Global EDF-based scheduling with laxity-driven priority promotion

Shinpei Kato\textsuperscript{a,b,}*, Nobuyuki Yamasaki\textsuperscript{c}

\textsuperscript{a}Department of Computer Science, The University of Tokyo, Tokyo, Japan
\textsuperscript{b}Department of Electrical and Computer Engineering, Carnegie Mellon University, United States
\textsuperscript{c}Department of Information and Computer Science, Keio University, Japan

\textbf{A R T I C L E  I N F O}

Article history:
Received 15 January 2010
Received in revised form 18 August 2010
Accepted 10 January 2011
Available online 22 January 2011

Keywords:
Real-time systems
Multiprocessor scheduling
Earliest Deadline First
Schedulability analysis
Sustainability analysis

\textbf{A B S T R A C T}

This paper presents an algorithm, called Earliest Deadline Critical Laxity (EDCL), for scheduling sporadic task systems on multiprocessors. EDCL is a derivative of the Earliest Deadline Zero Laxity (EDZL) algorithm. Each job is assigned a priority based on the well-known Global Earliest Deadline First (G-EDF) algorithm, as long as its laxity – the amount of time from the earliest possible time of job completion to the deadline of job – is above a certain value. The priority is however promoted to the highest level once the laxity falls below this certain value in order to meet the deadline. Priority promotions are aligned with arrivals and completions of jobs under EDCL to avoid additional scheduler invocations, while EDZL can promote priorities arbitrarily. As compared with EDZL, EDCL reduces runtime overhead and implementation cost, but still strictly dominates G-EDF in schedulability. Schedulability tests for EDCL are derived through theoretical analysis, and sustainability properties are also considered. Our simulation results demonstrate that EDCL is competitive to EDZL in schedulability with a smaller number of scheduler invocations, and it also outperforms traditional EDF-based algorithms.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Multiprocessor scheduling has been a primary concern for real-time systems, ever since the Earliest Deadline First (EDF) algorithm [1] turned out to be no longer optimal on multiprocessors [2]. On single processors, EDF is optimal for scheduling a set of independent periodic tasks, where each task has a relative deadline equal to its period, and its utilization – the ratio of its execution time and its period – is no greater than 1. Specifically, EDF can schedule any set of such tasks without missing deadlines, if the total utilization of tasks is no greater than 1. Unfortunately, this EDF optimality breaks down on multiprocessors. In particular, the Global EDF (G-EDF) algorithm that applies the EDF algorithm under the global scheduling discipline, where tasks are assigned uniquely global priorities across processors, was revealed to fail scheduling a set of periodic tasks on m identical processors, if the total utilization of tasks exceeds \( m \cdot (1 - u_{\text{max}}) + u_{\text{max}} \), where \( u_{\text{max}} \) is the maximum utilization of every individual task [3]. The worst case derived when \( u_{\text{max}} = 1 \) leads to that EDF causes periodic tasks to miss deadlines, if the total utilization of tasks is slightly greater than 1.

Traditionally, there are two design goals for multiprocessor real-time scheduling. One aims for optimal scheduling, where any set of periodic tasks is schedulable unless the total utilization of tasks exceeds \( m \cdot m \) processors. Pfair [4–6], LLREF [7], and EKG [8] are well-known optimal algorithms. The other type of algorithms seeks for a simple concept that allows practical schedulers to use algorithms readily. EDF-US[\( x \)] [9,10] and EDZL [11–13] are simple extensions to G-EDF, but improve schedulability. There is a trade-off between these two concepts: schedulability is improved at the expense of additional complexity. For practical use, we explore simple algorithms but improve schedulability close to optimal algorithms.

EDF-US[\( x \)] overcomes the weakness of G-EDF by assigning the static highest priority to the tasks with utilization greater than \( x \). In terms of the upper bound on the total utilization of tasks, the known best configuration is \( x = 1/2 \), which guarantees any set of periodic tasks to be schedulable on \( m \) processors, if the total utilization does not exceed \( (m + 1)/2 \) [10]. This schedulability improvement is commendable, but EDF-US[\( x \)] can cause deadline misses for the set of periodic tasks that is schedulable under G-EDF.

A scheduling algorithm \( A_1 \) is said to dominate another scheduling algorithm \( A_2 \), if any set of tasks that is schedulable under \( A_2 \) is also guaranteed to be schedulable under \( A_1 \). \( A_1 \) is also said to strictly dominate \( A_2 \), if (i) \( A_1 \) dominates \( A_2 \), and (ii) there exists a set of tasks schedulable under \( A_1 \) but is not schedulable under \( A_2 \). With the definitions above, EDF-US[\( x \)] with a static configuration of \( x \) does not dominate G-EDF.

An easy example is given below to show that EDF-US[\( x \)] does not dominate G-EDF for \( x = 1/2 \). Suppose that the an \( m \)-processor system executes one heavy and \( m \) light tasks. The heavy task has...
execution time $k$ and relative deadline $2k$, while each of the $m$ light tasks has execution time $k/2 + \epsilon$ and relative deadline $k + \epsilon$, where $\epsilon$ is a very small number: $\epsilon \ll 0$. This task set is schedulable under G-EDF according to the schedulability test [14], but is not under EDF-US [1] as described below. Consider that all the tasks are released simultaneously at time $t$. Since EDF-US [1] assigns the highest priority to the heavy task, one of the $m$ light tasks must finish at time $t + k + 2\epsilon$. Given its relative deadline equal to $k + \epsilon$, this task misses a deadline. It should be noted that EDF-US[$x$] can still dominate G-EDF, if $x$ is dynamically chosen for different task sets. In fact, EDF-US[$x$] is apparently as effective as EDF, if $x = 1$. Therefore, EDF-US[$x$] can dominate G-EDF if it changes the value of $x$ only when the task set is not schedulable under G-EDF, as considered in [15]. However, EDF-US[$x$] itself with static configuration of $x$ does not dominate G-EDF.

EDZL is another alternative of G-EDF, which strictly dominates G-EDF, namely, any set of tasks schedulable under G-EDF is also schedulable under EDZL. EDZL assigns the priority of a job based on the same policy as G-EDF, if its laxity the time interval from the earliest possible time of job completion to the deadline of job, is above zero. If a job reaches zero laxity, its priority is imperiously promoted to the highest level to meet the deadline. Despite such a minor modification, EDZL significantly outperforms G-EDF [11]. The worst-case upper bound on the total utilization of tasks to be schedulable under EDZL is still no greater than $(m + 1)/2$ [16], but efficient schedulability tests have been developed for different task sets, showing that EDZL provides schedulability close to optimal algorithms in most cases [12,13].

There is a downside for EDZL, as compared with G-EDF and EDF-US[$x$]. EDZL requires the scheduler to invoke every time a job reaches zero laxity. Since the laxity of job can become zero while the job is running, EDZL can invoke the scheduler for each job at most three times: (i) when the job arrives; (ii) when the job reaches zero laxity, and (ii) when the job completes. However, if all relative deadlines are equal to periods, the number of scheduler invocations per job is at most 2. Specifically, one is upon the arrival of job, and the other is either when the job reaches zero laxity or when the job completes, since if the laxity of job is zero, the completion of job will coincide with the arrival of the next job, and the scheduler invocation can be ascribed to the arrival of the next job. Meanwhile, the number of scheduler invocation is strictly bounded by 2 for G-EDF and EDF-US[$x$]: one is upon the arrival of job and the other is upon the completion of job.

System implementation is another concern for EDZL. Since EDZL needs to invoke the scheduler when the laxity of job becomes zero, high-resolution timers are required to implement EDZL, while G-EDF and EDF-US[$x$] do not need them, as long as the arrival of job is aligned with a scheduler tick, and the completion of job implicitly invokes the scheduler to dispatch the next job. Many real-time operating systems [17–23] provide the interface for user-programs to trigger scheduler invocations when jobs complete, and the scheduler tick can be set to even less than 1 ms, most systems can implement G-EDF and EDF-US[$x$] without high-resolution timers. In general, high-resolution timers are expensive to use, causing runtime overhead that may not be acceptable in real-time systems.

This paper presents an EDF-based algorithm, called Earliest Deadline Critical Laxity (EDCL), for efficiently scheduling sporadic task systems on multiprocessors. The basic idea behind EDCL is to align the priority promotion with the arrival or completion of job. When the scheduler is invoked upon the arrival or completion of job, EDCL looks ahead the schedule to detect jobs that can miss deadlines, with the current priority levels, before the next scheduler invocation. The priorities of such tasks are promoted to the highest level. Hence, EDCL remains with the same number of scheduler invocations as G-EDF and EDF-US[$x$], while improving schedulability competitive to EDZL. Schedulability tests for EDCL are developed based on those for EDZL. Sustainability properties [24,25] for EDCL are also considered to examine if EDCL can make the system remain to be schedulable under “better” or “easier” scenarios.

The rest of this paper is organized as follows. Our system model is defined in Section 2. Section 3 presents the EDCL algorithm. Section 4 and Section 5 provide schedulability and sustainability analyses for EDCL, respectively. Section 6 evaluates the schedulability and the scheduler invocation cost of EDCL though simulations, as compared with traditional EDF-based algorithms. Our concluding remarks are provided in Section 7.

### 2. System model

The system contains $m$ identical processors. There is a set of $n$ sporadic tasks, denoted by $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task $\tau_i$ is characterized by a $(c_i, d_i, p_i)$ tuple, where $c_i$ is its worst-case execution time (WCET), $d_i$ is its relative deadline, and $p_i$ is its minimum inter-arrival time (also called period). The execution time of $\tau_i$ must be no greater than the relative deadline and the period, i.e., $c_i \leq \min(d_i, p_i)$. The utilization of $\tau_i$ is denoted by $u_i = c_i/p_i$, and its density is denoted by $d_i = c_i/d_i$. We particularly restrict our attention to constrained-deadline task systems, where relative deadlines are no greater than periods, i.e., $d_i < p_i$. The contribution of this paper is, however, still applicable for arbitrary-deadline task systems, where relative deadlines can be greater than periods.

Each task sporadically produces an infinite sequence of jobs. The execution time of job produced by $\tau_i$ is no greater than $c_i$. The inter-arrival interval of two successive jobs produced by $\tau_i$ is separated by at least $p_i$. The job of $\tau_i$ that arrives at time $t$ has a deadline at time $t + d_i$. The remaining execution time of job of $\tau_i$ at time $t$ is denoted by $e_i(t)$, and its laxity is denoted by $x_i(t)$. For any time $t$, $t + e_i(t) + x_i(t) = d_i$ is satisfied. For the sake of generalization, the minimum of relative deadline and period of $\tau_i$ is defined as window $\Delta_i$, i.e., $\Delta_i = \min(d_i, p_i)$, and the minimum of utilization and density is defined as $\lambda_i = c_i/\Delta_i$. A set of ready tasks is denoted by $F$ at any time instant, and a set of such $m$ ready tasks in $F$ that have the $m$ earliest deadlines is denoted by $F(m)$, if any.

We assume the following. Tasks are independent and preemptive. An individual processor executes at most one task at the same time. Each job is never executed in parallel. Any job of a task is now allowed to start execution before the preceding job of the same task completes.

### 3. Algorithm description

This section describes the EDCL algorithm. EDCL is a derivative of EDZL. The priority of job is assigned based on the EDF policy, but can be exceptionally promoted to the highest level to improve schedulability, when the laxity of job reaches a certain value. This priority promotion is permitted only upon arrivals and completions of jobs for EDCL, while it is triggered when the laxity becomes zero for EDZL, which can occur at arbitrary time. Thus, the algorithm implementation for EDCL is simplified over EDZL, since there is no need to use high-resolution timers, while the scheduler can invoke only when jobs arrive and complete.

In order to reason about this design choice for EDCL, we first consider any scheduling point $t_s$, at which a job would miss a deadline if it is not dispatched for execution immediately. For simplicity of description, $t_s$ also represents the arrival time or completion time of any job.

#### Definition 1

A job of task $\tau_i$ is said to be critical, if its laxity satisfies inequality 1 at time $t_s$, where $\epsilon_{\min}$ denotes the minimum
remaining execution time of such \( m \) jobs that have the earliest deadlines.

\[
x_i(t_i) < e_{\text{min}}
\]  

(1)

**Definition 2.** The laxity \( x_i(t_i) \) of job of task \( t \), at time \( t_i \), is said to be **critical laxity**, if it satisfies Inequality 1.

It is clear that a critical job will miss a deadline unless it starts execution at the current scheduling point, since it will not be dispatched under the EDF policy until at least one of \( m \) earliest-deadline jobs completes. Fig. 1 shows an example where a deadline miss occurs on three processors. At time \( t_1 \), a gray-color job with deadline \( t_d \) is critical due to its laxity less than \( e_{\text{min}} \). However, this job is not dispatched for execution under the EDF policy at this scheduling, since there are other \( m \) jobs with earlier deadlines. As a result, a deadline miss is inevitably caused for this job. It should be noted that jobs whose laxity is equal to \( e_{\text{min}} \) are not critical according to our definition, because such tasks can complete at the very deadline, if they start execution immediately after the job of the minimum remaining execution time \( e_{\text{min}} \).

In order to avoid such a deadline miss, EDCL exceptionally assigns the highest priority to a critical job at time \( t_i \), a scheduling point upon the arrival or completion of job. If there are no critical jobs, the schedule produced by EDCL is exactly the same as that produced by G-EDF. Even though there are critical jobs, the scheduling points remain to be the same as G-EDF, while the resulting schedule is different. Hence, the number of scheduler invocations is also bounded to be the same as G-EDF.

**Theorem 1.** **EDCL strictly dominates G-EDF.**

**Proof 1.** The schedule produced by EDCL is exactly the same as that produced by G-EDF, if there are no critical jobs. As stated above, G-EDF cannot make critical jobs meet deadlines, while EDCL may be able to make them schedulable due to the priority promotion. In other words, any set of tasks schedulable under G-EDF is also schedulable under EDCL, and there could exist a set of tasks schedulable under EDCL, but is not under G-EDF. Hence, the theorem is true. \(\square\)

**Theorem 2.** **The number of scheduler invocations per job for EDCL is at most 2.**

**Proof 2.** Let \( j(A) \) be the number of jobs that arrive in any interval \( A \). Since EDCL invokes the scheduler only when jobs arrive and complete, the number of scheduler invocations in interval \( A \) is apparently bounded by \( 2j(A) \). Hence, the theorem is true. \(\square\)

Fig. 2 illustrates the pseudo-code of the EDCL algorithm. The scheduler is invoked only when jobs arrive or complete. If there are less than \( m \) ready tasks, all those ready tasks are dispatched. Else, the algorithm first selects such \( m \) ready tasks that have the

\[
m \text{ earliest deadlines. It next computes the minimum remaining execution time } e_{\text{min}} \text{ of those } m \text{ tasks. If there are ready jobs whose laxity is less than } e_{\text{min}}, \text{ they are critical jobs, and their priorities are promoted to the highest level to execute immediately. Non-critical earliest-deadline jobs are secondary choices in this case. Given that multiple jobs can become critical at the same time, we need to prioritize even critical jobs if there are more than } m \text{ critical jobs. Four prioritization policies are considered for those critical jobs:}
\]

1. **AR:** Priorities are given arbitrarily.
2. **RM:** Higher priorities are given to jobs with shorter remaining execution times.
3. **LX:** Higher priorities are given to jobs with less laxity.
4. **DL:** Higher priorities are given to jobs with earlier deadlines.

The AR policy provides a benefit that the scheduler only needs to dispatch the first \( m \) critical jobs examined by a linear search, meaning that there is no need to compare all critical jobs. The other three policies, on the other hand, are more time-consuming, but should perform better than the AR policy. Consider that there are more than \( m \) critical jobs. If the minimum remaining execution time of the dispatched \( m \) jobs is greater than \( e_{\text{min}} \), the remaining critical jobs will inevitably miss deadlines. The RM policy therefore prioritizes them in favor of shorter remaining execution times so that the minimum remaining execution time is likely less than \( e_{\text{min}} \), and the remaining critical jobs may still meet deadlines. The LX policy is encouraged by a similar idea to the RM policy. If higher priorities are given to those with less laxities, the remaining critical jobs have more laxities than the dispatched \( m \) jobs. Hence, there should be more chance for the remaining critical jobs to meet deadlines, if the minimum remaining execution time of the dispatched \( m \) jobs is less than \( e_{\text{min}} \). The DL policy is, meanwhile, encouraged by the known effectiveness of the G-EDF policy. These four prioritization policies for critical jobs are evaluated through our simulation study in Section 6.

EDZL is more expensive, since it needs to verify whether the laxities of some jobs will be zero before the next expected scheduler invocation. It hence needs to compare the laxity of each task with \( e_{\text{min}} \) to predict when exactly those laxities will be zero. If some jobs turn out to reach zero-laxity, EDZL next needs to compute the earliest time of zero-laxity occurrences, and configures a high-resolution timer to interrupt the current job and switch to the zero-laxity job. EDCL does not need this additional procedure, since priority promotions are aligned with arrivals and completions of jobs.

Figs. 3–5 illustrate how a set of five tasks, \( t_1 = t_2 = t_3 = t_4 = (3, 10, 10) \) and \( t_5 = (10, 15, 15) \), are scheduled on two processors under the G-EDF, EDF-US[1/2], and EDCL policies, respectively. In G-EDF scheduling, \( t_5 \) misses a deadline at time 15. In EDF-US[1/2]
scheduling, all five tasks meet deadlines, since the priority of \( t_5 \), which would miss a deadline at the current priority level, is promoted to the highest level at time 3. This task set is also schedulable under EDZL, but more scheduler invocations are required.

4. Schedulability analysis

This section provides a schedulability analysis for EDCL. It builds upon the previous approach presented for EDZL [12,13]. We focus on the first job that misses a deadline, and consider a necessary condition for this job to miss the deadline. Henceforth, this job is defined as a problem job. The time interval between the arrival time of the problem job and the missed deadline is also defined as a problem window. For simplicity of description, \( t_k \) denotes a task that contains the problem job, and \( t_d \) denotes its deadline.

Given the algorithm procedure described in Section 3, EDCL is a work-conserving algorithm. Hence, the problem job can miss a deadline only if it is blocked by competing jobs for a sufficient interval. Our schedulability analysis first explores a lower bound on the total amount of time that must be consumed by those competing jobs in the problem window to cause the problem job to miss a deadline. We next obtain an upper bound on the maximum amount of time for which the critical job can be blocked by other critical or strictly-critical jobs. Since EDCL is work-conserving, the problem job is only blocked by other problem jobs with higher or equal priorities, except for its preceding job. The gray-colored area indicates executions of such blocking jobs, and the area enclosed by dots indicates the time interval for which they effectively block the problem job.

Definition 3. A job of task \( t_k \) is said to be strictly-critical, if its laxity satisfies Inequality 2 at time \( t_s \), where \( \epsilon'_{\text{min}} \) denotes the minimum remaining execution time of jobs whose priority is less than that of the job of \( t_k \).

\[
\text{x}_k(t_s) < \epsilon'_{\text{min}}
\]  

Definition 4. The laxity \( \text{x}_k(t_s) \) of job of task \( t_k \) at time \( t_s \) is said to be strictly-critical laxity, if it satisfies Inequality 2.

It should be noted that jobs can be strictly critical without being critical. Therefore, the necessary condition for EDCL to cause a deadline miss is that there are \( m \) critical jobs and at least one strictly-critical job at the same time. Hereinafter, all such \( m + 1 \) jobs are considered to be problem jobs.

We next consider a condition for the problem jobs to be critical or strictly-critical, which is defined to be a critical condition. As stated in Section 3, jobs become critical only when they would miss deadlines under the EDF policy. Jobs also become strictly-critical in the same situation, but all other competing jobs must be critical.

Fig. 6 shows a critical condition for the problem job of task \( t_k \), where \( t_d \) denotes the deadline of the problem job, and \( t_t \) denotes the scheduling point at which the problem job becomes critical or strictly-critical. Since EDCL is work-conserving, the problem job is only blocked by other problem jobs with higher or equal priorities, except for its preceding job. The gray-colored area indicates executions of such blocking jobs, and the area enclosed by dots indicates the time interval for which they effectively block the problem job.

Definition 5. The total amount of time that can be contributed by task \( t_i \) in time interval \( [a, b] \) is defined as competing work \( W_i(a, b) \).

If \( d_k < p_i \) is satisfied, Fig. 6 implies that the problem job of \( t_k \) can become critical or strictly-critical, only when it is blocked for longer than a time interval of \( d_k - c_k \) in the problem window, and it is not dispatched at time \( t_k \). Even if \( t_k \) becomes critical or strictly-critical earlier, the maximum amount of time for which the critical job can execute in \( [t_d - p_i, t_t] \) is at most \( c_k \). The lower bound on the competing work in this interval is hence \( m(d_k - c_k) \). As a consequence, the following condition must be satisfied for the problem job of \( t_k \) to be critical or strictly-critical

\[
\sum_{i \neq k} W_i(t_d - d_k, t_d) > m(d_k - c_k)
\]

If \( d_k > p_i \) is satisfied, on the other hand, the problem job of \( t_k \) can become critical or strictly-critical, when it is blocked for longer than a time interval of \( p_i - c_k \) in \( [t_d - p_i, t_t] \), and it is not dispatched at time \( t_i \). This case includes the constraint of its preceding job, though this expectation is pessimistic as mentioned in [12]. Hence, the problem job of \( t_k \) can be critical or strictly-critical in \( [t_d - p_i, t_t] \), if the following condition is satisfied

\[
\sum_{i \neq k} W_i(t_d - p_i, t_d) > m(p_i - c_k)
\]
Using the same terminologies in [12], time interval \((t_d - A_k, t_d)\) is called an overload window, where \(A_k = \min(d_k, p_k)\). We can now derive a critical condition for \(t_k\) in \((t_d - A_k, t_d)\) by Inequality 3.

\[
\sum_{i = k}^{n} W_i(t_d - A_k, t_d) > m(A_k - c_k)
\]  

(3)

If Inequality 3 satisfies \(\neg\), the problem job of \(t_k\) can never be critical or strictly-critical. This is a different property than EDZL given in [12,13].

Conditions for \(t_k\) to be critical and strictly-critical can now be unified by Inequality 3, while an upper bound on the left-hand side depends on the number of competing critical jobs. This upper bound will be presented in Section 4.2. Finally, we derive the lemma regarding a necessary condition for EDCL to cause a deadline miss as follows.

**Lemma 1.** If a set of sporadic tasks is scheduled on \(m\) processors, at least \(m + 1\) tasks must satisfy Inequality 3 to cause a deadline miss under EDCL.

**Proof 3.** It is trivial from the above discussion. □

4.2. Upper bound

We now obtain an upper bound on competing work \(W_i(t_d - A_k, t_d)\) of each task \(t_k\), which causes the problem job of \(t_k\) to be critical or strictly-critical in overload window \((t_d - A_k, t_d)\). If the necessary condition in Lemma 1 is not satisfied with this upper bound, no task can miss a deadline.

Since the worst-case execution time \(c_i\) of each task \(t_i\) is static in this paper, the competing work cannot be increased by increasing the inter-arrival times of sporadic tasks, meaning that the competing work of each task in any overload window is never larger than the case in which each task periodically arrives at the minimum interval. Hence, there is no need to consider sporadic arrivals to derive an upper bound on the competing work of each task.

As for the contribution of jobs to the competing work, we only need to take into account such jobs that have deadlines after \(t_d - A_k\), since we assume that the first deadline miss is caused by the problem job, and no deadline is missed before the overload window. On the other hand, such jobs that have deadlines in the overload window or at time \(t_d\) contribute to the competing work.

However, such jobs that have deadlines after the overload window may or may not contribute to the competing work, depending on laxities. If jobs are critical, they are contributors. Otherwise, they are not. Therefore, we consider these two cases separately.

**Case 1**

We first consider the tasks that contain no critical jobs when the problem job is about to be critical. In this case, such jobs that have deadlines after the overload window do not interfere with the problem job. Hence, we only need to consider such jobs that have deadlines in the overload window, as studied in the EDZL analysis [12,13]. According to this analysis, the maximum competing work of each task \(t_i\) is never greater than the case in which all jobs of \(t_i\) execute as late as possible, and the deadlines of jobs of \(t_i\) are aligned with the deadline of the problem job.

![Fig. 7. Case 1 for the worst-case phase of task \(t_i\).](image)

**Fig. 7.** Case 1 for the worst-case phase of task \(t_i\).

**Fig. 8.** Case 2 for the worst-case phase of task \(t_k\).

**Lemma 2.** If a set of sporadic tasks is scheduled on \(m\) processors under EDCL, an upper bound on the competing work of task \(t_i\) in any overload window \(W_i(t_k)\) of the problem job of \(t_k\) is provided by Eq. 4, if a job of \(t_i\) is not critical when the problem job is about to be critical.

\[
W_i(t_k) = n_i c_i + \max\{c_i, \min\{0, A_k - n_i p_i\}\}
\]  

(4)

**Proof 4.** It is trivial from the preceding discussion. □

**Case 2**

We next consider the tasks that contain critical jobs when the problem job is about to be critical. In this case, such jobs that have deadlines after the overload window can affect the problem job, since these jobs may be assigned the highest priority due to the critical laxity. Therefore, the deadlines of \(t_i\) are not necessarily aligned with the deadline of the problem job to produce the worst-case phase of \(t_i\). In fact, if tasks are scheduled under EDZL, the worst-case phase of \(t_i\) matches Fig. 7, because such jobs that have deadlines after the overload window can interfere with the problem job, only when their laxities are zero, and they must complete at the very deadline. However, if tasks are scheduled under EDCL, such critical jobs that have deadlines after the overload window can contend with the problem job even before their deadlines. This is a main difference between EDCL and EDZL.

In order to maximize the competing work, the arrival time of \(t_i\) should still be periodic. We only need to consider moving the phase of arrivals from the case in Fig. 7. Since the job of \(t_i\) must be critical to affect the problem job of \(t_k\), the phase of \(t_i\) can be shifted early in time from the deadline by at most \(x_i(t_i)\). Hence, the worst-case phase is obtained when the deadline of the problem job is aligned with the completion of the last job of \(t_i\) that arrives before or in the overload window, as shown in Fig. 8. The following discussion reasons about the correctness of the above argument.

- If we shift later in time the phase of \(t_i\) by amount \(K < p_i - x_i(t_i)\) in Fig. 8, the competing work of \(t_i\) at the end of overload window is decreased by \(\min\{K, c_i\}\). This shifting can increase the competing work at the beginning of overload window by at most \(\min\{K, c_i\}\). Hence, the total competing work of \(t_i\) is not increased.

- If we shift earlier in time the phase of \(t_i\) by amount \(K < x_i(t_i)\) in Fig. 8, the competing work of \(t_i\) at the beginning of overload window is decreased, while no contribution is provided at the end of overload window. Hence, the competing work of \(t_i\) is not increased.
We now compute the competing work of $\tau_i$. Since the laxity is never decreased, the last job of $\tau_i$ that arrives before or in the overload window can be critical when its laxity is $x_i(t_d)$. Fig. 8 implies that the competing work of $\tau_i$ increases as $x_i(t_d)$ increases. Hence, the value of $x_i(t_d)$ is at most

$$x_i(t_d) \leq d_i - c_i.$$ 

In this situation, the problem job is not yet critical but is about to be critical. Hence, the laxity of the problem job must be no less than that of the last job of $\tau_i$. Otherwise, the problem job must be critical, which is contradict to the situation. Therefore, $x_i(t_d)$ must also satisfy

$$x_i(t_d) \leq d_i - c_i.$$ 

In addition, the laxity of job must be less than the minimum remaining execution time of the jobs with earlier deadlines to be critical. In the worst-case scenario, it is equal to the $m$th largest WCET, denoted as $C_{\text{max}}$, of all tasks other than $\tau_i$. Therefore, $x_i(t_d)$ must be in the range of

$$x_i(t_d) \leq C_{\text{min}}.$$ 

Finally, the upper bound of $x_i(t_d)$ for any $t_d$ of $\tau_i$, denoted by $X_i(t_d)$, is obtained by Eq. 5

$$x_i(t_d) = \min\{d_i - c_i, d_i - c_i, \min\{c_i | \tau_j \in \tau, j \neq i\}\}$$ (5)

Since the phase of $\tau_i$ is different from Case 1, we need to compute again the value of $n_i$ and the length of carry-in execution. Using $x_i(t_d)$ obtained above, the maximum number of jobs of $\tau_i$ that have both arrival times and deadlines in the overload window is given by

$$n_i = \frac{d_i - (p_i - X_i(t_d))}{p_i}.$$ 

The length of carry-in execution is thereby bounded by

$$\min\{c_i, \max\{d_i - n_i p_i - (p_i - X_i(t_d))\}\}.$$ 

**Lemma 3.** If a set of sporadic tasks is scheduled on $m$ processors under EDCL, an upper bound on the competing work of task $\tau_i$ in any overload window $W_B^i(t_d)$ of the problem job of $\tau_i$ is provided by Eq. 6, if a job of $\tau_i$ is critical when the problem job is about to be critical

$$W_B^i(t_d) = \min\{d_i, (n_i + 1)c_i + \min\{c_i, \max\{0, d_i - (n_i + 1)p_i + X_i(t_d)\}\}\}$$ (6)

**Proof 5.** It is trivial from the preceding discussion. □

### 4.3. Schedulability test

We have obtained an upper bound on the contribution of each task $\tau_i$ to the competing work in any overload window of the problem job. However, we still need to consider interference [14] for more precise analysis. The interference of $\tau_i$ on the problem job is the total interval in which the problem job is blocked by $m$ jobs, and a job of $\tau_i$ is one of these $m$ jobs that block the problem job.

According to Bertogna et al. [14], a sufficient condition to cause task $\tau_i$ to miss a deadline is that the interference of each $\tau_i$ on the problem job of $\tau_i$ in interval $(t_d - d_i, t_d)$, the problem window, is at least greater than $d_i - c_i$. This upper bound is intuitive from Fig. 6. Suppose that $\tau_i$ is not dispatched at time $t_i$. Then, $\tau_i$ will miss a deadline if the plotted area of Fig. 6 is greater than $m(d_i - c_i)$. Hence, it is sufficient for $\tau_i$ to consider only a duration of $d_i - c_i$, even though it can consume more processor time.

We now consider interference under EDCL. Let $W_B^i(t_d)$ unify $W_B^i(t_d)$ and $W_B^i(t_d)$ regardless of whether $\tau_i$ contains a critical job or not when $t_d$ is about to be critical. The lemma regarding a condition for $\tau_i$ to be critical or strictly-critical is provided as follows.

**Lemma 4.** In order for task $\tau_i$ to be critical or strictly-critical, $\tau_i$ must satisfy Condition 7, or it must satisfy Condition 8 when at least $m$ other tasks satisfy Condition 9

$$\sum_{i \neq k} \min\{W_B^i(t_d), d_k - c_k\} > m(A_k - c_k)$$ (7)

$$\sum_{i \neq k} \min\{W_B^i(t_d), d_k - c_k\} = m(A_k - c_k)$$ (8)

$$W_B^i(t_d) > d_k - c_k$$ (9)

**Proof 6.** We first take the minimum of $W_B^i(t_d)$ and $d_k - c_k$, because the interference for $\tau_k$ should not include the time that competing task $\tau_k$ executes in parallel with $\tau_k$. Recall that Inequality 3 is a necessary condition for $\tau_k$ to be critical or strictly-critical. Hence, $\tau_k$ needs to satisfy Inequality 7 to contain critical or strictly-critical jobs.

We next focus on the second condition with Eq. 8 and Inequality 9. If the total interference is no greater than $m(A_k - c_k)$, $\tau_k$ cannot miss a deadline. However, if more than $m$ tasks have competing work greater than $A_k - c_k$, the actual total interference can be greater than $m(A_k - c_k)$, even though $\sum_{i \neq k} \min\{W_B^i(t_d), d_k - c_k\} = m(A_k - c_k)$ is satisfied for those tasks. If one or more tasks has competing work less than or equal to $A_k - c_k$, the actual total interference should be equivalent to $m(A_k - c_k)$, and $\tau_k$ can never be critical or strictly-critical. □

Finally, Lemma 1 and Lemma 4 derive the following Theorem 3 with respect to a schedulability condition for a set of sporadic tasks under EDCL.

**Theorem 3.** A set of sporadic tasks is schedulable on $m$ processors under EDCL, if no more $m$ tasks satisfy the conditions derived in Lemma 4.

**Proof 7.** According to Lemma 1, there must exist at least $m + 1$ tasks that contain critical or strictly-critical jobs to cause a deadline miss under EDCL. Since conditions for task $\tau_i$ to be critical or strictly-critical are provided by Lemma 4, the theorem must be true. □

We now describe how to verify whether each task can contain critical or strictly-critical jobs. While we need the number of tasks that can contain critical or strictly-critical jobs to do so, obtaining this number itself also needs to verify whether each task can contain critical or strictly-critical jobs. Therefore, it is not easy to obtain a set of those tasks.

We first introduce a pessimistic but simple approach, which assumes all tasks to contain critical jobs. From the discussion in Section 4.2, the competing work of $\tau_i$ is maximized when the last job of $\tau_i$ that arrives before or in the overload window of the problem job is critical. Therefore, the total competing work cannot be greater than the one under this assumption.

**Theorem 4 (Pessimistic schedulability test).** A set of sporadic tasks is schedulable on $m$ processors under EDCL, if none of the following conditions is satisfied.

- At least $m + 1$ tasks satisfy $\sum_{i \neq k} \min\{W_B^i(t_d), d_k - c_k\} > m(A_k - c_k)$.
- At least $m + 1$ tasks satisfy $\sum_{i \neq k} \min\{W_B^i(t_d), d_k - c_k\} = m(A_k - c_k)$ when at least $m$ other tasks satisfy $W_B^i(t_d) > d_k - c_k$. 
We next derive a tight schedulability test that recursively searches for tasks that can contain critical or strictly-critical jobs. Let \( \alpha \) be a set of tasks that cannot contain critical or strictly-critical jobs, and \( \beta \) be a set of tasks that can contain those jobs. For simplicity of description, we define \( W^\alpha(\tau_k) \) and \( W^\beta(\tau_k) \) as follows, where \( W^s(\tau_k) \) is the total competing work of tasks that contain no critical jobs while \( W^i(\tau_k) \) is that of tasks that contain critical jobs, when \( \tau_k \) is about to be critical in any overload window.

Using those notations, the pseudo-code of our tight schedulability test repeats this procedure.

1. At least \( m + 1 \) tasks that can be critical were detected. This set of tasks is then considered not to be schedulable under EDCL.
2. No task was verified to be critical. This means that further recursion will never detect other tasks that can be critical. This set of tasks is then considered to be schedulable.

Since this procedure is repeated at most \( m + 1 \) times to verify the schedulability, the computation order of the tight schedulability test is \( O(n^2m) \), while that of the pessimistic test is \( O(n^3) \).

Theorem 5 (Tight schedulability test). A set of sporadic tasks is schedulable on \( m \) processors under EDCL, if it is accepted by the tight schedulability test illustrated in Fig. 9.

Proof 9. It is trivial from the preceding discussion.

Revision of EDZL schedulability test

We report that there is a pitfall in the schedulability analysis for EDZL presented in [12,13]. According to Refs. [12,13], a set of sporadic tasks is schedulable on \( m \) processors under EDZL, unless the following condition is satisfied for at least \( m + 1 \) tasks, and it holds strictly > for at least one of them, where \( \alpha(\tau_k) = W^i(\tau_k)/A_k \), where terminologies are slightly different from the original analysis but are consistent with this paper:

\[
\sum_{i=1}^m \min\{\alpha(\tau_i), 1 - \lambda_k\} \geq m(1 - \lambda_k) \tag{10}
\]

The above schedulability condition has a pitfall. Consider a set of \( m + 1 \) tasks. The left-hand side of Inequality 10 for each task \( \tau_k \) is at most \( 1 - \lambda_k \), since a sum target sums only \( m \) tasks, and each of them can hold a value at most \( 1 - \lambda_k \) due to a function of \( \min\{\alpha(\tau_k), 1 - \lambda_k\} \). Hence, none of those \( m + 1 \) tasks can have a sum strictly larger than \( m(1 - \lambda_k) \), which means that a set of \( m + 1 \) tasks is always verified to be schedulable. However, a set of \( m + 1 \) tasks can cause a deadline miss under EDZL [26]. The pitfall in the above schedulability analysis is an ignorance of the case in which all tasks take \( 1 - \lambda_k \) for \( \min\{\alpha(\tau_k), 1 - \lambda_k\} \) in Inequality 10. The schedulability test for EDZL must be revised as follows.

Theorem 6 (Revised EDZL schedulability test). A set of sporadic tasks is schedulable on \( m \) processors under EDZL, unless Inequality 10 holds for at least \( m + 1 \) tasks, and at least one of the \( m + 1 \) tasks satisfy:

- \( \sum_{i=1}^m \min\{\alpha(\tau_i), 1 - \lambda_k\} > m(1 - \lambda_k) \), or
- \( \sum_{i=1}^m \min\{\alpha(\tau_i), 1 - \lambda_k\} = m(1 - \lambda_k) \) when at least \( m \) other tasks satisfy \( \alpha(\tau_k) > 1 - \lambda_k \).

Proof 10. The proof follows the above discussion.
relative deadlines. The following analysis is based on the prior results for EDZL [13], but we consider necessary modifications for EDCL.

5.1.1. Later arrival times

Sustainability for later arrival times exactly follows the previous observation [13] that any scheduling policy is sustainable for sporadic task systems with respect to later arrival times. Hence, this property is proved as follows.

**Observation 1.** EDCL is sustainable for sporadic task systems with respect to later arrival times.

**Proof 11.** Let \( J \) denote a collection of jobs generated by a given task set \( \tau \). Let also \( J' \) denote any collection of jobs obtained from \( J \) by postponing the arrival times of one or more individual jobs of sporadic tasks. The collection \( J' \) of jobs could also have been generated by \( \tau \). □

5.1.2. Decreased execution times

Unlike the analysis for later arrival times, the proof of sustainability for EDCL with respect to decreased execution times is more complex than the previous work [13], since we need to consider that the priority promotion occurs with variable laxity for EDCL, while it occurs with a constant zero laxity for EDZL. A modification is required for a proof of this property.

Let \( J \) be a collection of jobs that is schedulable under the given scheduling algorithm, and \( J' \) be a collection of jobs obtained from \( J \) by decreasing the execution times of some jobs. For each job \( j \in J \), let \( j' \in J' \) denote the corresponding job obtained in this manner from \( J \). Let also \( \text{rem}(S, f, t) \) denote the remaining execution time of a job \( f \) in the schedule \( S \) at time \( t \), and \( d(f) \) denote the deadline of \( f \).

**Lemma 5.** EDCL is sustainable for sporadic task systems with respect to decreased execution times, if Inequality 11 is satisfied at any time \( t \)

\[
\forall f : \text{rem}(S, f, t) < \text{rem}(S, J, t)
\]  
(11)

where \( S \) and \( J \) denote the EDCL schedules for \( J \) and \( J' \), respectively.

**Proof 12.** Let \( J \) be any dispatched job in \( S \) other than \( J \), and \( J' \) be any dispatched job in \( S' \) other than \( J' \). Suppose that the lemma is false: i.e., there is a case that \( S' \) meets all deadlines while \( S \) does not when Inequality 11 is satisfied. By the property of EDCL, \( S' \) needs to satisfy

\[
\min(\text{rem}(S', J, t)) + \text{rem}(S', J, t) > d(J) - t
\]

for more jobs than \( S \). Given that \( \min(\text{rem}(S', J, t)) \leq \min(\text{rem}(S, J, t)) \) is always true from the lemma, if \( \min(\text{rem}(S', J, t)) + \text{rem}(S', J, t) < d(J) - t \) is true, \( \min(\text{rem}(S, J, t)) + \text{rem}(S, J, t) < d(J) - t \) is also true. We arrive at a contradiction to that \( S' \) satisfies \( \min(\text{rem}(S', J, t)) + \text{rem}(S', J, t) > d(J) \) for more jobs than \( S \). Hence, the lemma is true. □

**Lemma 5** indicates that EDCL has the same condition as EDZL to be sustainable with respect to decreased execution times. Now, we prove that Inequality 11 always holds true for EDCL. In the following, \( \text{run}(S, J, t) \) denotes the amount of the runtime that \( J \) has received in the schedule \( S \) over the interval \( [0, t) \), and \( \text{lat}(S, J, t) \) denotes the laxity of \( J \) in the schedule \( S \) at time \( t \): \( \text{lat}(S, J, t) = d(J) - t - \text{rem}(S, J, t) \).

**Lemma 6.** EDCL is sustainable for sporadic task systems with respect to decreased execution times, if deadline ties are broken in a deterministic manner.

**Proof 13.** Suppose that \( J \) is schedulable on \( m \) processors under EDCL. We provide a proof by contradiction. If the lemma is false, there exists a pair of \( J' \) and \( J'' \) that does not satisfy Inequality 11.

Given that (i) the execution requirement of \( J' \) is no greater than \( J \) due to the property of decreased execution times, (ii) \( \text{run}(S', J', 0) = \text{run}(S, J, 0) = 0 \), and (iii) \( d(J') = d(J) \) for any \( J \), Inequality 11 is true at time \( t = 0 \).

Let \( t_1 \) be the first time instant at which Inequality 11 is false, and \( J_1 \) be the job that makes it:

\[
\text{rem}(S', J_1, t_1) > \text{rem}(S, J_1, t_1).
\]

Since Inequality 11 is true at \( t_1 - 1 \) but is false at \( t_1 \), it must be the case that \( J_1 \) executes in \( S \), whereas \( J' \) does not execute in \( S' \) in interval \( [t_1 - 1, t_1) \). Note that we assume only integer times here.

From the definition of \( t_1 \), if some jobs in \( J \) are active at \( t_1 - 1 \), their corresponding jobs in \( J' \) are also active. Hence, the priority of \( J_1 \) in \( S \) must be higher than that of \( J_1 \) in \( S' \). Given that the deadlines of corresponding jobs are the same in both \( J \) and \( J' \), it must be the case:

1. \( J_1 \) is a critical job in \( S \) at time \( t_1 - 1 \), and
2. \( J_1 \) is not a critical job in \( S' \) at time \( t_1 - 1 \).

The second condition means that

\[
\min(\text{rem}(S, J_1, t_1) + \text{rem}(S', J_1, t_1 - 1)) < d(J_1) - (t_1 - 1)
\]

\( J_1 \) that contributed to \( \min(\text{rem}(S, J_1, t_1) + \text{rem}(S', J_1, t_1 - 1)) \) must execute in \([t_1 - 1, t_1) \) as well as \( J_1 \). Therefore, we derive from the above inequality that

\[
\min(\text{rem}(S', J_1, t_1)) + \text{rem}(S, J_1, t_1) < d(J_1) - t_1.
\]

On the other hand, since \( J_1 \) is a critical job in \( S \) at \( t_1 - 1 \), it also remains to be critical at \( t_1 \) by the property of EDCL that a job remains to be critical until it completes. Hence, we have

\[
\min(\text{rem}(S, J_1, t_1)) + \text{rem}(S, J_1, t_1) > d(J_1) - t_1.
\]

Since \( d(J_1) = d(J_1) \), it is a contradiction to \( \text{rem}(S', J_1, t_1) > \text{rem}(S, J_1, t_1) \). Hence, the lemma is true. □

5.1.3. Larger relative deadlines

We now focus on the sustainability property of EDCL with respect to larger relative deadlines; namely, the ability of EDCL to maintain the system to be schedulable by increasing (relaxing) relative deadlines. Unfortunately, we have not found the proof of this property yet. In fact, this property has not yet been found even for G-EDF and EDZL. We however sketch examples of some particular workload models for EDCL as well as G-EDF and EDZL, which cause deadline misses by increasing relative deadlines when all tasks arrive at the same time. These counterexamples do not necessarily mean that the scheduling algorithms are not sustainable. The algorithms may still be sustainable with these workloads due to sporadic arrivals, but the counterexamples indicate that we need careful consideration for sustainability with respect to larger relative deadlines.

**Observation 2.** G-EDF is not sustainable for arbitrarily-arriving job models with respect to larger relative deadlines.

**Proof 14.** Consider an EDF-scheduled system with \( m \) processors and a set \( \tau \) of tasks composed of \( m \) tasks, \( T_a = \{t_1, t_2, \ldots, t_m\} \), and one another task \( t_{m+1} \). For each task \( t_i \) in \( T_a \), \( dt_i = p_i \) and \( c_i = k \) are satisfied, where \( 2 < k < d_i - 1 \) and \( c_i/d_i < 0.5 \). Meanwhile, \( t_{m+1} \) has \( p_{m+1} = d_{m+1} = d_i - 1 \) and \( c_{m+1} = d_i - k + 2 \), where \( t_i \in T_a \).
This set of tasks is successfully scheduled under G-EDF when all tasks arrive at \( t_0 \). Because of \( d_{m+1} < d_i \), \( \tau_{m+1} \) is dispatched first on one processor \( P_i \), and the other \( m-1 \) tasks in \( T_a, t_1, t_2, \ldots, \tau_m \) are also dispatched on the remaining \( m-1 \) processors. These tasks apparently complete before their deadlines. At time \( t_0 + c_i \), the remaining one task \( \tau_a \) in \( T_a \) is dispatched on one processor \( P_2 \) (\( P_1 \rightarrow P_2 \)) and completes at time \( t_0 + 2c_i \). Since \( c_i/d_i \leq 0.5 \) is satisfied for all \( \tau \in T_a \), \( \tau_a \) also meets a deadline.

Now, we increase the relative deadline (and the period) of \( \tau_{m+1} \) by 2: i.e., \( d_{m+1} = d_i + 1 \). In this case, all the \( m \) tasks in \( T_a \) are dispatched before \( \tau_{m+1} \), because of \( d_{m+1} > d_i \), and they complete at time \( t_0 + c_i \). \( \tau_{m+1} \) is next dispatched at time \( t_0 + c_i \). However, because of \( c_i = k \) and \( c_{m+1} = d_i + k + 2 \), \( \tau_{m+1} \) completes at time \( d_i + 2 = d_{m+1} + 1 \). Hence, \( \tau_{m+1} \) misses its deadline by increasing its relative deadline.

**Observation 3.** EDZL is not sustainable for arbitrarily-arriving job models with respect to larger relative deadlines.

**Proof 15.** We first consider the case where a deadline is missed under EDZL. We next try to remove this deadline miss by decreasing the relative deadline of one task, meaning that there is a case where a system does not remain to be schedulable by increasing relative deadlines.

Consider an EDZL-scheduled system with \( m \) processors and a set \( \tau \) of tasks composed of two subsets \( T_a \) and \( T_s \). Tasks in \( T_a \) have the same timing properties \((c_i, d_i, p_i)\), and those in \( T_s \) also have the same timing properties \((c_i, d_i, p_i)\), where \( d_a \leq p_a \), \( d_s \leq p_s \), \( c_a \approx c_s \), and \( d_s < d_a \). 

The schedulability test of EDZL derived in Section 4.3 is not sustainable with respect to decreased execution times. In particular, \( W'_f \) is affected by \( c_i \) only when Eq. 6 evaluates \( A_k - (n_i + 1)p_i + \kappa(t_\tau) \) in the min/max function, and Eq. 5 evaluates \( d_k - c_k \) in the min/max function; i.e., \( W'_f \) is given by:

\[
A_k + (d_k - c_k) - (n_i + 1)p_i - c_i
\]

Having \( c_i \approx p_i \), the effect of \( \kappa \) is removed. Hence, if \( c_i \) is decreased by \( I \), the left-hand side of Condition 7 or Condition 8 in Theorem 3 can be increased by as much as \( (n_i - 1)I \), while its right-hand side can be increased by as much as \( m \times I \). As a consequence, a previously-satisfied inequality or equation may not hold, and a system considered to be schedulable may cease to be so after \( c_i \) is increased.

**Proof 16.** EDCL produces exactly the same schedule as Fig. 10, with a set of tasks introduced in Observation 3, and can avoid the deadline miss in the same manner.

**5.2. Sustainability of the schedulability test**

We now consider the sustainability of schedulability tests for EDCL, with respect to decreased execution times and larger relative deadlines.

**Lemma 7.** The schedulability tests for EDCL derived in Section 4.3 are not sustainable with respect to decreased execution times.

**Proof 17.** Let us assume that \( c_i \) is decreased by \( I \) for task \( \tau_i \). Since \( W_f(t_\tau) \) does not depend on the amount of \( c_\tau \), as implied by Eq. 4, we can focus on the case where competing tasks interfere with \( \tau_i \) by the amount of \( W_f \). In particular, \( W_f \) is affected by \( c_i \) only when Eq. 6 evaluates \( A_k - (n_i + 1)p_i + \kappa(t_\tau) \) in the min/max function, and Eq. 5 evaluates \( d_k - c_k \) in the min/max function; i.e., \( W'_f \) is given by:

\[
A_k + (d_k - c_k) - (n_i + 1)p_i - c_i
\]

Having \( c_i \approx p_i \), the effect of \( \kappa \) is removed. Hence, if \( c_i \) is decreased by \( I \), the left-hand side of Condition 7 or Condition 8 in Theorem 3 can be increased by as much as \( (n_i - 1)I \), while its right-hand side can be increased by as much as \( m \times I \). As a consequence, a previously-satisfied inequality or equation may not hold, and a system considered to be schedulable may cease to be so after \( c_i \) is increased.

**Proof 18.** Let us assume that \( d_i \) is increased by \( I \) for task \( \tau_i \). It is clear that \( W_f \) is increased by at most \( I \) when Eq. 4 evaluates \( A_k - np_i \) in the min/max function. Since \( \kappa(t_\tau) \) is also increased by at most \( I \) when Eq. 5 evaluates \( d_k - c_k \) in the min/max function, an increase in \( W'_f \) is also bounded by \( I \) as considered in Lemma 7.

**Fig. 10.** An example of a deadline miss caused by EDZL.
Hence, if \( d_k \) is increased by \( I \), the left-hand side of Condition 7 or Condition 8 in Theorem 3 can be increased by as much as \((n-1) \times I\), while its right-hand side can be increased by as much as \( m \times I \). The lemma is thus true.

**Theorem 7.** The schedulability test of EDCL is not sustainable.

**Proof 19.** It is led from Lemma 7 and 8. □

### 6. Simulation study

In this section, we evaluate the effectiveness of EDCL by simulations. We compare EDCL with the previous EDF-based algorithms, G-EDF, EDF-US\([1/2]\], EDZL, EDF-WM [27], and EDF-SS [28]. As mentioned before, EDF-US\([1/2]\] and EDZL are simple extensions to G-EDF. On the other hand, EDF-WM and EDF-SS are different types of algorithms based on a semi-partitioned scheduling discipline, where tasks are assigned to particular processors with a limited number of migrations across processors, which also improve schedulability with less runtime overheads than optimal algorithms.

#### 6.1. Simulation setup

We randomly generate 100,000 sets of sporadic tasks, each of which has the same total workload. The schedulability of each scheduling algorithm for the given workload is evaluated by simulating all the 100,000 task sets to obtain a success ratio:

- the number of successfully scheduled task sets.
- the number of scheduled task sets (100,000).

Each task set \( \tau \) with a total workload of \( U_{\text{tot}} \) is generated as follows. The utilization of task \( \tau_i \) is determined based on a uniform distribution within the range of \([0.1, 1.0] \). As long as \( \sum_{i \in \tau} u_i = U_{\text{tot}} \) is satisfied, \( \tau \) is inserted into \( \tau \). The utilization of the last task to be inserted is adjusted by \( u_l = U_{\text{tot}} - U(\tau) \) so that the total utilization is equal to \( U_{\text{tot}} \). The period of \( \tau \) is uniformly distributed in the range of \([1, 0.001, 100, 000] \), which simulates periods ranging from 1 ms to 100 ms. The WCET of \( \tau \) is computed as \( c_i = u_i \cdot p_i \).

We use two types of task sets with different deadline constraints. In an implicit-deadline scenario, the relative deadline of each task is set equal to the period, i.e., \( d_i = p_i \). In a constrained-deadline scenario, on the other hand, the relative deadline of each task is randomly set between the WCET and the period, i.e., \( d_i = [c, p_i] \).

The length of simulation time is \( \min(1, \text{cm} \cdot (p_i / c_i \in \tau))^2 \). Using such randomly-generated task sets, we evaluate schedulability test performance, runtime scheduling performance, and the average number of scheduler invocations per job. Schedulability test performance corresponds to the success ratio obtained by the schedulability tests of the algorithms. Runtime scheduling performance is estimated by the success ratio obtained by simulating all the schedules of the task sets. Theoretically, runtime scheduling performance should never be worse than schedulability test performance; otherwise, schedulability tests are incorrect.

For evaluating schedulability test performance of each algorithm, a task set is said to be successfully scheduled by one algorithm, if and only if the schedulability test accepted the task set. We run simulations for the following set of schedulability tests:

- **EDCL(Tight)** uses the tight test derived in Theorem 5.
- **EDCL(Pess)** uses the pessimistic test derived in Theorem 4.
- **EDZL** uses the EDF test revised in Theorem 6.
- **G-EDF** accepts a task set \( \tau \) if schedulability condition \( U(\tau) \leq m(1 - U_{\text{max}}) + U_{\text{max}} \) presented in [3] is satisfied, where \( U_{\text{max}} \) is the maximum utilization of every individual task. If this utilization-based test fails, we next apply the following schedulability conditions presented in [14]:
  - \( \sum_{i \in \tau} \min\{c_i(\tau_i), 1 - d_i\} < m(1 - d_i) \)
  - \( \sum_{i \in \tau} \min\{c_i(\tau_i), 1 - d_i\} < m(1 - d_i) \) and \( \exists \neq k: 0 < c_o(\tau_k) \)

- **EDF-US\([1/2]\)** uses the schedulability test of EDF-US\([1/2]\] considered in [15]. Specifically, it accepts a task set \( \tau \) if the \( n - h \) lowest-utilization tasks in \( \tau \) are accepted by the above G-EDF test for \( m - h \) processors, where \( h \) is the smaller of \( m - 1 \) or the number of tasks with utilization greater than \( 1/2 \) in \( \tau \).
- **EDF-SS(DTMIN)** accepts a task set \( \tau \) if all tasks in \( \tau \) are successfully assigned to particular processors, using the algorithm presented in [28], where a parametric length of the scheduling slot is set to \( \min(d_i, p_i) \), if all tasks in \( \tau \) are accepted by the above G-EDF test.
- **EDF-SS(DTMIN)** uses the tight test derived in Theorem 5, but the length of the scheduling slot is reduced to \( \min(d_i, p_i) / 4 \), as encouraged by the authors in [28].
- **EDF-SS** accepts a task set \( \tau \) if all tasks in \( \tau \) are successfully assigned to particular processors, using the algorithm presented in [27].

For evaluating runtime scheduling performance, meanwhile, a task set is said to be successfully scheduled by one algorithm, if and only if the task set was scheduled without missing any deadlines for the time interval of simulation. We use four variations of EDCL: **EDCL**, **EDCL(RM)**, **EDCL(LX)**, and **EDCL(DL)**. EDCL is a simple version that arbitrarily prioritizes critical jobs. EDCL(RM), EDCL(LX), and EDCL(DL), on the other hand, use the RM, the LX, and the DL policies presented in Section 3, respectively. During the runtime scheduling performance simulation, we also count the number of scheduler invocations and the number of jobs scheduled.

#### 6.2. Simulation results for implicit-deadline tasks

Figs. 11–14 show the results of simulations evaluating schedulability test performance for implicit-deadline task sets on 2, 4, 8, and 16 processors respectively. The algorithms are mostly classified into three groups in performance: (i) G-EDF and EDF-US\([1/2]\), (ii) EDCL(Tight), EDCL(Pess), and EDZL, and (iii) EDF-SS(DTMIN), EDF-SS(DTMIN), and EDF-WM. EDF-US\([1/2]\] in the first group are simple EDF-based algorithms, whereas EDCL and EDZL in the second group use laxity-driven priority promotions. Comparing these two groups, significant schedulability improvements are observed in EDCL and EDZL over G-EDF and EDF-US\([1/2]\]. Specifically, G-EDF and EDF-US\([1/2]\] increase the success ratio below 100% when the total workload reaches around a system utilization of 35–40%, whereas EDCL and EDZL maintain the success ratio at 100% until when the total workload reaches around a system utilization of 60–75. EDCL is consistently better than EDCL, but the tight schedulability test of EDCL shows very competitive performance to EDCL. As for the system utilization where success ratio starts dropping below 100%, the difference between EDCL(Tight) and EDCL is at most 5%. However, EDF-SS and EDF-WM in the third group, using semi-partitioned scheduling techniques, further outperform EDCL and EDZL in terms of schedulability test performance. This observation implies that semi-partitioned scheduling usually provides better schedulability than global scheduling. We claim that schedulability tests for global scheduling algorithms are often pessimistic, while those for semi-partitioned scheduling algorithms are tight. Specifically, task sets that are rejected by schedulability tests in global scheduling may be in fact schedulable at runtime, while those rejected by schedulability tests in semi-partitioned scheduling are often truly unschedulable. Hence, tightening schedulability tests will be a key issue for global scheduling.
Fig. 11. Schedulability test performance for implicit-deadline tasks on two processors.

Fig. 12. Schedulability test performance for implicit-deadline tasks on four processors.

Fig. 13. Schedulability test performance for implicit-deadline tasks on eight processors.

Fig. 14. Schedulability test performance for implicit-deadline tasks on 16 processors.
Fig. 15. Runtime scheduling performance for implicit-deadline tasks on two processors.

Fig. 16. Runtime scheduling performance for implicit-deadline tasks on four processors.

Fig. 17. Runtime scheduling performance for implicit-deadline tasks on eight processors.

Fig. 18. Runtime scheduling performance for implicit-deadline tasks on 16 processors.
Fig. 19. The average number of scheduler invocations for implicit-deadline tasks on two processors.

Fig. 20. The average number of scheduler invocations for implicit-deadline tasks on four processors.

Fig. 21. The average number of scheduler invocations for implicit-deadline tasks on eight processors.

Fig. 22. The average number of scheduler invocations for implicit-deadline tasks on 16 processors.
Fig. 23. Schedulability test performance for constrained-deadline tasks on two processors.

Fig. 24. Schedulability test performance for constrained-deadline tasks on four processors.

Fig. 25. Schedulability test performance for constrained-deadline tasks on eight processors.

Fig. 26. Schedulability test performance for constrained-deadline tasks on 16 processors.
Fig. 27. Runtime scheduling performance for constrained-deadline tasks on two processors.

Fig. 28. Runtime scheduling performance for constrained-deadline tasks on four processors.

Fig. 29. Runtime scheduling performance for constrained-deadline tasks on eight processors.

Fig. 30. Runtime scheduling performance for constrained-deadline tasks on 16 processors.
Fig. 31. The average number of scheduler invocations for constrained-deadline tasks on two processors.

Fig. 32. The average number of scheduler invocations for constrained-deadline tasks on four processors.

Fig. 33. The average number of scheduler invocations for constrained-deadline tasks on eight processors.

Fig. 34. The average number of scheduler invocations for constrained-deadline tasks on 16 processors.
algorithms. Semi-partitioned and global scheduling algorithms also have different advantages. For instance, global scheduling is more suitable when a system load needs to be balanced among processors, while semi-partitioned scheduling may impose heavy workloads on particular processors, since most tasks execute on particular processors without migrations.

Figs. 15–18 show the results of simulations that actually schedule the same sets of implicit-deadline tasks as those used in the previous experiments of schedulability test performance on 2, 4, 8, and 16 processors, respectively. Simulations exit when some job misses a deadline or the simulation duration expires. Unlike schedulability test performance, there is not a big difference among EDCL, EDZL, EDF-SS, and EDF-WM. This observation emphasizes that schedulability tests for global scheduling algorithms are pessimistic, and their potential runtime scheduling performance could be better than semi-partitioned scheduling algorithms. It is also interesting to see that EDCL is very competitive to EDZL, particularly when efficient prioritization policies for critical jobs are used, far beyond the performance of G-EDF and EDF-US[1/2]. The three prioritization policies, LX, RM, and DL, provide similar perfor-

![Graphs showing scheduler invocations per job for different processors.](image-url)

**Fig. A.35.** The average number of scheduler invocations for implicit-deadline tasks, including the EDF-SS algorithm.
mance benefits, but the LX policy is slightly better than the other two policies. While EDCL is very competitive to EDZL, EDZL is still better. EDZL can schedule mostly all the given task sets, even though a system utilization is close to 100%. However, we claim that EDZL needs to use high-resolution timers for practical implementation, which could pay more penalty than the performance benefit over EDCL.

Figs. 19–22 show the average number of scheduler invocations per job, measured during the same scheduling simulations performed above for 2, 4, 8, and 16 processors respectively. Due to the scale problems, we omit the results for EDF-SS here, since it causes 20–500 times as many scheduler invocations as the other algorithms. Interested readers are encouraged to see the Appendix section, showing the results including the EDF-SS test case in Fig. A.35. According to the simulation results, G-EDF, EDF-US[1/2], EDCL, and EDZL invoke the scheduler at most twice per job, while EDF-WM causes more scheduler invocations. It should be noted that when EDF-WM occasionally has zero scheduler invocations when the system utilization is 100%. This is because no task sets are successfully scheduled by EDF-WM, as shown in the previ-

![Graphs showing average number of scheduler invocations per job](image_url)
uous simulation results. Given that EDCL and EDZL show outstanding scheduling performance, as observed in the previous simulation results, with a small number of scheduler invocations, we conclude that EDCL and EDZL are very efficient algorithms for scheduling implicit-deadline tasks on multiprocessors.

6.3. Simulation results for constrained-deadline tasks

We next study simulations for constrained-deadline task sets. Fig. 23–26 show the results of simulations performing schedulability tests for constrained-deadline task sets on 2, 4, 8, and 16 processors respectively. Surprisingly, both EDCL and EDZL outperform even EDF-SS and EDF-WM for 2 processors. It is also interesting to see that EDF-WM is better than EDF-SS. Since EDF-SS decides the length of scheduling slot based on the minimum of relative deadlines and periods in all tasks, the length of scheduling slot is likely to be smaller, when relative deadlines can be as small as WCETs, producing many idle slots left unused by migratory tasks. Hence, the schedulability of EDF-SS tends to degrade for constrained-deadline tasks. EDF-WM also faces schedulability degrada-

tion for constrained-deadline tasks, since it splits relative deadlines of migratory tasks, which can result in having tasks with very short relative deadlines. On the other hand, EDCL and EDZL prevent schedulability degradation in a much better way than EDF-SS and EDF-WM. In particular, EDCL is still very competitive to EDZL, as the previous test case for implicit-deadline tasks. There is only a little difference between EDCL and EDZL, regarding the system utilization where a success ratio starts dropping below 100%. It is also surprising that the schedulability test of EDF-US[1/2] shows better performance than those of EDCL and EDZL, when a system utilization is close to 100% in the case of 2 processors. Therefore, there is a need to further explore if EDCL and EDZL strictly dominate EDF-US[x].

Fig. 27–30 show the results of simulations that actually schedule the same sets of constrained-deadline tasks as the previous experiments of schedulability test performance on 2, 4, 8, and 16 processors, respectively. Simulations exit when some job misses a deadline or the simulation duration expires. In this set of experiments, EDCL and EDZL significantly outperform other algorithms particularly for a larger number of processors. This implies that global scheduling algorithms could be potentially better than semi-partitioned scheduling algorithms for constrained-deadline tasks, since the difference between EDCL and EDZL is not trivial when the number of processors is 2 or 4, while there is less difference when the number of processors becomes larger. On 16 processors, for instance, EDCL provides almost the same schedulability as EDZL.

Fig. 31–34 show the average number of scheduler invocations per job, measured during the same scheduling simulations performed above for 2, 4, 8, and 16 processors, respectively. By the same reason as the case for implicit-deadline tasks, the results for EDF-SS are shown in the Appendix section (Fig. A.36). While G-EDF, EDF-US[1/2], and EDCL maintain the number of scheduler invocations at most twice per job, EDZL and EDF-WM cause more scheduler invocations. Unlike the previous case for implicit-deadline tasks, the completions of jobs with zero laxity are not aligned with the arrivals of the next jobs, since the deadlines may come earlier than the beginning of the next period for constrained-deadline tasks. Hence, the number of scheduler invocations per job for EDZL is increased to as much as 3, as we demonstrated in Section 1. Given that EDCL shows almost the same schedulability as EDZL for a large number of processors, EDCL is more beneficial for constrained-deadline tasks, when the cost of scheduler invocations is considered. As a consequence, EDCL is a useful contribution for scheduling sporadic task systems on multiprocessors.

7. Conclusion

We have presented an EDF-based algorithm, Earliest Deadline Critical Laxity (EDCL), for scheduling sporadic task systems on multiprocessors, which uses the laxity-driven priority promotion to improve schedulability. To the best of our knowledge, EDCL is the first algorithm that strictly dominates G-EDF, while bounding the number of scheduler invocations to be the same as G-EDF. We also have presented the pessimistic and tight schedulability tests for EDCL, which address the trade-off between the tightness of schedulability test and computation overhead. Furthermore, the sustainability of EDCL has been studied to show that EDCL is sustainable for sporadic task systems with respect to later arrival times and decreased execution times, while the sustainability property with respect to larger relative deadlines has not been evinced yet.

Our simulation studies demonstrated that EDCL is competitive to EDZL in schedulability, far beyond G-EDF and EDF-US[x]. Comparing with semi-partitioned scheduling algorithms, EDF-SS and EDF-WM, the schedulability tests for EDCL and EDZL are still inferior to those for EDF-SS and EDF-WM, but we showed that EDCL and EDZL could perform better than those algorithms at runtime, when tasks are actually scheduled. Therefore, it is important to explore tightening the schedulability tests for EDCL and EDZL. Our simulation tool used in this paper may be downloaded from our website at http://www.ece.cmu.edu/shinpei/sim/.

In future work, we will consider a utilization bound for EDFCL, which is an upper bound on system utilization, where any set of tasks is schedulable. While we expect that there is a set of tasks that is not schedulable under EDCL, if the total utilization exceeds (m + 1)/2, we will formalize this result to explicitly define a utilization bound for EDCL. We are also interested in mitigating a meaningful gap between schedulability test and runtime scheduling performance for EDCL. In addition, a domination property among EDF-based algorithms needs to be studied more. We will look into if EDCL and EDZL dominate EDF-US[x], and also if EDZL dominates EDCL.

Appendix A. Additional simulation results

Fig. A.35 shows simulation results of the average number of scheduler invocations per job for implicit-deadline tasks, including EDF-SS [28]. Since EDF-SS configures the length of scheduling slot for migratory tasks to be the minimum of relative deadlines and periods in all tasks, if there are tasks with short relative deadlines and periods on one processor, scheduling points are increased on all processors. Particularly, our implementation of EDF-SS invokes the scheduler at the beginning and the end of every slot on all processors, even though there are no tasks to be scheduled. Therefore, the number of scheduler invocations for EDF-SS may be reduced by using more efficient implementation. However, we claim that such scheduling points need to be made by using high-resolution timers. Since the next arrival time of job is not known precisely for the sporadic task model, when the scheduler is invoked, scheduler invocations at the beginning and the end of all scheduling slots may eventually be required. Fig. A.36 shows simulation results of the average number of scheduler invocations per job for constrained-deadline tasks, including the EDF-US algorithm. Since relative deadlines may be much shorter than periods in those test cases, the length of scheduling slot for EDF-US is likely to be much smaller than the previous experiments. As a result, the number of scheduler invocations per job is also increased dramatically.

References
