Non-Preemptive Real-Time Multiprocessor Scheduling Beyond Work-Conserving

Hyeongboo Baek
Dept. of Computer Science and Engineering
Incheon National University (INU)
Republic of Korea
hbbaek@inu.ac.kr

Jaeheon Kwak
School of Computing
KAIST
Republic of Korea
0jaehunny0@kaist.ac.kr

Jinkyu Lee†
Dept. of Computer Science and Engineering
Sungkyunkwan University (SKKU)
Republic of Korea
jinkyu.lee@skku.edu

***Abstract*—Although essential for inherently non-preemptive tasks and favorable to tasks with large preemption/migration overheads, non-preemptive scheduling has not been thoroughly studied compared to preemptive scheduling. In particular, existing studies for non-preemptive scheduling could not effectively exploit being *non-work-conserving* (i.e., idling processor(s) intentionally), failing to achieve its full schedulability capability. In this paper, we propose the first non-preemptive scheduling framework that covers *work-conserving-infeasible* task sets (each of which is proven unschedulable by *every* work-conserving non-preemptive scheduling), without knowledge of future release patterns of tasks (i.e., without *clairvoyance*). To this end, we first discover the following principle: without clairvoyance, it is *impossible* to generate a feasible schedule for work-conserving-infeasible task sets on a *uniprocessor* platform. To make it *possible* on a *multiprocessor* platform, we design the `NWC(N)-NP-`$*$ framework that systematically idles up to `N` processors so as to enable `N` designated tasks (that yield work-conserving-infeasibility) to be schedulable without clairvoyance, and derive important properties of the framework. We then target the framework associated with fixed-priority scheduling (as a prioritization policy), and develop its schedulability test by utilizing the framework's properties. Our simulation results demonstrate that the proposed framework successfully covers a number of work-conserving-infeasible task sets, none of which can be deemed schedulable by any previous approach.**

I. Introduction

Since the seminal work in [1], which provides timing guarantees for periodic/sporadic jobs invoked by a set of real-time tasks, preemptive scheduling has been extensively studied from uniprocessor to multiprocessor platforms. On the other hand, although non-preemptive scheduling is essential to inherently non-preemptive tasks (e.g., interrupts and transactional operations) and favorable to tasks with large preemption/migration overheads, its underlying theories have not yet matured, especially for multiprocessor scheduling. Moreover, most existing studies concerning non-preemptive multiprocessor scheduling have been biased toward work-conserving scheduling, under which no processor can be left idle as long as there is at least one pending job with remaining execution (e.g., [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14] and many others), while there have been only several studies regarding non-work-conserving scheduling [15], [16], [17], [18], [19], [20], [21].

Different from a set of sporadic preemptive tasks (assuming no preemption/migration overhead), it has been known that idling processor(s) can be beneficial in meeting job deadlines of a set of sporadic non-preemptive real-time tasks under the knowledge of future job release patterns (i.e., *clairvoyance*) [15], but it has not been discovered whether the same holds without such knowledge. Therefore, existing studies have failed to not only identify but also achieve full schedulability capability of non-preemptive scheduling. In this paper, we focus on a situation in which knowledge of future job release patterns is not available (i.e., *non-clairvoyance*), and aim at exploring non-work-conserving global scheduling for a set of sporadic non-preemptive tasks on a multiprocessor platform, by addressing the following questions.

- Q1. Is it possible for non-work-conserving scheduling to generate a feasible schedule for a *work-conserving-infeasible* task set (which is proven to be unschedulable by *every* work-conserving non-preemptive scheduling)?
- Q2. If it is possible, how can we systematically determine the processor idling time without knowledge of future job release patterns?
- Q3. After designing a framework that addresses Q2, how can we provide timing guarantees for a task set under a framework associated with a target prioritization policy?

To answer Q1, we first derive a condition for a work-conserving-infeasible task set from the following example. Suppose that there are three jobs J_1 (release time = x, execution time = 2, deadline = $x + 12$), J_2 (0, 12, 22) and J_3 (0, 12, 22) executed on two processors, where $x \geq 1$. By every work-conserving scheduling, J_2 and J_3 start their execution at $t = 0$ and finish their execution at $t = 12$, and J_1 cannot start its execution until $t = 12$, which yields its deadline miss if $x = 1$. Using such a work-conserving-infeasible example, we prove that it is *impossible* for non-clairvoyant non-preemptive scheduling to yield a feasible schedule for a work-conserving-infeasible task set on a *uniprocessor* platform. To make it *possible* on a *multiprocessor* platform, we need to analyze why a job deadline miss happens in the example, which is due to non-preemptiveness of lower-priority jobs' execution that starts before a higher-priority job's release. To break such a fundamental limit of work-conserving non-preemptive scheduling, we consider idling processor(s), as it can prevent lower-priority job(s) from starting their execution before a higher-priority job's release.

†Jinkyu Lee is the corresponding author.

2576-3172/20/$31.00 ©2020 IEEE
DOI 10.1109/RTSS49844.2020.00020

However, it is generally challenging to decide when and how long we idle processor(s), because (i) the decision should yield not only timely execution of the job of interest (such as J_1) but also a predictable effect on timely execution of other jobs (such as J_2 and J_3) and (ii) the decision should be made without knowing future job release patterns. In the previous example with $x = 1$, suppose that we reserve a processor for J_1 by idling it until J_1 is released. Then, at $t = 0$, only one of J_2 or J_3 starts its execution, and J_1 starts its execution as soon as it is released at $t = 1$ on the reserved processor; once J_1 finishes at $t = 3$, one of J_2 or J_3 (that did not start its execution at $t = 0$) can start its execution and finish it before its deadline. Although such a processor idling policy seems perfect for the example with $x = 1$, it does not work with $x \geq 9$ (meaning that J_1 is released no earlier than $t = 9$). This is because, once J_1 is released no earlier than $t = 9$, starts its execution on the reserved processor, and finishes its execution after two time units, one of J_2 or J_3 (that did not start its execution at $t = 0$) cannot start its execution before $t = 11$, yielding its deadline miss. Addressing this challenge along with Q2, we propose a non-work-conserving non-preemptive scheduling framework, called NWC(N)-NP-$*$, under which the timely execution of N designated tasks is guaranteed as long as the number of processors (m) is no smaller than $2 \cdot$ N. By assigning tasks that inevitably miss its deadline under every work-conserving scheduling to the designated tasks, the proposed framework has a potential to make work-conserving-infeasible task sets schedulable. This is possible by carefully idling at most N processors out of m processors. We design the NWC(N)-NP-$*$ framework under the following two principles regarding the schedulability of the N designated tasks and the other tasks, respectively.

- P1. We guarantee to always provide one chance to start the execution of every job of the N designated tasks no later than a time instant at which the job will inevitably miss its deadline without starting its execution, and execute the job at the chance (if needed), yielding no job deadline miss of the designated tasks.
- P2. We make the effect of P1 predictable so as to give potential timing guarantees for jobs of the non-designated tasks.

To achieve P1 and P2 without knowledge of future job release patterns, we exploit the notion of a time stamp for each designated task. We now explain a core principle of NWC(N)-NP-$*$ that utilizes the time stamp, using the previous example with unknown release time of J_1. Once we idle a processor from $t = 0$, only one of J_2 or J_3 can start its execution at $t = 0$ (we assume that J_2 starts its execution, without loss of generality); in this case, the time stamp for J_1 records the finishing time of J_2, which is $t = 12$. Then, instead of keeping idling a processor until J_1 is released, we can relax the idling from $t = 2$. This relaxation comes from the time stamp for J_1. That is, since J_1 can start its execution at $t = 12$ (indicated by its time stamp) in any case without reserving a processor, we can calculate the earliest time instant when we do not need to idle a processor for J_1, which is $t = 2$, calculated by (absolute deadline - release time - execution time) of J_1 ahead of $t = 12$, i.e., $12-(x+12-x-2) = 2$. As a result, if J_1 is released no later than $t = 2$, it can immediately start its execution on the reserved processor. Otherwise, J_1 can start its execution at $t = 12$ on the processor where J_2 finishes its execution; in this case, J_3 can start its execution at $t = 2$ on the processor whose idling is finished at $t = 2$. In both cases, there is no J_1's deadline miss, which yields the achievement of P1. At the same time, since we do not idle a processor for J_1 in $[2, 12)$, we can yield the processor to J_3 during $[2, 12)$, which helps address P2. Although the core principle of NWC(N)-NP-$*$ seems simple to implement for the job set in the example, it is complicated to formalize details of the principle for a set of recurrent real-time tasks, which will be presented in Section IV.

To address Q3, we derive the three following important properties for the proposed framework, which also prove the accomplishment of P1 and P2: (i) every job of the N designated tasks always meets its deadline; (ii) we can upper-bound the time for idling a processor for each designated task, in a similar manner as to a traditional sporadic real-time task; and (iii) the actual execution of a designated task and the processor idling time reserved by the task cannot occur at the same time. These three properties facilitate the development of a schedulability test for NWC(N)-NP-$*$, as follows. From (i), we do not need to consider the schedulability of the N designated tasks. From (ii), the schedulability test to be developed can accommodate the effect of the processor idling time as if it were an additional sporadic real-time task. Finally, (iii) allows to upper-bound the effects of the actual execution and the processor idling time for each designated task, making the schedulability test tighter.

As an example, we consider NWC(N)-NP-FP, which is NWC(N)-NP-$*$ associated with a prioritization policy of fixed priority (FP) [1]. We first recapitulate an existing schedulability test for work-conserving non-preemptive FP scheduling (denoted by WC-NP-FP) and its improved version. Utilizing the three properties of the NWC(N)-NP-$*$ framework, we finally develop schedulability tests for NWC(N)-NP-FP based on the existing schedulability tests.

To demonstrate the effectiveness of the proposed framework, we generate a large number of task sets and check whether each of them is schedulable by the schedulability tests for NWC(N)-NP-FP and its counterpart WC-NP-FP. Our simulation results demonstrate that the proposed framework successfully covers a number of work-conserving-infeasible tasks sets, none of which are deemed schedulable by any existing approach, which is a main contribution of this paper.

In summary, this paper makes the following contributions:

- Establishment of a theoretical foundation of non-work-conserving non-preemptive scheduling,
- Development of the first non-work-conserving non-clairvoyant non-preemptive scheduling framework, which is capable of finding feasible schedules for work-conserving-infeasible task sets,
- Derivations of the framework's properties to be used for developing a schedulability test for a framework associated with a target prioritization policy,
- Development of schedulability tests for the proposed non-work-conserving framework associated with FP (i.e., NWC(N)-NP-FP), based on these properties, and
- Demonstration of the effectiveness of the proposed framework via simulations.

The remainder of the paper is organized as follows. Section II presents the system model, assumptions, and notations.

Section III explains our motivation of idling processor(s) for non-preemptive scheduling. Section IV proposes the NWC(N)-$*$ framework and derives its properties. Section V develops its schedulability test when FP is employed as a prioritization policy. Section VI evaluates the effectiveness of the proposed framework, and finally Section VII concludes the paper.

II. System Model, Assumptions, and Notations

In this paper, we focus on a sporadic real-time task model [22] in which each task τ_i in a task set τ is represented by finite positive values T_i (the minimum separation), C_i (the worst-case execution time), and D_i (the relative deadline). Let $|\tau|$ denote[1] the number of tasks in τ. We restrict our attention to implicit- and constrained-deadline tasks, satisfying $D_i = T_i$ and $D_i \leq T_i$, respectively. We consider a quantum-based time slot, and let the length of one quantum be equal to one time unit, without loss of generality; all parameters of T_i, C_i, and D_i are integer values. Each task invokes a series of jobs; a job invoked by τ_i is released at least T_i time units after the release time of the preceding job of τ_i, and should finish its execution within D_i time units. We call a job of τ_i (or τ_i itself) *active* at t if the job of τ_i has remaining execution at t. For predictability, we assume that every job of $\tau_i \in \tau$ executes for *exactly* C_i time units; if the actual execution time is $(C_i - x)$ $(0 < x < C_i)$, then we delay the job state transition from execution to completion, by x time units. This can be simply implemented in the scheduler: once τ_i's job starts its execution on a processor, we set a timer for C_i and let the processor not service any other job until the timer expires. We assume that each job is independent and cannot be executed on more than one processor at the same time. We consider a multiprocessor system consisting of m (≥ 2) identical processors. We assume there are at least $m + 1$ tasks in τ; otherwise, τ is trivially schedulable.

Concerning the preemption policy and task affinity, we consider *non-preemptive global* scheduling. Unlike for preemptive scheduling, *non-preemptive* scheduling prevents any job from preempting a currently-executing job. Thus, it is possible for lower-priority jobs to block higher-priority jobs. By *global* scheduling, we mean that a job is allowed to start its execution on any processor, while partitioned scheduling restricts the execution of jobs of a task to only one designated processor. We consider *non-clairvoyant* scheduling, meaning that there is no prior knowledge of future job arrival times. Instead, we assume that the scheduler only knows the static task parameters T_i, C_i and D_i for every $\tau_i \in \tau$. A scheduling algorithm is said to be *work-conserving* if any processor *cannot* be left idle as long as there is at least one pending active job; a scheduling algorithm is said to be *non-work-conserving*, otherwise. For ease of presentation, we may omit the terms *non-preemptive*, *global* and/or *non-clairvoyant*, when we refer scheduling or a scheduling algorithm.

While there are infinitely many instances of job sets invoked by a given task set (due to sporadic job releases), we define the *schedulability* of a task set as follows: a task set τ is said to be *schedulable* by a scheduling algorithm A, if every instance of job sets invoked by τ does not have any single job deadline miss under A. In this paper, we define a *work-conserving-infeasible* task set, as one that is unschedulable by every work-conserving non-preemptive global scheduling.

[1] In this paper, $|A|$ denotes the number of elements in A.

III. Why Idling for Non-Preemptive Scheduling?

In this section, we discuss the limitations of work-conserving scheduling as well as the potential of non-work-conserving scheduling with its related work.

A. Limitation of work-conserving scheduling

While the work-conserving property always yields better timing guarantees than the non-work-conserving one under preemptive scheduling (assuming no preemption/migration cost), it is well known that the same cannot be said under non-preemptive scheduling, as follows.

Example 1: A task set $\tau = \{\tau_1 = (T_1 = 12, C_1 = 2, D_1 = 12), \tau_2 = \tau_3 = (22, 12, 22)\}$ is scheduled on a two-processor platform. Suppose that the first jobs of τ_2 and τ_3 are released at $t = 0$, and that of τ_1 is released at $t = 1$. Then, every work-conserving non-preemptive scheduling algorithm makes the two jobs of τ_2 and τ_3 start their executions at $t = 0$. Due to the two jobs' execution in $[0, 12)$, the first job of τ_1 does not have any chance to begin its execution until $t = 12$, yielding a deadline miss as shown in Fig. 1(a).

From the observation in Example 1, we present a condition for a work-conserving-infeasible task set.

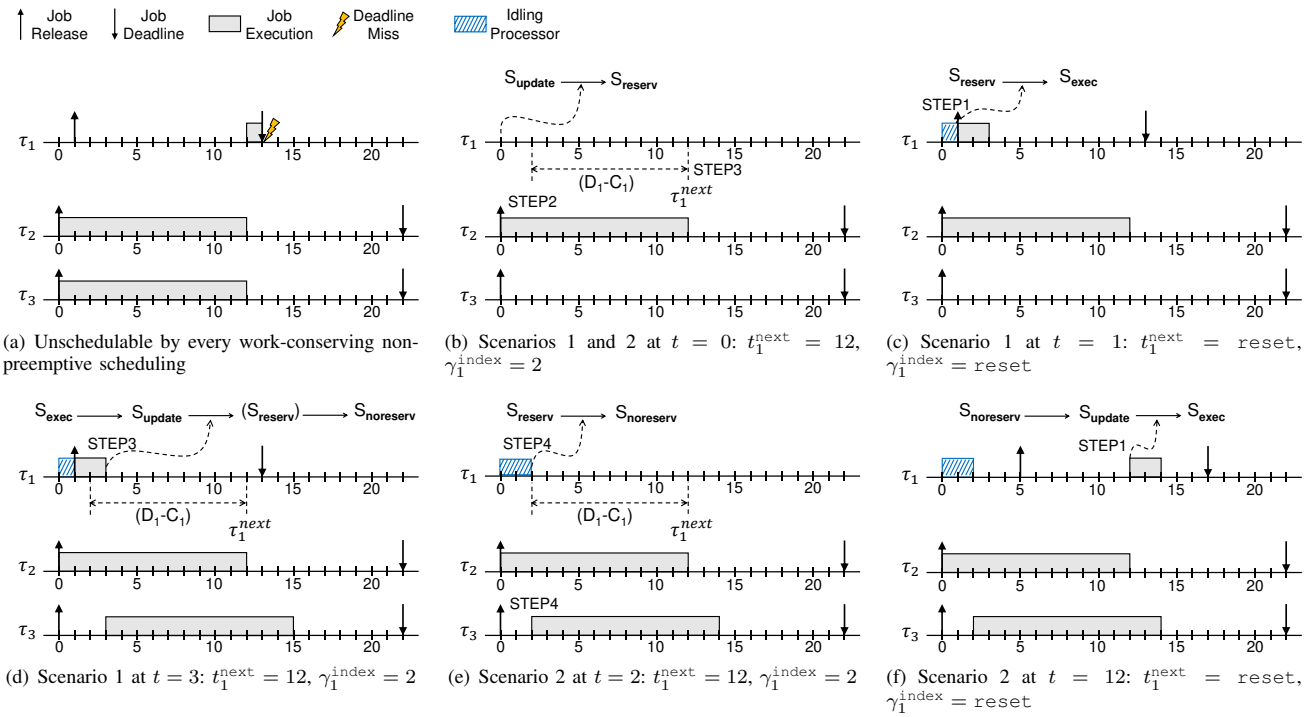
Lemma 1: A task set τ cannot be schedulable by any work-conserving non-preemptive global scheduling, if there exists a task τ_x that satisfies $C_i > D_x - C_x + 1$ for m other tasks $\tau_i \in \tau \setminus \{\tau_x\}$.[2]

Proof: Suppose that m jobs are released at $t = 0$, and a job of τ_x is released at $t = 1$. Then, every work-conserving non-preemptive scheduling starts the execution of the m jobs at $t = 0$. If the m jobs' execution time is strictly larger than $(D_x - C_x + 1)$, then the job of τ_x cannot begin its execution before or at $t = D_x - C_x + 1$, which yields the execution of strictly less than C_x time units until its absolute deadline $t = D_x + 1$. The lemma holds by the existence of this scenario. ■

In contrast to work-conserving scheduling, non-work-conserving scheduling can avoid the deadline miss situation in Example 1, as shown by the following example.

Example 2: Consider the same task set, job release times and platform as those in Example 1. Among the two jobs released at $t = 0$, we intentionally delay the start time of the execution of τ_3's job until $t = 3$, leading one processor to be idle in $[0, 1)$. Then, the first job of τ_1 can perform its execution in $[1, 3)$, and the job of τ_3 performs its execution in $[3, 15)$, resulting in no deadline miss for any job, as shown in Fig. 1(d).

[2] The lemma has been presented in some studies [6], [10], [21] either implicitly or explicitly.



(a) Unschedulable by every work-conserving non-preemptive scheduling

(b) Scenarios 1 and 2 at $t = 0$: $t_1^{\texttt{next}} = 12$, $\gamma_1^{\texttt{index}} = 2$

(c) Scenario 1 at $t = 1$: $t_1^{\texttt{next}} = \texttt{reset}$, $\gamma_1^{\texttt{index}} = \texttt{reset}$

(d) Scenario 1 at $t = 3$: $t_1^{\texttt{next}} = 12$, $\gamma_1^{\texttt{index}} = 2$

(e) Scenario 2 at $t = 2$: $t_1^{\texttt{next}} = 12$, $\gamma_1^{\texttt{index}} = 2$

(f) Scenario 2 at $t = 12$: $t_1^{\texttt{next}} = \texttt{reset}$, $\gamma_1^{\texttt{index}} = \texttt{reset}$

Fig. 1. Schedules of three tasks on two processors under work-conserving and non-work-conserving non-preemptive scheduling (details with notations to be explained in Section IV-A)

B. Potential of non-work-conserving scheduling and related work

While Example 2 demonstrates the effectiveness of idling a processor for non-preemptive scheduling, it seems that the example only works if we know that the job of τ_1 will be released at $t = 1$. Also, the example offers timing guarantees of one job per task only, which raises the following question. "Can we design a scheduling framework that determines when and how much we should idle processor(s) so as to meet every job deadline without knowledge of future job release patterns?" The following lemma presents the impossibility of such a framework on a uniprocessor platform.

Lemma 2: Suppose that the information concerning future job release times is not available. If Lemma 1 with $m = 1$ judges that τ is unschedulable by every work-conserving non-preemptive scheduling on a uniprocessor platform, then it is also *impossible* for a non-work-conserving non-preemptive scheduling algorithm to make τ schedulable.

Proof: Lemma 1 with $m = 1$ implies that there exist τ_x and τ_i that satisfy $C_i > D_x - C_x + 1$. We prove that it is possible to generate a scenario that yields a deadline miss. Suppose that a job of τ_i is released at $t = 0$.

Case 1. Suppose that the job of τ_i starts its execution at $t = t'$ for $t' \geq 0$. Then, if a job of τ_x is released at $t = t' + 1$, then the job will miss its deadline.

Case 2. Suppose that the job of τ_i does not start its execution. Then, it is sure for the job of τ_i to miss its deadline eventually (note that all the task parameters are *finite* positive values as mentioned in Section II).

In every case, a job misses its deadline. ■

Different from a uniprocessor platform, non-work-conserving scheduling on a multiprocessor platform can overcome the work-conserving-infeasible condition.

Claim 1: Suppose that the information on future job releases is not available. There exists a global non-work-conserving non-preemptive scheduling algorithm that makes a feasible schedule for a work-conserving-infeasible task set, which Lemma 1 deems unschedulable by every work-conserving non-preemptive global scheduling on a multiprocessor platform.

We show that the set of tasks with their job releases in Example 1 is schedulable on a two-processor platform by the following non-work-conserving scheduling algorithm with a prioritization policy of fixed priority scheduling (where τ_1 and τ_3 are the highest- and lowest-priority tasks, respectively), as shown in Fig. 1(d). Whenever a job of τ_3 (*likewise* τ_2) is ready to start its execution (in the example, the jobs of τ_2 and τ_3 are released at $t = 0$), it is checked whether a job of τ_2 (*likewise* τ_3) is executed. If not executed, the job of τ_3 (*likewise* τ_2) starts its execution; note that both jobs of τ_2 and τ_3 are ready, the job of τ_2 is selected to start its execution according to the fixed-priority prioritization policy (in the example, the job of τ_2 starts its execution at $t = 0$). If executed, the job of τ_3 (*likewise* τ_2) also checks the finishing time of the currently-executing job of τ_2 (*likewise* τ_3) (in the example, the job of τ_2 will finish its execution at $t = 12$), and does not start its execution by intentionally idling a processor until $D_1 - C_1 = 10$ time units ahead of the finishing time (in the example, until $t = 12 - 10 = 2$). If a job of τ_1 is released during the

processor idling time, the job starts its execution immediately (in the example, the job of τ_1 starts its execution at $t = 1$). Otherwise, the job can start its execution at the finishing time of the job of τ_2 or τ_3.

The scheduling idea does not require any knowledge of future job release times. For example, consider another job release pattern for τ_1: the first job of τ_1 is released at $t = 5$. Without knowing the job of τ_1's release time, we idle a processor until $t = 12 - (D_1 - C_1) = 2$ as shown in Fig. 1(e). Since the processor idling time is finished at $t = 2$, the processor services a job of τ_3 from $t = 2$. Even without processor idling, the job of τ_1 released at $t = 5$ can start its execution at $t = 12$, because a job of τ_2 finishes its execution at the time instant, as shown in Fig. 1(f), which implies no job deadline miss for all tasks. In fact, the scheduling algorithm explained so far is NWC(1)-NP-$*$, to be developed in the next section.

Related work. In the literature, there have been several attempts to design/analyze non-work-conserving non-preemptive scheduling on a uniprocessor platform, assuming that the future job release pattern is available. For example, Ekelin developed a scheduling algorithm for a set of independent jobs [15], while Nasri *et al.* designed a scheduling algorithm for the periodic (loose-)harmonic task model [16], [17], which was subsequently improved to support the periodic task model [19]. Also, there has been work in the direction of developing a new type of schedulability test that can be utilized for timing guarantees of existing non-work-conserving scheduling [18], [20]. Beyond uniprocessor scheduling, one study discussed non-work-conserving non-preemptive scheduling for mixed-criticality tasks [23], while another study focused on non-work-conserving non-preemptive scheduling on a multiprocessor platform, which requires information of future job release patterns [21].

On the other hand, there has been no study regarding the design of a non-work-conserving non-preemptive scheduling, without relying on information of future job release patterns, as explained in [24]. Therefore, this paper aims at improving the schedulability of a set of non-preemptive tasks on a multiprocessor platform without prior knowledge of future job release patterns, which can be achieved by systematically idling processor(s).

IV. NWC(N)-NP-$*$: Non-Work-Conserving Non-Preemptive Scheduling Framework

In this section, we propose a non-work-conserving framework for non-preemptive scheduling, called NWC(N)-NP-$*$. We then derive properties of this framework to be used for developing schedulability tests for the framework associated with a target prioritization policy.

A. NWC(N)-NP-$*$ framework

In the previous section, we proved a task set τ containing any task τ_x that satisfies the condition in Lemma 1 (i.e., $C_i > D_x - C_x + 1$ holds for m other tasks $\tau_i \in \tau \setminus \{\tau_x\}$) cannot be schedulable by every work-conserving non-preemptive global scheduling. Thus, it is necessary to deal with every τ_x that satisfies the condition in Lemma 1 in order to make τ schedulable. We refer to such τ_x as a designated task. We

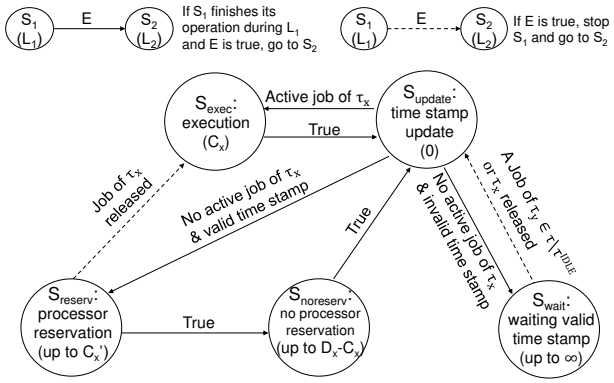


Fig. 2. State diagram for $\tau_x \in \tau^{\texttt{IDLE}}$

now describe the design of NWC(N)-NP-$*$, which allows up to N processors to remain idle for N designated tasks. For notational convenience, we let $\tau^{\texttt{IDLE}}$ denote a set of the N designated tasks, each of which incurs processor idling by the NWC(N)-NP-$*$ framework.

The proof of Lemma 1 indicates a job of $\tau_x \in \tau$ which satisfies the work-conserving-infeasible condition misses its deadline when the m other tasks' jobs begin their executions just before the job release of τ_x. Then, the job of τ_x does not have any chance to start its execution until the time instant at which the job will miss its deadline unless it starts its execution (i.e., the instant is its worst-case execution time ahead of its absolute deadline). To avoid the situation, we establish the two design principles P1 and P2, presented in Section I.

The key idea for achieving P1 and P2 is to exploit the notion of a time stamp for every designated task $\tau_x \in \tau^{\texttt{IDLE}}$ in order to systematically reduce the potential blocking time that hinders the execution of τ_x, without knowledge of future job release patterns. A time stamp $t_x^{\texttt{next}}$ of τ_x is the finishing time of a currently-executing job of a non-designated task, and therefore it indicates a time instant at which a processor will be available in any case. Note that a valid $t_x^{\texttt{next}}$ is assigned only after the "currently-executing job" starts its execution, by calculating the time to start its execution plus its execution time, which does not require any prior knowledge of release patterns and scheduling policies; therefore, $t_x^{\texttt{next}}$ may have no "currently-executing job", which makes $t_x^{\texttt{next}}$ invalid (to be explained in Algorirthm 1). Also, $t_x^{\texttt{next}}$ is associated with a task index y of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ (by assigning y to $\gamma_x^{\texttt{index}}$), which implies that a processor is guaranteed to be available for τ_x at $t_x^{\texttt{next}}$ when a job of τ_y completes its execution. If there are multiple designated tasks (i.e., $|\tau^{\texttt{IDLE}}| \geq 2$), then each $t_x^{\texttt{next}}$ has a unique $\gamma_x^{\texttt{index}}$, implying that each τ_x is guaranteed to execute after the corresponding τ_y finishes its execution. Then, NWC(N)-NP-$*$ ensures that the duration for each job (to be released in the future) of every τ_x to be blocked is no larger than $(D_x - C_x)$, as follows. NWC(N)-NP-$*$ reserves a processor for τ_x by idling it until $t_x^{\texttt{next}} - (D_x - C_x)$, and a job of τ_x immediately starts its execution if it is released in the middle of the processor idling time for τ_x. On the other hand, τ_x does not idle a processor in $[t_x^{\texttt{next}} - (D_x - C_x), t_x^{\texttt{next}})$. This does not yield a deadline miss of a job of τ_x released in the interval, because the job of τ_x can start its execution at $t_x^{\texttt{next}}$

and therefore the duration for the job of τ_x to be blocked is limited to $(D_x - C_x)$. The following example demonstrates how a time stamp $t_x^{\texttt{next}}$ is managed by NWC(N)-NP-$*$ with Figs. 1(b), (c), (e) and (f).

Example 3: Consider the task set from Example 1, and $\tau^{\texttt{IDLE}} = \{\tau_1\}$ and $\tau \setminus \tau^{\texttt{IDLE}} = \{\tau_2, \tau_3\}$ hold. In Fig. 1(b), although individual jobs of both τ_2 and τ_3 are released at $t = 0$, the job of τ_2 is executed on a processor and τ_1 idles the other processor for up to two time units (i.e., until $t_1^{\texttt{next}} - (D_1 - C_1) = 12 - 10 = 2$), where $t_1^{\texttt{next}}$ indicates the finishing time of a job of τ_2 (i.e., $t = 12$). A job of τ_1 immediately executes if it is released no later than $t = 2$ (as shown in Fig. 1(c)); otherwise (i.e., if it is released after $t = 2$), it executes at or after $t_x^{\texttt{next}}$ (as shown in Figs. 1(e) and (f)). Then, the timely execution of a job of τ_1 is guaranteed because the blocking time that hinders the execution of the job of τ_1 is upper-bounded by $(D_1 - C_1) = 10$ time units in both cases.

With the time stamp $t_x^{\texttt{next}}$, $\tau_x \in \tau^{\texttt{IDLE}}$ belongs to one of the following five states at t: execution $\texttt{S}_{\texttt{exec}}$, processor reservation (idling) $\texttt{S}_{\texttt{reserv}}$, no processor reservation $\texttt{S}_{\texttt{noreserv}}$, waiting for a valid time stamp $\texttt{S}_{\texttt{wait}}$, and time stamp update $\texttt{S}_{\texttt{update}}$, as shown in Fig. 2. Note that when the system starts, every $\tau_x \in \tau^{\texttt{IDLE}}$ starts with the state of $\texttt{S}_{\texttt{wait}}$. Then, $\tau_x \in \tau^{\texttt{IDLE}}$ exhibits different behaviors in different states, as follows.

- $\texttt{S}_{\texttt{update}}$ (whose duration is zero) checks whether there is an active job of τ_x. If so, the state of τ_x is transited to $\texttt{S}_{\texttt{exec}}$. Otherwise, we update the time stamp: if it is *valid* or *invalid*, then we change τ_x's state to $\texttt{S}_{\texttt{reserv}}$ or $\texttt{S}_{\texttt{wait}}$, respectively. Here, a *valid* time stamp $t_x^{\texttt{next}}$ indicates the finishing time of a currently-executing job of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ such that the job of τ_y does not yield a time stamp for any task other than τ_x. If there exists no valid time stamp for τ_x, then we set the time stamp to invalid.
- $\texttt{S}_{\texttt{reserv}}$ compares the current time t with $t_x^{\texttt{next}} - (D_x - C_x)$. If the former is earlier than the latter, then τ_x idles a processor until the latter. Otherwise, the state of τ_x is immediately changed to $\texttt{S}_{\texttt{noreserv}}$ (with a duration of zero for $\texttt{S}_{\texttt{reserv}}$). If there is an event of a job release of τ_x during $\texttt{S}_{\texttt{reserv}}$, then the state of τ_x is changed to $\texttt{S}_{\texttt{exec}}$; otherwise, after idling a processor until $t_x^{\texttt{next}} - (D_x - C_x)$, $\texttt{S}_{\texttt{reserv}}$ is changed to $\texttt{S}_{\texttt{noreserv}}$.
- Once reaching $\texttt{S}_{\texttt{noreserv}}$, τ_x just waits until $t_x^{\texttt{next}}$ without idling a processor, and then changes its state to $\texttt{S}_{\texttt{update}}$ at $t_x^{\texttt{next}}$.
- $\texttt{S}_{\texttt{exec}}$ performs the execution of a job of τ_x during C_x time units, and then transits to $\texttt{S}_{\texttt{update}}$.
- $\texttt{S}_{\texttt{wait}}$ waits until a job of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ or τ_x is released, and then is changed to $\texttt{S}_{\texttt{update}}$.

Then, P1 is satisfied as follows. If a job of τ_x is released, it begins its execution immediately unless the current state is $\texttt{S}_{\texttt{noreserv}}$. If the current state is $\texttt{S}_{\texttt{noreserv}}$, then it begins its execution at $t_x^{\texttt{next}}$ without missing any deadline (because the job does not wait more than $(D_x - C_x)$ time units after its release). Furthermore, P2 is satisfied as follows. As shown in Fig. 2, once τ_x starts to idle a processor in $\texttt{S}_{\texttt{reserv}}$, the idling time is upper-bounded by C'_x (to be calculated in Section IV-B). In addition, after the idling time, no processor is idle for τ_x during either C_x time units with $\texttt{S}_{\texttt{exec}}$ or $(D_x - C_x)$ time units with $\texttt{S}_{\texttt{noreserv}}$. We will formally prove the achievement of P1 and P2 in Section IV-B.

Algorithm 1 NWC(N)-NP-$*$ framework

Handling every finished job: For every job which is finished at t,

1: Remove the job from RQ (running queue).
2: **if** the removed job was invoked by $\tau_x \in \tau^{\texttt{IDLE}}$ **then**
3: $t_x^{\texttt{next}} \leftarrow$ invalid; $\gamma_x^{\texttt{index}} \leftarrow$ invalid
4: **end if**

Handling every released job: For every job which is released at t,

1: Insert the job into WQ (wait queue).

Scheduling: If there is at least one job which is finished or released at t, or any processor idling for $\tau_x \in \tau^{\texttt{IDLE}}$ is finished (i.e., $t_x^{\texttt{next}} - (D_x - C_x) = t$), then the following steps are performed:

1: // STEP 1: Start execution of jobs of $\tau_x \in \tau^{\texttt{IDLE}}$ in WQ
2: **for** every $\tau_x \in \tau^{\texttt{IDLE}}$ whose job is in WQ **do**
3: **if** $t_x^{\texttt{next}} = t$, $t_x^{\texttt{next}} =$ invalid, or $t_x^{\texttt{next}} - (D_x - C_x) > t$ holds **then**
4: Move the job of τ_x from WQ to RQ; start its execution; $t_x^{\texttt{next}} \leftarrow$ reset; $\gamma_x^{\texttt{index}} \leftarrow$ reset
5: **end if**
6: **end for**
7: // STEP 2: Fill up to $(m - \texttt{N})$ jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ in RQ
8: $N_{\texttt{RQ}} \leftarrow$ the number of jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ in RQ
9: $N_{\texttt{WQ}} \leftarrow$ the number of jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ in WQ
10: Move the $\max\left(0, \min(N_{\texttt{WQ}}, m - \texttt{N} - N_{\texttt{RQ}})\right)$ highest-priority job(s) of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ from WQ to RQ; start their executions
11: // STEP 3: Update $t_x^{\texttt{next}}$ for $\tau_x \in \tau^{\texttt{IDLE}}$
12: **for** every $\tau_x \in \tau^{\texttt{IDLE}}$ satisfying $t_x^{\texttt{next}} = t$ or $t_x^{\texttt{next}} =$ invalid **do**
13: $JS \leftarrow$ a set jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ in RQ such that there does not exist $\gamma_{x'}^{\texttt{index}} = y$ for $\tau_{x'} \in \tau^{\texttt{IDLE}}$
14: **if** $|JS| \geq 1 + m - 2 \cdot \texttt{N}$ **then**
15: $t_x^{\texttt{next}} \leftarrow$ the earliest finishing time of jobs in JS; $\gamma_x^{\texttt{index}} \leftarrow$ the task index of the job
16: **else**
17: $t_x^{\texttt{next}} \leftarrow$ invalid; $\gamma_x^{\texttt{index}} \leftarrow$ invalid
18: **end if**
19: **end for**
20: // STEP 4: Allocate the remaining available processors to jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$
21: $N_{\texttt{IDLE}} \leftarrow$ the number of jobs of $\tau_x \in \tau^{\texttt{IDLE}}$ with $t_x^{\texttt{next}} - (D_x - C_x) > t$
22: $N_{\texttt{WQ}} \leftarrow$ the number of jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ in WQ
23: Move the $\max\left(0, \min(N_{\texttt{WQ}}, m - N_{\texttt{IDLE}} - |\texttt{RQ}|)\right)$ highest-priority job(s) of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ from WQ to RQ; start their executions

Using the five states for every $\tau_x \in \tau^{\texttt{IDLE}}$, we present the NWC(N)-NP-$*$ framework in Algorithm 1. For every job that finishes its execution at t, we remove it from RQ (running queue); if the job belongs to $\tau_x \in \tau^{\texttt{IDLE}}$, we set $t_x^{\texttt{next}} \leftarrow$ invalid and $\gamma_x^{\texttt{index}} \leftarrow$ invalid. For every job released at t, we insert the job to WQ (wait queue).

If there is at least one job which is finished or released at t, or a job of $\tau_x \in \tau^{\texttt{IDLE}}$ relaxes the reserved (idling) processor (i.e., $t = t_x^{\texttt{next}} - (D_x - C_x)$), we perform STEPS 1–4. Note that when the system starts, we set $t_x^{\texttt{next}} \leftarrow$ invalid and $\gamma_x^{\texttt{index}} \leftarrow$ invalid for every $\tau_x \in \tau^{\texttt{IDLE}}$, and the state of every $\tau_x \in \tau^{\texttt{IDLE}}$ is assigned to $\texttt{S}_{\texttt{wait}}$.

- In STEP 1 (Lines 2–6), the following three situations

result in starting the execution of jobs of $\tau_x \in \tau^{\texttt{IDLE}}$ in WQ: transitions from $\texttt{S}_{\texttt{noreserv}}$ ($t_x^{\texttt{next}} = t$), $\texttt{S}_{\texttt{wait}}$ and then $\texttt{S}_{\texttt{update}}$ ($t_x^{\texttt{next}} = \texttt{invalid}$), and $\texttt{S}_{\texttt{reserv}}$ ($t_x^{\texttt{next}} - (D_x - C_x) > t$). Once $\tau_x \in \tau^{\texttt{IDLE}}$ starts its execution, its time stamp is reset, i.e., $t_x^{\texttt{next}} \leftarrow \texttt{reset}$ and $\gamma_x^{\texttt{index}} \leftarrow \texttt{reset}$.

- In STEP 2 (Lines 8–10), we fill up to $(m - \texttt{N})$ jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ in RQ, which is necessary for every $\tau_x \in \tau^{\texttt{IDLE}}$ to update its time stamp $t_x^{\texttt{next}}$ as valid if possible.
- In STEP 3 (Lines 12–19), we update $t_x^{\texttt{next}}$ for every $\tau_x \in \tau^{\texttt{IDLE}}$ that satisfies $t_x^{\texttt{next}} = t$ or $t_x^{\texttt{next}} = \texttt{invalid}$, which is performed in $\texttt{S}_{\texttt{update}}$. Let JS denote a set of jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ in RQ such that there does not exist $\gamma_{x'}^{\texttt{index}} = y$ for $\tau_{x'} \in \tau^{\texttt{IDLE}}$. If there exists at least $(1 + m - 2 \cdot \texttt{N})$ jobs in JS,[3] we assign a valid time stamp to $t_x^{\texttt{next}}$, by setting $t_x^{\texttt{next}}$ and $\gamma_x^{\texttt{index}}$ to the earliest finishing time of jobs in JS and the task index of the job with the earliest finishing time, respectively. Otherwise, we assign an invalid time stamp to $t_x^{\texttt{next}}$, by setting $t_x^{\texttt{next}} \leftarrow \texttt{invalid}$ and $\gamma_x^{\texttt{index}} \leftarrow \texttt{invalid}$.
- In STEP 4 (Lines 21–23), we allocate the remaining available processors to jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ after updating $t_x^{\texttt{next}}$ for $\tau_x \in \tau^{\texttt{IDLE}}$. We first calculate the number of jobs in $\tau_x \in \tau^{\texttt{IDLE}}$ each of which reserves (idles) a processor (denoted by $N_{\texttt{IDLE}}$). Then, we additionally execute jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ up to as many as $(m - N_{\texttt{IDLE}} - |\texttt{RQ}|)$.

Now, we present an example of how the NWC(N)-NP-$*$ framework operates, with Figs. 1(b)–(f).

Example 4: Recall the task set, job release times, and platform from Example 1. We apply NWC(1)-NP-$*$ with the prioritization policy of FP, where τ_1 and τ_3 have the highest and lowest priorities, respectively. Also $\tau^{\texttt{IDLE}}$ is set to $\{\tau_1\}$. We present two scenarios: (Scenario 1) Fig. 1(b)→(c)→(d) and (Scenario 2) Fig. 1(b)→(e)→(f), where a job of τ_1 is released during and after idling a processor, respectively. In both scenarios, we explain Algorithm 1 with τ_1's state transition.

Scenario 1 is as follows.

- Fig. 1(b): When the system begins at $t = 0$, $t_1^{\texttt{next}} \leftarrow \texttt{invalid}$ and $\gamma_1^{\texttt{index}} \leftarrow \texttt{invalid}$ are conducted. At $t = 0$, there is no action for STEP 1, and a job of τ_2 starts its execution by STEP 2. By STEP 3, $t_1^{\texttt{next}}$ is set to 12, which is the finishing time of the job of τ_2 (and $\gamma_1^{\texttt{index}} \leftarrow 2$ is conducted); $\texttt{S}_{\texttt{update}}$ is changed to $\texttt{S}_{\texttt{reserv}}$. Also, no action is taken for STEP 4. Because $t_1^{\texttt{next}} - (D_1 - C_1) = 2 > t$ holds, a processor starts to be idle until $t = 2$.
- Fig. 1(c): At $t = 1$, the newly released job of τ_1 starts its execution on the reserved (idling) processor by STEP 1; $\texttt{S}_{\texttt{reserv}}$ is changed to $\texttt{S}_{\texttt{exec}}$. In addition, $t_1^{\texttt{next}} \leftarrow \texttt{reset}$ and $\gamma_1^{\texttt{index}} \leftarrow \texttt{reset}$ are conducted. No actions are taken for STEPS 2–4.
- Fig. 1(d): At $t = 3$, the job of τ_1 is finished and its time stamp is set to invalid; $\texttt{S}_{\texttt{exec}}$ is changed to $\texttt{S}_{\texttt{update}}$. Then, no actions are taken for STEPS 1 and 2. In STEP 3, $t_1^{\texttt{next}}$ is set to 12 again, which is the finishing time of the job of τ_2 (and $\gamma_1^{\texttt{index}} \leftarrow 2$ is conducted); $\texttt{S}_{\texttt{update}}$ is changed to $\texttt{S}_{\texttt{noreserv}}$. In STEP 4, a job of τ_3 starts its execution because $t_1^{\texttt{next}} - (D_1 - C_1) = 2 \leq t$, meaning that τ_1 does not idle a processor.

Scenario 2 is as follows.

- Fig. 1(b): The actions are the same as those of Scenario 1.
- Fig. 1(e): At $t = 2$, no action is taken for STEPS 1–3. In STEP 4, a job of τ_3 starts its execution because $t_1^{\texttt{next}} - (D_1 - C_1) = 2 \leq t$, meaning that τ_1 does not idle a processor; $\texttt{S}_{\texttt{reserv}}$ is changed to $\texttt{S}_{\texttt{noreserv}}$.
- Fig. 1(f): At $t = 5$, a job of τ_1 is released, but it does not start its execution; τ_1's state is not changed (i.e., $\texttt{S}_{\texttt{noreserv}}$).
  At $t = 12$, $\texttt{S}_{\texttt{noreserv}}$ is changed to $\texttt{S}_{\texttt{update}}$, and a job of τ_1 starts its execution in STEP 1 (and $t_1^{\texttt{next}} \leftarrow \texttt{reset}$ and $\gamma_1^{\texttt{index}} \leftarrow \texttt{reset}$ are conducted); $\texttt{S}_{\texttt{update}}$ is changed to $\texttt{S}_{\texttt{exec}}$.
  At $t = 14$, the job of τ_1 completes its execution; $\texttt{S}_{\texttt{exec}}$ is changed to $\texttt{S}_{\texttt{update}}$. Then, $t_1^{\texttt{next}}$ is invalid since Line 14 of Algorithm 1 is not satisfied; therefore, $t_1^{\texttt{next}} \leftarrow \texttt{invalid}$ and $\gamma_1^{\texttt{index}} \leftarrow \texttt{invalid}$ are conducted according to Line 17 of the algorithm.

Although we design the NWC(N)-NP-$*$ framework to achieve P1 and P2, we do not formally discuss how and why the framework achieves these. In the next subsection, we prove that P1 and P2 hold under a certain condition, which helps to develop a schedulability test for the framework associated with a target prioritization policy.

B. Properties of NWC(N)-NP-$*$ framework

In this subsection, we derive properties of the NWC(N)-NP-$*$ framework, which are useful for developing schedulability tests for the framework that employs a target prioritization policy. First, the following lemma states that P1 holds under $m \geq 2 \cdot \texttt{N}$.

Lemma 3: Under NWC(N)-NP-$*$ on a multiprocessor platform with $m \geq 2 \cdot \texttt{N}$, every job of $\tau_x \in \tau^{\texttt{IDLE}}$ cannot miss its deadline.

Proof: We consider two cases at t: (Case i) there are at least $(m - \texttt{N})$ currently-executing jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$; and (Case ii) otherwise.

(Case i) In this case, all N tasks $\tau_x \in \tau^{\texttt{IDLE}}$ have a valid time stamp $t_x^{\texttt{next}}$ by STEP 3 of Algorithm 1 and the condition of $(m - \texttt{N} \geq \texttt{N})$. Therefore, τ_x idles a processor until $t_x^{\texttt{next}} - (D_x - C_x)$. If a new job of $\tau_x \in \tau^{\texttt{IDLE}}$ is released before $t_x^{\texttt{next}} - (D_x - C_x)$, it immediately starts its execution. If a new job of $\tau_x \in \tau^{\texttt{IDLE}}$ is released in $[t_x^{\texttt{next}} - (D_x - C_x), t_x^{\texttt{next}}]$, it starts its execution at $t_x^{\texttt{next}}$. If a new job of $\tau_x \in \tau^{\texttt{IDLE}}$ is not released until $t_x^{\texttt{next}}$, it will update its time stamp at $t_x^{\texttt{next}}$. In every case, there is no deadline miss for $\tau_x \in \tau^{\texttt{IDLE}}$.

(Case ii) In this case, since there are at most $(m - \texttt{N} - 1)$ currently-executing jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$, there exists at least

[3] Although it is more intuitive to apply "one job" instead of "$(1 + m - 2 \cdot \texttt{N})$ jobs" in the phrase, the latter makes it possible to reduce the maximum idling period compared to the former, to be used to derive Lemma 4. Briefly speaking, the latter can allow not to select the $(m - 2 \cdot \texttt{N})$ latest finishing times of jobs in JS as a time stamp in Line 15.

one processor which is neither reserved/executed by any $\tau_x \in \tau^{\texttt{IDLE}}$ nor executed by any $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$; let β denote the number of such processors. Then, by STEP 3 of Algorithm 1, the number of $\tau_x \in \tau^{\texttt{IDLE}}$ each of which has an invalid time stamp cannot be larger than β.

Therefore, if a job of $\tau_x \in \tau^{\texttt{IDLE}}$ has a valid time stamp, then the proof is the same as Case i. Otherwise, at least a processor does not service any job until a new job of any task is released. If the newly-released job is a job of τ_x itself, it immediately starts its execution; otherwise, τ_x will update its time stamp. In every case, there is no deadline miss for $\tau_x \in \tau^{\texttt{IDLE}}$. ■

From the idling mechanism of $\tau_x \in \tau^{\texttt{IDLE}}$, we can upper-bound the interval length between the current time instant t and $t_x^{\texttt{next}}$, if $m \geq 2 \cdot \texttt{N}$ holds. That is, the interval length between t and $t_x^{\texttt{next}}$ (if valid) is upper-bounded by the $(m-2\cdot\texttt{N}+1)^{th}$ largest C_y among $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$. This comes from Lines 14–18 of Algorithm 1; the $(m-2\cdot\texttt{N})$ latest finishing times of jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ cannot be selected as a time stamp due to the condition of Line 14. For example, if $m = 8$ and $\texttt{N} = 2$, the interval lengths between t and the two time stamps are upper-bounded by the $(8-2*2+1) = 5^{th}$ and 6^{th} largest C_i among $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$, respectively. Then, since $\tau_x \in \tau^{\texttt{IDLE}}$ idles a processor until $t_x^{\texttt{next}} - (D_x - C_x)$, the longest consecutive processor idling time is the upper-bound of the interval length minus $(D_x - C_x)$. Also, followed by the processor idling time for $\tau_x \in \tau^{\texttt{IDLE}}$, we always either execute a job of τ_x for C_x time units or wait for $(D_x - C_x)$ time units. By combining all the above properties, the following lemma provides an upper-bound on the amount of processor idling time.

Lemma 4: Under NWC(N)-NP-$*$ on a multiprocessor platform with $m \geq 2 \cdot \texttt{N}$, the amount of the processor idling time for $\tau_x \in \tau^{\texttt{IDLE}}$ in an interval of length L is upper-bounded by $E(T'_x, C'_x, L)$ where $C'_x = \big($the $(m-2\cdot\texttt{N}+1)^{th}$ largest C_y among $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}\big) - (D_x - C_x)$ and $T'_x = C'_x + \min(D_x - C_x, C_x)$, where

$$E(T,C,L) = \left\lfloor \frac{L}{T} \right\rfloor \cdot C + \min\left(C, L - \left\lfloor \frac{L}{T} \right\rfloor \cdot T\right). \quad (1)$$

Proof: We first prove that the time interval length between the current time t and $t_x^{\texttt{next}}$ is at most the $(m-2\cdot\texttt{N}+1)^{th}$ largest C_y among $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$. There are two cases at t: (Case i) there are at least $(m-\texttt{N})$ currently-executing jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$; and (Case ii) otherwise.

(Case i) Since we choose the earliest finishing time of jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$, $\tau_x \in \tau^{\texttt{IDLE}}$ chooses the 1^{st}, 2^{nd}, ..., $\texttt{N}^{th}$ earliest finishing time of jobs among at least $(m-\texttt{N})$ jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$. Therefore, the $(m-\texttt{N}-\texttt{N}) = (m-2\cdot\texttt{N})$ latest finishing time of jobs among at least $(m-\texttt{N})$ jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ cannot be selected as time stamps, which proves the supposition.

(Case ii) If there are $(m-\texttt{N}-a)$ currently-executing jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ (where $0 < a <= m-\texttt{N}$), $\tau_x \in \tau^{\texttt{IDLE}}$ choose the 1^{st}, 2^{nd}, ..., $(\texttt{N}-a)^{th}$ earliest finishing time of jobs among $(m-\texttt{N}-a)$ jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$. Therefore, the $(m-\texttt{N}-a-(\texttt{N}-a)) = (m-2\cdot\texttt{N})$ latest finishing time of jobs among $(m-\texttt{N}-a)$ jobs of $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}$ cannot be selected as a time stamp, which proves the supposition.

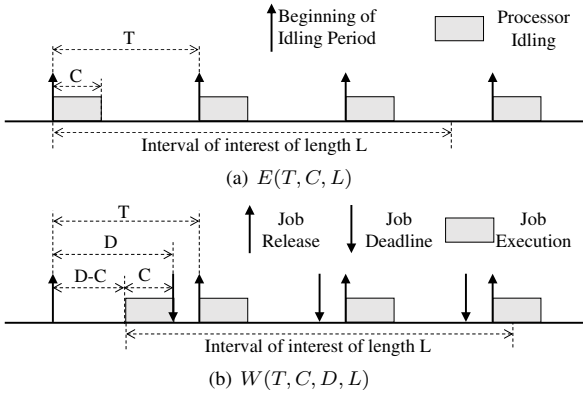


Fig. 3. Interference bound functions $E(T,C,L)$ and $W(T,C,D,L)$

Considering that $\tau_x \in \tau^{\texttt{IDLE}}$ idles a processor until $t_x^{\texttt{next}} - (D_x - C_x)$, the longest consecutive processor idling time for τ_x is $C'_x = \big($the $(m-2\cdot\texttt{N}+1)^{th}$ largest C_y among $\tau_y \in \tau \setminus \tau^{\texttt{IDLE}}\big) - (D_x - C_x)$. Also, followed by idling a processor by $\tau_x \in \tau^{\texttt{IDLE}}$, we always either execute a job of τ_x for C_x time units or waits for $(D_x - C_x)$ time units. Therefore, $\tau_x \in \tau^{\texttt{IDLE}}$ will start its next processor idling time at least $T'_x = C'_x + \min(D_x - C_x, C_x)$ time units after the beginning of the current processor idling time.

Then, if the processor idling duration is exactly C'_x, the amount of the processor idling time for $\tau_x \in \tau^{\texttt{IDLE}}$ in an interval of length L is upper-bounded by $E(T'_x, C'_x, L)$, as shown in Fig. 3(a). If the current processor idling duration is $C'_x - \epsilon$ ($\epsilon > 0$) (and thus the period is $T'_x - \epsilon$), this is equivalent to shifting all the following idling duration to the left ϵ in Fig. 3(a). If the duration of an idling instance within the interval of interest is reduced by ϵ, the contribution of the following idling instance newly included in the interval to the amount of processor idling time is at most ϵ. Therefore, $E(T'_x, C'_x, L)$ remains an upper-bound on the amount of processor idling time for $\tau_x \in \tau^{\texttt{IDLE}}$ in an interval of length L, in any case. ■

Finally, the following lemma proves the concurrency of τ_x's actual execution and its processor idling.

Lemma 5: Under NWC(N)-NP-$*$, the following statement holds for every $\tau_x \in \tau^{\texttt{IDLE}}$: any execution of jobs of τ_x and the processor idling time for τ_x cannot be overlapped.

Proof: As shown in Algorithm 1 and the state diagram in Fig. 2, $\tau_x \in \tau^{\texttt{IDLE}}$ does not reserve (idle) a processor when a job of τ_x is being executed. ■

Lemmas 3, 4, and 5 facilitate the adaption of existing schedulability tests for work-conserving non-preemptive scheduling algorithms, to the NWC(N)-$*$ framework as follows. First, we do not need to consider the schedulability of the N designated tasks. Second, the schedulability test can accommodate the effect of the processor idling time using $E(T'_x, C'_x, L)$. Finally, we can upper-bound the effect of the actual execution of a designated task and the processor idling time for the designated task, making the schedulability test tighter. Utilizing these three properties, the next section demonstrates how to apply an existing schedulability test for

a corresponding work-conserving scheduling, to the NWC(N)-$*$ framework associated with a target prioritization policy.

V. Schedulability Analysis

In this section, we consider NWC(N)-NP-FP, which is the NWC(N)-NP-$*$ framework employing FP as a prioritization policy. In FP, job priorities are determined by pre-defined task priorities. To derive a schedulability test for NWC(N)-NP-FP, we first recapitulate existing schedulability tests for the work-conserving non-preemptive FP scheduling algorithm (denoted by WC-NP-FP). By utilizing the properties of NWC(N)-NP-$*$ explained in Section IV-B and the existing schedulability tests for WC-NP-FP, we develop schedulability tests for NWC(N)-NP-FP.

Here, we define some notations for FP. Let $\tau^{\mathrm{HI}(\tau_k)}$ and $\tau^{\mathrm{LO}(\tau_k)}$ denote sets of tasks in τ, whose priorities are higher and lower than τ_k, respectively. Recall that τ^{IDLE} denotes a set of the N designated tasks, each of which incurs processor idling by the NWC(N)-NP-$*$ framework.

A. Existing schedulability tests for WC-NP-FP

Different from preemptive scheduling, a job under non-preemptive scheduling does not pause once it has started its execution. Therefore, we can judge whether a job of τ_k of interest is schedulable or not, by checking whether the job completes the first unit of its execution until $(D_k - C_k + 1)$ time units after its release time [6], [10]. Let I_k denote the cumulative length of intervals such that a job of τ_k of interest cannot execute in an interval between the release time of the job and $(D_k - C_k + 1)$ time units after the release time. Then, a job of τ_k of interest is schedulable if and only if the following inequality holds (shown in [6], [10] in a different form):

$$I_k < D_k - C_k + 1. \tag{2}$$

To upper-bound I_k under FP, we need to calculate how much execution can be performed in an interval of length $(D_k - C_k + 1)$. Let $W(T, C, D, L)$ denote the maximum amount of execution of jobs of a task with parameters (the minimum separation T, the worst-case execution time C, the relative deadline D) in an interval of length L, which is calculated as follows [25], [26]:

$$W(T,C,D,L) = \left\lfloor \frac{L+D-C}{T} \right\rfloor \cdot C + \min\left(C, L+D-C-\left\lfloor \frac{L+D-C}{T} \right\rfloor \cdot T\right). \tag{3}$$

As shown in Fig. 3(b), $W(T, C, D, L)$ describes the situation where the executions of the first and last jobs start as late and early as possible, respectively, and the interval of interest of length L starts at the beginning of the first job's execution [25], [26]. Then, $W(T_i, C_i, D_i, D_k - C_k + 1)$ is an upper bound of the amount of execution of jobs of τ_i in an interval of length $(D_k - C_k + 1)$.

Even though the priority of τ_k is higher than that of τ_i, a job of τ_i can block the execution of a job of τ_k of interest if the job of τ_i starts before the release of the job of τ_k. Therefore, the number of such blocking jobs is at most m (the number of processors), and the blocking time by each job of τ_i is up to $(C_i - 1)$ time units.

Considering that a job of τ_k cannot be executed in a time slot only if there are m other jobs to be executed in that slot, we can upper-bound I_k as follows [6], [10]:

$$I_k \le \frac{1}{m}\Big(\sum_{\tau_i \in \tau^{\mathrm{HI}(\tau_k)}} \min\big(W(T_i, C_i, D_i, D_k - C_k + 1), D_k - C_k + 1\big) + \sum_{\tau_i \in \tau^{\mathrm{LO}(\tau_k)} | m \text{ largest } C_i} \min\big(C_i - 1, D_k - C_k + 1\big)\Big). \tag{4}$$

Note that schedulability tests in [6], [10] exhibit a different form of the above formula. Also, the min operation holds because jobs of a given task cannot be executed for more than $(D_k - C_k + 1)$ time units within an interval of length $(D_k - C_k + 1)$.

Using Eq. (4), the lemma records a schedulability test for WC-NP-FP [6], [10].

Lemma 6 (Lemma 4 in [10] with $\ell = D_k - C_k + 1$): τ is schedulable by WC-NP-FP on an m-processor platform, if every task $\tau_k \in \tau$ satisfies the following inequality.[4]

$$\text{The RHS of (4)} < D_k - C_k + 1. \tag{5}$$

Proof: The lemma holds by Eqs. (2) and (4). ■

To improve the schedulability, a recent study [27] developed another upper-bound on I_k for the m highest-priority tasks.

Lemma 7 (Theorem 1 in [27]): Let n_k denote the number of tasks belonging to $\tau^{\mathrm{HI}(\tau_k)}$. Under WC-NP-FP, the following inequality holds for every τ_k with $n_k \le m - 1$:

$$I_k \le (m - n_k)^{th} \text{ longest } (C_i - 1) \text{ among } \tau_i \in \tau^{\mathrm{LO}(\tau_k)}. \tag{6}$$

Note that if there exists no such task described in the RHS of Eq. (6), the RHS is 0.

Proof: Because τ_k has n_k higher-priority tasks, its higher-priority tasks can prevent a job of τ_k from executing only on n_k processors. This means that, on the other $(m - n_k)$ processors, its lower-priority tasks are the only tasks which can prevent a job of τ_k from execution. Because every job of a lower-priority task τ_i can block the job of τ_k during at most $(C_i - 1)$ time units only if it starts its execution before the job of τ_k's release, the job of τ_k starts its execution no later than the $(m - n_k)^{th}$ largest $(C_i - 1)$ among the lower-priority tasks $\tau_i \in \tau^{\mathrm{LO}(\tau_k)}$ after its release time. ■

Incorporating Lemma 7 into Lemma 6, we can present an improved schedulability test for WC-NP-FP, which is similar to the existing study [27].

Lemma 8: Let n_k denote the number of tasks belonging to $\tau^{\mathrm{HI}(\tau_k)}$. Then, τ is schedulable by WC-NP-FP on an m-processor platform, if every τ_k with $n_k \le m - 1$ satisfies Eq. (7) and every τ_k with $n_k \ge m$ satisfies Eq. (5):

$$\min\big(\text{the RHS of Eq. (4)}, \text{the RHS of Eq. (6)}\big) < D_k - C_k + 1. \tag{7}$$

Proof: The theorem holds by Lemmas 6 and 7. ■

[4]The RHS and LHS stand for the right-hand side and the left-hand side, respectively.

B. Schedulability test for NWC(N)-NP-FP

Now, we develop a schedulability test for NWC(N)-NP-FP utilizing the three properties explained in Section IV-B. First, we do not need to consider the schedulability of every $\tau_x \in \tau^{\mathrm{IDLE}}$ by Lemma 3. Second, the amount of idling time for $\tau_x \in \tau^{\mathrm{IDLE}}$ is upper-bounded by $E(T'_x, C'_x, L)$ in an interval of length L. Third, the actual execution of and idling time for $\tau_x \in \tau^{\mathrm{IDLE}}$ cannot occur at the same time. Incorporating these three properties into Lemma 6, we develop a schedulability test for NWC(N)-NP-FP as follows.

Lemma 9: A task set τ is schedulable by NWC(N)-NP-FP on a multiprocessor platform with $m \geq 2 \cdot \mathtt{N}$, if every task $\tau_k \in \tau \setminus \tau^{\mathrm{IDLE}}$ satisfies the following inequality.

$$\frac{1}{m}\Bigg(\sum_{\tau_i \in \tau^{\mathrm{IDLE}}} \min\Big(D_k - C_k + 1, W(T_i, C_i, D_i, D_k - C_k + 1) + E(T'_i, C'_i, D_k - C_k + 1)\Big) + \sum_{\tau_i \in \tau^{\mathrm{HI}(\tau_k)} \setminus \tau^{\mathrm{IDLE}}} \min\Big(W(T_i, C_i, D_i, D_k - C_k + 1), D_k - C_k + 1\Big) + \sum_{\tau_i \in \tau^{\mathrm{LO}(\tau_k)} \setminus \tau^{\mathrm{IDLE}} | m \text{ largest } C_i} \min\Big(C_i - 1, D_k - C_k + 1\Big)\Bigg) < D_k - C_k + 1, \tag{8}$$

where $C'_x = \big(\text{the } (m - 2 \cdot \mathtt{N} + 1)^{th} \text{ largest } C_y \text{ among } \tau_y \in \tau \setminus \tau^{\mathrm{IDLE}}\big) - (D_x - C_x)$ and $T'_x = C'_x + \min(D_x - C_x, C_x)$.

Proof: By Lemma 3, every $\tau_k \in \tau^{\mathrm{IDLE}}$ is schedulable. To judge the schedulability of every $\tau_k \in \tau \setminus \tau^{\mathrm{IDLE}}$, we need to calculate the amount of other tasks' execution that prevents the job of τ_k of interest from executing in an interval of length $(D_k - C_k + 1)$ that starts at the release time of the job of τ_k of interest. We classify other tasks than τ_k into three categories: τ^{IDLE}, $\tau^{\mathrm{HI}(\tau_k)} \setminus \tau^{\mathrm{IDLE}}$ and $\tau^{\mathrm{LO}(\tau_k)} \setminus \tau^{\mathrm{IDLE}}$.

In an interval of length $(D_k - C_k + 1)$, each $\tau_i \in \tau^{\mathrm{IDLE}}$ performs its actual execution during at most $W(T_i, C_i, D_i, D_k - C_k + 1)$ time units, and idles a processor during at most $E(T'_i, C'_i, D_k - C_k + 1)$ time units calculated by Lemma 4. Because the actual execution and idling time cannot occur at the same time according to Lemma 5, we can upper-bound the sum of both by the interval length $(D_k - C_k + 1)$. The amount of execution of $\tau^{\mathrm{HI}(\tau_k)} \setminus \tau^{\mathrm{IDLE}}$ and $\tau^{\mathrm{LO}(\tau_k)} \setminus \tau^{\mathrm{IDLE}}$ that prevents the job of τ_k of interest from executing is the same as that of $\tau^{\mathrm{HI}(\tau_k)}$ and $\tau^{\mathrm{LO}(\tau_k)}$ in Eq. (4) under WC-NP-FP.

Considering that a job of τ_k cannot be executed in a time slot only if there are m other jobs executed in the slot, the lemma holds by Eq. (2). ■

Next, we prove that the improved schedulability test condition in Lemma 7 can be applied to NWC(N)-NP-FP.

Lemma 10: Let n_k denote the number of tasks belonging to $\tau^{\mathrm{IDLE}} \cup \tau^{\mathrm{HI}(\tau_k)}$. Under NWC(N)-NP-FP on a multiprocessor platform with $m \geq 2 \cdot \mathtt{N}$, the following inequality holds for every $\tau_k \in \tau \setminus \tau^{\mathrm{IDLE}}$ with $n_k \leq m - 1$:

$$I_k \leq (m - n_k)^{th} \text{ longest } (C_i - 1) \text{ among } \tau_i \in \tau^{\mathrm{LO}(\tau_k)} \setminus \tau^{\mathrm{IDLE}}. \tag{9}$$

Proof: Because τ_k has n_k tasks each of which either has a higher priority than τ_k or belongs to τ^{IDLE}, the n_k tasks can prevent a job of τ_k from executing only on n_k processors. This means that on the other $(m - n_k)$ processors, tasks belonging to $\tau^{\mathrm{LO}(\tau_k)} \setminus \tau^{\mathrm{IDLE}}$ are the only tasks which can prevent a job of τ_k from executing. Then, the remainder of the proof is the same as that of Lemma 7. ■

Using Lemmas 9 and 10, we develop an improved schedulability test for NWC(N)-NP-FP.

Theorem 1: Let n_k denote the number of tasks belonging to $\tau^{\mathrm{IDLE}} \cup \tau^{\mathrm{HI}(\tau_k)}$. Then, τ is schedulable by NWC(N)-NP-FP on a multiprocessor platform with $m \geq 2 \cdot \mathtt{N}$, if every $\tau_k \in \tau \setminus \tau^{\mathrm{IDLE}}$ with $n_k \leq m - 1$ satisfies Eq. (10) and every $\tau_k \in \tau \setminus \tau^{\mathrm{IDLE}}$ with $n_k \geq m$ satisfies Eq. (8).

$$\min\Big(\text{the LHS of Eq. (8)}, \text{the RHS of Eq. (9)}\Big) < D_k - C_k + 1. \tag{10}$$

Proof: First, by Lemma 3, every $\tau_k \in \tau^{\mathrm{IDLE}}$ is schedulable. Second, every $\tau_k \in \tau \setminus \tau^{\mathrm{IDLE}}$ is schedulable by Lemmas 9 and 10. ■

Now, we present an example of how to apply Theorem 1.

Example 5: Consider the task set and platform from Example 1. We apply NWC(1)-NP-FP with $\tau^{\mathrm{IDLE}} = \{\tau_1\}$, where τ_1 and τ_3 have the highest and lowest priorities, respectively. Then, $C'_1 = C_2$ (or C_3) $- (D_1 - C_1) = 2$ and $T'_1 = C'_1 + \min(D_1 - C_1, C_1) = 4$. For τ_2, $\min(W(T_1, C_1, D_1, D_2 - C_2 + 1) + E(T'_1, C'_1, D_2 - C_2 - 1), D_2 - C_2 + 1) = \min(4 + 6, 11) = 10$ and $\min(C_3 - 1, D_2 - C_2 + 1) = 11$; therefore $I_2 \leq 21/2 < 11$, meaning τ_2 is schedulable. For τ_3, $\min(W(T_1, C_1, D_1, D_3 - C_3 + 1) + E(T'_1, C'_1, D_3 - C_3 - 1), D_3 - C_3 + 1) = \min(4 + 6, 11) = 10$ and $\min(W(T_2, C_2, D_2, D_3 - C_3 + 1), D_3 - C_3 + 1) = 11$; therefore $I_3 \leq 21/2 < 11$, meaning that τ_3 is schedulable. Therefore, τ is schedulable by NWC(1)-NP-FP with $\tau^{\mathrm{IDLE}} = \{\tau_1\}$.

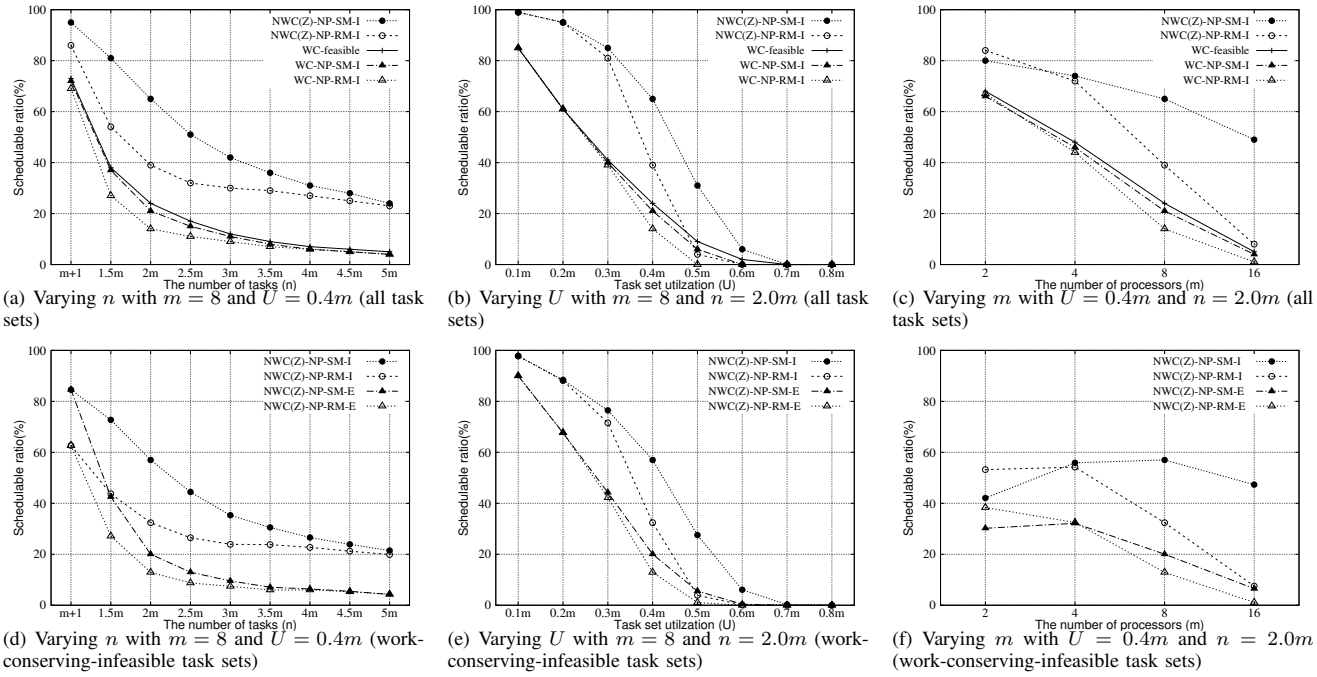
VI. EVALUATION

In this section, we evaluate the schedulability performance of NWC(Z)-NP-FP with its schedulability tests. After explaining the target schedulability tests to be compared and the task set generation process, we discuss the effectiveness of the NWC(Z)-NP-$*$ framework, in terms of empirical schedulability performance.

Target schedulability tests. We compare the following schedulability tests.

- WC-NP-$*$-E: the existing schedulability test for WC-NP-FP with the task priority assignment of $*$, i.e., Lemma 6,
- WC-NP-$*$-I: the improved schedulability test for WC-NP-FP with the task priority assignment of $*$, i.e., Lemma 8,
- NWC(Z)-NP-$*$-E: the plain schedulability test for NWC(Z)-NP-FP with the task priority assignment of $*$, i.e., Lemma 9, and
- NWC(Z)-NP-$*$-I: the improved schedulability test for NWC(Z)-NP-FP with the task priority assignment of $*$, i.e., Theorem 1,

where Z denotes the number of tasks that satisfy the work-conserving-infeasible condition shown in Lemma 1 in each



(a) Varying n with $m = 8$ and $U = 0.4m$ (all task sets)
(b) Varying U with $m = 8$ and $n = 2.0m$ (all task sets)
(c) Varying m with $U = 0.4m$ and $n = 2.0m$ (all task sets)
(d) Varying n with $m = 8$ and $U = 0.4m$ (work-conserving-infeasible task sets)
(e) Varying U with $m = 8$ and $n = 2.0m$ (work-conserving-infeasible task sets)
(f) Varying m with $U = 0.4m$ and $n = 2.0m$ (work-conserving-infeasible task sets)

Fig. 4. Schedulability performance varying n, U and m for all task sets and work-conserving-infeasible task sets

task set. Note that a task set with $Z > 0$ is unschedulable by every work-conserving non-preemptive scheduling including WC-NP-$*$-E and WC-NP-$*$-I, and a task set with $2 \cdot Z > m$ is unschedulable by all the target schedulability tests. For the task priority assignment policy for FP, we consider RM (rate monotonic; a task with a smaller T_i has a higher priority) and SM (slack monotonic; a task with a smaller $(T_i - C_i)$ has a higher priority).

Task set generation. We generate task sets using a popular task set generation method for a multiprocessor platform, called UUnifast-discard [28]. There are three input parameters: (i) the number of processors m (2, 4, 8, and 16), (ii) the number of tasks n ($m + 1$, $1.5m$, $2.0m$, $2.5m$, $3.0m$, $3.5m$, $4.0m$, $4.5m$, and $5.0m$), and (iii) the task set utilization $U = \sum_{\tau_i \in \tau} C_i/T_i$ ($0.1m$, $0.2m$, $0.3m$, $0.4m$, $0.5m$, $0.6m$, $0.7m$, and $0.8m$).[5] We generate 10,000 task sets for every three-tuple (m, n, U). For each tuple, UUnifast-discard [28] generates each task's utilization for each task set. For a given task utilization for τ_i (u_i), T_i is uniformly selected in $[1, 1000]$; C_i is computed based on the given utilization and T_i (i.e., $C_i = T_i \cdot u_i$); and D_i is set to T_i.

For all generated task sets with the three-tuple (i.e., $10,000 \cdot 4 \cdot 9 \cdot 8$ task sets), we observe that the schedulability performance of NWC(Z)-NP-$*$-I (*likewise* NWC(Z)-NP-$*$-E) is superior to that of its corresponding schedulability test WC-NP-$*$-I (*likewise* WC-NP-$*$-E) in terms of schedulability, which substantiates the effectiveness of the proposed NWC(Z)-NP-$*$ framework. To explain the detailed evaluation results, we choose the representative results shown in Fig. 4. That is, Figs. 4(a) and (d) fix $m = 8$ and $U = 0.4m$, and show the schedulable ratios for different n; Figs. 4(b) and (e) fix $m = 8$ and $n = 2.0m$, and show schedulable ratio for different U; and Figs. 4(c) and (f) fix $n = 2.0m$ and $U = 0.4m$, and show schedulable ratio for different m.

Schedulability performance of NWC(Z)-NP-$*$ compared to the corresponding work-conserving scheduling. Figs. 4(a), (b) and (c) plot the ratio schedulable by NWC(Z)-NP-SM-I, NWC(Z)-NP-RM-I, WC-NP-SM-I, WC-NP-RM-I, and WC-feasible, respectively, among all the 10,000 task sets per each point. Note that WC-feasible denotes a test that passes task sets which are not proven infeasible by Lemma 1, and therefore the schedulable ratio of WC-feasible is an upper bound on those of WC-NP-SM-I and WC-NP-RM-I.[6] On the other hand, Figs. 4(d), (e), and (f) plot the ratio schedulable by NWC(Z)-NP-SM-I, NWC(Z)-NP-RM-I, NWC(Z)-NP-SM-E, and NWC(Z)-NP-RM-E, among the task sets which are deemed unschedulable by Lemma 1 (i.e., work-conserving-infeasible task sets). Note that we do not plot WC-NP-$*$-I and WC-NP-$*$-E, because they cannot deem any single work-conserving-infeasible task set to be schedulable. From the figures, we make the following observations.

First, focusing on Figs. 4(a), (b) and (c), we observe that NWC(Z)-NP-$*$ significantly improves the schedulability of its corresponding work-conserving scheduling WC-NP-$*$ in every case. The result comes from the fact that the proposed framework makes a number of tasks that cannot be schedulable by every work-conserving non-preemptive scheduling schedulable.

[5] Since non-preemptive scheduling is difficult to meet timing requirements of a task set with high utilization, the utilization choices from $0.1m$ to $0.8m$ suffice to analyze the schedulability performance according to varying utilization.

[6] This does not mean that the schedulable ratio of WC-feasible is a lower bound of those of NWC(Z)-NP-SM-I and NWC(Z)-NP-RM-I; see U=0.5m in Fig. 4(b).

Second, focusing Figs. 4(d), (e), and (f), we conclude that NWC(Z)-NP-$*$ can cover a number of work-conserving-infeasible task sets, which successfully achieves the aim of this paper. Note again that all the task sets covered by the schedulability test of NWC(Z)-NP-$*$ in Figs. 4(d), (e) and (f) are unschedulable by every work-conserving scheduling.

Third, as m, n, or U becomes larger, the absolute value for the schedulability ratio of every schedulability test decreases while the rate between the schedulability ratio of NWC(Z)-NP-$*$-I and that of WC-NP-$*$-I increases, which is given in Figs. 4(a), (b) and (c). For example, in Fig. 4(a), the rate between the schedulability ratio of NWC(Z)-NP-SM-I and that of WC-NP-SM-I is $95.4/72.1 = 132.3\%$ with $n = m + 1$, $42.2/11.2 = 376.7\%$ with $n = 3.0m$, and $24.4/4.2 = 580.9\%$ with $n = 5.0m$. This is because the number of work-conserving-infeasible task sets increases as m, n or U increases (imagine a line of 100% minus the schedulable ratio of WC-feasible in Figs. 4(a), (b) and (c)). While every work-conserving scheduling cannot make any single work-conserving-infeasible task set schedulable (yielding the schedulable ratio of WC-NP-$*$-I no larger than that by WC-feasible), NWC(Z)-NP-$*$-I is able to cover a number of work-conserving-infeasible task sets. For example, in Fig. 4(d), the schedulability ratio of NWC(Z)-NP-SM-I is 84.6% with $n = m+1$, 35.3% with $n = 3.0m$, and 21.5% with $n = 5.0m$.

In addition, we can observe that the schedulable ratio of SM is mostly (but not always) higher than that of RM, which indicates the effectiveness of SM in satisfying the timing requirements of non-preemptive tasks. Also, if we compare the schedulable ratio of NWC(Z)-NP-$*$-I with that of NWC(Z)-NP-$*$-E in Figs. 4(d), (e) and (f), the former is observed to be significantly larger than the latter, which demonstrates the effectiveness of the improved schedulability test in providing timing guarantees.

VII. Conclusion

In this paper, we developed a non-work-conserving non-preemptive global scheduling framework, which is capable of yielding feasible schedules of task sets which are never schedulable by every work-conserving non-preemptive scheduling. By deriving useful properties of the framework, we developed a schedulability test for the framework associated with FP. Our evaluation results demonstrated that the proposed framework successfully covers a number of work-conserving-infeasible task sets, all of which cannot be schedulable by every existing study.

While this paper expanded the domain of task sets proven to be schedulable by any non-preemptive scheduling (regardless of being work-conserving), the exact domain of task sets that are schedulable by any non-preemptive scheduling remains unknown. In the future, we would like to develop a tight necessary feasibility condition for non-preemptive scheduling, motivated by our necessary feasibility condition for work-conserving non-preemptive global scheduling.

Acknowledgement

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2019R1A2B5B02001794, 2019R1F1A1059663, 2017H1D8A2031628). Jinkyu Lee is the corresponding author.

References

[1] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] S. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Systems*, vol. 32, no. 1-2, pp. 9–20, 2006.

[3] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu, "New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2008, pp. 137–146.

[4] H. Leontyev and J. H. Anderson, "A unified hard/soft real-time schedulability test for global edf multiprocessor scheduling," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2008, pp. 375–384.

[5] N. Guan, W. Yi, Q. Deng, Z. Gu, and G. Yu, "Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling," *Journal of Systems Architecture*, vol. 57, no. 5, pp. 536–546, 2011.

[6] J. Lee and K. G. Shin, "Controlling preemption for better schedulability in multi-core systems," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2012, pp. 29–38.

[7] G. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems: a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[8] R. I. Davis, A. Burns, J. Marinho, V. Nelis, S. M. Petters, and M. Bertogna, "Global fixed priority scheduling with deferred preemption," in *Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2013, pp. 1–11.

[9] J. Marinho, V. Nelis, S. M. Petters, M. Bertogna, and R. I. Davis, "Limited pre-emptive global fixed task priority," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2013, pp. 182–191.

[10] J. Lee and K. G. Shin, "Improvement of real-time multi-core schedulability with forced non-preemption," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1233–1243, 2014.

[11] R. I. Davis, A. Burns, J. Marinho, V. Nelis, S. M. Petters, and M. Bertogna, "Global and partitioned multiprocessor fixed priority scheduling with deferred preemption," *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 3, pp. 47:1–47:28, 2015.

[12] J. Lee, "Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1816–1823, 2017.

[13] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Partitioned fixed-priroity scheduling of parallel tasks without preemptions," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 421–433.

[14] B. Yalcinkaya, M. Nasri, and B. B. Brandenburg1, "An exact schedulability test for non-preemptive self-suspending real-time tasks," in *Proceedings of Design Automation and Test in Europe Conference (DATE)*, 2019, pp. 1228–1233.

[15] C. Ekelin, "Clairvoyant non-preemptive EDF scheduling," in *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)*, 2006, pp. 23–32.

[16] M. Nasri and M. Kargahi, "Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks," *Real-Time Systems*, vol. 50, no. 4, pp. 548–584, 2014.

[17] M. Nasri and G. Fohler, "Non-work-conserving scheduling of non-preemptive hard real-time tasks based on fixed priorities," in *International Conference on Real-Time Networks and Systems*, 2015, pp. 309–318.

[18] M. Nasri and B. Brandenburg, "Offline equivalence: A non-preemptive scheduling technique for resource-constrained embedded real-time systems," in *Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2017, pp. 75–86.

[19] M. Nasri and G. Fohler, "Non-work-conserving non-preemptive scheduling: motivations, challenges, and potential solutions," in *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 165–175.

[20] M. Nasri and B. Brandenburg, "An exact and sustainable analysis of non-preemptive scheduling," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 12–23.

[21] H. Lee and J. Lee, "Limited non-preemptive EDF scheduling for a real-time system with symmetry multiprocessors," *Symmetry*, vol. 12, no. 1, pp. 172:1–172:19, 2020.

[22] A. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Ph.D. dissertation, Massachusetts Institute of Technology, 1983.

[23] M. Nasri and F. Gerhard, "Open problems on non-preemptive scheduling of mixed-criticality real-time systems," in *Real-Time Scheduling Open Problems Seminar*, 2015, pp. 17–18.

[24] M. Nasri, G. Nelissen, and B. Brandenburg, "A response-time analysis for non-preemptive job sets under global scheduling," in *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)*, 2018, pp. 9:1–9:23.

[25] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 149–158.

[26] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 553–566, 2009.

[27] H. Baek and J. Lee, "Improved schedulability test for non-preemptive fixed-priority scheduling on multiprocessors," *IEEE Embedded Systems Letters*, vol. 12, no. 4, 2020.

[28] R. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 2009, pp. 398–409.