New Schedulability Test Conditions for Non-preemptive Scheduling on Multiprocessor Platforms

Nan Guan¹, Wang Yi¹,², Zonghua Gu¹, Qingxu Deng¹ and Ge Yu¹

¹ Northeastern University, Shenyang, China
² Uppsala University, Uppsala, Sweden
³ Hong Kong University of Science and Technology, Hong Kong, China

Abstract

We study the schedulability analysis problem for non-preemptive scheduling algorithms on multiprocessors. To our best knowledge, the only known work on this problem is the test condition proposed by Baruah [1] (referred to as [BAR-EDFnp]) for non-preemptive EDF scheduling, which will reject a task set with arbitrarily low utilization if it contains a task whose execution time is equal or greater than the minimal relative deadline among all tasks. In this paper, we firstly derive a linear-time test condition which avoids the problem mentioned above, by building upon the work in [2] for preemptive multiprocessor scheduling. This test condition works on not only non-preemptive EDF, but also any other work-conserving non-preemptive scheduling algorithms. Then we improve the analysis and present test conditions of pseudo-polynomial time-complexity for Non-preemptive Earliest Deadline First scheduling (EDFnp) and Non-preemptive Fixed Priority scheduling (FPnp) respectively. Experiments with randomly generated task sets show that our proposed test conditions, especially the improved test conditions, have significant performance improvements compared with [BAR-EDFnp].

1 Introduction

Compared with preemptive scheduling, non-preemptive scheduling and schedulability analysis have received considerable less attention in the research community. However, non-preemptive scheduling is widely used in industry, and it may be preferable to preemptive scheduling for a number of reasons [3]: Non-preemptive scheduling algorithms are easier to implement and have lower runtime overhead than preemptive scheduling algorithms; the overhead of preemptive scheduling algorithms is more difficult to characterize and predict than that of non-preemptive scheduling algorithms due to inter-task interference caused by caching and pipelining. These benefits of non-preemptive scheduling are even more important on multiprocessor platforms, since the task migration overhead is higher and more difficult to predict. However, this problem is much less severe for non-preemptive scheduling, where each task instance runs to completion on one processor, and task migration only happens at task instance boundaries.

Non-preemptive scheduling is considered inferior for time critical systems (partially) because of its poor responsiveness. On a single processor platform, the most urgent task may not get the processor for quite a long time due to non-preemptive blocking, which could be deadly harmful for hard real-time systems. However, the natural parallelism of multiprocessors can mitigate the penalty of non-preemptive blocking. Even if several processors are longtime occupied by tasks with large execution time, urgent tasks can execute on other processors. We have conducted simulation experiments to compare the performance of two well-known preemptive scheduling algorithms EDF and DM, and their non-preemptive versions EDFnp and DMnp on multiprocessors (we refer interested readers to the technical report [4] for details). To our surprise, for task sets in which the range of task execution time is not very wide, the performance of EDFnp is very close to EDF, and DMnp actually performs better than DM. Note that we even have not taken the context switch overhead into account in the simulations. So we believe that, for a considerable part of real-life applications on multiprocessor platforms, non-preemptive scheduling could be a better choice with respect to real time performance. This is yet another motivation for us to study non-preemptive scheduling on multiprocessors.

In this work, we study the schedulability analysis problem for non-preemptive scheduling of sporadic task sets on identical multiprocessors. We focus on work-conserving scheduling algorithms, i.e., it is not allowed to idle a processor if there is some task instance awaiting for execution. Note that in the context of multiprocessor scheduling, a work-conserving algorithm is necessarily global [5]. The only known work on
this problem is the test condition proposed by Baruah [1] (referred to as [BAR-EDF_np]) for non-preemptive EDF scheduling (referred to as EDF_np), which has quite poor performance and suffers from the disadvantage that task sets with arbitrarily low utilization will be rejected if \( C_{max} \geq D_{min} \), where \( C_{max} \) is the maximal execution time among all tasks and \( D_{min} \) is the minimal relative deadline. In this paper we develop new sufficient schedulability tests by building upon the work in [6, 2] for preemptive scheduling algorithms. At first we derive a test condition of linear-time complexity, which works on any work-conserving non-preemptive scheduling algorithm, and can avoid [BAR-EDF_np]'s disadvantage mentioned above. Then we derive improved test conditions for EDF_np and FP_np respectively, which are both of pseudo-polynomial time complexity, but have significant performance improvement compared with [BAR-EDF_np].

The paper is structured as follows: Section 2 presents the related work. Section 3 introduces the system model and our analysis framework. We present our first general test condition in Section 4, and then improve it for EDF_np and FP_np in Section 5 and Section 6 respectively. The robust property of the proposed tests is proved in Section 7. Section 8 presents performance evaluation results, and finally, conclusions are given in Section 9.

2 Related Work

Preemptive Scheduling. All scheduling algorithms mentioned in this paragraph are within the context of "global preemptive", for example, we refer to the "global preemptive EDF" as "EDF" for short. Goossens et al. [7] introduced a schedulability test of polynomial time-complexity for periodic task sets scheduled by EDF based on the resource-augmentation techniques [8]. Similar techniques are also used in [9] to derive schedulability tests for tasks with limited utilization scheduled by RM. Baker [6] presented schedulability tests for both EDF and DM by assuming that a given task \( \tau_k \)'s task instance misses deadline, and then determining necessary conditions on the parameters of all the tasks to cause such a deadline miss. Based on Baker’s idea, Bertogna et al. [10] observed that the work done in parallel with a task instance do not need to be counted into its interference, and provided a new test condition of polynomial time-complexity, which can occasionally outperform Baker’s test condition. Baruah [2] extended Baker’s approach to reduce the overestimation of the so-called "carry-in", and provided a test condition of pseudo-polynomial time-complexity, which has much higher acceptance ratio than previous test conditions for task systems satisfying the following conditions: the number of tasks \( n \) is significantly larger than the number of processors \( m \) (i.e., \( n \gg m \)), or the parameters of different tasks have widely varying orders of magnitude. Recently, Baruah et al. [11, 12, 13] developed a new approach based on Baker’s idea, which provides test conditions (of pseudo-polynomial time-complexity or polynomial time-complexity depending on different accuracy degrees) for both EDF and DM, as well as some interesting resource augmentation bounds. Andersson et al. [14] is the first to use the approximate response time analysis for multiprocessor scheduling, which was later improved by Bertogna et al. [15] by using their observation in [10] and exploring task slack time to reduce the pessimistic degree in the computation of the approximate response time. Similar to the exact response time analysis for single-processor scheduling, the time-complexity of their approaches is pseudo-polynomial.

Non-Preemptive Scheduling. Baruah [1] proposed a sufficient but not necessary polynomial-time schedulability test condition [BAR-EDF_np] for global EDF_np for periodic task sets, and it can be easily generalized to sporadic task sets. [BAR-EDF_np] used the technique similar to [7] and took into account the extra interference time caused by non-preemption. [BAR-EDF_np] showed that a task set \( \tau \) is EDF_np schedulable on \( m \) processors if

\[
V_{sum}(\tau) \leq m - (m - 1)V_{max}(\tau)
\]

where

\[
V_{sum}(\tau) = \sum_{\tau_i \in \tau} V_i, \quad V_{max}(\tau) = \max_{\tau_i \in \tau} V_i
\]

and \( C_{max} \) is the maximum execution time among all tasks. It is obvious that a task set with arbitrarily low utilization cannot pass the test if \( C_{max} \geq D_{min} \), where \( D_{min} \) denotes the minimal \( D_i \) among all tasks. Intuitively, it means that for any task instance \( J_k \), if there is some task with execution time large enough to cover its relative deadline \( D_k \), \( J_k \) will definitely miss its deadline. This is true for single processor scheduling, but not necessarily true for multiprocessor scheduling, since even if there are some processors long-time occupied by a task instance with large \( C_i \), other task instances can execute on other processors to meet their deadlines.

3 System Model and Analysis Framework

We adopt the discrete time model in this paper, i.e., any time value \( t \) involved in scheduling and any task parameter is assumed to be a non-negative integer value.

We assume that a multiprocessor platform consists of \( m \) identical processors. A sporadic task set \( \tau \) consists of \( n \) sporadic tasks. A sporadic task is denoted by \( \tau_i = (C_i, D_i, T_i) \), where \( C_i \) is the worst-case execution time, \( D_i \) is the relative deadline and \( T_i \) is the minimum inter-release separation, which is also referred to as the period of the task, and we assume \( D_i \leq T_i \). We define \( S_i = D_i - C_i \). The utilization of task \( \tau_i \) is defined as \( U_i = \frac{S_i}{T_i} \), and we use \( U(\tau) \) to denote the sum of \( U_i \) of all \( \tau_i \in \tau \).

Such a sporadic task \( \tau_i \) generates a potentially infinite sequence of task instances (also known as jobs) with successive
releases separated by at least $T_i$ time units. We use $J_i^p$ to denote the $p^{th}$ instances of $\tau_i$. We also use $J_i$ to denote $\tau_i$’s instance in general if we do not want to specify which instance it is. Each task instance has a release time (arrival time) $r_i$ and an absolute deadline $d_i = r_i + D_i$. We use $l_i = d_i - C_i$ to denote the latest start time of task instance $J_i$, i.e., if $J_i$ starts execution after $l_i$, it must miss its deadline.

For fixed priority scheduling, we use $P(\tau_i)$ to denote $\tau_i$’s priority. We assume that every task has a unique priority in our task system, and use $P(\tau_i) > P(\tau_j)$ to denote that $\tau_i$’s priority is higher than $\tau_j$’s.

We aim to analyze the schedulability of a sporadic task set on multiprocessors with non-preemptive scheduling. As shown in [3], exact feasibility-analysis of periodic non-preemptive task systems is highly intractable, even on single-processors. So our goal is to obtain sufficient, rather than exact, schedulability conditions.

In the following we will introduce the general framework of our analysis, which is closely related to the work in [2] for preemptive scheduling. Suppose a task set $\tau$ is non-schedulable, and let $J_k$ be the first task instance that misses deadline. Let $l_o$ denote the latest time-instance earlier than $r_k$ at which at least one processor is idle and let $A_k = r_k - l_o$. Since all processors are idle when the system starts, there must exist such a $l_o$. Since preemption is not allowed, once a task instance starts execution, it must run to completion without interruption. So if $J_k$ starts to execute before its latest start time $l_k$, it must be able to finish execution before deadline $d_k$. Therefore, in order for $J_k$ to miss its deadline, all $m$ processors must be continuously busy in the time interval $[l_o, l_k]$. What happens after $l_k$ has no effect on the schedulability of $J_k$. We name the time interval $[l_o, l_k]$ as problem window, as shown in Figure 1-(a).

The definition of problem window in this paper is different from the one in [2] for preemptive scheduling, where all $m$ processors must be continuously busy in the time interval $[r_k, d_k]$, but does not have to be continuously busy in the time interval $[r_k, d_k]$ as long as the sum of the busy segments (shadowed area in the figure) is large enough to cause $\tau_k$ to miss its deadline, as shown in Figure 1-(b).

The necessary condition for the deadline miss to occur is that the worst-case work done in the problem window by all other task instances in the task set $\tau$ except $J_k$, is larger than $(A_k + S_k) \times m$ (the shadowed area in Figure 1-(a)). There is no known method to find the critical instant in multiprocessor scheduling except exhaustively simulating the system. So we will compute the worst-case work done by each task in the problem window, denoted by $I(\tau_i)$, and use the sum of each $I(\tau_i)$ as an upper bound of the overall worst-case work done in the problem window. The work done by a task $\tau_i$ in the problem window can be categorized into three types:

- **body**: the contribution of all task instances (called body instance) with both release time and deadline in the problem window; each task instance contributes to the workload in that interval with a complete execution time $C_k$;
- **carry-in**: the contribution of at most one task instance (called carry-in instance) with release time earlier than $l_o$ and deadline in the problem window; this task instance contributes with the fraction of its execution time actually executed in the problem window;
- **carry-out**: the contribution of at most one task instance (called carry-out instance) with release time in the problem window and deadline later than $l_k$; this task instance contributes with the fraction of its execution time actually executed in the problem window.

We always consider the work of a carry-in instance is executed as late as possible and a carry-out instance is executed as early as possible, as shown in Figure 2. This is a pessimistic but safe approximation to account the work done by a task.

Since there is at least one processor idled at $l_o$, so at most $m - 1$ tasks may cause the carry-in, and the remaining $(m - m + 1)$ tasks has no carry-in. We use $I_1(\tau_i)$ to denote $I(\tau_i)$ if $\tau_i$ has no carry-in instance, and use $I_2(\tau_i)$ to denote $I(\tau_i)$ if $\tau_i$ has a carry-in instance, and define:

$$I_{df}(\tau_i) = I_2(\tau_i) - I_1(\tau_i)$$

We sort all $I_{df}(\tau_i)$ in a non-increasing list, and use $\Delta^{m-1}_{\tau_i}$ to denote the sum of the first $(m - 1)$ elements in this list, then we can get a sufficient condition for $\tau$ to be schedulable:

![Figure 1. the problem window in preemptive and non-preemptive scheduling.](image1.png)

![Figure 2. body, carry-in and carry-out of a task in the problem window $[l_o, l_k]$.](image2.png)
Lemma 1. A task set \( \tau \) is schedulable with work-conserving non-preemptive scheduling algorithms on \( m \) processors, if for any task \( \tau_k \) and any \( A_k \geq 0 \) the following condition is satisfied:

\[
\sum_{\tau_i \in \tau} I_1(\tau_i) + \Delta m \leq (A_k + S_k) \cdot m
\]

and since \( A_k \geq 0 \) and \( S_k \geq S_{\min} \), we get

\[
U(\tau) < m - \frac{\sum C_i + \Delta C_i^{-1}}{S_{\min}}
\]

which contradicts our assumption that \( \tau \) satisfies Inequality 4.

\[\Box\]

4 The First Schedulability Test

To use Lemma 1 for schedulability test, we should compute the left-hand side of Inequality 3 as well as solve the unknown variable \( A_k \) in the inequality (The left-hand side of the inequality also implicitly contains \( A_k \)).

Simple upper bounds of \( I_1(\tau_i) \) and \( I_2(\tau_i) \) can be obtained by pessimistically counting both the carry-in and carry-out of a task \( \tau_i \) as \( C_i \) and counting the work done by its body instances as \( \frac{A_k + S_k}{T_i} \cdot C_i \). At the same time, we observed that as \( A_k \) increases, the proportion of the carry-in and carry-out in the overall work done by a task in the problem window tends to decrease, which implies that the adverse effect of the overestimation of the carry-in and carry-out is more severe with small \( A_k + S_k \) values, of which the extreme case is \( A_k = 0 \). We get our first test condition based on this observation:

Theorem 1. [TEST-1] A task set \( \tau \) is schedulable with work-conserving non-preemptive scheduling algorithms on \( m \) processors if:

\[
U(\tau) < m - \frac{\sum C_i + \Delta C_i^{-1}}{S_{\min}}
\]

where \( \sum C_i \) is the sum of all tasks’ \( C_i \), \( S_{\min} \) is the minimal \( S_i \) among all tasks; we sort all \( C_i \) in a non-increasing list, and use \( \Delta C_i^{-1} \) to denote the sum of the first \( (m-1) \) elements in this list.

Proof. We prove the theorem by contradiction. Assume a task set \( \tau \) satisfies Inequality 4 but it is non-schedulable, and task \( \tau_k \) missing its deadline.

Since \( \tau_k \) is non-schedulable, by Lemma 1 we know there is an \( A_k \) such that:

\[
\sum_{\tau_i \in \tau} I_1(\tau_i) + \Delta m \leq (A_k + S_k) \cdot m
\]

The number of \( \tau_i \)’s body instances in the problem window is at most \( \frac{A_k + S_k}{T_i} \), and the carry-in and the carry-out are both at most \( C_i \), so by the definition of \( I_1(\tau_i) \) and \( I_2(\tau_i) \), we have:

\[
I_1(\tau_i) \leq \left( \frac{A_k + S_k}{T_i} \right) * C_i + C_i \leq \frac{A_k + S_k}{T_i} * C_i + C_i \Rightarrow I_1(\tau_i) \leq (A_k + S_k)U_i + C_i
\]

We also know that the upper bound of the carry-in done by \( m-1 \) tasks is \( \Delta C_i^{-1} \).

Therefore, we have

\[
(A_k + S_k)U(\tau) + \sum_{i=1}^{n} C_i + \Delta C_i^{-1} \geq (A_k + S_k) \cdot m
\]

Note that [TEST-1] does not suffer from the disadvantage in [BAR-EDF_np] that any task set with \( C_{\max} \geq D_{\min} \) will be rejected.

[TEST-1] works on any work-conserving non-preemptive scheduling algorithm, since it does not rely on any scheduling algorithm-specific property except requiring that no processor can be idle if there is some task instance awaiting for execution.

\( U_i, \sum C_i \) and \( S_{\min} \) all can be computed in linear time, and we can use linear-time selection [16] to compute \( \Delta C_i^{-1} \), so [TEST-1] has linear-time complexity, which is the same as that of [BAR-EDF_np]. [TEST-1] is still deeply pessimistic, since it used very coarse bounds on \( I_1(\tau_i) \) and \( I_2(\tau_i) \). In Section 5 and 6, we will present less-pessimistic test conditions for EDF_np and FP_np, by deriving more precise bounds on \( I_1(\tau_i) \) and \( I_2(\tau_i) \), where we assume each task \( \tau_i \) exactly executes for \( C_i \) and its instances are exactly released with separations of \( T_i \), and later in Section 7 we will show the test conditions are still correct if this assumption is broken.

5 The Improved Test for EDF_np

In the last section, we pessimistically assumed that every carry-out instance contributes to the overall work in the problem window. Actually, a carry-out instance can execute in the problem window only if it can interfere with \( J_k \). otherwise, it must execute after \( l_k \). Now we discuss the possible interference on a task instance \( J_k \) in EDF_np. We assume the priority ties are broken arbitrarily in the EDF_np scheduler.

Lemma 2. For EDF_np, if \( D_i > D_k \), the necessary condition for \( J_k \) to cause interference to \( J_k \) is \( r_i < r_k \), i.e., \( J_i \) must be released earlier than \( J_k \); if \( D_i \leq D_k \), the necessary condition for \( J_k \) to cause interference to \( J_k \) is \( d_i \leq d_k \), i.e., \( J_i \)’s absolute deadline must be no later than that of \( J_k \).
than \( d_k \), so it also can not cause the first type of interference. If \( D_i \leq D_k \), suppose \( d_i > d_k \), then \( J_i \) can not cause the first type of interference; since \( D_i \leq D_k \), \( J_i \)’s release time must be later than \( r_k \), so it also can not cause the second type of interference.

5.1 Computing \( I_1(\tau_i) \) for EDF

![Figure 3. the worst case of \( I_1(\tau_i) \) if \( i = k \).](image)

At first we compute \( I_1(\tau_i) \) with \( i = k \), i.e., the worst-case work done by \( \tau_k \)’s body instances. As shown in Figure 3, the number of \( \tau_k \)’s body instances is \( \left\lfloor \frac{A_k}{T_k} \right\rfloor \), so we have

\[
I_1^1(\tau_i) = \left\lfloor \frac{A_k}{T_k} \right\rfloor \ast C_k \tag{8}
\]

Next we will compute \( I_1(\tau_i) \) with \( i \neq k \). The following Lemma shows the worst case of \( I_1(\tau_i) \) of a task \( \tau_i \) with \( i \neq k \).

**Lemma 3.** The worst case of \( I_1(\tau_i) \) (\( i \neq k \)) occurs when one of \( \tau_i \)’s instances is released at the time-instant \( t_o \).

**Proof.** To prove the worst case of \( I_1(\tau_i) \) occurs when one of \( \tau_i \)’s instances is released at the time-instant \( t_o \), we should prove that based on this case, moving all \( \tau_i \)’s releases rightwards for a distance \( x (x < T_i) \) will not increase \( I_1(\tau_i) \)\(^1\).

We use \( J_i^p \) to denote the first body instance of \( \tau_i \). At the left end of the problem window, \( J_i^p \)’s release time is \( t_o - T_i + x \) after being moved rightwards for \( x \). Since \( x < T_i \), \( J_i^p \)’s release time is still earlier than \( t_o \), and by the definition of \( I_1(\tau_i) \), \( J_i^p \) has no contribution to \( I_1(\tau_i) \) after being moved, so there is no increase to \( I_1(\tau_i) \) at the left end of the problem window. At the right end of the problem window, trivially there is no increase to \( I_1(\tau_i) \). So moving all \( \tau_i \)’s releases rightwards for a \( x (x < T_i) \) will not increase \( I_1(\tau_i) \).

Now we compute \( I_1(\tau_i) \) (\( i \neq k \)) in this worst case:

1. \( D_i \leq D_k \). By Lemma 2, we know a task instance of \( \tau_i \) with \( D_i \leq D_k \) can interfere with \( J_k \) only if its deadline is no later than \( d_k \). We use \( \alpha_1 \) to denote the distance between \( t_o \) and the deadline of \( \tau_i \)’s last instance released before \( t_o \):

\[
\alpha_1 = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor \ast T_i + D_i \tag{9}
\]

Moving \( \tau_i \)'s releases for any distance can be transformed to an equal form of moving them rightwards (or leftwards) for \( x (x < T_i) \).

![Figure 4. the worst case of \( I_1(\tau_i) \) if \( D_i \leq D_k \).](image)

(a) \( \alpha_1 > A_k + D_k \). As shown in Figure 4-(a), the deadline of \( \tau_i \)’s last instance released in the problem window is later than \( d_k \), so it has no contribution to \( I_1(\tau_i) \). The number of \( \tau_i \)’s instances contributing to \( I_1(\tau_i) \) is \( \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor \). So we have:

\[
I_1^2(\tau_i) = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor \ast C_i \tag{10}
\]

(b) \( \alpha_1 \leq A_k + D_k \). As shown in Figure 4-(b), the deadline of \( \tau_i \)’s last instance released in the problem window is no later than \( d_k \), so it contributes to \( I_1(\tau_i) \), and the contribution is bounded by both \( C_i \) and \((A_k + S_k) \mod T_i \). So we have:

\[
I_1^2(\tau_i) = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor \ast C_i + \min(C_i, (A_k + S_k) \mod T_i) \tag{11}
\]

2. \( D_i > D_k \). By Lemma 2, we know an instance of \( \tau_i \) with \( D_i > D_k \) can interfere with \( J_k \) only if its release time is earlier than \( r_k \). We use \( \alpha_2 \) to denote the distance between \( t_o \) and the release time of \( \tau_i \)’s last instance released in the problem window:

\[
\alpha_2 = \left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor \ast T_i \tag{12}
\]

(a) \( A_k = 0 \). If \( A_k = 0 \), then \( t_o = r_k \). Since \( D_i \geq D_k \), any task instance released no earlier than \( t_o \) has deadline later than \( d_k \), so it can not interfere with \( J_k \). In this case \( I_1(\tau_i) = 0 \).

(b) \( \alpha_2 \geq A_k \). As shown in Figure 5(a), the release time of \( \tau_i \)’s last instance released in the problem window is no earlier than \( r_k \), so it can not interfere with \( J_k \). The number of \( \tau_i \)’s instances contribut-
ing to $I_1(\tau_i)$ is $\left\lfloor \frac{A_k + S_k}{T_i} \right\rfloor$. So in this case $I_1(\tau_i)$ is computed by Equation 10.

(c) $\alpha_2 < A_k$. As shown in Figure 5(b), the release time of $\tau_i$'s last instance released in the problem window is earlier than $r_k$, so it contributes to $I_1(\tau_i)$, and its contribution is bounded by both $C_i$ and $(A_k + S_k) \mod T_i$. So in this case $I_1(\tau_i)$ is computed by Equation 11.

By the discussions above, we can compute $I_1(\tau_i)$ for EDF$_{np}$ by:

\[
I_1(\tau_i) = \begin{cases} 
0 & D_i > D_k \land A_k = 0 \\
I_1^1(\tau_i) & i = k \\
I_1^2(\tau_i) & (i \neq k \land D_i \leq D_k \land A_k > A_k + D_k) \\
I_1^3(\tau_i) & \forall(D_i > D_k \land \alpha_2 > A_k > 0) \\
I_1^4(\tau_i) & \text{otherwise} 
\end{cases}
\]

(13) where $I_1^1(\tau_i)$, $I_1^2(\tau_i)$, $I_1^3(\tau_i)$, $\alpha_1$ and $\alpha_2$ are defined in Equation 8, 10, 11, 9 and 12 respectively.

5.2 Computing $I_2(\tau_i)$ for EDF$_{np}$

At first we compute $I_2(\tau_i)$ with $i = k$, i.e., the worst-case woke done by $\tau_k$'s carry-in and body instances. As shown in Figure 6, if we take the interval between the deadlines of two adjoining instances of $\tau_i$ as a "unit", there are $\left\lfloor \frac{A_k + D_k}{T_k} \right\rfloor$ such "units" in the time interval $[t_o, d_k]$. The carry-in is bounded by both $C_k$ and $(A_k + D_k) \mod T_k$. At the same time, the work done by $J_k$ itself should be subtracted. So for $\tau_i$ with $i = k$, we have:

\[
I_2^1(\tau_i) = \left\lfloor \frac{A_k + D_k}{T_k} \right\rfloor \times C_k + \min(C_k, (A_k + D_k) \mod T_k) - C_k 
\]

(14) In the following we will compute $I_2(\tau_i)$ with $i \neq k$ in Lemma 4, 5 and 6.

**Lemma 4.** The worst case of $I_2(\tau_i) (i \neq k)$ occurs when one of $\tau_i$'s instance has its deadline at $d_k$, if

\[
D_i \leq D_k \land S_i > C_k
\]

and in this case we can compute $I_2(\tau_i)$ by:

\[
I_2^2(\tau_i) = \left\lfloor \frac{A_k + D_k}{T_i} \right\rfloor \times C_i + \min(C_i, (A_k + D_k) \mod T_i)
\]

(16)

**Proof.** Let $J^p_i$ be the instance with its deadline at $d_k$, and by Lemma 2, we know $J^p_i$ may interfere with $J_k$. Since $S_k \geq C_i$, $J^p_i$’s contribution to $I_2(\tau_i)$ is $C_i$.

Now we examine whether $I_2(\tau_i)$ will be increased if we move all $\tau_i$’s releases leftwards for $x < T_i$.

After moving leftwards for $x (x < T_i)$, at the right end of the problem window, the contribution of $J^p_i$ is still $C_i$; the deadline of $J_i^{p+1}$ is at $d_k + T_i - x$, and since $x < T_i$, it is still later than $d_k$, so has no contribution to $I_2(\tau_i)$, so $I_2(\tau_i)$ will not be increased at the right end of the problem window. At the left end, trivially $I_2(\tau_i)$ is not increased. So $I_2(\tau_i)$ will not be increased after moving $\tau_i$’s releases leftwards for $x (x < T_i)$, so $d^o_i = d_k$ is the worst case for $I_2(\tau_i)$ for tasks with $D_i \leq D_k \land S_i > C_k$.

As shown in Figure 7, we take the interval between the deadlines of two adjoining instances of $\tau_i$ as a "unit", there are $\left\lfloor \frac{A_k + D_k}{T_k} \right\rfloor$ such "units" in the time interval $[t_o, d_k]$. The carry-in is bounded by both $C_k$ and $(A_k + D_k) \mod T_k$. So we can compute $I_2(\tau_i)$ by Equation 16.

**Lemma 5.** The worst case of $I_2(\tau_i) (i \neq k)$ occurs when one of $\tau_i$’s instance is released at $r_k - 1$, if

\[
D_i > D_k \land S_i \geq C_i
\]

(17) and in this case we can compute $I_2(\tau_i)$ by:
which equals to max(0, (A_k - 1) mod T_i - (T_i - D_i)). \hfill \Box

Lemma 6. The worst case of \( I_2(\tau_i) \) (i \( \neq k \)) occurs when one of \( \tau_i \)'s instances is released at \( x_k \) - \( C_i \), if
\[
(D_k \leq D_k \wedge S_i \leq C_i) \vee (D_k > D_k \wedge S_k < C_i)
\]
and in this case we compute \( I_2(\tau_i) \) by:
\[
I_2^b(\tau_i) = \begin{cases} 
A_k + S_k & A_k + S_k \leq C_i \\
\left\lfloor \frac{A_k + S_k - C_i}{T_i} \right\rfloor + 1 + C_i + \omega & A_k + S_k > C_i
\end{cases}
\]
where
\[
\omega = \min(C_i, \max(0, (A_k - 1) mod T_i - (T_i - D_i)))
\]
\begin{equation}
(21)
\end{equation}

Proof by Figure 9, it is easy to see that if \( A_k + S_k \leq C_i \), the worst-case \( I_2(\tau_i) \) (i \( \neq k \)) is \( A_k + S_k \). In the following we will elaborate on the case of \( A_k > 0 \).

Let \( J^p \) be \( \tau_i \)'s task instance released at \( r_k - 1 \), as shown in Figure 8. By Lemma 2 we know \( J^p \) can interfere with \( J_k \). Since \( S_k \geq C_i \), the contribution of \( J^p \) is \( C_i \).

Now we examine whether \( I_2(\tau_i) \) will be increased if we move all \( \tau_i \)'s releases leftwards for \( x < T_i \).

After moving leftwards for \( x \) (\( x < T_i \)), at the right end of the problem window, the contribution of \( J^p \) is still \( C_i \); the release time of \( J^p+1 \) is \( r_k - 1 + T_i - x \), since \( x < T_i \), it is still not earlier than \( r_k \), so \( J^p+1 \) has no contribution to \( I_2(\tau_i) \). Therefore, \( I_2(\tau_i) \) will not be increased at the right end of the problem window. At the left end, trivially the contribution will not be increased. So \( I_2(\tau_i) \) will not be increased after moving \( \tau_i \)'s releases leftwards for \( x < T_i \), so \( r^p = r_k - 1 \) is the worst case for \( I_2(\tau_i) \) for tasks with \( D_k > D_k \wedge S_k \leq C_i \).

If \( A_k > 0 \), as shown in Figure 8, the number of \( \tau_i \)'s body instances is \( \left\lfloor \frac{A_k - 1}{T_i} \right\rfloor \), the carry-out is \( C_i \), the carry-in is bounded by both \( C_i \) and the distance between \( t_k \) and the deadline of the carry-in instance, which equals to max(0, \( (A_k - 1) \) mod \( T_i - (T_i - D_i) \)).

\begin{equation}
(19)
\end{equation}
In summary, \( I_2(\tau_i) \) will not be increased if we move \( \tau_i \)'s releases leftwards for \( x < T_i \). So \( r^p_i = l_k - C_i \) is the worst case for \( I_2(\tau_i) \).

The number of body instances is \[ \sum \frac{A_k + S_k - C_i}{T_i} \]; the carry-out is \( C_i \); the carry-in is bounded by both \( C_i \) and the distance between \( t_i \) and \( D_i \), and the deadline of the carry-in instance, which equals to \( \max(0, (A_k + S_k - C_i) \mod T_i - (T_i - D_i)) \). So we can compute \( I_2(\tau_i) \) by Equation 21.

By the discussions above, we can compute \( I_2(\tau_i) \) for EDF for \( m \) processors by:

\[
I_2(\tau_i) = \begin{cases} 
I_2^1(\tau_i) & i = k \\
I_2^2(\tau_i) & i \neq k \land D_i \leq D_k \land S_i \geq C_k \\
I_2^3(\tau_i) & D_i > D_k \land S_i \geq C_i \\
I_2^4(\tau_i) & \text{otherwise}
\end{cases}
\]

where \( I_2^2(\tau_i), I_2^3(\tau_i), I_2^4(\tau_i) \) and \( I_2^5(\tau_i) \) are defined in Equation 14, 16, 18, 21 respectively.

5.3 A New Test Condition for EDF

By now we have obtained a sufficient schedulability test condition for EDF for \( m \) processors by Lemma 1 and the computation of \( I_1(\tau_i) \) and \( I_2(\tau_i) \) above.

**Theorem 2.** [TEST-EDF] A task set is EDF schedulable on \( m \) identical processors if for any task \( \tau_k \) and for any \( A_k \) we have:

\[
\sum_{\tau_i \in \tau} I_1(\tau_i) + \Delta^{m-1}_{I_{g}} < (A_k + S_k) \cdot m
\]

where \( I_1(\tau_i) \) and \( I_2(\tau_i) \) are computed by Equations 13 and 23.

For any given \( \tau_k \) and \( A_k \), the left-hand side of Condition 24 can be evaluated in time \( \mathcal{O}(n) \), since the time for computing \( I_1(\tau_i) \), \( I_2(\tau_i) \) and \( I_4(\tau_i) \) for each \( \tau_i \) is \( \mathcal{O}(n) \), and we can use linear-time selection [16] to compute \( \Delta^{m-1}_{I_{g}} \). The next theorem tells us the range of \( A_k \) that should be tested:

**Theorem 3.** For any task set with \( U(\tau) < m \), if condition 24 is to be violated for any \( A_k \), then it is violated for some \( A_k \) that satisfies the condition below:

\[
A_k \leq \frac{\sum^n a_i + \Delta^{m-1}_{C_i}}{m - U(\tau)} - S_k
\]

**Proof.** As shown in Section 4 (Inequality 6), we know:

\[
I_1(\tau_i) \leq (A_k + S_k) \cdot U_i + C_i
\]

Similarly, we know that:

\[
I_2(\tau_i) \leq (A_k + S_k) \cdot U_i + 2C_i
\]

If Condition 24 is violated, it must be true that

\[
\sum^n a_i + \Delta^{m-1}_{C_i} + (A_k + S_k)U(\tau) \geq (A_k + S_k)m
\]

Since \( A_k \) is a non-negative integer, Condition 24 can be checked in time pseudo-polynomial in the task parameters, for all task systems \( \tau \) of which \( U(\tau) \) is bounded by a constant strictly less than the number of processors \( m \).

As mentioned in Section 4, the effect of the over-estimation of the carry-in and carry-out is more severe with smaller \( A_k \) and \( S_k \) values. So one should check the tasks in increasing order of their \( S_k \)s, and in increasing order of \( A_k \)s for each task, to advance the testing efficiency. We have tested task sets consisting of one hundred of tasks with each \( T_i \) uniformly distributed in \([10, 2000]\). The tests of 1,000,000 such task sets are finished in several minutes. We believe this rates suggest that [TEST-EDF] is not only applicable to off-line schedulability test, but also a good candidate for on-line admission control with moderate-scale task systems.

6 The Improved Test for FP

We can also obtain a test condition for FP by computing the worst-case \( I_1(\tau_i) \) and \( I_2(\tau_i) \) in FP. We omit the analysis procedure due to space limitation and refer interested readers to the technical report [4] for details.

**Theorem 4.** [TEST-FP] A task set is FP schedulable on \( m \) processors if for any task \( \tau_k \) and for any \( A_k \) we have:

\[
\sum_{\tau_i \in \tau} I_1(\tau_i) + \Delta^{m-1}_{I_{g}} < (A_k + S_k) \cdot m
\]

where

\[
I_1(\tau_i) = \begin{cases} 
0 & P(\tau_i) < P(\tau_k) \land A_k = 0 \\
I_1^1(\tau_i) & i = k \\
I_1^2(\tau_i) & P(\tau_i) < P(\tau_k) \land \alpha_2 \geq A_k > 0 \\
I_1^3(\tau_i) & \text{otherwise}
\end{cases}
\]

\[
I_2(\tau_i) = \begin{cases} 
I_2^1(\tau_i) & i = k \\
I_2^2(\tau_i) & P(\tau_i) < P(\tau_k) \land S_i \geq C_i \\
I_2^3(\tau_i) & \text{otherwise}
\end{cases}
\]

\(I_1^1(\tau_i), I_1^2(\tau_i), I_1^3(\tau_i), I_2^1(\tau_i), I_2^2(\tau_i), I_2^3(\tau_i)\) and \(\alpha_2\) are defined in Section 5.

7 Robustness

A scheduling algorithm is said to be execution time robust (inter-release separation robust) if decreasing the execution times (increasing the inter-release separation) of task instances in a schedulable task set does not lead to deadline violations.

The robustness property is important. If a scheduling algorithm is robust, then the system designer only needs to consider the boundary values \((C_i, T_i)\) to determine if the system is
8 Performance Evaluation

The only known schedulability test condition for EDF$\_np$ is [BAR-EDF$\_np$], and there is no known test condition for FP$\_np$ to our best knowledge. So we will compare [BAR-EDF$\_np$] with our proposed test conditions [TEST-1], [TEST-EDF$\_np$] and [TEST-FP$\_np$]. [TEST-EDF$\_np$] and [TEST-FP$\_np$] are superior to [TEST-1], which means a task set accepted by [TEST-1] can also be accepted by [TEST-EDF$\_np$] and [TEST-FP$\_np$]. In general, our new test conditions is not comparable to [BAR-EDF$\_np$], i.e., we can construct a task set accepted by our test conditions but rejected by [BAR-EDF$\_np$], as well as a task set accepted by [BAR-EDF$\_np$] but rejected by our tests. In the following we will use randomly generated task sets to compare the average performances of these test conditions in terms of acceptance ratio. Additionally, we will also compare the performance of the test conditions for EDF$\_np$ with the test condition (referred to as [BASE-EDF]) in [2] for preemptive EDF, which is the base of the analysis in this paper.

We follow the method in [17] to generate task sets: A task set of $m + 1$ tasks was generated and tested. Then we increase the number of tasks by 1 to generate a new task set, and all the schedulability tests were run on the new task set. This process was repeated until the total processor utilization exceeded $m$. The whole procedure was then repeated, starting with a new task set of $m + 1$ tasks, until 1,000,000 task sets have been generated and tested. This method of generating random task sets produces a fairly uniform distribution of total utilizations, except at the extreme end of low utilization.

The task parameter setting in Figure 11(a) is as follows: The processor number is 6; for each task, $T_i$ is uniformly distributed in $[10, 20]$, the ratio between $D_i$ and $T_i$ is uniformly distributed in $[0.8, 1]$, and $U_i$ is uniformly distributed in $[0.2, 0.4]$. In this experiment, [TEST-1] performs slightly better than [BAR-EDF$\_np$], and they are clearly outperformed by the improved test conditions. [BASE-EDF] outperforms all the test conditions for non-preemptive scheduling, since more interference caused by non-preemption blocking should be taken into account in the tests for non-preemptive scheduling.

In Figure 11(b), we change the range of $U_i$ to $[0.1, 0.6]$ and keep other settings the same as in Figure 11(a). It is shown that as the average task utilization increases, the performance of [BAR-EDF$\_np$] degrades, while the effect on our proposed test conditions is much smaller. The reason is that [BAR-EDF$\_np$] is more sensitive to the $C_{\text{max}}$ value. An interesting phenomena shown in this experiment is that [TEST-EDF$\_np$] performs very close to (even a little better than) [BASE-EDF]. The reason is that the length of the problem window in the analysis of non-preemptive scheduling is $C_i$ shorter than in the analysis of preemptive scheduling, which compensates the extra interference caused by non-preemption blocking in some degree. As discussed in Section 5.3, most of the failures in the testing occur with small $A_k + S_k$ values, so this compensation is significant when $S_k$ is relatively small.

In Figure 11(c), we change the range of $T_i$ to $[10, 100]$ based
Figure 11. performance comparison of the test conditions

on the setting in Figure 11(a), which means different tasks have a wider varying scale range. In this case, the performance of all test conditions for non-preemptive scheduling degrades rapidly, while the effect on [BASE-EDF] is trivial. This result accords with the intuition that tasks with long execution time are harmful to the schedulability of short-urgent tasks due to non-preemptive blocking.

From the above experiments, we can see that the test conditions proposed in this paper, especially the improved test conditions, have a significant performance improvement compared with [BAR-EDFnp]. In general, the performance of our proposed test conditions is inferior to the test condition [BASE-EDF] for preemptive scheduling, while in some special cases, the performance of [TEST-EDFnp] is close to (or better than) [BASE-EDF].

9 Conclusions

As observed by Baruah [2], "global scheduling is fundamentally different from, and seems much more difficult than partitioned scheduling". In this paper, we take another stab at this tough problem by presenting new schedulability test conditions for work-conserving (necessarily global) non-preemptive scheduling on multiprocessor platforms, by building upon the techniques of Baker [6] and Baruah [2]. We firstly derive a linear-time test condition which works on any work-conserving non-preemptive scheduling algorithms. Then we improve the analysis and present test conditions of pseudo-polynomial time-complexity for EDFnp and FPnp, which significantly outperform the existing result.

The pessimism of the analysis in this paper mainly comes from the assumption that worst-case interferences by all tasks happen simultaneously, which is actually not necessary. As for future work, we plan to employ ILP or SAT methods to identify impossible scenarios, in order to obtain less-pessimistic schedulability tests.

References