamar.edu/~sandrei/taskSetsBenchmark.zip

[11] J. Ras and A.M.K. Cheng. Response Time Analysis for the Abort and-Restart Task Handlers of the Priority-Based Functional Reactive Programming (P-FRP) Paradigm. In IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pages 305-314, August 2009.

[12] J. Ras and A.M.K. Cheng. Response Time Analysis of the Abort – and Restart Model under Symmetric Multiprocessing. In Proceedings *CIT*, pages 1954-1961, 2010.

[13] Hing Choi Wong. Schedulability Analysis for the Abort-and-Restart Model, *Doctor of Philosophy Thesis*, University of York, 2014

[14] Jia Xu, Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations, IEEE Trans. Software Eng., 19 (2): 139-154 (1993)